Notes du Cours de Sémantique

Roberto Di Cosmo

27 mai 2004

Chapitre 1

Σ-algèbres

Les Σ-algèbres et les algèbres de termes

Les signatures et les termes finis

Définition 1.0.1 (Signature) *Une signature est constituée de deux ensembles :*
- *un ensemble fini $S = \{s_1, \ldots, s_k\}$ de symboles de* sortes
- *un ensemble non vide de symboles de fonction t.q. chaque $f \in \Sigma$ possède une arité $s_1 * \ldots * s_n, s$.*

Définition 1.0.2 (Σ-algèbre) *Etant donnée une signature Σ, une Σ-algèbre A est donnée par*
- *un ensemble A_{s_i} pour toute sorte s_i de Σ (cet ensemble est appelé le "support" de s_i)*
- *pour chaque symbole de fonction $f \in \Sigma$ d'arité $s_1 * \ldots * s_n, s$, une fonction $f^A : A_{s_1} \times \ldots \times A_{s_n} \to A_s$, appelée interprétation de f.*

Exemple : a faire...

Définition 1.0.3 (Les termes sur une signature Σ) *Soit Σ une signature, et $\mathcal{X}$ une famille d'ensembles $\mathcal{X}_s$ de variables pour chaque sorte s de Σ. L'ensemble $\mathcal{T}_\Sigma(\mathcal{X})$ de termes sur $\mathcal{X}$ et Σ est défini par :*

- Toute variable $x \in \mathcal{X}_s$ est un terme de sorte s dans $\mathcal{T}_\Sigma(\mathcal{X})$

– Si f est d'arité $s_1 * \ldots * s_n, s$, et $t_1, \ldots, t_n$ sont dans $\mathcal{T}_\Sigma(\mathcal{X})$ de sorte $s_1, \ldots s_n$ respectivement, alors $f(t_1, \ldots, t_n)$ est un terme de sorte s dans $\mathcal{T}_\Sigma(\mathcal{X})$

On note $Var(t)$ l'ensemble de variables du terme t. Un terme t est clos si $Var(t) = \emptyset$. L'ensemble de termes clos est noté $\mathcal{T}_\Sigma(\emptyset)$ ou $\mathcal{T}(\Sigma)$.

Exercice : Vérifiez que :

– Les termes sur la signature Σ et variables $\mathcal{X}$, $\mathcal{T}_\Sigma(\mathcal{X})$, forment bien une Σ-algèbre (A_{s_i} est l'ensemble de termes de sorte s_i, et $f^{\mathcal{A}}(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$). Cette algèbre est aussi dite algèbre syntaxique.

– Les termes clos $\mathcal{T}(\Sigma)$ forment bien une Σ-algèbre : (A_{s_i} est l'ensemble de termes clos de sorte s_i $f^{\mathcal{A}}(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$). Cette algèbre est aussi dite algèbre syntaxique close.

Dans la suite, par simplicité, on travaillera souvent sur des algèbres mono-sortées (i.e. ayant une seule sorte, que l'on peut alors ne pas indiquer explicitement).

Les suites d'entiers, les positions d'un terme

L'ensemble $\mathbb{N}^*$ de suites sur $\mathbb{N}$ est le plus petit ensemble t.q.

– $\epsilon \in \mathbb{N}^*$

– Si $i \in \mathbb{N}$ et $p \in \mathbb{N}^*$, alors $ip \in \mathbb{N}^*$

L'ensemble $Pos(t)$ de positions d'un terme t est un sous-ensemble de $\mathbb{N}^*$ défini par :

– $\epsilon \in Pos(t)$

– Si $t = f(t_1, \ldots, t_n)$ et $p \in Pos(t_i)$, alors $ip \in Pos(t)$ pour tout $1 \leq i \leq n$.

La relation $\leq_{pref}$ sur les suites d'entiers

L'opération de concatenation sur deux positions de $\mathbb{N}^*$ est définie par induction comme suit :

$$\begin{array}{lcl} \epsilon.q & = & q \\ (ip).q & = & i(p.q) \end{array}$$

La relation préfixe $\leq_{pref}$ sur $\mathbb{N}^* \times \mathbb{N}^*$ est définie par : $p \leq_{pref} q$ ssi $\exists r \in \mathbb{N}^*$ t.q. $p.r = q$

Les positions p et q sont incompatibles ou parallèles, noté $p \bowtie q$, ssi $p \not\leq_{pref} q$ et $p \not\leq_{pref} q$.

Les sous-termes d'un terme

Soit t un terme. L'ensemble $ST(t)$ de sous-termes de t est défini par :

- $ST(x) = \{x\}$.
- $ST(f(t_1, \ldots, t_n)) = \{f(t_1, \ldots, t_n)\} \cup \{v \mid v \in ST(t_i)\}$.

L'ensemble de sous-termes stricts d'un terme t est l'ensemble de sous-termes auquel on a enlevé t.

On écrit $t \trianglerighteq s$ (resp. $t \triangleright s$) si s est un sous-terme (resp. strict) de t.

Sous-terme à une position

Soit t un terme et $p \in Pos(t)$. Le sous-terme de t à la position p, noté $t|_p$, est défini par récurrence sur p par :

- $t|_\epsilon = t$.
- $f(t_1, \ldots, t_n)|_{i.q} = t_i|_q$.

Le remplacement du sous-terme $t|_p$ par un terme v, noté $t[p \leftarrow v]$ ou $t[v]_p$, est défini comme suit :

- $t[v]_\epsilon = v$
- $f(t_1, \ldots, t_n)[v]_{i.p} = f(t_1, \ldots, t_i[v]_p, \ldots, t_n)$

Sous-algèbres

Une Σ-algèbre $\mathcal{A}_1$ est une sous-algèbre de la Σ-algèbre $\mathcal{A}_2$ ssi

- $\mathbf{A_1} \subseteq \mathbf{A_2}$
- pour tout $n \geq 0$, pour tout symbole $f \in \Sigma$ d'arité n et pour tout $a_1, \ldots, a_n \in \mathbf{A_1}$ on a $f^{\mathcal{A}_1}(a_1, \ldots, a_n) = f^{\mathcal{A}_2}(a_1, \ldots, a_n)$.

Congruences

Soit R une relation sur une Σ-algèbre $\mathcal{A}$.
On dit que $f \in \Sigma$ est monotone par rapport à R ssi pour tout $i = 1 \ldots n$, $a_i \; R \; a'_i$ implique $f^{\mathcal{A}}(a_1, \ldots, a_i, \ldots, a_n) \; R \; f^{\mathcal{A}}(a_1, \ldots, a'_i, \ldots, a_n)$.

Une congruence $\sim$ sur une Σ-algèbre est une relation d'équivalence (réflexive, symétrique, transitive), t.q. tout symbole $f \in \Sigma$ est monotone par rapport à $\sim$.

Notation 1.0.4 *$S/_\sim$ est l'ensemble de classes d'équivalence d'une Σ-algèbre S modulo une congruence $\sim$ sur S.*

L'algèbre quotient

Si $\sim$ est une congruence sur une Σ-algèbre $\mathcal{A}$, alors $\mathcal{A}^{\sim}$ est une Σ-algèbre, dite algèbre quotient sur $\mathcal{A}$ t.q.

– son domain est $\mathcal{A}/_{\sim}$
– pour chaque $f \in \Sigma$, $f^{\mathcal{A}^{\sim}}([a_1], \ldots, [a_n]) = [f^{\mathcal{A}}(a_1, \ldots, a_n)]$.

Homomorphismes, endomorphismes, isomorphismes

Soit $\mathcal{A}$ et $\mathcal{B}$ deux Σ-algèbres. Un homomorphisme (ou morphisme) est une fonction $\Phi : \mathcal{A} \to \mathcal{B}$ t.q. pour tout $n \geq 0$, pour tout symbole $f \in \Sigma$ d'arité n et pour tout $a_1, \ldots, a_n \in \mathbf{A}$ on a

$$\Phi(f^{\mathcal{A}}(a_1, \ldots, a_n)) = f^{\mathcal{B}}(\Phi(a_1), \ldots, \Phi(a_n))$$

Un endomorphisme sur une Σ-algèbre $\mathcal{A}$ est un morphisme de $\mathcal{A}$ sur elle même. Un isomorphisme est un morphisme bijectif (injectif et surjectif).

Algèbres initiale

Une Σ-algèbre $\mathcal{I}$ est initiale dans $\mathcal{K}$ ssi pour toute Σ-algèbre $\mathcal{A} \in \mathcal{K}$ il existe un unique morphisme $\Phi : \mathcal{I} \to \mathcal{A}$.

Assignations

Soit $\mathcal{A}$ une Σ-algèbre et soit $\mathcal{X}$ un ensemble de variables.

Une $\mathcal{A}$-assignation est une application $\sigma : \mathcal{X} \to \mathcal{A}$.

Théorème 1.0.5 *Pour toute $\mathcal{A}$-assignation $\sigma : \mathcal{X} \to \mathcal{A}$, il existe un unique morphisme $\widehat{\sigma} : \mathcal{T}_{\Sigma}(\mathcal{X}) \to \mathcal{A}$, t.q.*

$$\begin{array}{lcl} \widehat{\sigma}(x) & = & \sigma(x) \\ \widehat{\sigma}(f(t_1, \ldots, t_n)) & = & f^{\mathcal{A}}(\widehat{\sigma}(t_1), \ldots, \widehat{\sigma}(t_n)) \end{array}$$

Notation 1.0.6 *On confond σ et $\widehat{\sigma}$*

Substitutions

Une substitution est un endomorphisme de $\mathcal{T}_{\Sigma}(\mathcal{X})$ en $\mathcal{T}_{\Sigma}(\mathcal{X})$.

Une substitution close est un morphisme de $\mathcal{T}_{\Sigma}(\mathcal{X})$ en $\mathcal{T}(\Sigma)$.

Le domaine d'une substitution θ est l'ensemble $Dom(\theta) = \{x \in \mathcal{X} \mid \theta(x) \neq x\}$.

Une substitution est finie si son domain est fini. Dans ce cas, on note $\theta = \{x_1/t_1, \ldots, x_n/t_n\}$ si $\theta(x_i) = t_i$ pour tout $1 \leq i \leq n$.

L'image d'une substitution est l'ensemble $Im(\theta) = \{\theta(x) \mid x \in Dom(\theta)\}$. On note $VarIm(\theta)$ est l'ensemble $\bigcup_{x \in Dom(\theta)} Var(\theta(x))$.

Un renommage θ est une substitution bijective de $Dom(\theta)$ sur $VarIm(\theta)$.

Unification

Deux termes s et t sont unifiables ss'il existe une substitution t.q. $\theta(s) = \theta(t)$ (θ est donc un unificateur de s et t).

La composition de deux substitutions θ et τ est définie par $(\theta \circ \tau)(x) = \widehat{\theta}(\tau(x))$ pour toute variable $x \in \mathcal{X}$.

Soient θ et τ deux substitutions. θ est une instance de τ (ou τ est plus générale que θ) ss'il existe une substitution ρ t.q. pour toute variable $x \in \mathcal{X}$, $(\rho \circ \tau)(x) = \theta(x)$.

Soit $\mathcal{S}$ un ensemble de substitutions. Une substitution $\theta \in \mathcal{S}$ est principale ssi toute substitution $\tau \in \mathcal{S}$ est une instance de θ.

Unificateur principal

Théorème 1.0.7 *Soit $\mathcal{E}$ l'ensemble (non vide) d'unificateurs de deux termes s et t. Alors il existe un unificateur $\theta \in \mathcal{E}$ appelé unificateur principal t.q. pour tout $\tau \in \mathcal{E}$, θ est plus général que τ. De plus, cet unificateur principal est unique à renommage près.*

Une substitution θ est idempotente ssi $\theta \circ \theta = \theta$.

Théorème 1.0.8 *Si s et t sont unifiables, alors il existe un unificateur principal de s et t qui est idempotent.*

La logique équationnelle

Équations : syntax et sémantique

Une Σ-équation est une paire de termes noté $s \doteq t$.

Une Σ-algèbre $\mathcal{A}$ est un modèle d'une Σ-équation $s \doteq t$, noté $\mathcal{A} \models s \doteq t$, ssi pour toute $\mathcal{A}$-assignation σ on a $\widehat{\sigma}(s) = \widehat{\sigma}(t)$.

On dit aussi que $s \doteq t$ est valide pour $\mathcal{A}$.

Une Σ-algèbre $\mathcal{A}$ est un modèle d'un ensemble de Σ-équations $\mathcal{E}$, noté $\mathcal{A} \models \mathcal{E}$, ssi $\mathcal{A}$ est un modèle de toute équation de $\mathcal{E}$.
On dit aussi que $\mathcal{E}$ est valide pour $\mathcal{A}$.

Consequénce sémantique

Soit $\mathcal{E}$ un ensemble de Σ-équations et $s \doteq t$ une Σ-équation quelconque.

L'équation $s \doteq t$ est une consequénce logique de l'ensemble $\mathcal{E}$, noté $\mathcal{E} \models s \doteq t$, ssi tout modèle de $\mathcal{E}$ est aussi un modèle de $s \doteq t$.

La théorie équationnelle engendrée par $\mathcal{E}$ est l'ensemble $\doteq_{\mathcal{E}} = \{(s,t) \in \mathcal{T}_\Sigma(\mathcal{X}) \times \mathcal{T}_\Sigma(\mathcal{X}) \mid \mathcal{E} \models s \doteq t\}$.

Exercice : Montrer que la relation $\doteq_{\mathcal{E}}$ est une congruence sur $\mathcal{T}_\Sigma(\mathcal{X})$.

Règles syntaxiques pour le raisonnement équationnel

$$\frac{s \doteq t \in \mathcal{E}}{s \doteq t} \text{ (axiome)} \qquad \frac{}{s \doteq s} \text{ (réflexivité)}$$

$$\frac{s \doteq t}{t \doteq s} \text{ (symétrie)} \qquad \frac{s \doteq t \quad t \doteq u}{s \doteq u} \text{ (transitivité)}$$

$$\frac{s \doteq t}{\sigma(s) \doteq \sigma(t)} \text{ (substitution)} \qquad \frac{s \doteq t}{u[s]_p \doteq u[t]_p} \text{ (contexte)}$$

Dérivation

Une dérivation de l'équation $s \doteq t$ à partir d'un ensemble $\mathcal{E}$ est un arbre d'équations t.q.

- La racine est $s \doteq t$
- Si E est une feuille, alors $E \in \mathcal{E}$ ou E est une équation $s \doteq s$.
- Si $E_1, \ldots, E_n$ sont les fils de E, alors E est obtenue à partir de $E_1, \ldots, E_n$ par une règle syntaxique.

L'équation $s \doteq t$ est dérivée à partir de $\mathcal{E}$, noté $\mathcal{E} \vdash s \doteq t$, ssi il existe une dérivation de $s \doteq t$ à partir de $\mathcal{E}$.

La relation $\leftrightarrow_{\mathcal{E}}^*$

$$\frac{s \doteq t \in \mathcal{E}}{\sigma(s) \rightarrow_{\mathcal{E}} \sigma(t)} \quad (\forall \sigma) \qquad \frac{s \rightarrow_{\mathcal{E}} t}{u[s]_p \rightarrow_{\mathcal{E}} u[t]_p} \quad (\forall u \forall p)$$

$\leftrightarrow_{\mathcal{E}}$ est la fermeture symétrique de $\rightarrow_{\mathcal{E}}$
$\leftrightarrow_{\mathcal{E}}^*$ est la fermeture réflexive, symétrique et transitive de $\rightarrow_{\mathcal{E}}$

À propos de $\leftrightarrow_{\mathcal{E}}^*$

Exercice : Montrer que $\leftrightarrow_{\mathcal{E}}^*$ est une congruence sur $\mathcal{T}_\Sigma(\mathcal{X})$.

Exercice : Montrer que $\mathcal{E} \vdash s \doteq t$ ssi $s \leftrightarrow_{\mathcal{E}}^* t$.

Lemme de substitution (i)

Soit $\mathcal{A}$ une Σ-algèbre, σ une $\mathcal{A}$-assignation et θ une substitution. Alors pour tout terme $t \in \mathcal{T}_\Sigma(\mathcal{X})$ on a

$$\sigma(\theta(t)) = \sigma'(t)$$

où $\sigma'(x) = \sigma(\theta(x))$ pour toute variable $x \in \mathcal{X}$.

Lemme de substitution (ii)

Considérons la Σ-algèbre $\mathcal{T}_\Sigma(\mathcal{X})^{\leftrightarrow_{\mathcal{E}}^*}$.

Soit σ une $\mathcal{T}_\Sigma(\mathcal{X})^{\leftrightarrow_{\mathcal{E}}^*}$-assignation et θ une substitution t.q $\sigma(x) = [\theta(x)]$ pour toute variable $x \in \mathcal{X}$. Alors pour tout terme $t \in \mathcal{T}_\Sigma(\mathcal{X})$ on a

$$\sigma(t) = [\theta(t)]$$

Le modèle $\mathcal{T}_\Sigma(\mathcal{X})^{\leftrightarrow_{\mathcal{E}}^*}$

Exercice : L'algèbre $\mathcal{T}_\Sigma(\mathcal{X})^{\leftrightarrow_{\mathcal{E}}^*}$ est un modèle de $\mathcal{E}$.

Théorème d'adéquation (Birkhoff 1933)

(Correction) Si $\mathcal{E} \vdash s \doteq t$, alors $\mathcal{E} \models s \doteq t$.

(Complétude) Si $\mathcal{E} \models s \doteq t$, alors $\mathcal{E} \vdash s \doteq t$.

Ce premier théorème de Birkhoff nous dit que la derivabilité syntaxique et la conséquence sémantique coincident, dans la logique équationnelle.

Un deuxième théorème de Birkhoff nous donne les propriétés intrinseques de la classe de Σ-algèbres qui sont les modèles d'une théorie équationnelle donnée.

Définition 1.0.9 (Produit direct de Σ-algèbres) *Soit $A_1, \ldots, A_k$ une famille finie de Σ-algèbres. Le produit direct P de $A_1, \ldots, A_k$ est la Σ-algèbre définie comme suit :*

- $P_{s_i} = A_{1 s_i} \times \ldots \times A_{k s_i}$
- *pour f de signature $s_1 * \ldots * s_n, s$ on définit f^P comme*

$$f^P(< a_1^1, \ldots, a_k^1 >, \ldots, < a_1^n, \ldots, a_k^n >) = \\ < f^{A_1}(a_1^1, \ldots, a_1^n), \ldots, f^{A_k}(a_k^1, \ldots, a_k^n) >$$

Exercice : Vérifier que le produit direct de k Σ-algèbres est bien une Σ-algèbre.

Définition 1.0.10 (Variété) *Une classe $\mathcal{K}$ de Σ-algèbres est une variété si ell'est* close *par les opérations suivantes :*

sousalgèbre *(i.e., si $A \in \mathcal{K}$, et B est une sousalgèbre de A, alors $B \in \mathcal{K}$)*

image homomorphe *(i.e., si $A \in \mathcal{K}$, B une Σ-algèbre, et $B = \Phi(A)$ pour un homomorphisme Φ, alors $B \in \mathcal{K}$))*

produit direct *(i.e., si $A_1, \ldots, A_k \in \mathcal{K}$ est P est le produit direct de $A_1, \ldots, A_k$, alors $P \in \mathcal{K}$*

Définition 1.0.11 (Classe des modèles) *On note $\mathcal{M}od(\mathcal{E})$ la classe des Σ-algèbres qui satisfont les équations $\mathcal{E}$ (les modèles de $\mathcal{E}$).*

Théorème 1.0.12 (Birkhoff, 1935) *Une classe $\mathcal{K}$ de Σ-algèbres est $\mathcal{M}od(E)$ pour un quelque ensemble d'équations $\mathcal{E}$) ssi ell'est une variété.*

L'application la plus intéressante du théorème de Birkhoff est la preuve que certaines opérations ne sont pas axiomatisables par le biais d'une théorie équationnelle.

Exercice : Vérifié que la conditionnelle stricte définie par les équations conditionnelles suivantes n'est pas axiomatisable par une théorie équationnelle.

$$
\begin{array}{lcll}
ITE(b, e1, e2) & = & \bot & \text{si } b = \bot \text{ ou } e1 = \bot \text{ ou } e2 = \bot \\
ITE(true, e1, e2) & = & e1 & \\
ITE(false, e1, e2) & = & e2 &
\end{array}
$$

Chapitre 2

Le lambda calcul

Dans ce chapitre, nous allons étudier le lambda calcul, noyau de tout langage fonctionnel. On énonçe quelques propriétés fondamentales, comme la confluence et la Turing-équivalence, puis on étudie les stratégie d'évaluation, par le biais de la sémantique à petis et grands pas. On va ensuite mettre en relation ces deux sémantiques.

2.1 Syntaxe du lambda calculs

a rédiger : syntaxe, substitution, convention de Barendregt, syntaxe à la Krivine, règle beta.

Codage des fonctions récursives.

2.2 Confluence

à rédiger : schéma de la preuve à la Tait/Martin-Lof.

2.3 Turing équivalence

à rédiger : codage des fonctions recursives.

2.3.1 Sémantique opérationnelle structurée (SOS) du lambda calcul non typé

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

L'appel par nom : $\Rightarrow_n$

application :

$$(\lambda x.M)N \Rightarrow_n M[N/x] \qquad \frac{e_1 \Rightarrow_n e'}{e_1 e_2 \Rightarrow_n e' e_2}$$

L'appel par valeur : $\Rightarrow_v$

Remarque 2.3.1 (Valeurs) *On appelle ici valeurs les abstractions, et on les marque dans ce qui suit par des noms commençant par v.*

application :

$$(\lambda x.M)v \Rightarrow_v M[v/x] \qquad \text{(si } v \text{ est un valeur)}$$

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

Exercice 2.3.2 *Est-ce que l'appel par valeur et par nom donnent des résultats différents dans la lambda calcul non typé ? Pourquoi ?*

Théorème 2.3.3 (Confluence) *Si $M \Rightarrow N'$ et $M \Rightarrow N''$, il existe N''' t.q. $N' \Rightarrow N'''$ et $N'' \Rightarrow N'''$.*

Preuve. Conséquence du fait que la réduction est deterministe. □

Exercice 2.3.4 (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 $v \equiv v'$*

2.3.2 Sémantique opérationnelle naturelle du lambda calcul non typé

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

L'appel par nom : $\Downarrow_n$

abstraction fonctionnelle et application :

$$\lambda x.M \ \Downarrow_n \ \lambda x.M$$

$$\frac{e_1 \ \Downarrow_n \ \lambda x.e_0 \qquad e_0[e_2/x] \ \Downarrow_n \ v}{e_1(e_2) \ \Downarrow_n \ v}$$

L'appel par valeur : $\Downarrow_v$

abstraction fonctionnelle et application :

$$\lambda x.M \ \Downarrow_v \ \lambda x.M$$

$$\frac{e_1 \ \Downarrow_v \ \lambda x.e_0 \qquad e_2 \ \Downarrow_v \ v_2 \qquad e_0[v_2/x] \ \Downarrow_v \ v_0}{e_1(e_2) \ \Downarrow_v \ v_0}$$

2.3.3 Quelques résultats

Définition 2.3.5 (Valeurs) *On définit* valeurs *les abstractions.*

Proposition 2.3.6 (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 2.3.7 (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. □

2.4 Relation entre les sémantiques

Proposition 2.4.1 (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 $M \Rightarrow_* v$

□

Corollaire 2.4.2 ($\Rightarrow$ sur termes clos) *En appel par nom ou par valeur, pour tout terme $clos$ (i.e. sans variables libres) M, il existe un valeur v t.q. $M \Rightarrow v$.*

Preuve. Par induction sur la longueur de la réduction. □

Corollaire 2.4.3 ($\Downarrow$ sur termes clos) *En appel par nom ou par valeur, pour tout terme $clos$ (i.e. sans identificateurs non déclarés) M, il existe un valeur v t.q. $M \Downarrow v$.*

Preuve. Utiliser le corollaire précédent. □

Exercice 2.4.4 *Montrez que pour tout terme clos e, $e \Rightarrow_v v$ si et seulement si $e \Downarrow_n v$.*

2.5 Equivalence Observationnelle

Comme la sémantique opérationnelle ne donne pas vraiement de sémantique aux termes sans entrées, il arrive souvent que elle ne soit pas adequate pour identifier termes que calculent la même fonction. Par exemple, dans n'importe laquelle des sémantiques operationnelles déjà vues, on a que :

$$\lambda x.(\lambda y.y)x \Downarrow \lambda x.(\lambda y.y)x$$

$$\lambda x.x \Downarrow \lambda x.x$$

On a très envie de identifier λ x.x et λ x.(λ y.y) x, vu que les deux calculent bien la fonction identité, mais le problème est que, pour s'en apercevoir, il faut appliquer les deux termes à un argument.

C'est bien cela l'idée qui est à la base de la notion d'equivalence observationnelle : on cherche à définir une notion d'équivalence de termes qui identifie exactement ces (fragments de) termes "qui font la même chose" du point de vue d'un observateur choisi.

Définition 2.5.1 (Contexte) *Un* $\mathit{contexte}$ *avec un trou, noté* $C[\ ]$*, est un (fragment de) terme où une variable spéciale distinguée* $[\ ]$ *apparait au plus une fois. On note* $C[M]$ *le (fragment de) terme obtenu en remplaçant* $[\ ]$ *par* M *textuellement (i.e. , sans faire les* α*-conversions comme dans la substitution du* λ*-calcul).*

Définition 2.5.2 (Equivalence observationnelle) *Fixée une sémantique opérationnelle* S*, une notion de* $\mathit{observation}$ O_S*, et une notion de* terme*, deux fragments de termes* M *et* N *sont dit equivalent observationnellement, par rapport à* O_S *et* S*, ssi :*

$$\forall C[] \text{ t.q. } C[M] \text{ et } C[N] \text{ sont } \mathit{termes} \text{ observables par } O_S$$

$$O_S(C[M]) \Longleftrightarrow O_S(C[N])$$

Dans ce cas, on écrit

$$M \approx_{O_S,S} N$$

Remarque 2.5.3 (Notions d'observation) *Il y a plusieurs notions d'observation que l'on trouve utilisées dans la literature. En voici qques exemples :*

– $O_S(P)$ *= le terme* P *termine avec la sémantique* S

– $O_S(P)$ = *le terme P de type de base donne comme résultat, avec la sémantique S, une constante donnée*

Remarque 2.5.4 *On va voir plus avant que, par rapport aux notions d'observation O précédentes, et toutes les sémantiques opérationnelles S déjà vues,*

$$\lambda x.x \quad \approx_{O,S} \quad \lambda x.(\lambda y.y)x$$

2.5.1 Rélations entre sémantiques opérationnelles

Si on a plusieurs sémantiques opérationnelles d'un langage (ou si on a deux langages dont un est inclus dans l'autre) on peut se poser plusieurs questions sur la relation entre les sémantiques. C'est pour cela que l'on introduit les notions suivantes :

Définition 2.5.5 *Soit L_1 un langage equipé d'une sémantique opérationnelle $\Downarrow_1$ et inclus dans un langage L_2 equipé d'une sémantique opérationnelle $\Downarrow_2$. On dit alors que :*

$\Downarrow_2$ **est adequate pour le calcul vis à vis de** $\Downarrow_1$ *si pour tout terme P (de L_1),*

$$O_{\Downarrow_1}(P) \Longleftrightarrow O_{\Downarrow_2}(P)$$

(On peut travailler aussi bien dans L_2 si on s'intéresse à observer un terme).

$\Downarrow_2$ **est equationnellement correcte vis à vis de** $\Downarrow_1$ *si*

$$M \approx_{O,\Downarrow_2} N \text{ implique } M \approx_{O,\Downarrow_1} N$$

(On peut travailler aussi bien dans L_2 si on s'intéresse à prouver l'equivalence observationnelle).

$\Downarrow_2$ **preserve l'abstraction de** $\Downarrow_1$ *si ell'est equationnellement correcte et en plus*

$$M \approx_{O,\Downarrow_1} N \text{ implique } M \approx_{O,\Downarrow_2} N$$

(Les deux sémantiques génèrent la même equivalence observationnelle sur L_1).

Si les deux langages coincident, les deux dernières proprietés decoulent de la première.

Remarque 2.5.6 *Dans le cas du lambda calcul non typé, toutes les sémantiques que l'on a vu ne sont pas toujours equivalentes par rapport a tous ces critéres.*

- *par rapport à la terminaison : il y a une difference entre appel par nom et par valeur, parce-que en appel par nom on peut éviter des dérivations infinies qui se cachent dans les arguments des fonctions, si ces arguments sont jetés.*

Chapitre 3

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.

3.1 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.

3.1.1 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))`

3.1.2 Typage

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

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

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 `id :` Id`[type]` qui donnent le type des identificateurs libres du programme.

Remarque 3.1.1 *On demande que chaque identificateur apparaisse une seule fois dans* Γ.

Proposition 3.1.2 (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.

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

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

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

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

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

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

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

Remarque 3.1.3 *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.*

Remarque 3.1.4 *On laisse a vous de compléter les règles dans les cas des opérations binaires et unaires sur entiers et booléens.*

Exercice 3.1.5 *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 3.1.6 *Écrire une dérivation formelle complète en utilisant le règles de typage du fait que*

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

3.1.3 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 :

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

Remarque 3.1.7 (Environnements) *On n'utilise pas les environnements dans l'évaluation.*

Exercice 3.1.8 *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 :

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

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

conditionnel :

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

3.1.4 Quelques résultats

Définition 3.1.9 (Valeurs) *On définit* valeurs *les constantes ou les abstractions.*

Proposition 3.1.10 (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 3.1.11 (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. □

3.1.5 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 :

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

conditionnel :

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

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

L'appel par valeur : $\Rightarrow_v$

Remarque 3.1.12 (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 :

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

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

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

Exercice 3.1.14 *Compléter les sémantique en rajoutant les cas pour les opérations sur les entiers et les booléens, et aussi les couples avec projections.*

Exercice 3.1.15 *Est-ce que l'appel par valeur et par nom donnent des résultats différents dans notre langage TFun ? Pourquoi ?*

3.1.6 Quelques résultats

Proposition 3.1.16 (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 3.1.17 (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 3.1.18 (Normalisation forte) *Toute séquence de réduction $M \ \Rightarrow$ . . . termine.*

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

Théorème 3.1.19 (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 3.1.20 (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 ?). □

Corollaire 3.1.21 ($\Rightarrow$ 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. □

Corollaire 3.1.22 ($\Downarrow$ 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. □

Exercice 3.1.23 *Montrez que si M a type de base, alors $M \Rightarrow_n v$ et $M \Rightarrow_v v'$ implique $v \equiv v'$*

Exercice 3.1.24 *Montrez que pour tout programme clos e, $e \Rightarrow_v v$ si et seulement si $e \Downarrow_n v$.*

3.2 Sémantique dénotationnelle de TFun dans Set

On rappelle que la sémantique dénotationnelle est la donnée de la syntaxe abstraite du langage (ce qu'on a déjà fait plus haut pour TFun), des domaines d'interprétation sémantique et des morphismes *compositionnelles* d'interprétation qui associent à chaque élément des différentes catégories syntaxiques un objet dans les domaines.

3.2.1 Domaines sémantiques

Nous allons interpréter TFun dans les ensembles, et plus spécifiquement dans une structure connue comme la "Full type hierarchy" sur certains ensembles de base.

Définition 3.2.1 (Fonctionnels de type fini) *La full type hierachy basée sur des ensembles non vides $B_1, \ldots, B_n$ pour les types de base est une famille*

$$\mathcal{F}_{B_1,\ldots,B_n} = (\mathcal{F}_t)_{t \in Types}$$

définie comme suit par induction sur les types :

- $\mathcal{F}_{\texttt{bi}}$ = B_i *si* `bi` *est un type de base*
- $\mathcal{F}_{\texttt{t1->t2}}$ = *toute fonction totale entre* $\mathcal{F}_{\texttt{t1}}$ *et* $\mathcal{F}_{\texttt{t2}}$
- $\mathcal{F}_{\texttt{t1*t2}}$ = *le produit cartésien de* $\mathcal{F}_{\texttt{t1}}$ *et* $\mathcal{F}_{\texttt{t2}}$

3.2.2 Morphismes d'interprétation

On va maintenant donner notre sémantique sur $\mathcal{F}_{\mathbb{N},\{true,false\}}$, que l'on va noter FTH.

T⟦⟧ : Types → FTH

- $T⟦$`int`$⟧$ = $\mathbb{N}$
- $T⟦$`bool`$⟧$ = $\{true, false\}$
- $T⟦$`t1 -> t2`$⟧$ = l'ensemble des fonctions totales entre $T⟦$`t1`$⟧$ et $T⟦$`t2`$⟧$
- $T⟦Exp$`[t]`$⟧$ = T⟦t⟧
- $T⟦Id$`[t]`$⟧$ = T⟦t⟧

G⟦⟧ : Environnements → FTH L'interprétation d'un environnement de typage est les produit de l'interprétations de types des identificateurs qu'il déclare :

- $G⟦\Gamma⟧$ = $T⟦$`t1`$⟧ \times \cdots \times T⟦$`tn`$⟧$ if $\Gamma =$ `x1` $:Id$`[t1]`$, \cdots,$ `xn` $:Id$`[tn]`

E⟦⟧ : Expressions → FTH La donnée des morphismes d'interprétation sur les expressions est faite par induction sur la structure de la dérivation de typage[2] : à une assertion de la forme

$$\Gamma \vdash \texttt{e} : Exp\texttt{[t]}$$

on associe une fonction $E⟦\Gamma \vdash \texttt{e} : Exp\texttt{[t]}⟧$ entre l'interprétation de Γ et l'interprétation du type $Exp[t]$. On notera $E⟦\Gamma \vdash \texttt{e} : Exp\texttt{[t]}⟧_a$ pour la valeur de la fonction $E⟦\Gamma \vdash \texttt{e} : Exp\texttt{[t]}⟧$ sur a.

- $E⟦\Gamma \vdash \texttt{i}: Exp\texttt{[int]}⟧_e$ = i (pour `i` constante entière)
- $E⟦\Gamma \vdash \texttt{true}: Exp\texttt{[bool]}⟧_e$ = $true$,
- $E⟦\Gamma \vdash \texttt{false}: Exp\texttt{[bool]}⟧_e$ = $false$,
- $E⟦\Gamma \vdash \texttt{id}: Exp\texttt{[t]}⟧_e$ = $\pi_i^n(e)$ si `id` est le i-ème identificateur dans Γ (de longueur n)
- $E⟦\Gamma \vdash \texttt{fun x :t1->M}: Exp\texttt{[t1->t2]}⟧_e$ = $[a \mapsto f(e,a)]$ où $f \in (G⟦\Gamma⟧ \times T⟦\texttt{t1}⟧ \to T⟦\texttt{t2}⟧)$ est $E⟦\Gamma, \texttt{x} : Id\texttt{[t1]} \vdash \texttt{M}: Exp\texttt{[t2]}⟧$
- $E⟦\Gamma \vdash \texttt{M N}: Exp\texttt{[t]}⟧_e$ = $(E⟦\Gamma \vdash \texttt{M}: Exp\texttt{[t1->t]}⟧_e)\ E⟦\Gamma \vdash \texttt{N}: Exp\texttt{[t1]}⟧_e$

Remarque 3.2.2 (Projections) *Ici, on assume que π_i^n soit défini comme :*

$$\pi_i^n = \begin{cases} \pi_1^1 = id & \\ \pi_n^n = snd & n > 1 \\ \pi_i^n = \pi_{i-1}^{n-1} \circ fst & \textit{sinon} \end{cases}$$

On peut aussi le définir comme on fait souvent en thèorie des catégories :

$$\pi_i^n = \begin{cases} \pi_n^n = snd & \\ \pi_i^n = \pi_{i-1}^{n-1} \circ fst & \textit{sinon} \end{cases}$$

Mais dans ce cas, un environnement ne peut jamais être de taille plus petite que 1, et on est forcé de poser :

$$G⟦\Gamma⟧ = 1 \times T⟦\texttt{t1}⟧ \times \cdots \times T⟦\texttt{tn}⟧$$

où 1 est l'ensemble à un seul élément.

Voici quelques relations typiques entre syntaxe et sémantique :

Lemme 3.2.3 (Substitution sémantique)

$$E⟦\Gamma, \texttt{x} : \texttt{t1} \vdash \texttt{M}: Exp\texttt{[t2]}⟧_{(e, E⟦\Gamma \vdash \texttt{N}: Exp\texttt{[t1]}⟧_e)} = E⟦\Gamma \vdash \texttt{M[N/x]}: Exp\texttt{[t2]}⟧_e$$

[2] Ce qui est cohérent en raison de l'unicité de la dérivation.

Preuve. Longue induction sur la structure de M. □

Lemme 3.2.4 (Indépendance sémantique) *Si e et e' coïncident sur les variables libres de M, alors*

$$E[\![\Gamma \vdash \texttt{M:}Exp\texttt{[t]}]\!]_e = E[\![\Gamma \vdash \texttt{M:}Exp\texttt{[t]}]\!]_{e'}$$

Notation 3.2.5 *Pour breveté, dans ce qui suit, on va souvent oublier le contexte de typage Γ, s'il n'est pas relevant dans le contexte. Aussi, on ne va pas toujours distinguer les morphismes d'interprétation : on écrira juste $[\![]\!]$.*

3.3 Relations entre sémantique opérationnelle et dénotationnelle

Il y a plusieurs choses que l'on veut pouvoir dire sur la relation entre une sémantique opérationnelle et une sémantique dénotationnelle d'un langage donné. En voici quelques unes :

Définition 3.3.1 *Soit L un langage équipé d'une sémantique opérationnelle $\Downarrow$, avec une équivalence observationnelle $\approx$, et d'une sémantique dénotationnelle $[\![]\!]$. On dit alors que :*

$[\![]\!]$ est adéquate pour le calcul vis à vis de $\Downarrow$ *si :*

$$[\![\texttt{M}]\!] = [\![\texttt{V}]\!] \Rightarrow M \Downarrow v \quad v \text{ un valeur}$$

(on peut regarder les dénotations si on cherche juste le résultat)

$[\![]\!]$ est correcte pour $\Downarrow$ vis à vis de $\approx$ *si :*

$$[\![\texttt{M}]\!] = [\![\texttt{N}]\!] \Rightarrow M \approx N$$

(on peut utiliser les dénotations pour prouver l'équivalence observationnelle)

$[\![]\!]$ est complète pour $\Downarrow$ vis à vis de $\approx$ *si :*

$$M \approx N \Rightarrow [\![\texttt{M}]\!] = [\![\texttt{N}]\!]$$

(on peut utiliser les dénotations pour prouver la non-équivalence observationnelle)

$[\![]\!]$ **est complètement abstraite pour** $\Downarrow$ **vis à vis de** $\approx$ *si ell'est correcte et complète.*
(on peut utiliser les dénotations à la place de l'équivalence observationnelle)

Remarque 3.3.2 *L'équivalence observationnelle est très difficile à établir à partir de la définition : il y a une quantification sur tous les contextes ! C'est pour cela que une sémantique dénotationnelle complètement abstraite (ou même juste correcte), est très utile et recherchée. Par contre, la définition est utile pour prouver la non équivalence : il suffit de trouver un contexte départageant les deux (fragments de) programmes.*

Remarque 3.3.3 *L'abstraction complète n'est pas une propriété courante, mais par contre, si la définition de la sémantique dénotationnelle est donnée de façon compositionnelle, on a facilement la correction.*

Notation 3.3.4 *On écrit souvent*

$$\mathcal{M} \models M = N$$

à la place de $[\![\texttt{M}]\!]=[\![\texttt{N}]\!]$*, où* $\mathcal{M}$ *est le modèle utilisé pour la sémantique dénotationnelle.*

Proposition 3.3.5 (Consistance) *Dans TFun, si* $\Gamma \vdash \texttt{M:}Exp\texttt{[t]}$ *et* $M \Downarrow_v v$*, alors* $[\![\texttt{M}]\!] = [\![\texttt{v}]\!]$.

Preuve. Induction sur la taille de $M \Downarrow_v v$. Voyons juste le cas de l'application :

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

On sait par hypothèse d'induction que :
– $[\![e_1]\!]$ = $[\![\texttt{fun}\ x : A\texttt{->}e_0]\!]$
– $[\![e_2]\!]$ = $[\![v_2]\!]$
– $[\![e_0[v_2/x]]\!]$ = $[\![v_0]\!]$
Donc on peut dériver que :

$$\begin{aligned}
[\![e_1(e_2)]\!](e) &= ([\![e_1]\!](e))([\![e_2]\!](e)) \\
&= ([\![\texttt{fun}\ x : A\texttt{->}e_0]\!](e))([\![v_2]\!](e)) \\
&= ([a \mapsto [\![e_0]\!](e, a)])([\![v_2]\!](e))
\end{aligned}$$

$$\begin{aligned} &= [\![e_0]\!](e, ([\![v_2]\!](e))) \\ &= [\![e_0[v_2/x]]\!](e) \\ &= [\![v_0]\!](e) \end{aligned}$$

et on a fini. □

Exercice 3.3.6 *Compléter la preuve, et montrer le même résultat pour* $\Downarrow_n$.

Lemme 3.3.7 (L'égalité dans [[]] est une *congruence***)**
Si [[M]] = [[N]]*, alors* [[C[M]]] = [[C[N]]] *pour tout contexte* $C[\,]$.

Preuve. Par induction sur la structure de $C[\,]$. Cela se passe bien en raison de la compositionalité de la définition de [[]]. □

Corollaire 3.3.8 [[]] *est correcte vis à vis de* $\Downarrow$ *, soit en appel par nom que par valeur, si on observe les types de base.*

Preuve. Prouvons la contrapositive : soit un contexte qui départage M et N, donc tel que $C[M] \Downarrow v$, $C[N] \Downarrow v'$, et supposons [[M]] = [[N]]. Alors, pour tout contexte $C[\,]$, [[C[M]]] = [[C[N]]].

Mais par la consistance, [[v]] = [[C[M]]] = [[C[N]]] = [[v']].

Or, sur les types de base, $v \equiv v' \Longleftrightarrow [\![\mathtt{v}]\!] = [\![\mathtt{v'}]\!]$, et on obtient une contradiction. □

Corollaire 3.3.9 *Si* $[\![\mathtt{M}]\!] = n$ *(*n *un booléen ou un entier), alors* $M \Downarrow n$.

Preuve. Parce-que alors M et n sont équivalents observationnellement, et étant de type de base, $M \Downarrow n$ est conséquence de la définition en prenant $C[\,]$ comme le contexte vide. □

Exercice 3.3.10 *Vérifier que l'équivalence suivante (règle* η*)*

```
fun x :A->x   ≈O,S   fun x :A->(fun y :A-> x y)
```

est valide si l'on observe les types de base.

Exercice 3.3.11 *Montrer que, si l'on observe les valeurs aux types supérieurs,* η *n'est pas valide.*

3.3.1 Conclusion

Dans le cas de TFun, on a donc prouvé que :
– $[\![]\!]$ est correcte vis à vis de l'observation des types de base
– $M \Downarrow_{v,n} n$ ssi $[\![\texttt{M}]\!] = n$
($[\![]\!]$ est aussi correcte par rapport à la terminaison, mais pour des raison triviales pour TFun).

Exercice 3.3.12 *Montrez que $[\![]\!]$ n'est pas correcte si on observe des valeurs qui ne sont pas de base.*

3.3.2 Égalités prouvables

Un autre atout de la sémantique dénotationnelle est la possibilité d'utiliser le modèle pour *valider* des règles equationnelles que l'on peut utiliser pour prouver l'équivalence observationnelle, à l'aide d'une théorie equationnelle (notez que les règles de congruence sont toujours admissibles, si $[\![]\!]$ est donnée de façon compositionnelle).

Par exemple :

Exercice 3.3.13 *Prouvez que L'égalité β suivante est validée par $[\![]\!]$:*

$$\texttt{(fun x :A->M)N} = \texttt{M[N/x]}$$

en déduire par raisonnement equationnel que

$$\texttt{fun x :A->x} \quad \approx_{O,S} \quad \texttt{fun x :A->(fun y :A->y) x}$$

Faites de même, pour η :

$$\texttt{fun x :A->x} = \texttt{fun x :A->(fun y :A-> x y)}$$

et retrouvez le résultat de l'exercice 3.3.10.

3.3.3 Les egalités prouvables dans $\mathcal{F}$

Le modèle ensembliste que l'on a vu, si on ne conidère que les fonctions, et pas les opèraions de base, ne nous donne guère plus que β et η. Pour voir cela, nous nous appuyons sur des résultats classiques du λ-calcul :

Notation 3.3.14 *Nous disons que $M = N$ est prouvable via $\beta\eta$ si M et N sont égaux modulo α-conversion dans la théorie engendrée par β et η et la fermeture par contexte.*

Théorème 3.3.15 (Friedman's completeness) *Si X est un ensemble infini, alors :*

$$\mathcal{F}_X \models M = N \quad \Longleftrightarrow M = N \text{ est prouvable via } \beta\eta$$

Théorème 3.3.16 (Plotkin's completeness)

$$(\forall X\ finite. \mathcal{F}_X \models M = N) \quad \Longleftrightarrow M = N \text{ est prouvable via } \beta\eta$$

Corollaire 3.3.17 (Égalités prouvables dans $\mathcal{F}_{\mathbb{N}}$) *$[\![]\!]$ dans $\mathcal{F}_{\mathbb{N}}$ identifie seulement les programmes qui sont égaux modulo $\beta\eta$.*

Remarque 3.3.18 *Il y a plusieurs extensions du théorème de Friedman [Fri75] pour des langages avec produits, sommes etc. Voir par exemple [DS95] pour quelques résultats récents.*

3.4 Extension avec non terminaison : TFun$_{loop}$

Des phénomènes plus intéressants se manifestent si l'on introduit dans le langage la possibilité d'écrire des programmes qui ne terminent pas. Pour l'instant, on s' intéresse juste à la non-terminaison en elle même, et non pas à la façon d'écrire un programme qui ne termine pas (par exemple à l'aide de définitions recursives qui ne sont pas légales dans TFun).

Pour cela, on va rajouter juste un symbole loop à chaque type, qui représente un programme de ce type là qui ne termine pas. Voici les modifications à apporter aux différentes sémantiques opérationnelles :

syntaxe de TFun$_{loop}$ même que TFun, mais en plus pour tout type `t`, on rajoute

$$\Gamma \vdash \texttt{loop:}Exp\texttt{[t]}$$

appel par nom (sos et naturel) aucune modification

appel par valeur (sos et naturel) aucune modification

Voyons comment les relations entre sémantiques se modifient :

Remarque 3.4.1

- *TFun$_{loop}$ avec $\Downarrow_v$ ou $\Downarrow_n$ est adéquate pour le calcul vis à vis de TFun avec $\Downarrow_v$ ou $\Downarrow_n$ (les programmes sont les mêmes, donc sans loop, pas de surprise)*
- *sur TFun$_{loop}$, $\Downarrow_v$ et $\Downarrow_n$ ne génèrent plus la même équivalence observationnelle (que l'on observe la terminaison ou le types de base) :*

  `(fun x :t->n)loop` $\Downarrow_n$ n *mais* `(fun x :t->n)loop` $\not\Downarrow_v$ n

Exercice 3.4.2 *En programmant un peu, montrez que observer les types de base ou la terminaison sur le types de base sont équivalents.*

Exercice 3.4.3 *Que se passe-t-il si on décide de observer la terminaison à un type quelconque ? (i.e. on observe "le programme M a un valeur", où* $\Gamma \vdash$ `M:`Exp`[t]` *pour un quelque type* `t`*)*

3.4.1 Sémantique dénotationnelle avec non-terminaison

Si les programmes peuvent ne pas terminer, il faut pouvoir le prendre en compte au niveau des domaines sémantiques : pour cela, on *rajoute* dans l'interprétation de chaque catégorie syntaxique une dénotation spéciale pour ceux programmes qui ne terminent pas, et on le note $\perp$.
Cela peut se faire à l'aide de l'union disjointe d'ensembles, $+$, et en étendant la définition de $\mathcal{F}$ avec la formation d'union disjointe :

- $\mathcal{F}_{\texttt{t1+t2}}$ = l'union disjointe de $\mathcal{F}_{\texttt{t1}}$ et $\mathcal{F}_{\texttt{t2}}$ (on peut la définir formellement comme $\{(a, true) | a \in \mathcal{F}_{\texttt{t1}}\} \cup \{(b, false) | b \in \mathcal{F}_{\texttt{t2}}\}$)

Voici comment on voit se modifier la sémantique dénotationnelle :

3.4.2 Appel par valeur

Avec `loop` dans le langage, une expression peut ne pas terminer, donc on doit modifier l'interprétation de Exp[t] comme suit :

- $[\![Exp\texttt{[t]}]\!]$ = $[\![\texttt{t}]\!]$ + $\{\perp\}$

Aussi, une fonction prend en paramètre un *valeur* (donc qque chose qui est le résultat d'une évaluation qui s'est bien terminée), mais elle peut ne pas terminer, donc nous voilà forcés à modifier aussi l'interprétation des types fonctionnels :

- $[\![\texttt{t1->t2}]\!]$ = $[\![\texttt{t1}]\!] \to [\![Exp\texttt{[t2]}]\!]$

Au contraire de ce qui se passe au niveau opérationnel, ces modifications nous obligent à retravailler toutes nos définitions dénotationnelles :

- $[\![\Gamma \vdash \texttt{i:}Exp\texttt{[int]}]\!]_e$ = $in_1(i)$ (pour `i` constante entière)

- $[\![\Gamma \vdash \texttt{true:}Exp\texttt{[bool]}]\!]_e$ = $in_1(true)$,
- $[\![\Gamma \vdash \texttt{false:}Exp\texttt{[bool]}]\!]_e$ = $in_1(false)$,
- $[\![\Gamma \vdash \texttt{loop:}Exp\texttt{[t]}]\!]_e$ = $in_2(\bot) \in [\![Exp\texttt{[t]}]\!] = [\![\texttt{t}]\!] + \{\bot\}$,
- $[\![\Gamma \vdash \texttt{id:}Exp\texttt{[t]}]\!]_e$ = $in_1(\pi_i(e))$ si `id` est le i-ème identificateur dans Γ
- $[\![\Gamma \vdash \texttt{fun x :t1->M:}Exp\texttt{[t1->t2]}]\!]_e$ = soit $f \in ([\![\Gamma]\!] \times [\![\texttt{t1}]\!] \rightarrow [\![Exp\texttt{[t2]}]\!])$ l'interprétation $[\![\Gamma, \texttt{x :}Id\texttt{[t1]} \vdash \texttt{M:}Exp\texttt{[t2]}]\!]$ dans $in_1([a \mapsto f(e,a)])$
- $[\![\Gamma \vdash \texttt{M N:}Exp\texttt{[t]}]\!]_e = \begin{cases} ha & \text{si } (fe) = in_1(h) \text{ et } (ge) = in_1(a) \\ in_2(\bot) & \text{sinon} \end{cases}$

  où $f = [\![\Gamma \vdash \texttt{M:}Exp\texttt{[t1->t]}]\!]$ et $g = [\![\Gamma \vdash \texttt{N:}Exp\texttt{[t1]}]\!]$.

3.4.3 Appel par nom

En appel par nom aussi, une expression peut ne pas terminer, donc on doit modifier l'interprétation de Exp[t] comme suit :

- $[\![Exp\texttt{[t]}]\!]$ = $[\![\texttt{t}]\!]$ + $\{\bot\}$

Au contraire de l'appel par valeur, une fonction ne prend pas en paramètre un *valeur* (donc qque chose qui est le résultat d'une évaluation qui s'est bien terminée), et en plus elle peut ne pas terminer, et donc dans l'environnement on n'associe plus des valeurs aux paramètres, mais plutôt des expressions qui peuvent ne pas terminer :

- $[\![Id\texttt{[t1]}]\!]$ = $[\![\texttt{t1}]\!]$ + $\{\bot\}$

Pour la même raison, nous voilà forcés à modifier aussi l'interprétation des types fonctionnels :

- $[\![\texttt{t1->t2}]\!]$ = $[\![Id\texttt{[t1]}]\!] \rightarrow [\![Exp\texttt{[t2]}]\!]$ = $[\![Exp\texttt{[t1]}]\!] \rightarrow [\![Exp\texttt{[t2]}]\!]$

Au contraire de ce qui se passe au niveau opérationnel, ces modifications nous obligent à retravailler toutes nos définitions dénotationnelles :

- $[\![\Gamma \vdash \texttt{i:}Exp\texttt{[int]}]\!]_e$ = $in_1(i)$ (pour `i` constante entière)
- $[\![\Gamma \vdash \texttt{true:}Exp\texttt{[bool]}]\!]_e$ = $in_1(true)$,
- $[\![\Gamma \vdash \texttt{false:}Exp\texttt{[bool]}]\!]_e$ = $in_1(false)$,
- $[\![\Gamma \vdash \texttt{loop:}Exp\texttt{[t]}]\!]_e$ = $in_2(\bot) \in [\![Exp\texttt{[t]}]\!] = [\![\texttt{t}]\!] + \{\bot\}$,
- $[\![\Gamma \vdash \texttt{id:}Exp\texttt{[t]}]\!]_e$ = $\pi_i(e)$ si `id` est le i-ème identificateur dans Γ
- $[\![\Gamma \vdash \texttt{fun x :t1->M:}Exp\texttt{[t1->t2]}]\!]_e$ = $in_1([a \mapsto f(e,a)])$
  où $f \in ([\![\Gamma]\!] \times [\![Id\texttt{[t1]}]\!] \rightarrow [\![Exp\texttt{[t2]}]\!])$ est l'interprétation $[\![\Gamma, \texttt{x :}Id\texttt{[t1]} \vdash \texttt{M:}Exp\texttt{[t2]}]\!]$
- $[\![\Gamma \vdash \texttt{M N:}Exp\texttt{[t]}]\!]_e = \begin{cases} h(ge) & \text{si } (fe) = in_1(h) \\ in_2(\bot) & \text{sinon} \end{cases}$

  où $f = [\![\Gamma \vdash \texttt{M:}Exp\texttt{[t1->t]}]\!]$ et $g = [\![\Gamma \vdash \texttt{N:}Exp\texttt{[t1]}]\!]$

Remarque 3.4.4 (Dénotationnel vs. opérationnel) *On peut prouver dans ce cas aussi que l'égalité dénotationnelle implique l'équivalence observationnelle si l'on observe les types de base. Cependant, c'est la sémantique dénotationnelle par valeur (nom) qui va avec la sémantique opérationnelle par valeur (nom) : il n'y a plus une seule sémantique qui fait tout (c'est d'ailleurs normale, vu que les équivalences observationnelles ne sont plus les mêmes !)*

3.4.4 L'abstraction complète

Pour ce qui en est de l'abstraction complète, en présence de fonctions d'ordre supérieure et de la non-terminaison, ici on peut prouver (avec pas mal de technicisme quand même), qu'elle ne vaut pas : le modèle est trop riche et arrive à distinguer certains fonctions d'ordre supérieure, tel les *gouteurs d'ou parallèle*. Pour en savoir plus sur ce sujet et sur les thèmes reliés de la sequentialité, on peut regarder [Cur93, SAM94].

3.5 Un Metalangage pour FTH (Set)

Pour donner la sémantique dénotationnelle, nous avons utilisé des fonctions qui sont bien disponibles dans Set (en réalité nous avons travaillé dans FTH), mais nous n'avons pas fait cela à l'aide d'un langage vraiment formel. Le but de cette section est de formaliser les outils que nous avons utilisé, et d'introduire un métalangage formel qui le contient.

3.5.1 Les structures

On a utilisé plus en haut les structures suivantes, avec certaines opérations qui leur sont naturellement associées :

booléens

type : $Bool$

opérations : $if\ \_\ then\ e1\ else\ e2 : Bool \to C$ si $e1 : C$ et $e2 : C$, et aussi $true : Bool$, $false : Bool$

entiers

type : Nat

opérations : $+, -, *, /, 0, 1, \cdots$

espace de fonctions

type : $A \to B$

opérations :

- application : $app(\_, \_) : ((A \to B) \times A) \to B$
- abstraction : pour toute expression e de type B qui dépend d'une variable libre $x : A$, $\lambda x.e : A \to B$ est la fonction $[a \mapsto e[a/x]]$

produit cartésien

type : $A \times B$

opérations :

- projections : $\pi_1 : A \times B \to A$ et $\pi_2 : A \times B \to A$
- formation de couples : (_,_) : $A, B \to A \times B$

union disjointe

type : $A + B$

opérations :

- injections : $in_1(\_) : A \to A + B$ et $in_2(\_) : B \to A + B$
- définition par cas : $case(\_, \lambda x : A + B.e1, \lambda y : A + B.e2) : A + B \to C$ si $\lambda x : A+B.e1 : A+B \to C$, $\lambda y : A+B.e2 : A+B \to C$

Or, tout ce qu'on a utilisé est là, et on peut très bien définir un *métalangage* qui capture exactement ces fonctions (dans le sens que la signification de chaque expression dans ce métalangage est immédiate au niveau du modèle). Mais, attention :

Remarque 3.5.1 (Les fonctions utilisées) *Avec l'abstraction, on peut construire seulement des fonctions qui peuvent être calculées par le mécanisme de la substitution. Cela est très loin de suffire pour capturer toutes les fonctions du modèle.*

Exercice 3.5.2 *Montrez qu'il n'est pas possible de capturer avec un métalangage toutes les fonctions disponibles dans* $\mathcal{F}_{\mathbb{N},\{true,false\}}$.

Cela revient à définir un simple noyau fonctionnel typé que l'on peut mettre par exemple en correspondance directe avec un sous-ensemble de ML (qu'il s'agisse de Miranda, Hope, Haskell, SML, Gopher ou CamlLight, cela ne change rien).

3.5.2 Sémantique comme traduction

Si l'on pousse cet approche, on arrive a décomposer la tache de donner une sémantique dénotationnelle en deux phases :

choix du modèle et du métalangage : on choisi un modèle suffisamment riche pour le métalangage que l'on a besoin d'utiliser. Il peut y en avoir plusieurs.

traduction du langage dans le métalangage : on traduit les programmes du langage dans le métalangage. Cela donne, par composition, une sémantique dénotationnelle au niveau du langage.

L'avantage de cet approche est la possibilité de pouvoir choisir des modèles différents, tout en maintenant le même métalangage, ce qui permet de ne pas réécrire la sémantique à chaque fois.

En plus, si le métalangage est exécutable, on peut en extraire immédiatement un interprète pour le langage : c'est bien cela que l'on va faire en utilisant CamlLight.

3.5.3 Sémantique executable de TFun

Voici donc la sémantique dénotationnelle de TFun écrite avec le metalangage dans la syntaxe de CamlLight :

```
#(*                                                                    *)
#(* Syntaxe Abstraite du Langage                                       *)
#(*                                                                    *)
#
#type Tfuntype  = TBool | TNat | Arr of Tfuntype * Tfuntype   (* les types
#
#and  Id       == string                                     (* identificate
#
#and  Exp       = INT of int | BOOL of bool                  (* les expressi
#              | Iden of Id
#              | App of (Exp * Exp)
#              | Abs of ((Id * Tfuntype) * Exp)
#              | IF of (Exp * Exp * Exp)
#;;
Type Tfuntype defined.
Type Id defined.
Type Exp defined.

#
```

```
#(*                                                                    *)
#(*            TFun : l'Interpreteur                                   *)
#(*                                                                    *)
#
#(* Ce code est une traduction en ML immediate de la Semantique       *)
#(* Denotationnelle Directe de TFun qu'on a donne' en cours            *)
#(* Un grande avantage de cet approche est la clarte du code,         *)
#(* qui se commente tout seul                                         *)
#(* On remarquera cependant que on n'utilise pas ici les types du langage au  *)
#(* contraire de ce que l'on fait dans la semantique mathematique.    *)
#(* En effet, pour simplicite, on n'utilise pas ici la derivation de typage   *)
#(* pour la definition inductive, mais seulement la structure des programmes. *)
#(* Cela ne change pas vraiement les fonctions semantiques, mais on permet de *)
#(* accepter comme parametre des programmes qui sont mal types, ce qui        *)
#(* apparait en trois points : l'erreur possible dans projectvalue et les deux *)
#(* match incomplets pour Arrow et Bool. En effet, sans derivation de typage, *)
#(* on ne peut pas garantir que le programme passe en parametre fournisse les *)
#(* bons valeurs dan ces cas *)
#
#(* Definition des Domaines Semantiques :                              *)
#
#type FTH = Int of int | Bool of bool (* les valeurs de base : ici les entiers et bool
#         | Arrow of (FTH -> FTH) | Prod of FTH * FTH;;
Type FTH defined.

#
#let rec projectvalue (i,g,e) =
#      match (g,e) with
#        ([],[])        -> failwith "Environnement mal forme"
#      | (id : :g1,a : :e1) -> if id = i then a else projectvalue (i, g1, e1);;
Toplevel input :
>......match (g,e) with
>        ([],[])        -> failwith "Environnement mal forme"
>      | (id : :g1,a : :e1) -> if id = i then a else projectvalue (i, g1, e1
)...
Warning : this matching is not exhaustive.
projectvalue : 'a * 'a list * 'b list -> 'b = <fun>

#
#
#let rec E gamma exp e =
#        match exp with
#          App (exp1,exp2) -> let (Arrow f) = (E gamma exp1 e) in f (E gamma exp2 e)
#        | Abs ((id,t),exp)-> Arrow(fun a -> (E (id : :gamma) exp (a : :e)))
#        | Iden(id)        -> projectvalue(id,gamma,e)
#        | INT i           -> Int i
```

```
#        | BOOL b          -> Bool b
#        | IF(b,exp1,exp2) -> let (Bool bv) = (E gamma b e) in
#                             if bv then (E gamma exp1 e)
#                             else (E gamma exp2 e);;
Toplevel input :
>.........................let (Bool bv) = (E gamma b e) in
>                            if bv then (E gamma exp1 e)
>                            else (E gamma exp2 e)..
Warning : this matching is not exhaustive.
Toplevel input :
>          App (exp1,exp2) -> let (Arrow f) = (E gamma exp1 e) in f (E g
amma exp2 e)
>                            ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^^^^^^^^^
Warning : this matching is not exhaustive.
E : Id list -> Exp -> FTH list -> FTH = <fun>

#
#(* Programme bien type' *)
#E [] (App(Abs(("x",TBool),IF(Iden "x",INT 1,INT 2)),BOOL true)) [];;
- : FTH = Int 1

#
#(* Programme mal type'  *)
#E [] (App(INT 1, INT 2))[];;
Uncaught exception : Match_failure ("", 2478, 2532)
```

On a vu auparavant comment, sur une structure très simple comme celle des fonctions totales que l'on retrouve dans Set (notre modèle FTH), on peut donner la sémantique a un simple langage fonctionnel typé sans récursion, même si l'on permet d'écrire des programmes qui ne terminent pas toujours (ce que l'on a fait a l'aide de `loop`).

L'iteration dans Set

Si on cherche bien, on peut aller un peu plus loin dans Set et arriver à interpréter des langages un peu plus puissants que TFun : en effet, dans Set, ils existent bien, à tous les types, des fonctions totales tel que l'itération et la récursion primitive.

itération Il s'agit de la fonction d'ordre supérieur

$$iter(a, f, n) = f^n(a)$$

Avec $iter$, on peut, par exemple, écrire le factoriel de façon itérative comme

$$fact\ n = snd(iter((1,1), \texttt{fun}\ \ (x,y) : I\!N \times I\!N \texttt{->}(x+1, x*y), n))$$

(vérifiez cela en CamlLight en programmant $iter$)

récursion primitive Il s'agit de la fonction d'ordre supérieur

$$R(u,v,0) = u \qquad R(u,v,n+1) = v(R(u,v,n),n)$$

Avec R, on peut, par exemple, écrire le factoriel de façon itérative comme

$$fact\ n = R(1, \texttt{fun}\ \ (x,y)\texttt{->}(x*y), n)$$

(vérifiez cela en CamlLight en programmant R)

Remarque 3.5.3 (Pouvoir d'expression) *On peut montrer ([GLT90]) que si on rajoute R (ou aussi $iter$: ils ont le même pouvoir expressif, comme l'on peut montrer en programmant un peu) à TFun, on peut décrire toutes les fonctions $totales$ dont la terminaison est prouvable dans l' arithmétique de Peano. En effet, TFun+R est aussi connu comme le système T de Gödel.*

3.6 Limitations de Set

Cependant, même si l'on rajoute la récursion primitive, il y a deux hypothèses cachées dans ce que l'on a fait qui ne sont pas toujours raisonnables si on veut pouvoir traiter les vrai langages de programmation :

pas de récursion on n'a pas, dans TFun, la possibilité d'écrire des boucles recursives non primitives, ni à l'aide de définitions recursives ni à travers des constructions tel que `repeat` ou `while`

typage on a pu donner la sémantique par induction sur la dérivation de typage, mais cela n'est pas possible pour les langages non typés ou dont le typage n'a pas des bonnes propriétés

Nous allons voir que Set n'est pas adéquat si on laisse tomber ces hypothèses.

3.7 La récursion

Une définition recursive d'une fonction dans un langage de programmation est normalement censée identifier une et une seule fonction à l'exécution. Toutefois, dans Set, certaines définitions sont satisfaites soit par trop soit par pas assez de fonctions.

trop c'est le cas de la définition

```
let rec f x = f x
```

Toute fonction satisfait cette définition dans le modèle Set, et on ne sait donc pas laquelle lui associer. Cependant, du point de vue opérationnel, la seule fonction raisonnable qui est calculée est celle qui diverge partout, et donc on devrait lui associer $\perp$.

pas assez c'est le cas de la définition

```
let rec f x = (f x)+1
```

Aucune fonction satisfait cette définition dans le modèle Set, Cependant, du point de vue opérationnel, la seule fonction raisonnable qui est calculée est ici aussi celle qui diverge partout, et donc on devrait lui associer $\perp$.

Même si on décide, de façon arbitraire, d'associer $\perp$ à ces deux fonctions là, on n'a pas résolu le vrai problème, qui est de trouver une façon *uniforme* d'associer une sémantique à un programme.

En réalité, dans le mathématique on à bien l' habitude de traiter des objets définis de façon recursive à travers la notion de point fixe, et on peut adopter ici la même technique. Voyons comment on procède normalement :

passage de la récursion au point fixe partant d'une définition recursive donnée, on la transforme, à l'aide de quelques manipulation formelle, en une équation de point fixe. Par exemple, pour nos définitions précédentes on a :

```
let rec f x = f x                          =
let rec f = fun x -> f x                   =  égalité β
let rec f = (fun g -> (fun x -> g x)) f    =  f est un point fixe de
let f = FIX(fun g -> (fun x -> g x))          (fun g -> (fun x -> g x))
```

et :

```
let rec f x = (f x)+1                          =
let rec f = fun x -> (f x)+1                   =  égalité β
let rec f = (fun g -> (fun x -> (g x)+1)) f    =  f est un point fixe de
let f = FIX(fun g -> (fun x -> (g x)+1))          (fun g -> (fun x -> (g x)+1))
```

Le procédé, uniforme, consiste en construire (à l'aide de la β-expansion sur la définition recursive), un fonctionnel pour lequel la fonction définie de façon recursive est un point fixe.
Cependant, comme dans le cas des fonctions mathématiques, il se peut que il y aie plusieurs ou aucun point fixe pour un fonctionnel donné. En effet, dans Set cette manipulation formelle ne nous a fait rien gagner : le premier fonctionnel a pour point fixe toute fonction, et le deuxième aucune. Or, en Analyse, on s'en sorte en identifiant certains classes de fonctions qui admettent toujours un point fixe, et, au cas ou on en trouve plusieurs, on choisi souvent le plus petit (c'est le cas des fonction Lipschitziennes de coefficient strictement inférieur a 1, par exemple).
Dans le cas de la sémantique dénotationnelle, on cherchera de même à définir des classes de fonctions qui ont la propriété de toujours admettre des point fixes. Pour cela, il faudra équiper nos ensembles avec une structure (topologique) appropriée.

calcul du point fixe il ne faut pas oublier non plus que nous sommes en train de construire des modèles de programmes, qui ont un contenu calculatoire effectif. Donc, même si d'un point de vue strictement mathématique l'existence d'un point fixe est suffisante, d'un point de vue calculatoire, on souhaiteen plus que ce point fixe soit *calculable* de façon effective. Cette exigence aussi va imposer le choix d'une structure appropriée sur nôtre modèle.

La récursion pour les boucles

La nécessité d'avoir recours à des fonctions recursive dans la sémantique dénotationnelle n'est pas liée seulement à la présence de définitions recursives dans le langages : il suffit d'avoir une construction comme le `while` pour qu'elle se manifeste. Voici un premier essai de sémantique pour le `while` :

$[\![$`while B do C`$]\!]$(s) = if $[\![$`B`$]\!]$(s) = true
then $[\![$`while B do C`$]\!]$($[\![$`C`$]\!]$(s))
else s

Cela *semble* capturer notre intuition opérationnelle, mais cette définition (qui est, elle, bien recursive) viole le principe fondamentale de la *compositionalité*, que l'on a vu être, entre autres choses, indispensable pour assurer que $[\![]\!]$ soit une congruence. On est donc obligé de modifier la définition comme suit :

$[\![$`while B do C`$]\!](s) =$ f
where rec f = if $[\![$`B`$]\!](s)$ = true
then f($[\![$`C`$]\!](s)$)
else s

Cette définition est compositionnelle, mais elle fait apparaître une définition recursive de la fonction f dans le domaine.

3.8 Le fonctions non typées et l'auto-application

Un problème similaire, mais pas identique, peut se présenter quand on essaye de donner la sémantique de langages qui permettent de passer des procédures et fonctions en paramètre à procédures et fonctions, ou de les renvoyer comme résultat de l'appel d'une fonction.
Dans le cas de notre langage TFun, où l'on permet en effet de faire ça, nous avons pu utiliser la sémantique ensembliste en raison du typage simple du langage : chaque fois qu'il y a une application `e1 e2`, le type de `e1` est de la forme $t \to t'$ où t est le type de `e2`, et on peut donc donner une interprétation "stratifiée" dans FTH où chaque fonction est plongée dans l'interprétation de son type[3].
Cependant, si le langage n'est pas typé[4], on se retrouve obligés à plonger toutes les fonctions/procédures dans un *même* domaine d'interprétation D. Mais alors, comment interpréter l'application ? Dans le cas le pire, on peut se retrouver avec un programme de la forme `(fun x->e) (fun x->e)`, qui nous oblige à savoir regarder $[\![$`fun x->e`$]\!]$ à la fois comme un élément de D et comme un élément de $D \to D$, ce qui revient à établir une correspondance uniforme entre D et $D \to D$, qui prend la forme d'une *équation de domaine* :

$$D = D \to D$$

Or, pour des raisons de cardinalité, cette égalité ne peut pas être un isomorphisme ensembliste : en raison du théorème de Cantor, le seul ensemble

[3] Notons que dans TFun on ne peut pas passer une fonction en paramètre à elle même : `(fun x :A->e) (fun x :A->e)` ne peut pas être bien typé.

[4] Ou si le système de types permet l'autoapplication de fonctions et/ou procédures.

E isomorphe à $E \to E$ est l'ensemble réduit à un seul élément.

D'un autre coté, les fonctions *définissables* (i.e. que l'on peut programmer dans un langage de programmation) entre D et D ne dépassent jamais la cardinalité des entiers (pourquoi ?), qui est normalement aussi la cardinalité des élément définissables de D. Donc le problème de cardinalité semble vraiment souligner une inadéquation du modèle ensembliste, qui admets trop de fonctions l'espace de fonctions pleins d'éléments que l'on ne pourra pas, de toutes façon, définir, plutôt qu'une limitation de la technique dénotationnelle : il s'agit d'identifier des modèle sémantiques dans lesquels on puisse construire des domaines satisfaisant des équations *recursives* de domaines.

Pour cela, on fait recours à une idée qui est utilisée assez souvent en mathématique : si on veut réduire la dimension d'un ensemble, on essaye de lui imposer une structure. C'est bien cela qui va se passer si on rajoute une structure d'ordre aux ensembles pour obtenir des CPO, et on interprète la $\to$ comme la formation de l'espace des fonctions *continues* par rapport à la topologie induite par l'ordre.

En résumant, il s'agit ici aussi de savoir traiter de façon uniforme des définitions recursives, mais non plus des fonctions, sinon des domaines eux mêmes. Le même type de problématique se manifeste quand on doit traiter des langages où l'on peut définir des types recursifs (comme c'est bien le cas de ML, par exemple).

3.9 Une première solution pour les définitions recursives : le théorème de Tarski

On va rappeler ici un certain nombre de notions et résultats qui nous donnent une première solution au problème des définitions recursives de fonctions. Ces résultats, qui sont important pour la sémantique que l'on va développer ensuite, sont présentés de façon plus étendue dans [DP90, AG92], auquels on renvoie pour plus d'approfondissement.

Définition 3.9.1 (Ordre partiel et totale) *Une* relation d'ordre partiel $\leq$ *sur un ensemble* D *est une relation binaire qui a les propriétés :*

reflexive $\forall a \in D.a \leq a$

anti-symmetrique $\forall a, a' \in D.a \leq a', a' \leq a \Rightarrow a = a'$

transitive $\forall a, b, c \in D.a \leq b, b \leq c \Rightarrow a \leq c$

Une relation d'ordre partiel se dit totale *si en plus*

$$\forall d, d' \in D.d \leq d' \text{ ou } d' \leq d$$

Notation 3.9.2 *Un ensemble D équipé d'une relation d'ordre partiel $\leq$ est souvent appelé "un ordre partiel" et noté $(D, \leq)$. Aussi, on trouve souvent employé dans la literature le terme* poset *pour indiquer un ensemble équipé d'un ordre partiel.*

Définition 3.9.3 (Fonction monotone) *Si $(D, \leq)$ et $(D', \leq')$ sont ordres partiels, une fonction $f : D \to D'$ est* monotone *si*

$$\forall d, d' \in D.d \leq d' \Rightarrow f(d) \leq' f(d')$$

Définition 3.9.4 (Chaîne) *On appelle une* chaîne *dans $(D, \leq)$ un sous-ensemble non vide C de D qui est totalement ordonné par la relation $\leq$.*

Définition 3.9.5 (max,min,limites) *Soit $(D, \leq)$ un ordre partiel, et S un sous-ensemble de D. On dit que un élément $a \in D$ est un :*

limite supérieure *de S si $\forall d \in S.d \leq a$*

limite inférieure *de S si $\forall d \in S.a \leq d$*

maximum *de S si $\forall d \in S.d \leq a$ et $d \in S$*

minimum *de S si $\forall d \in S.a \leq d$ et $d \in S$*

sup *de S s'il est la plus petite limite supérieure*

inf *de S s'il est la plus grande limite inférieure*

Définition 3.9.6 (Treilli) *Un treilli est un ensemble D équipé d'une relation d'ordre partiel et sur lequel tout sous-ensemble* fini *admet sup et inf (ou, cela revient au même (pourquoi ?) tel que $\forall d, d' \in D.sup(d, d')$ et $inf(d, d')$ sont définis).*

Définition 3.9.7 (Treilli complet) *Un treilli D est dit* complet *si tout sous-ensemble S de D admet sup et inf.*

Remarque 3.9.8 *Tout treilli complet a maximum $\top = sup(D)$ et minimum $\bot = sup(\{\})$.*

Théorème 3.9.9 (Tarski) *Toute fonction monotone $f : D \to D$ où D est un treilli* complet *admet un point fixe.*

Preuve. La preuve mérite d'être détaillée. Un *pre-point-fixe* de f est un élément $d \in D$ t.q. $d \leq f(d)$. Considérons l'ensemble des pre-point-fixes de f :

$$X = \{x | x \leq f(x)\}$$

On va prouver que le point $a = sup(X)$ est en effet un point fixe de f, à travers la suite d'implications :

$\forall x \in X.x \leq a$	$\Rightarrow$	def. sup
$\forall x \in X.f(x) \leq f(a)$	$\Rightarrow$	monotonie
$\forall x \in X.x \leq f(x) \leq f(a)$	$\Rightarrow$	def. X
$a \leq f(a)$	$\Rightarrow$	def. sup
$f(a) \leq f(f(a))$	$\Rightarrow$	monotonie
$f(a) \in X$	$\Rightarrow$	def. X
$f(a) \leq a$	$\Rightarrow$	def. sup
$f(a) = a$		

□

Remarque 3.9.10 *En effet, a est (par définition de sup) le* plus grand *point fixe de f. On peut aussi trouver le plus petit point fixe :*

$$p = inf\{x | f(x) \leq x\}$$

On peut prouver un résultat plus fort, qui était bien dans l'article de Tarski, mais que l'on retrouve rarement cité ensuite :

Théorème 3.9.11 (Tarski) *Toute fonction monotone $f : D \to D$ où D est un treilli* complet *admet un ensemble de points fixes qui forme un treilli complet ([Tar55]).*

Remarque 3.9.12 *Historiquement, les premiers approches à la sémantique dénotationnelle étaient bien basés sur les treillis complets (mais avec les fonctions continues) : en effet, notre métalangage peut être interprété avec des fonctions monotones sur un modèle fait des treilli complets, ce qui permet, grâce au théorème de Tarski, de pouvoir définir le point fixe de n'importe lequel fonctionnel construit à travers le métalangage. Cependant, imposer la structure de treilli sur les domaines est souvent trop demander, et en plus, si les fonctions sont seulement monotone, on n'a pas moyen de* calculer *de façon finitaire un point fixe, en général, et on préfère donc travailler avec les fonctions continues.*

3.10 La solution standard : les Ordres Partiels Complets (CPO)

En réalité, il y a des structures plus simples que les treillis complets avec les fonctions monotones sur lequels on peut construire notre sémantique : il s'agit des ordres partiels complets.

Définition 3.10.1 (CPO) *Un ordre partiel* $(D, \leq)$ *est dit* complet *si pour toute chaîne* C *dans* D *existe* $sup(C) \in D$.

Définition 3.10.2 (CPO pointé) *Un Cpo* D *est dit* pointé *s'il existe dans* D *un élément minimum, noté* $\bot$ *(lire "bottom").*

Définition 3.10.3 (fonction stricte) *Une fonction* $f : A \to B$ *entre deux cpos pointés* A *et* B *est dite* stricte *si* $f(\bot) = \bot$.

Notation 3.10.4 *On écrira souvent* $f(C)$ *pour* $\{f(d)|d \in C\}$.

La notion de fonction monotone définie pour les treillis reste bien la même pour les cpo's, mais il est utile de définir aussi ce que c'est une fonction *continue* :

Définition 3.10.5 (fonction continue) *Une fonction* $f : D \to D'$ *entre deux cpo's* D *et* D' *se dit* continue *si pour toute chaîne* C *dans* D *on a*

$$f(sup(C)) = sup(\{f(d)|d \in C\})$$

Proposition 3.10.6 *Une fonction continue est monotone.*

Pour travailler sur les CPO, il faut savoir manipuler aisement les chaînes et leur sup's. Pour cela, on a une batterie de lemmes utiles, dont en voici déjà quelques uns :

Lemme 3.10.7 (échange) *Si un ensemble d'élements* $\{a_{i,j}|i \in I, j \in J\}$, *indexé sur des ordres partiels* I *et* J, *est tel que* $sup_{j\in J}\{a_{i,j}\}$ *et* $sup_{i\in I}\{a_{i,j}\}$ *existent, alors*

$$\sup_{i\in I}\ \sup_{j\in J}\{a_{i,j}\} = \sup_{j\in J}\ \sup_{i\in I}\{a_{i,j}\}$$

Preuve. Notez d'abord que on à forcément $a_{i,j} \leq a_{i',j}$, si $i \leq_I i'$ et $a_{i,j} \leq a_{i,j'}$, si $j \leq_J j'$, donc on peut vérifier que $sup_{i\in I}\, sup_{j\in J}\{a_{i,j}\}$ et $sup_{j\in J}\, sup_{i\in I}\{a_{i,j}\}$ existent.

Ensuite, on vérifie immédiatement que pour tout i', j' on a $sup_{i\in I}\{a_{i,j'}\} \geq a_{i',j'}$, d'où, pour tout i', $sup_{j\in J}\, sup_{i\in I}\{a_{i,j}\} \geq sup_{j\in J}\{a_{i',j}\}$, et finalement, par définition de sup $sup_{j\in J}\, sup_{i\in I}\{a_{i,j}\} \geq sup_{i\in I}\, sup_{j\in J}\{a_{i',j}\}$, □

Lemme 3.10.8 (diagonale)

$$\sup_{i\in I\times I}\{a_{i,i}\} = \sup_{i\in I}\,\sup_{j\in I}\{a_{i,j}\} = \sup_{j\in I}\,\sup_{i\in I}\{a_{i,j}\}$$

Preuve. Analogue à la précédente. □

Lemme 3.10.9 (Continuité par composantes) *Une fonction de n arguments est continue si et seulement si elle l'est par composantes.*

Théorème 3.10.10 (Continuité de la composition) *Si $f : A \to B$ et $g : B \to C$ sont fonction continues, alors leur composition $g \circ f : A \to C \equiv [x \mapsto g(fx)]$ est une fonction continue.*

Lemme 3.10.11 (Itération vs. limites) *Si $F = \{f_i | i \in I\}$ est une chaîne de fonctions continues, alors*

$$\sup_{f\in F}(f^n) = (\sup_{f\in F} f)^n$$

Preuve. On procède par induction sur n.
– $n = 1$, trivial
– $n = m + 1$, on a alors :

$$\begin{aligned}
(\sup_{f\in F} f)^n &= (\sup_{g\in F} g) \circ (\sup_{f\in F} f)^{(n-1)} \\
&= (\sup_{g\in F} g) \circ (\sup_{f\in F} f^{(n-1)}) \\
&= \sup_{g\in F}\,\sup_{f\in F}(g \circ f^{(n-1)}) \\
&= \sup_{f\in F}(f \circ f^{(n-1)}) \\
&= \sup_{f\in F}(f^n)
\end{aligned}$$

□

Existence du point fixe

Théorème 3.10.12 *Toute fonction continue $f : D \to D$ sur un cpo D pointé admet un plus petit point fixe a.*

Preuve. Soit $a = sup\{f^n(\bot)|n \in I\!N\}$. On prouve que a est un point fixe :

$$\begin{array}{lcl} f(a) & = & \text{def. } a \\ f(sup\{f^n(\bot)|n \in I\!N\}) & = & \text{continuité} \\ sup\{f^{(n+1)}(\bot)|n \in I\!N\} & = & \text{def. } \bot \\ sup(\{f^{(n+1)}(\bot)|n \in I\!N\} \cup \{\bot\}) & = & \text{def. } f^0(\bot) \\ sup\{f^n(\bot)|n \in I\!N\} & = & \text{def. } a \\ a & & \end{array}$$

Pour prouver qu 'il est plus petit que tout autre point fixe b, il suffit de faire passer à la limite l'inégalité $\bot \leq b$. □

Pourquoi le plus petit point fixe ?

Il nous reste quand même à justifier le choix, parmi les nombreux point fixes disponibles, du plus petit. Il y a plusieurs raisons à cela, mais la plus importante reste la nécessité de garantir que *fix* soit un opérateur *uniforme*.

Définition 3.10.13 (opérateur de point fixe uniforme) *Un opérateur de point fixe fix est dit* uniforme *si pour toute paire de fonctions continues $f : A \to A$ et $g : B \to B$ (A, B pointés), et fonction stricte $h : A \to B$ t.q.*

$$\begin{array}{ccc} A & \xrightarrow{f} & A \\ {\scriptstyle h}\downarrow & & \downarrow{\scriptstyle h} \\ B & \xrightarrow{g} & B \end{array}$$

on a que $h(fix(f)) = fix(g)$.

Théorème 3.10.14 `fix` *est l'unique opérateur de point fixe uniforme.*

Si h n'est pas stricte, il arrive des fois qu'on trouve quand même $h(\bot) = g(\bot)$ et alors

Proposition 3.10.15 (Égalité utile) *Si $h(\bot) = g(\bot)$ et*

$$\begin{array}{ccc} A & \xrightarrow{f} & A \\ {\scriptstyle h}\downarrow & & \downarrow{\scriptstyle h} \\ B & \xrightarrow{g} & B \end{array}$$

on a aussi que $h(fix(f)) = fix(g)$.

3.10.1 Raisonner sur un point fixe

Pour pouvoir raisonner sur un objet qui est défini par point-fixe, il est utile d'avoir des principes d'induction adaptés, comme celui que l'on présente ici de suite.

Définition 3.10.16 (Ensemble inclusif) *Un sous-ensemble* $S \subseteq C$ *d'un cpo* C *pointé est dit* inclusif pour f *s'il satisfait ces conditions :*

- $\bot \in S$
- $x \in S \Rightarrow f(x) \in S$
- S *contient le* sup *de toute chaîne* $C \subseteq S$

Théorème 3.10.17 (Induction de point fixe) *Si* C *est un cpo pointé, et* $S \subseteq C$ *est inclusif pour une fonction continue* $f : C \to C$*, alors* $fix(f) \in S$

Normalement, l'ensemble S est défini à partir d'un quelque prédicat $P : C \to Bool$ sur C, comme $\{x|P(x)\}$, et on trouve dans la literature des définitions correspondantes de *prédicat inclusif* pour les prédicats P t.q. $\{x|P(x)\}$ est inclusif.

On rémarque aussi le critère suivant :

Théorème 3.10.18 (Égalité de point fixes) *Soit* $a = fix(F)$ *et* $b = fix(G)$*. Si l'on a* $a = G(a)$ *et* $b = F(b)$ *alors on a* $a = b$*.*

3.10.2 Intermezzo : topologie de Scott, I

Comme en analyse, ici aussi on peut capturer la notion de continuité de façon plus abstraite en utilisant des outils topologiques plutôt que la notion de préservation des limites. Mais, comme la structure des cpo's est bien plus simple que celle des réels, ils nous suffira d'utiliser des espaces topologiques bien plus faibles que ceux qui sont associés aux espaces métriques de l'analyse. Il s'agira en effet de topologies *T0*.

Définition 3.10.19 (Topologie) *Une topologie* $\mathcal{T}_E$ *sur un ensemble* E*, est la donnée d'une collection de sous-ensembles de* E*, nommés* ouverts*, qui ont les propriétés suivantes :*

- E *et* $\emptyset$ *appartiennent a* $\mathcal{T}_E$
- $\mathcal{T}_E$ *est fermée par intersection finie*

– *$\mathcal{T}_E$ est fermée par union arbitraire*

Définition 3.10.20 (cône supérieur) *Soit D un cpo et $d \in D$. On appelle* cône supérieur *de d l'ensemble $\breve{d} = \{d' | d \leq d'\}$.*

Définition 3.10.21 (Topologie de Scott pour les CPO) *Pour un cpo D, la topologie de Scott a pour ouverts les les sous-ensembles $A \subseteq D$ t.q. :*

fermeture vers le haut *si $x \in A$ alors $\breve{d} \subseteq A$*

si D est une chaîne non vide et $sup(D) \in A$, alors $D \cap A \neq \emptyset$

Proposition 3.10.22 *La topologie de Scott est bien une topologie.*

Preuve. exercice □

Définition 3.10.23 (continuité topologique) *Une fonction $f : D \to D'$ entre deux espaces topologiques D et D' est dite continue si l'image inverse à travers f de tout ouvert de D' est un ouvert A de D, i.e.*

$$\forall A \in T_{D'}. f^{-1}(A) \in T_D$$

Proposition 3.10.24 (préservation des limites vs. topologie de Scott) *Une fonction $f : D \to D'$ entre deux ordres partiels complets est continue ssi ell'est continue pour la topologie de Scott induite sur D et D' par la structure d'ordre.*

Preuve. Prouvez d'abord que $\{x | x \not\leq y\}$ est un ouvert, et montrez à l'aide de ce résultat qu'une fonction f continue topologique est monotone (et donc, que l'image d'une chaîne à travers f reste une chaîne). □

3.11 Construction de domaines et métalangage

L'intuition derrière la structure d'ordre des cpo's que l'on utilise dans la sémantique dénotationnelle, est qu'il s'agit d'un ordre (partiel) sur le contenu d'information. Par exemple, $\perp$ représente un (fragment) de programme qui est totalement indéfini (par exemple, s'il boucle sans rien faire comme `loop`), et qui contient donc le minimum possible d'information.

Dans cette section, on va voir comment, sur les cpo's *avec les fonctions continues*, on peut définir les mêmes opérateurs que l'on à utilisé dans le chapitre précédent pour construire FTH, tout en gardant le même métalangage. Cela nous permettra de "recuperer" les définitions sémantiques données pour TFun, et de rajouter, grâce aux point fixes qui existent sur les cpo's pointés, des définitions recursives à TFun.

3.11.1 Domaines de base

Une façon tout à fait standard d'obtenir un cpo C à partir d'un ensemble E consiste en utiliser sur E l'ordre *discret*, i.e. $\leq = \{(e,e)|e \in E\}$. Sur ces domaines, les seules chaines sont de la forme $\{e\}$ pour $e \in E$, et donc toute fonction $f : E \to D$ est trivialement continue. Par exemple, la structure des entiers $\mathbb{N}$ comme cpo avec l'ordre discret est la suivante :

$$0 \qquad 1 \qquad 2 \qquad \cdots$$

L'intuition étant que, si on compare les entiers *par rapport à leur contenu d'information*, ils sont tous incomparables.

3.11.2 Lifting

Si D est un domaine, alors $D_\perp$ (le "lifting" de D) est le domaine obtenu en rajoutant un (eventuellement nouveau) élément minimal sous D, et en rajoutant à la relation d'ordre $\leq$ de D les couples $\{(\perp,e)|e \in D\}$. C'est la façon canonique d'obtenir un cpo pointé à partir de un qui ne l'est pas.

metalangage Le lifting vient avec les opérations (continues) :

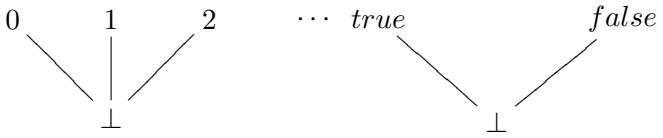
l'inclusion $up : A \to A_\perp$

si $f : A \to B_\perp$ alors $lift(f) : A_\perp \to B_\perp$, avec

$$lift(f)x = \begin{cases} \perp & \text{if } x = \perp \\ f(x) & \text{sinon} \end{cases}$$

3.11.3 Domaines plats

Une façon tout à fait standard d'obtenir un cpo *pointé* à partir d'un ensemble E consiste en "lifter" le cpo C associé a E. Cela s'appelle un domaine *plat*. Par exemple, la structure des entiers $\mathbb{N}$ et des valeurs de verité $Bool$ comme cpo plat $\mathbb{N}_\perp$ et $Bool_\perp$ est la suivante :



L'intuition étant que, si on compare les entiers *par rapport à leur contenu d'information*, ils sont encore tous incomparables, mais plus définis que $\perp$.

3.11.4 Produit

Si C et D sont deux cpo's, on définit alors le support du produit comme

$$\{(x, y)|x \in C, y \in D\}$$

et pour l'ordre, on le donne de la façon naturelle :

$$(x, y) \leq_{C\times D} (x', y') \Longleftrightarrow x \leq_C x', y \leq_D y'$$

métalangage Le produit vient équipé des mêmes opérations de projection et de formation de couple que l'on avait sur FTH, mais sur une structure plus riche. En particulier, on peut prouver que :

Théorème 3.11.1 (continuité) *La formation de couples et les fonctions $\pi_1 : A \times B \to A, \pi_2 : A \times B \to B$ sont continues.*

Sur les cpo (en raison de la faible topologie T0 dont ils sont équipés), on peut prouver le résultat suivant (qui est faux, par exemple, sur des ensembles équipés de topologies plus fortes, tels les espaces de Haussdorf).

Théorème 3.11.2 (Continuité par composantes) *Une fonction $f : A \times B \to C$ est continue ssi ses composantes*

$$f(\_, b) : A \to D \qquad f(a, \_) : B \to C$$

sont continues (pour tout $a \in A$ et $b \in B$).

3.11.5 Fonction

Si C et D sont deux cpo's, on définit alors

$$C \to D = \{f : C \to D|f \text{ continue}\}$$

avec l'ordre *par points* :

$$f \leq_{C\to D} g \Longleftrightarrow \forall x \in C.f(x) \leq_D g(x)$$

métalangage on retrouve ici l'abstraction et l'application que l'on avait sur FTH, avec la même signification, mais sur une structure plus riche. En particulier, on peut prouver que :

Théorème 3.11.3 (continuité) *La fonction* $app(\_,\_)$ *et la fonction* $\lambda x.e = [x \mapsto e]$ *(*e *expression du métalangage) sont continues.*

Mais on a aussi à notre disposition l'opérateur de point fixe, au cas où on travaille sur des domaines pointés :

Proposition 3.11.4 (Continuité de *fix*) *La fonction* $fix : (A \to A) \to A$ *qui associe à chaque fonction continue* $f : A \to A$ *(*A *pointé) son* plus petit *point fixe est continue.*

3.11.6 Somme

Si C et D sont deux cpo's, on définit alors le support de la somme comme la somme disjointe des supports de C et D, ce qui peut aussi se formaliser comme

$$\{x_C | x \in C\} \cup \{y_D | y \in D\} \cup \{\bot\}$$

et pour l'ordre, on a

$$x_C \leq_{C+D} x'_C \Longleftrightarrow x \leq_C x', \quad y_D \leq_{C+D} y'_D \Longleftrightarrow y \leq_D y'$$

métalangage Les sommes viennent équipés des mêmes opérations de injection et de analyse par cas que l'on avait sur FTH, mais sur une structure plus riche. En particulier, on peut prouver que :

Théorème 3.11.5 (continuité) *La fonction* $case$ *et les fonctions* in_1, in_2 *sont continues.*

Remarque 3.11.6 (CPO est une catégorie) *Il est facile de voir que les cpo's avec les fonctions continues forment une catégorie (on à déjà vérifié que la composition est continue, par exemple). En programmant un peu dans notre métalangage, il est possible de montrer que dans notre modèle des cpo's avec les fonctions continues on trouve aussi* $curry$, $apply$, fst, snd, $< \_,\_ >$ *et* $!A$ *qui satisfont les axiomes des catégories cartésiennes fermées. Cela nous montre que on à bien là une catégorie cartésienne fermée (CCC). Même plus : les sommes sont des vrais sommes catégoriques, et on a donc une catégorie bi-cartesienne fermée (BiCCC).*

Autres domaines utiles

3.11.7 Séquences

Si D est un domaine, alors D^* est le domaine des listes de longueur *finie* d'éléments de D. Une liste l est plus petite que l' si elles ont la même longueur *et* chaque élément de l est plus petit du correspondant élément dans l'.

metalangage Les sequences viennent avec les opérations (continues) :
- $nil : D^*$
- $hd : D^* \to D_\perp$
- $tl : D \to D^*$
- $cons : D \times D^* \to D^*$

3.11.8 One point CPO

C'est le cpo $1 = \{\perp\}$ avec un seul élément.

3.11.9 Fonctions strictes

Si C et D sont pointés, on note souvent par $C \circ\!\!\to D$ le sous-cpo de $C \to D$ qui est formé par le seules fonctions *strictes* entre C et D. Il y a une inclusion continue entre $C \circ\!\!\to D$ et $C \to D$.

Il est utile de savoir transformer en stricte une fonction quelconque.

Proposition 3.11.7 (Continuité de *strict*) *La fonction* strict $:(C \to D) \to (C \circ\!\!\to D)$ *définie comme*

$$strict(f)(x) = \begin{cases} f(x) & \text{si } x \neq \perp \\ \perp & \text{si } x = \perp \end{cases}$$

est continue.

Isomorphismes

Sur les cpo's la notion d'isomorphisme se doit de prendre en compte l'ordre.

Définition 3.11.8 (Isomorphismes entre cpo's) *Deux cpo's C et D sont isomorphes si il y a deux fonctions* continues $f : C \to D$ *et* $g : D \to C$ *t.q.* $f \circ g = id_D$ *et* $g \circ f = id_C$.

Déjà avec notre ensemble pas trop étendu de constructeurs de domaines, on a la possibilité d'établir une série d'isomorphismes intéressants.

- $A \times B \to C \cong A \to B \to C$
- $A \to B_\perp \cong A_\perp \circ\!\!\to B_\perp$
- $A \times (B \times C) \cong (A \times B) \times C$
- $A \times B \cong B \times A$
- $A + (B + C) \cong (A + B) + C$
- $A + B \cong B + A$
- $\mathbb{N} \cong \mathbb{N} \times \mathbb{N}$
- $\mathbb{N} \cong \mathbb{N} + \mathbb{N}$
- $Bool \cong 1 + 1$

3.11.10 Métalangage : sommaire

Pour facilité de référence, nous rappellons ici tout le métalangage que nous avons introduit pour noter les éléments des CPO's.

Syntaxe :

```
t : : = x | λ x∈ C.t | app(t,t)
    | lift(t) | up(t)
    | (t,t) | π1(t) | π2(t)
    | in1(t) | in2(t) | case(x,t,t)
    | nil | hd(t) | tail(t) | cons(t,t)
    | true | false | if t then t else t
    | fix(t)
```

où :

app :(A→B)×A→B est la fonction $app(f, a) = f(a)$

λ**x**$\in$ **A.e :(A→B)** est la fonction $f(x) = e$, pour x qui varie sur A, si $e[a/x]$ est un élément de B pour tout $a \in$ A

lift :(A→B$_\perp$)→(A$_\perp$→B$_\perp$) est l'injection entre (A→B$_\perp$) et (A$_\perp$→B$_\perp$)

up :A→A$_\perp$ est l'injection entre A et A$_\perp$

(e1,e2) :A×B est l'élément de $A \times B$ qui a pour composantes $e1 \in A$ et $e2 \in B$

etc.

Définition 3.11.9 (Expressions continues) *On dit que une expression e dont les variables libres sont parmi $x_1, \ldots, x_n$ est continue si $sup(e[\overrightarrow{a_i}/\overrightarrow{x_i}]) = e[\overrightarrow{sup(a_i)}/\overrightarrow{x_i}]$.*

Théorème 3.11.10 (Métalangage et continuité) *Toute expression bien typée $e : A$ du métalangage est continue.*

Preuve. On peut prouver cela par induction sur la structure de e, tout en prouvant aussi que toute instantiation de e avec des éléments des domaines appartient bien au domaine A. On ne detailera pas cette démonstration ici. □

Notations

Pour simplifier la notation dans le métalangage, on oublie souvent de noter les isomorphismes ou les inclusions, ce qui nous amène, par exemple, à généraliser les produit et la somme aux n-uplets.

n-uplets

$D_1 \times \cdots \times D_n$ avec les fonctions $(\_, \cdots, \_)$ et π_i

somme n-aire

$D_1 + \cdots + D_n$ avec les fonctions in_i et $case(\_, e_1, \cdots, e_n)$

En plus, on se donne les abréviations suivantes :

application

$(f\ e) = app(f, e)$

fonction produit

$< f_1, \cdots, f_n >$ = $\lambda x.(f_1(x), \cdots, f_n(x))$
$\lambda(x_1, \ldots, x_n).f$ = $\lambda x.(\lambda x_1. \ldots . \lambda x_n.f)(\pi_1(x)) \ldots (\pi_n(x)))$

produit de fonctions

$f_1 \times \cdots \times f_n$ = $\lambda x.(f_1(pi_1 x), \cdots, f_n(\pi_n x))$

fonction d'analyse par cas

$[f_1, \cdots, f_n] = \lambda x.case(x, f_1, \cdots, f_n)$

let

let $x = e$ in $e1$ *(ou aussi* $e1$ where $x = e$) = $(\lambda x.e1)e$
let $[x] = e$ in $e1$ = $(lift(\lambda x.e1))e$

extension de fonction

$f[l \mapsto e]$ = ($\lambda l'$.if $l = l'$ then e else $f(l)$)
N.B. : cette fonction ne peut-être continue que si l appartient à un cpo de base, en raison du test d'égalité, qui n'est pas continu en général

Relation entre CPO et CPO$_\perp$ (les CPO's toujours pointés)

Dans un certain nombre de livres de référence en litérature, on trouve ce mêmes conceptes dévéloppés non pas dans la BiCCC CPO des cpo's *sans* $\perp$ comme il est fait ici, mais dans la CCC CPO$_\perp$ des cpo's *avec* $\perp$ (i.e. la définition de cpo donnée pour CPO$_\perp$ inclut toujours un élément minimal, ce qui n'est pas notre cas.)

Il y a plusieurs raisons pour avoir choisi les cpo's sans bottom : l'existence d'un coproduit (la somme : montrez que dans CPO$_\perp$ il n'y a pas de coproduit), la possibilité d'avoir un contrôle plus fin sur l'élément minimal dans un cpo, ce qui nous permet par exemple de dériver de façon tout à fait simple, à partir de notre produit, somme et lifting, les différents produits (*direct* et *smashed*) et les différentes sommes (*direct* et *coalesced*) que l'on est obligés d'introduire sinon.

Exercice 3.11.11 (Définition des chaînes) *Expliquer pourquoi il ne serait pas bien de permettre, dans la définition de chaîne, la chaîne vide (qui est bien exclue ici).*

Exercice 3.11.12 (Quel cpo est pointé ?) *Donnez quelque règle pour d'eduire, en sachant si C et D sont pointés, si $D * C$ est pointé, où $* = \times, +, \rightarrow, \circ\!\!\rightarrow$.*

Exercice 3.11.13 *Montrez que, si A', B' sont des cpo's pointés dans la CCC des cpo's toujours pointés, et $A' = A_\perp$, $B' = B_\perp$ pour A, B dans la BiCCC des cpo's non pointés, alors on a :*

- $A' \times B' \cong (A_\perp \times B_\perp)_\perp$
- $A' \otimes B' \cong (A \times B)_\perp$
- $A' + B' \cong (A_\perp + B_\perp)_\perp$
- $A' \oplus B' \cong (A + B)_\perp$

Et un plus, on a aussi les produits semi-smashed

$$(A \times B_\perp)_\perp \qquad (A_\perp \times B)_\perp$$

et les sommes semi-coalesced

$$(A + B_\perp)_\perp \qquad (A_\perp + B)_\perp$$

3.12 Sémantique dénotationnelle de TFun dans CPO

On a déjà à notre disposition tout ce que l'on avait utilisé dans FTH, ce qui nous permet de recuperer sur les cpo's la sémantique de TFun$_{loop}$ presque inchangée, mais en traitant avec attention le $\perp$, qui est ici un élément minimale par rapport à l'ordre d'information, et non plus une constante introduite de façon arbitraire comme on avait fait dans FTH.

3.12.1 Appel par valeur

T$[\![]\!]$: Types $\rightarrow$ CPO
- $T[\![\texttt{int}]\!] = I\!N$
- $T[\![\texttt{bool}]\!] = Bool$
- $T[\![\texttt{t1 -> t2}]\!] = T[\![Id\texttt{[t1]}]\!] \rightarrow T[\![Exp\texttt{[t2]}]\!]$ (cpo des fonctions continues)
- $T[\![Exp\texttt{[t]}]\!] = \texttt{T}[\![\texttt{t}]\!]_\perp$
- $T[\![Id\texttt{[t]}]\!] = \texttt{T}[\![\texttt{t}]\!]$

G$[\![]\!]$: Environnements $\rightarrow$ CPO L'interprétation d'un environnement de typage est les produit de l'interprétations des types des identificateurs qu'il déclare :
- $G[\![\Gamma]\!] = T[\![Id\texttt{[t1]}]\!] \times \cdots \times T[\![Id\texttt{[tn]}]\!]$ if $\Gamma = \texttt{x1 :}Id\texttt{[t1]}, \cdots, \texttt{xn :}Id\texttt{[tn]}$

E$[\![]\!]$: Expression Judgements $\rightarrow$ CPO
- $[\![\Gamma \vdash \texttt{i:}Exp\texttt{[int]}]\!]_e = up(i)$ (pour `i` constante entière)
- $[\![\Gamma \vdash \texttt{true:}Exp\texttt{[bool]}]\!]_e = up(true)$,
- $[\![\Gamma \vdash \texttt{false:}Exp\texttt{[bool]}]\!]_e = up(false)$,
- $[\![\Gamma \vdash \texttt{loop:}Exp\texttt{[t]}]\!]_e = \perp \in [\![Exp\texttt{[t]}]\!]$,
- $[\![\Gamma \vdash \texttt{id:}Exp\texttt{[t]}]\!]_e = up(\pi_i(e))$ si `id` est le i-ème identificateur dans Γ
- $[\![\Gamma \vdash \texttt{fun x :t1->M:}Exp\texttt{[t1->t2]}]\!]_e = up(\lambda a.f(e,a))$
  où $f \in ([\![\Gamma]\!] \times [\![\texttt{t1}]\!] \rightarrow [\![Exp\texttt{[t2]}]\!])$ est l'interprétation $[\![\Gamma, \texttt{x :}Id\texttt{[t1]} \vdash \texttt{M:}Exp\texttt{[t2]}]\!]$

– $[\![\Gamma \vdash$ `M N:`Exp`[t]`$]\!]_e$ = let [h] = (f e) in let [a] = (g e) in h a

où $f = [\![\Gamma \vdash$ `M:`Exp`[t1->t]`$]\!]$ et $g = [\![\Gamma \vdash$ `N:`Exp`[t1]`$]\!]$.

Il est important pour bien suivre les définitions dénotationnelles de savoir quand les injections et les isomorphismes peuvent être oubliées sans ambiguité ; par exemple, dans ce qui precède, $up(true)$ pourrait s'abréger en $true$ sans ambiguité, vu que en regardant les domaines on peut reconstruire le up qui manque. Il en est de même pour l'espace des fonctions : $up(\lambda a.f(e,a))$ et $\lambda a.f(e,a)$ indiquent bien le même objet, vu que en regardant les domaines on s'aperçoit que même si on découvre que $\lambda a.f(e,a) = \bot \in [\![$`t1`$]\!] \to [\![$`t2`$]\!]_\bot$, cette fonction indéfinie partout ne peut pas être confondue avec $\bot \in ([\![$`t1`$]\!] \to [\![$`t2`$]\!]_\bot)_\bot$, qui n'est *pas* un valuer fonctionnel. Dans les chapitres ce qui suivent, on pourra, pour brevité, utiliser ces abréviations.

3.12.2 Appel par nom

T$[\![]\!]$: Types $\to$ CPO

– $T[\![$`int`$]\!]$ = $\mathbb{N}$
– $T[\![$`bool`$]\!]$ = $Bool$
– $T[\![$`t1 -> t2`$]\!]$ = $T[\![Id$`[t1]`$]\!] \to T[\![Exp$`[t2]`$]\!]$ (cpo des fonctions continues)
– $T[\![Exp$`[t]`$]\!]$ = `T`$[\![$`t`$]\!]_\bot$
– $T[\![Id$`[t]`$]\!]$ = `T`$[\![$`t`$]\!]_\bot$

G$[\![]\!]$: Environnements $\to$ CPO L'interprétation d'un environnement de typage est les produit de l'interprétations des types des identificateurs qu'il déclare :

– $G[\![\Gamma]\!]$ = $T[\![Id$`[t1]`$]\!] \times \cdots \times T[\![Id$`[tn]`$]\!]$ if $\Gamma =$ `x1 :`Id`[t1]`$, \cdots,$ `xn :`Id`[tn]`

E$[\![]\!]$: Expression Judgements $\to$ CPO

– $[\![\Gamma \vdash$ `i:`Exp`[int]`$]\!]_e = up(i)$ (pour `i` constante entière)
– $[\![\Gamma \vdash$ `true:`Exp`[bool]`$]\!]_e = up(true)$,
– $[\![\Gamma \vdash$ `false:`Exp`[bool]`$]\!]_e = up(false)$,
– $[\![\Gamma \vdash$ `loop:`Exp`[t]`$]\!]_e = up(\bot \in [\![Exp$`[t]`$]\!])$,
– $[\![\Gamma \vdash$ `id:`Exp`[t]`$]\!]_e = \pi_i(e)$ si `id` est le i-ème identificateur dans Γ
– $[\![\Gamma \vdash$ `fun x :t1->M:`Exp`[t1->t2]`$]\!]_e = up(\lambda a.f(e,a)))$
où $f \in ([\![\Gamma]\!] \times [\![Id$`[t1]`$]\!] \to [\![Exp$`[t2]`$]\!])$ est l'interprétation $[\![\Gamma,$ `x :`Id`[t1]` $\vdash$ `M:`Exp`[t2]`$]\!]$
– $[\![\Gamma \vdash$ `M N:`Exp`[t]`$]\!]_e$ = let [h] = (f e) in let a = (g e) in h a

où $f = [\![\Gamma \vdash \texttt{M:}Exp\texttt{[t1->t]}]\!]$ et $g = [\![\Gamma \vdash \texttt{N:}Exp\texttt{[t1]}]\!]$

3.12.3 Définitions recursives

En plus, on a la possibilité de former des fonctions recursives a tous les types. Cela nous suffit pour donner maintenat la sémantique (sur les cpo's avec les fonctions continues) d'une extension TFun$_{rec}$ de TFun avec des fonctions recursives :

syntaxe de TFun$_{rec}$ même que TFun, mais en plus pour tout type `t`, on rajoute

$$\frac{\Gamma, \texttt{f :}Id\texttt{[t1->t2]}, \texttt{x :}Id\texttt{[t1]} \vdash \texttt{M:}Exp\texttt{[t2]}}{\Gamma \vdash \texttt{f where rec f(x) = M:}Exp\texttt{[t1->t2]}}$$

pour lequel on peut donner la sémantique operationnelle comme suit :

CBV : on rajoute les règles

$$\texttt{f where rec f(x) = M} \Downarrow_v \texttt{f where rec f(x) = M}$$

$$\frac{e_1 \Downarrow_v \texttt{f where rec f(x) = e}_0 \quad e_2 \Downarrow_v v_2 \quad e_0[v_2/x; \texttt{f where rec f(x) = e}_0/f] \Downarrow_v v_0}{e_1(e_2) \Downarrow_v v_0}$$

CBN : on rajoute les règles

$$\texttt{f where rec f(x) = M} \Downarrow_n \texttt{f where rec f(x) = M}$$

$$\frac{e_1 \Downarrow_n \texttt{f where rec f(x) = e}_0 \quad e_0[e_2/x; \texttt{f where rec f(x) = e}_0/f] \Downarrow_n v}{e_1(e_2) \Downarrow_n v}$$

et la sémantique denotationnelle comme suit :

CBV

$$[\![\Gamma \vdash \texttt{f where rec f(x) = M:}Exp\texttt{[t1->t2]}]\!]_e =$$
$$up(fix(\lambda h.\lambda a.[\![\Gamma, \texttt{f :}Id\texttt{[t1->t2]}, \texttt{x :}Id\texttt{[t1]} \vdash \texttt{M:}Exp\texttt{[t2]}]\!](e, h, a)))$$

on remarque ici que l'on prend un point fixe sur un fonctionnel dans le cpo pointé $([\![\texttt{t1}]\!] \to [\![\texttt{t2}]\!]_\perp) \to ([\![\texttt{t1}]\!] \to [\![\texttt{t2}]\!]_\perp)$

CBN

$$[\![\Gamma \vdash \texttt{f where rec f(x) = M:}Exp\texttt{[t1->t2]}]\!]_e =$$
$$fix(\lambda h.up(\lambda a.[\![\Gamma, \texttt{f :}Id\texttt{[t1->t2]}, \texttt{x :}Id\texttt{[t1]} \vdash \texttt{M:}Exp\texttt{[t2]}]\!](e, h, a))))$$

on remarque ici que l'on prend un point fixe sur un fonctionnel dans le cpo pointé $([\![\texttt{t1}]\!]_\perp \to [\![\texttt{t2}]\!]_\perp)_\perp \to ([\![\texttt{t1}]\!]_\perp \to [\![\texttt{t2}]\!]_\perp)_\perp$

Mais on peut aussi (et il est fort pratique de le faire, si on veut ensuite implanter la sémantique dans un langage Call By Value comme OCaml), utiliser la définition alternative suivante :

$$[\![\Gamma \vdash \texttt{f where rec f(x) = M:}Exp\texttt{[t1->t2]}]\!]_e =$$
$$up(fix(\lambda h.\lambda a.[\![\Gamma, \texttt{f :}Id\texttt{[t1->t2]}, \texttt{x :}Id\texttt{[t1]} \vdash \texttt{M:}Exp\texttt{[t2]}]\!](e, up(h), a)))$$

on remarque ici que l'on prend un point fixe sur un fonctionnel dans le cpo pointé $([\![\texttt{t1}]\!]_\perp \to [\![\texttt{t2}]\!]_\perp) \to ([\![\texttt{t1}]\!]_\perp \to [\![\texttt{t2}]\!]_\perp)$ et après on lifte le résultat.

3.12.4 Discussion

Notons là dans quelles parties de la sémantique l'on prend les décision qui caractèrisent :

- l'appel par valeur (CBV) se caractèrise par le choix d'interpreter les paramètres formels sur un cpo non pointé et l'application par une fonction stricte dans les deux arguments
- l'appel par nom (CBN) se caractèrise par le choix d'interpreter les paramètres formels sur un cpo pointé et l'application par une fonction stricte dans le premier argument seulement
- la reduction faible (pas de reduction sous les lambdas) se voit dans l'utilisation de up pour interpreter les abstractions : une abstraction est toujours $up(f) \in [\![Exp\texttt{[t1->t2]}]\!]$, tandis que un valeur fonctionnel indéfini est bien $\perp \in [\![Exp\texttt{[t1->t2]}]\!]$, ce qui departage par exemple `loop` de `fun x :A-> loop`.
- la reduction forte (reduction sous les lambdas) s'obtiendrait en envoyant $\perp \in [\![\texttt{t1->t2}]\!]$ dans $\perp \in [\![Exp\texttt{[t1->t2]}]\!]$.

Conclusion

On a montré comment le metalangage que l'on avait extrait de FTH peut s'interpreter sur les cpo's avec les fonctions continues, une structure plus riche qui nous permet d'avoir le point fixe de toute fonction, et en particulier de toute fonction que l'on peut decrire à l'aide du metalangage. On a pu ainsi repondre à la premiere limitation de Set, notamment l'inadequation à la modelisation des définitions recursives. Cela nous suffit déjà pour traiter des langages plus réalistes, et on peut s'attaquer maintenant à la sémantique des autres constructions disponibles dans un langage de programmation, tels les declarations de constantes et variables, les effets de bord, les exceptions, le nondeterminisme et la concurrence.

3.13 Les déclarations

Dans notre langage TFun, on ne permet pas de déclarer explicitement des variables, locales ou globales. Et cependant, on peut facilement coder tout type de déclaration à l'aide des fonctions de TFun. Par exemple,

```
let x = e1 in e
```

peut être vu comme une abbreviation de

```
(fun x->e) e1
```

et aussi, pour les déclarations simultanés, on peu traiter

```
let x = e1 and y = e2 in e
```

comme une abréviation de

```
(fun x->fun y->e) e1 e2
```

ce qui est bien différent de

```
let x = e1 in y = e2 in e
```

qui est vu comme une abréviation de

```
(fun x->((fun y->e) e1)) e2
```

En effet, même si on ne souhaite pas faire appel à cette traduction, le mechanisme d'environnements mis en place pour traiter les fonctions est plus que suffisant pour gérer tout type de déclaration.

La formalisation directe des déclarations peut être très facilement dérivée de ces traductions.

syntaxe de TFun$_{decl}$ on étende celle de TFun comme suit :

expressions on a la nouvelle expression `let id : t = M1 in M2`

typage on rajoute pour tout type t la règle

$$\frac{\Gamma \vdash \texttt{M1:}Exp\texttt{[t1]} \quad \Gamma, x : Id[t1] \vdash \texttt{M2:}Exp\texttt{[t]}}{\Gamma \vdash \texttt{let x : t1 = M1 in M2:}Exp\texttt{[t]}}$$

Sémantique

appel par valeur

$$\begin{aligned}
&[\![\Gamma \vdash \texttt{let x :t1 = M1 in M2:}Exp\texttt{[t]}]\!]_e \\
&= \; lift(\lambda a.[\![\Gamma, x : Id[t1] \vdash \texttt{M2:}Exp\texttt{[t]}]\!](e, a))\ [\![\Gamma \vdash \texttt{M1:}Exp\texttt{[t1]}]\!]_e \\
&= \; \texttt{let}\ [a] = [\![\Gamma \vdash \texttt{M1:}Exp\texttt{[t1]}]\!]_e\ \texttt{in}\ [\![\Gamma, x : Id[t1] \vdash \texttt{M2:}Exp\texttt{[t]}]\!](e, a)
\end{aligned}$$

appel par nom

$$[\![\Gamma \vdash \texttt{let x :t1 = M1 in M2:}Exp\texttt{[t]}]\!]_e = \\ [\![\Gamma, x : Id[t1] \vdash \texttt{M2:}Exp\texttt{[t]}]\!](e, [\![\Gamma \vdash \texttt{M1:}Exp\texttt{[t1]}]\!]_e)$$

Exercice 3.13.1 *Donner la sémantique d'une construction pour la déclaration simultanée :*
`let id1 : t1 = M1 and id2 : t2 = M2 in M`

3.14 TFun$_{err}$ Un langage avec erreurs

Dans notre TFun, on n'a pas non plus la notion de *erreur* qui est aussi fondamentale dans les langages de programmation impératifs. Même dans des langages applicatifs, pour des raisons d'efficacité, il est certains fois utile d'avoir à disposition des fonctions pour generer et traiter des erreurs (qui s'appellent souvent *exceptions* dans le monde fonctionnel).

Nous nous intéressons ici d'abords aux erreurs et ensuite aux exceptions. De façon générale, il y a deux technique de modélisation adaptées au traitement des erreurs et des exceptions : les *drapeaux* (anglais : *flags*) et les *continuations*, ces dernières étant une technique plus complexe et générale qui ne devient nécessaire que pour traiter les sauts non structurés. Ici, on fera recours à la première technique.

syntaxe de TFun$_{err}$ on étende celle de TFun comme suit :

catégories syntaxiques on a une nouvelle catégorie syntaxique $Erreurs$ de *noms d'erreurs*

expressions on a des nouvelles expressions `error ern`, où `ern` et un nom d'erreur

typage on rajoute pour tout type t la règle

$$\frac{\texttt{ern} \in Erreurs}{\Gamma \vdash \texttt{error ern:}Exp\texttt{[t]}}$$

L'intuition opérationnelle étant que l'effet de `error ern` est d'arreter l'execution et renvoyer comme résultat de tout le programme le nom d'erreur `ern`.

Sémantique

Ici aussi, la possibilité pour toute expression de se terminer abnormalement sur un erreur nous oblige à modifier notre sémantique en profondeur : le résultat d'une expression, une fois que l'on lui donne un environnement de valeurs pour les variables libres, peut être maintenant soit un valuer, soit un erreur, ce qui nous suggère de modifier la sémantique de façon que la dénotation d'une expression rende en sortie le résultat du calcul *ou* un erreur.

$$E[\![]\!] : \text{Expression Judgements} \rightarrow Env \rightarrow (CPO + Err)$$

Ou pire, si on veut avoir des programmes qui peuvent ne pas terminer :

$$E[\![]\!] : \text{Expression Judgements} \rightarrow Env \rightarrow (CPO + Err)_{\perp}$$

Où Err est un domaine standard défini comme suit :

Err (de *Erreur*)	$=$	le cpo abstrait de noms d'erreurs avec l'ordre discret, i.e. sans bottom et avec tous les éléments incomparables, qui vient avec une égalité qui est bien continue en raison de l'ordre choisi

Voici la sémantique des types (pour le langage sans non-terminaison) :
- $T[\![\texttt{int}]\!]$ = $\mathbb{N}$
- $T[\![\texttt{bool}]\!]$ = $Bool$
- $T[\![\texttt{t1 -> t2}]\!] = T[\![Id\texttt{[t1]}]\!] \rightarrow T[\![Exp\texttt{[t2]}]\!]$ (cpo des fonctions continues)
- $T[\![Exp\texttt{[t]}]\!]$ = T$[\![$t$]\!]$+Err
- $T[\![Id\texttt{[t]}]\!]$ = T$[\![$t$]\!]$

et des nouvelles catégories syntaxiques et expressions :
- noms d'erreur :

$$[\![\texttt{ern}]\!] = ern \in Err$$

- génération d'un erreur

$$E[\![\Gamma \vdash \texttt{error ern:}Exp\texttt{[t]}]\!]_e = in_2([\![\texttt{ern}]\!])$$

Mais, et cela est très négatif, *toutes* les définitions sémantiques que l'on à déjà donné doivent changer, à partir de celles de constantes, vu qu'il faut, partout, vérifier si la sémantique d'une expression est un valeur ou un erreur.

- $[\![\Gamma \vdash \texttt{i:}Exp\texttt{[int]}]\!]_e = in_1(i)$ (pour `i` constante entière)
- $[\![\Gamma \vdash \texttt{true:}Exp\texttt{[bool]}]\!]_e = in_1(true)$,
- $[\![\Gamma \vdash \texttt{false:}Exp\texttt{[bool]}]\!]_e = in_1(false)$,
- $[\![\Gamma \vdash \texttt{id:}Exp\texttt{[t]}]\!]_e = in_1(\pi_i(e))$ (`id` le i-ème identificateur dans Γ)
- $[\![\Gamma \vdash \texttt{fun x :t1->M:}Exp\texttt{[t1->t2]}]\!]_e = in_1(\lambda a.h(e,a))$
  où $f \in ([\![\Gamma]\!] \times [\![\texttt{t1}]\!] \to [\![Exp\texttt{[t2]}]\!])$ est l'interprétation $[\![\Gamma, \texttt{x :}Id\texttt{[t1]} \vdash \texttt{M:}Exp\texttt{[t2]}]\!]$
- $[\![\Gamma \vdash \texttt{M N:}Exp\texttt{[t]}]\!]_e =$ case(f(e),λf'$\in[\![\texttt{t1->t2}]\!]$. case(g(e),$\lambda$v'$\in[\![\texttt{t1}]\!]$.
  (f' v') ,λn'$\in$ Err. $in_2(n')$) ,λn$\in$ Err. $in_2(n)$) où $f = [\![\Gamma \vdash \texttt{M:}Exp\texttt{[t1->t]}]\!]$
  et $g = [\![\Gamma \vdash \texttt{N:}Exp\texttt{[t1]}]\!]$.

Erreurs et nonterminaison

Ici on donne la sémantique pour le langage avec erreurs *et* nonterminaison : voici comment la sémantique des types change

- $T[\![\texttt{int}]\!] = \mathbb{N}$
- $T[\![\texttt{bool}]\!] = Bool$
- $T[\![\texttt{t1 -> t2}]\!] = T[\![Id\texttt{[t1]}]\!] \to T[\![Exp\texttt{[t2]}]\!]$ (cpo des fonctions continues)
- $T[\![Exp\texttt{[t]}]\!] = (T[\![\texttt{t}]\!] + Err)_\perp$
- $T[\![Id\texttt{[t]}]\!] = T[\![\texttt{t}]\!]$

et pour les nouvelles catégories syntaxiques et expressions :

- noms d'erreur :

$$[\![\texttt{ern}]\!] = ern \in Err$$

- génération d'un erreur

$$E[\![\Gamma \vdash \texttt{error ern:}Exp\texttt{[t]}]\!]_e = up(in_2([\![\texttt{ern}]\!]))$$

Encore une fois, *toutes* les définitions sémantiques que l'on à déjà donné doivent changer, à partir de celles de constantes, vu qu'il faut, partout, expliciter si la sémantique d'une expression est un valeur ou un erreur ou un calcul indéfini. Voici en détail le cas de l'appel par valeur.

- $[\![\Gamma \vdash \texttt{i:}Exp\texttt{[int]}]\!]_e = up(in_1(i))$ (pour `i` constante entière)
- $[\![\Gamma \vdash \texttt{true:}Exp\texttt{[bool]}]\!]_e = up(in_1(true))$,
- $[\![\Gamma \vdash \texttt{false:}Exp\texttt{[bool]}]\!]_e = up(in_1(false))$,
- $[\![\Gamma \vdash \texttt{loop:}Exp\texttt{[t]}]\!]_e = \perp \in [\![Exp\texttt{[t]}]\!]$,
- $[\![\Gamma \vdash \texttt{id:}Exp\texttt{[t]}]\!]_e = up(in_1(\pi_i(e)))$ (`id` le i-ème identificateur dans Γ)
- $[\![\Gamma \vdash \texttt{fun x :t1->M:}Exp\texttt{[t1->t2]}]\!]_e = up(in_1(\lambda a.h(e,a)))$
  où $f \in ([\![\Gamma]\!] \times [\![\texttt{t1}]\!] \to [\![Exp\texttt{[t2]}]\!])$ est l'interprétation $[\![\Gamma, \texttt{x :}Id\texttt{[t1]} \vdash \texttt{M:}Exp\texttt{[t2]}]\!]$

- $[\![\Gamma \vdash \texttt{M N:}Exp\texttt{[t]}]\!]_e =$ let [h] = f(e) in case(h,λf'$\in[\![\texttt{t1->t2}]\!]$.let [v] = g(e) in case(v,λv'$\in[\![\texttt{t1}]\!]$. f'(v') ,λn'$\in$ Err. $in_2(n')$) ,λn$\in$ Err. $in_2(n)$) où $f = [\![\Gamma \vdash \texttt{M:}Exp\texttt{[t1->t]}]\!]$ et $g = [\![\Gamma \vdash \texttt{N:}Exp\texttt{[t1]}]\!]$.
- et pour la recursion

  $$[\![\Gamma \vdash \texttt{f where rec f(x) = M:}Exp\texttt{[t1->t2]}]\!]_e = up(in_1(fix(\lambda f.\lambda a.[\![\Gamma, \texttt{f :}Id\texttt{[t1->t2]}, \texttt{x :}Id\texttt{[t1]} \vdash \texttt{M:}Exp\texttt{[t2]}]\!](e, f, a))))$$

  on remarque ici que l'on prend un point fixe sur un fonctionnel dans le cpo pointé $([\![\texttt{t1}]\!] \to [\![Exp\texttt{[t2]}]\!]) \to ([\![\texttt{t1}]\!] \to [\![Exp\texttt{[t2]}]\!])$

Exercice 3.14.1 (CBN) *Adapter la sémantique de TFun$_{rec}$ en CBN aux erreurs, analoguement à ce que l'on vient de faire pour la CBV ici.*

Remarque 3.14.2 (Manque de modularité) *Ici on voit bien que, même si l'approche dénotationnel direct classique est assez élégant et concis si on doit donner la sémantique d'un langage figé, il est par contre très décevant quand il s'agit de modifier une sémantique déjà donnée. Il faudra faire mieux que cela si on veut pouvoir étudier la sémantique de façon modulaire. En effet, en se basant sur des notions catégoriques, Eugenio Moggi à ouvert la voie à un approche de la sémantique dénotationnelle qui suit les lignes du génie logiciel.*

3.15 TFun$_{exc}$ Un langage avec exceptions

Le mecanisme mis en place pour modéliser les erreurs est suffisant pour traiter aussi les *exceptions*

syntaxe de TFun$_{exc}$ on étende celle de TFun$_{err}$ comme suit :

expressions on a des nouvelles expressions `handle(ern,M,N)` où `ern` est un nom d'erreur

typage on rajoute pour tout type t la règle

$$\frac{\texttt{ern} \in Erreurs \quad \Gamma \vdash \texttt{M:}Exp\texttt{[t]} \quad \Gamma \vdash \texttt{N:}Exp\texttt{[t]}}{\Gamma \vdash \texttt{handle(ern,M,N):}Exp\texttt{[t]}}$$

L'intuition opérationnelle étant que l'effet de `handle(ern,M,N)` est de donner comme résultat le résultat de l'execution de `M`, sauf si celui se termine sur l'erreur `ern`, et dans ce cas on renvoie le résultat de l'execution de `N`.

Sémantique

On rajoute à la sémantique de TFun$_{err}$ la clause suivante pour `handle(ern,M,N)` :
– $[\![\Gamma \vdash$ `handle(ern,M,N)` $: Exp$`[t]`$]\!]_e$ = let [m] = f(e) in case(m,λm'$\in[\![$`t`$]\!]$.up($in_1(m')$), λern'$\in$ Err. if ern' = ern then g(e) else up($in_2(ern')$))

3.16 TFun$_{ref}$ Un langage avec état

Dans notre TFun, on n'a pas la notion de *case mémoire* qui est fondamentale dans les langages de programmation impératifs. Même dans des langages applicatifs, pour des raisons d'efficacité, il est certains fois utile d'avoir à disposition des cases mémoire, et en effet on retrouve dans pas mal des dialects de ML une notion de référence qui n'est pas lointaine de celle que l'on va introduire ici. La sémantique que l'on va donner se base sur une manipulation *directe* de la modélisation de la mémoire, et ell'est bien connue comme *sémantique directe* pour cela en literature, en opposition à la sémantique par continuations que l'on va voir plus avant.

syntaxe de TFun$_{ref}$ on étende celle de TFun comme suit :

expressions on a les nouvelles expressions `ref e`, `!e` et `id :=e`

typage on rajoute pour tout type t que l'on veut mémoriser un type $t\ ref$, et les règles

$$\frac{\Gamma \vdash \texttt{M:}Exp\texttt{[t]} \quad \texttt{t mémorisable}}{\Gamma \vdash \texttt{ref M:}Exp\texttt{[t ref]}}$$

$$\frac{\Gamma \vdash \texttt{M:}Exp\texttt{[t ref]}}{\Gamma \vdash \texttt{!M:}Exp\texttt{[t]}}$$

Pour modifier une case mémoire, sans introduire tout de suite la notion de *commande*, on fait recours à un type bidon $unit$ avec un seul élement $* : unit$ et à la règle

$$\frac{\Gamma \vdash \texttt{id:}Exp\texttt{[t ref]} \quad \Gamma \vdash \texttt{M:}Exp\texttt{[t]}}{\Gamma \vdash \texttt{id := M:}Exp\texttt{[unit]}}$$

donc *l'expression* `id :=M` renvoie un valuer inintéressant $*$, et modifie la case mémoire associée à `id`.

Sémantique

Il est clair que, contrairement aux déclarations, la présence de cases mémoire nous oblige à modifier notre sémantique assez en profondeur : une expression ne se limite plus à nous renvoyer un résultat une fois que l'on lui donne un environnement de valeurs pour les variables libres, mais elle peut modifier les cases mémoire, qui constituent un *état*.

La solution traditionnelle est de modifier la sémantique de façon que la dénotation d'une expression prenne en paramétre l'etat courant, et rende en sortie le résultat du calcul et un état eventuellement modifié. Dans le cas simple de TFun sans non-terminaison plus état, on se retrouve avec une fonction d'interprétation sémantique :

$E[\![]\!]$: Expression Judgements $\rightarrow\ Env \rightarrow\ State \rightarrow\ (CPO \times\ State)$

Si les aspects impurs se limitent aux cases mémoires, on peut ne mettre dans l'état que une modélisation abstraite de la mémoire, vue comme fonction qui associe à chaque case son contenu.

Voilà quelques domaines qui sont standards pour réaliser cette modélisation (qui, pour simplicité, ne prend pas en compte la possibilité d'avoir epuisé la mémoire).

State = S(tore)	$=$	Loc $\rightarrow$Sv
Sv (de *Storable Values*)	$=$	l'union disjointe des interprétations des types mémorisables (par exemple, dans notre exemple en bas, on prend $\mathbb{N}$)
Loc (de *Location*)	$=$	un cpo abstrait de cases mémoire qui vient équipé d'une fonction predéfinie spéciale *newloc* : S $\rightarrow$Loc, qui est censée renvoyer une nouvelle case libre (i.e. n'ayant pas de valeur dans S)

Définition 3.16.1 (Manipulation de la mémoire (store) S) *On définit les suivantes fonctions de manipulation de la mémoire (store) :*

alloc-init :S$\rightarrow$Sv$\rightarrow$Loc$\times$S *modifie le store S en rajoutant une nouvelle case mémoire, initialisée avec la valeur passée, et renvoie la nouvelle case et le nouveau store :*

$$\textbf{alloc-init} = \lambda s \in S.\lambda v \in Sv.(\texttt{let l =}(newloc\ s)\ \texttt{in}\ \ (l, s[l \mapsto v]))$$

set :S→Sv→Loc→S *modifie un store en inserant la valeur passée en paramètre dans la case specifiée*

$$\textbf{set} = \lambda s \in S.\lambda v \in Sv.\lambda l \in Loc.s[l \mapsto v]$$

get :S→Loc→Sv *renvoie la valeur mémorisée dans l'etat et la case mémoire passés en paramètres :*

$$\textbf{get} = \lambda s \in S.\lambda l \in Loc.s(l)$$

Remarque 3.16.2 *Le domaine Sv est en general une somme de domaines, et alors dans get et alloc-init on doit rajouter le necessaire in_D pour indiquer que v est dans le domaine D.*

Voici comment changer la sémantique des types

- $T[\![\texttt{unit}]\!]$ = $\mathbb{1}$ = $\{\perp_{\mathbb{1}}\}$
- $T[\![\texttt{int ref}]\!] = Loc$
- $T[\![\texttt{int}]\!]$ = $I\!N$
- $T[\![\texttt{bool}]\!]$ = $Bool$
- $T[\![\texttt{t1 -> t2}]\!]$ = $T[\![Id\texttt{[t1]}]\!] \to T[\![Exp\texttt{[t2]}]\!]$ (cpo des fonctions continues)
- $T[\![Exp\texttt{[t]}]\!]$ = State→($T[\![\texttt{t}]\!]\times$State)
- $T[\![Id\texttt{[t]}]\!]$ = $T[\![\texttt{t}]\!]$

et des nouvelles expressions :

- initialisation d'une référence : on évalue d'abord M et ensuite on alloue la nouvelle case

$$E[\![\Gamma \vdash \texttt{ref M:}Exp\texttt{[int ref]}]\!]_e = \lambda s \in State.$$
$$\texttt{let } (v, s') = E[\![\Gamma \vdash \texttt{M:}Exp\texttt{[int]}]\!](e)(s)$$
$$\texttt{in } (\texttt{alloc-init}(s')(v))$$

- inspection d'une référence : on évalue d'abord M pour trouver la case et ensuite on renvoie son contenu

$$E[\![\Gamma \vdash \texttt{!M:}Exp\texttt{[int]}]\!]\_e = \lambda\ s \in State.$$
$$\texttt{let (l,s') = } E[\![\Gamma \vdash \texttt{M:}Exp\texttt{[int ref]}]\!](e)(s)$$
$$\texttt{in (get(s')(l),s')}$$

- modification d'une référence : on évalue d'abord M et ensuite on modifie le contenu de la case

$$E[\![\Gamma \vdash \texttt{id :=M:}Exp\texttt{[unit]}]\!]_e = \lambda s \in State.$$

$$\begin{array}{l}\texttt{let } (v, s') = E[\![\Gamma \vdash \texttt{M:}Exp\texttt{[t]}]\!](e)(s)\\ \texttt{in } (\bot_{\mathbb{1}}, \texttt{set}(s')(v)(\pi_i(e)))\end{array}$$

(id le i-ème identificateur dans Γ)

Malheureusement, *toutes* les définitions sémantiques que l'on à déjà donné doivent changer, à partir de celles de constantes, et deviennent

- $[\![\Gamma \vdash \texttt{i:}Exp\texttt{[int]}]\!]_e = \lambda s.(i, s)$ (pour i constante entière)
- $[\![\Gamma \vdash \texttt{true:}Exp\texttt{[bool]}]\!]_e = \lambda s.(true, s)$,
- $[\![\Gamma \vdash \texttt{false:}Exp\texttt{[bool]}]\!]_e = \lambda s.(false, s)$,
- $[\![\Gamma \vdash \texttt{id:}Exp\texttt{[t]}]\!]_e = \lambda s.(\pi_i(e), s)$ (id le i-ème identificateur dans Γ)
- $[\![\Gamma \vdash \texttt{fun x :t1->M:}Exp\texttt{[t1->t2]}]\!]_e = \lambda s.(\lambda a.\lambda s'.f(e, a)(s'), s)$
  où $f \in ([\![\Gamma]\!] \times [\![\texttt{t1}]\!] \to [\![Exp\texttt{[t2]}]\!])$ est l'interprétation $[\![\Gamma, \texttt{x :}Id\texttt{[t1]} \vdash \texttt{M:}Exp\texttt{[t2]}]\!]$
- $[\![\Gamma \vdash \texttt{M N:}Exp\texttt{[t]}]\!]_e =$
  $\lambda s.\ \texttt{let } (f', s') = f(e)(s) \texttt{ in let } (v, s'') = g(e)(s') \texttt{ in } f'(v)(s'')$

  où $f = [\![\Gamma \vdash \texttt{M:}Exp\texttt{[t1->t]}]\!]$ et $g = [\![\Gamma \vdash \texttt{N:}Exp\texttt{[t1]}]\!]$.

3.16.1 État et nonterminaison ensemble

Maintenant, on peut se démander que se passet-il quand on a à la fois un état et la possibilité de programmer des fonctions recursives. La sémantique doit changer pour prendre en compte la possibilité de nonterminaison, ce qui nous amène à définir la fonction d'interpretation sémantique comme suit :

$E[\![]\!]$: Expression Judgements $\to\ Env \to\ State \to\ (CPO \times\ State)_\perp$

Pour la sémantique des types (ici, on traite le cas CBV), on doit alors changer la définition de $[\![Exp\texttt{[t]}]\!]$:

- $T[\![Exp\texttt{[t]}]\!]$ = State→(T$[\![\texttt{t}]\!]$×State)$_\perp$

(ce qui à pour effet de changer aussi le domaine d'interprétation de la flèche).

Et, à nouveau, *toutes* les définitions sémantiques que l'on à déjà donné doivent changer, à partir de celles de constantes, et deviennent (dans le cas de l'appel par valeur) :

- initialisation d'une référence : on évalue d'abord M et ensuite on alloue la nouvelle case

$$\begin{array}{l}E[\![\Gamma \vdash \texttt{ref M:}Exp\texttt{[t ref]}]\!]_e = \lambda s \in State.\\ \quad \texttt{let } [(v, s')] = E[\![\Gamma \vdash \texttt{M:}Exp\texttt{[t]}]\!](e)(s)\\ \quad \texttt{in } up((\texttt{alloc-init}(s')(v)))\end{array}$$

– inspection d'une référence : on évalue d'abord M pour trouver la case et ensuite on renvoie son contenu

$$E[\![\Gamma \vdash \texttt{!M:}Exp\texttt{[t]}]\!]_e = \lambda s \in State.$$
$$\texttt{let } [(l, s')] = E[\![\Gamma \vdash \texttt{M:}Exp\texttt{[t ref]}]\!](e)(s)$$
$$\texttt{in } up((\texttt{get}(s')(l), s'))$$

– modification d'une référence : on évalue d'abord M pour trouver la case et ensuite on modifie son contenu

$$E[\![\Gamma \vdash \texttt{id :=M:}Exp\texttt{[unit]}]\!]_e = \lambda s \in State.$$
$$\texttt{let } [(v, s')] = E[\![\Gamma \vdash \texttt{M:}Exp\texttt{[t]}]\!](e)(s)$$
$$\texttt{in } up((\bot_{\mathbb{1}}, \texttt{set}(s')(v)(\pi_i(e))))$$

(id le i-ème identificateur dans Γ)

– $[\![\Gamma \vdash \texttt{i:}Exp\texttt{[int]}]\!]_e = \lambda s.up((i, s))$ (pour i constante entière)
– $[\![\Gamma \vdash \texttt{true:}Exp\texttt{[bool]}]\!]_e = \lambda s.up((true, s))$,
– $[\![\Gamma \vdash \texttt{false:}Exp\texttt{[bool]}]\!]_e = \lambda s.up((false, s))$,
– $[\![\Gamma \vdash \texttt{loop:}Exp\texttt{[t]}]\!]_e = \lambda s.(\bot \in ([\![\texttt{t}]\!] \times S)_\bot) \equiv \bot \in [\![Exp\texttt{[t]}]\!]$,
– $[\![\Gamma \vdash \texttt{id:}Exp\texttt{[t]}]\!]_e = \lambda s.up((\pi_i(e), s))$ (id le i-ème identificateur dans Γ)
– $[\![\Gamma \vdash \texttt{fun x :t1->M:}Exp\texttt{[t1->t2]}]\!]_e = \lambda s.up((\lambda a.\lambda s'.f(e, a)(s'), s))$
où $f \in ([\![\Gamma]\!] \times [\![\texttt{t1}]\!] \to [\![Exp\texttt{[t2]}]\!])$ est l'interprétation $[\![\Gamma, \texttt{x :}Id\texttt{[t1]} \vdash \texttt{M:}Exp\texttt{[t2]}]\!]$. Au fait, $\lambda s'.f(e, a)(s') = f(e, a)$, mais on a donné la version plus redondante pour clarté.
– $[\![\Gamma \vdash \texttt{M N:}Exp\texttt{[t]}]\!]_e =$
$\lambda s.\ \texttt{let } [(f', s')] = f(e)(s) \ \texttt{in let } [(v, s'')] = g(e)(s') \ \texttt{in } f'(v)(s'')$

où $f = [\![\Gamma \vdash \texttt{M:}Exp\texttt{[t1->t]}]\!]$ et $g = [\![\Gamma \vdash \texttt{N:}Exp\texttt{[t1]}]\!]$.
– et pour la recursion

$[\![\Gamma \vdash \texttt{f where rec f(x) = M:}Exp\texttt{[t1->t2]}]\!]_e =$
$\lambda s.up(((fix(\lambda f.\lambda a.[\![\Gamma, \texttt{f :}Id\texttt{[t1->t2]}, \texttt{x :}Id\texttt{[t1]} \vdash \texttt{M:}Exp\texttt{[t2]}]\!](e, f, a))), s))$

Ici l'on prend un point fixe sur un fonctionnel dans le cpo pointé

$$([\![\texttt{t1}]\!] \to [\![Exp\texttt{[t2]}]\!]) \to ([\![\texttt{t1}]\!] \to [\![Exp\texttt{[t2]}]\!])$$

(il est pointé parceque $[\![Exp\texttt{[t]}]\!] = State \to (T[\![\texttt{t}]\!] \times State)_\bot$ l'est).

Exercice 3.16.3 (CBN) *Adapter la sémantique de TFun$_{rec}$ en CBN aux références, analoguement à ce que l'on vient de faire pour la CBV ici.*

Exercice 3.16.4 (Mémoire et S) *On a dans notre approche fait une assumption très simpliste sur les états $s : Loc \to Sv$: en effet, on a supposé qu'ils sachent renvoyer toujours un valeur pour toute location, même une location qui n'a pas été encore initialisée, ce qui n'est raisonnable que si toute location contient le même type de valeur. Modifiez la définition des domaines sémantiques pour que l'on puisse renvoyer un erreur sur les cases non initialisées, et modifiez la sémantique du langage en consequence. Discutez l'importance des modifications nécéssaires.*

Remarque 3.16.5 (Manque de modularité) *Même remarque que pour les erreurs.*

3.17 Un langage avec commandes

Il n'est pas difficile, maintenant, de rajouter des *commandes* à TFun$_{ref}$: il ne s'agira que de fonctions dont le résultat n'est pas interessant (et on leur donnera donc le type $unit$), qui travaillent seulement sur l'état.

syntaxe on étende celle de TFun$_{ref}$ comme suit :

expressions on a des nouvelles expressions `while e do c`, `repeat c until e` et `c1 ;c2`

typage on rajoute les règles

$$\frac{\Gamma \vdash \texttt{c1:}Exp\texttt{[unit]} \quad \Gamma \vdash \texttt{c2:}Exp\texttt{[unit]}}{\Gamma \vdash \texttt{c1 ;c2:}Exp\texttt{[unit]}}$$

pour typer la séquentialisation de deux commandes

$$\frac{\Gamma \vdash \texttt{b:}Exp\texttt{[bool]} \quad \Gamma \vdash \texttt{c:}Exp\texttt{[unit]}}{\Gamma \vdash \texttt{while b do c:}Exp\texttt{[unit]}}$$

pour typer une boucle *while*

$$\frac{\Gamma \vdash \texttt{b:}Exp\texttt{[bool]} \quad \Gamma \vdash \texttt{c:}Exp\texttt{[unit]}}{\Gamma \vdash \texttt{repeat c until b:}Exp\texttt{[unit]}}$$

pour typer une boucle *repeat*

Sémantique

La séquentialisation de deux commandes revient à exécuter le deuxième dans l'état modifié par le premier (les résultats, qui sont inintéressants, sont jetés).

On à déjà discuté de façon informelle la sémantique du `while`, qui fera recours au point fixe. Ici, on peut finalement la formaliser.

– $[\![\Gamma \vdash \texttt{c1;c2:}Exp\texttt{[unit]}]\!]_e = \lambda s \in S.$
```
          let [(v,s')] = [[Γ ⊢ c1:Exp[unit]]](e)(s)
          in [[Γ ⊢ c2:Exp[unit]]](e)(s')
```

– $[\![\Gamma \vdash$ `while b do c:`Exp`[unit]`$]\!]_e =$
```
           fix(λ h∈ S → (𝟙 × S)⊥.λ s∈ S.
               let [(v,s')] = [[Γ ⊢ b:Exp[bool]]](e)(s)
               in if v then let [(v',s'')] = [[Γ ⊢ c:Exp[unit]]](e)(s')
                            in h(s'')
                       else (⊥𝟙,s'))
```

– $[\![\Gamma \vdash$ `repeat c until b:`Exp`[unit]`$]\!]_e =$
```
           fix(λ h∈ S → (𝟙 × S)⊥.λ s∈ S.
               let [(v,s')] = [[Γ ⊢ c:Exp[unit]]](e)(s)
               in let [(v',s'')] = [[Γ ⊢ b:Exp[bool]]](e)(s')
                  in  if v' then (⊥𝟙,s''))
                            else h(s'')
```

Exercice 3.17.1 *Montrez que, pour tout commande* c *et expression booléenne* b *on peut prouver l'egalité suivante.*

$$[\![\texttt{repeat c until b}]\!] = [\![\texttt{c ;while (not b) do c}]\!]$$

Exercice 3.17.2 (Sémantique opérationnelle) *Reflechissez à une sémantique opérationnelle pour TFunref avec commandes, dans le but d'arriver à prouver la correction de la sémantique dénotationnelle que l'on vient de donner par rapport à l'equivalence observationnelle induite par l'observation des valeurs de base.*

3.18 Sémantique avec continuations

Dans la plupart des langages de programmation modernes il y a des mécanismes dont le but est de permettre de modifier l'ordre normale d'exe-

cution des instructions d'un programme. Cela n'a pas (ou plus) le but d'utiliser systematiquement des *goto* la où des constructions plus propres tels les boucles *while* sont appropriées (et cela depuis le travail fondamentale de Böhm et Iacopini), mais plutôt de permettre de traiter de façon efficace des situations exceptionnelles (tels les erreurs et les exceptions).

On a déjà vu comment la technique des drapeaux permet de traiter les erreurs et les exceptions : essentialement, chaque clause sémantique *propage* les erreurs où les exceptions jusqu'au point ils poivent être traités. Cela alourdit remarquablement la définition de la sémantique, et ne suffit pas pour traiter les mécanismes de saut non structurés qui existent encore dans les langages de programmation. Pour cela on doit faire recours à la technique des *continuations*.

Informellement, une continuation k est une formalisation du "reste du programme" : si une expression (ou commande) e produit une valeur v, le "reste du programme" sera bien une fonction qui prend cette valeur v et fabrique le résultat finale du programme tout entier, celuici étant normalement défini par un type (ou ensemble) R de *résultats* (souvent aussi nommé *Ans*, de "Answers", dans la literature anglaise).

L'avantage d'avoir sous la main le "reste du programme" à chaque coup est que on peut le jeter (si il y a un erreur), le sauver dans un environnement (utile pour traiter les etiquettes des *goto*), le restaurer à partir d'un environnement etc.

Sémantique

Voyons comment on se doit de modifier notre sémantique pour introduire les continuations. On avait auparavant une fonction sémantique de la forme

$$E[\![\_ : Exp\texttt{[t]}]\!] : \text{Expression Judgements} \to Env \to \mathcal{T}([\![\texttt{t}]\!])$$

où $\mathcal{T}(\_)$ etait soit $\__\perp$ ou $\_ + Err$ ou $S \to (\_ \times S)$ ou même qque chose de plus complexe, en fonction des caracteristiques des programmes à modeliser, et correspondait de façon assez precise à l'intuition de "calcul de type t" implicite dans la catégorie syntaxique Exp[t].

Si on souhaite utiliser des *continuations*, on va prendre en compte le fait que le calcul d'une expression *depende* du "reste du programme", capturé par la continuation.

$$E[\![\_ : Exp\texttt{[t]}]\!] : \text{Expression Judgements} \to Env \to ([\![\texttt{t}]\!] \to \mathcal{T}(\mathcal{R})) \to \mathcal{T}(\mathcal{R})$$

Cette modification se voit aussi dans la sémantique de Exp[t], qui change pour prendre en compte le fait que le calcul d'une expression *depende* de la continuation.

Voyons cela sur un exemple très simple, TFun sans divergence, en commençant par la sémantique des types :

- $T[\![\texttt{int}]\!] = \mathbb{N}$
- $T[\![\texttt{bool}]\!] = Bool$
- $T[\![\texttt{t1 -> t2}]\!] = T[\![Id\texttt{[t1]}]\!] \to T[\![Exp\texttt{[t2]}]\!]$ (cpo des fonctions continues)
- $T[\![Exp\texttt{[t]}]\!] = (T[\![\texttt{t}]\!] \to R) \to R$
- $T[\![Id\texttt{[t]}]\!] = \texttt{T}[\![\texttt{t}]\!]$

Voici comment change la sémantique des expressions (pour la CBV) :

- $[\![\Gamma \vdash \texttt{i:}Exp\texttt{[int]}]\!]_e = \lambda k.k(i)$
- $[\![\Gamma \vdash \texttt{true:}Exp\texttt{[bool]}]\!]_e = \lambda k.k(true)$
- $[\![\Gamma \vdash \texttt{false:}Exp\texttt{[bool]}]\!]_e = \lambda k.k(false)$
- $[\![\Gamma \vdash \texttt{id:}Exp\texttt{[t]}]\!]_e = \lambda k.k(\pi_i(e))$ (id le i-ème identificateur dans Γ)
- $[\![\Gamma \vdash \texttt{fun x :t1->M:}Exp\texttt{[t1->t2]}]\!]_e = \lambda k.k(\lambda a.f(e,a))$
  où $f \in ([\![\Gamma]\!] \times [\![\texttt{t1}]\!] \to [\![Exp\texttt{[t2]}]\!])$ est l'interprétation $[\![\Gamma, \texttt{x } :Id\texttt{[t1]} \vdash \texttt{M:}Exp\texttt{[t2]}]\!]$
- $[\![\Gamma \vdash \texttt{M N:}Exp\texttt{[t]}]\!]_e = \lambda k.f(e)(\lambda h.g(e)(\lambda v.(h\ v)k))$ où $f = [\![\Gamma \vdash \texttt{M:}Exp\texttt{[t1->t]}]\!]$ et $g = [\![\Gamma \vdash \texttt{N:}Exp\texttt{[t1]}]\!]$.

Remarque 3.18.1 *Pour les expressions de type de base, on a souvent le resultat recherché si on évalue la sémantique sur la continuation identité $\lambda x.x$. Que faut-il faire si on a des expressions de type non de base ?*

Remarque 3.18.2 *Ici on a pris, informellement, $R = CPO$, mais dans les langages imperatifs, R est souvent juste l'état.*

Sémantique en CBN pour donner une sémantique en CBN, on doit changer la clause de l'application, pour ne pas forcer l'évaluation de l'argument, et on n'a pas besoin de lifter les identificateurs.

- $[\![\Gamma \vdash \texttt{id:}Exp\texttt{[t]}]\!]_e = \pi_i(e)$ (id le i-ème identificateur dans Γ)
- $[\![\Gamma \vdash \texttt{M N:}Exp\texttt{[t]}]\!]_e = \lambda k.f(e)(\lambda h.h(g(e)k))$ où $f = [\![\Gamma \vdash \texttt{M:}Exp\texttt{[t1->t]}]\!]$ et $g = [\![\Gamma \vdash \texttt{N:}Exp\texttt{[t1]}]\!]$.

3.19 Une pause de reflexion

Maintenant que l'on a vu comment traiter a peu près tous les aspects des langages de programmation (sequentiels), nous pouvons marquer une

pause de reflexion et regarder la tête que nos définitions sémantiques ont pris. On a déjà vu qu'il y avait un manque de modularité, mais voyons cela mieux maintenant sur deux cas significatifs : comment change la sémantique d'une *constante*, et comment change la sémantique d'une application de fonction.

3.19.1 Sémantique d'une constante entière

$[\![Exp\texttt{[t]}]\!]$	$[\![\Gamma \vdash \texttt{i:}Exp\texttt{[int]}]\!]_e$
$[\![\texttt{t}]\!]$	i
$[\![\texttt{t}]\!]_\perp$	$up(i)$
$[\![\texttt{t}]\!] + Err$	$in_1(i)$
$S \to ([\![\texttt{t}]\!] \times S)$	$\lambda s.(i, s)$
$([\![\texttt{t}]\!] \to R) \to R$	$\lambda k.k(i)$

Ici, c'est assez clair que ce que l'on fait sur la constante i revient juste à la plonger dans le domaine des calculs associé à Exp[int], en passant toute la structure supplementaire inchangée. Si on veut donner une presentation modulaire de la sémantique, il nous faudra rajouter à notre métalangage une fonction

$$val : [\![\texttt{t}]\!] \to [\![Exp\texttt{[t]}]\!]$$

qui va abstraire cette opération de plongement. La fonction val devra être specifiée avec Exp[t].

3.19.2 Sémantique de l'application de fonction

$[\![Exp\texttt{[t]}]\!]$	$[\![\Gamma \vdash \texttt{M N:}Exp\texttt{[t]}]\!]_e$	
$[\![\texttt{t}]\!]$	$([\![\texttt{M}]\!]_e)([\![\texttt{N}]\!]_e)$	
$[\![\texttt{t}]\!]_\perp$	$let[f] = [\![\texttt{M}]\!]_e \ in \ let[v] = [\![\texttt{N}]\!]_e \ in \ fv$	(CBV)
	$let[f] = [\![\texttt{M}]\!]_e \ in \ f([\![\texttt{N}]\!]_e)$	(CBN)
$[\![\texttt{t}]\!] + Err$	$case([\![\texttt{M}]\!]_e, \lambda f.f([\![\texttt{N}]\!]_e)\lambda err.in_2(err))$	(CBN)
	$case([\![\texttt{M}]\!]_e, \quad \lambda f.case([\![\texttt{N}]\!]_e. \quad \lambda v.fv$	
	$\lambda err.in_2(err))$	(CBV)
	$\lambda err.in_2(err))$	
$S \to ([\![\texttt{t}]\!] \times S)$	$let(f, s') = [\![\texttt{M}]\!](e)(s) \ in \ let(v, s'') = [\![\texttt{N}]\!](e)(s') \ in \ (fv)s''$	(CBV)
	$let(f, s') = [\![\texttt{M}]\!](e)(s) \ in \ (f[\![\texttt{N}]\!](e))s'$	(CBN)
$([\![\texttt{t}]\!] \to R) \to R$	$\lambda k.[\![\texttt{M}]\!]_e(\lambda h.[\![\texttt{N}]\!]_e(\lambda v.(h \ v)k))$	(CBV)
	$\lambda k.[\![\texttt{M}]\!]_e(\lambda h.h([\![\texttt{N}]\!]_e)k)$	(CBN)

Encore une fois, comme pour la fonction val, on voudrait trouver ici une façon uniforme d'écrire la sémantique de l'application en utilisant une quelque forme de abstraction (une autre fonction qui dependra de Exp[t] à rajouter à notre métalangage), qui ne change pas quand on change Exp[t]. Ici, c'est plus difficile de reperer une seule structure sousjacente à ces expressions apparemment très differentes, mais on peut verifier qu'il suffit d'une seule fonction :

$$Lift : ([\![\texttt{t}]\!] \to [\![Exp\texttt{[t']}]\!]) \to ([\![Exp\texttt{[t]}]\!] \to [\![Exp\texttt{[t']}]\!])$$

3.19.3 La sémantique modulaire

Pour notre approche modulaire à la sémantique on introduira alors un type de donnée abstrait qui sera constitué de :

- un type polymorphe $T(X)$
- une fonction $val : X \to T(X)$
- une fonction $Lift : (X \to T(Y)) \to (T(X) \to T(Y))$

Notation 3.19.1 *On écrira* `let x ⇐ c in e` *pour* $(Lift(\lambda x.e))c$.

Ce type de donnée est censée encapsuler toute l'information nécessaire à donner la sémantique de la partie de langage qui ne depende pas de la structure de Exp[t]. Et en effet, on pourra écrire :

- $[\![\texttt{Expt}]\!] = T([\![\texttt{t}]\!])$
- $[\![\Gamma \vdash \texttt{i:}Exp\texttt{[int]}]\!]_e = val(i)$
- $[\![\Gamma \vdash \texttt{M N:}Exp\texttt{[t]}]\!]_e =$
  $let\ f \Leftarrow [\![\Gamma \vdash \texttt{M:}Exp\texttt{[t->t1]}]\!]\ in\ (let\ v \Leftarrow ([\![\Gamma \vdash \texttt{N:}Exp\texttt{[t1]}]\!])\ in\ fv)$
  (CBV)
- $[\![\Gamma \vdash \texttt{M N:}Exp\texttt{[t]}]\!]_e =$
  $let\ f \Leftarrow [\![\Gamma \vdash \texttt{M:}Exp\texttt{[t->t1]}]\!]\ in\ f([\![\Gamma \vdash \texttt{N:}Exp\texttt{[t1]}]\!])$ (CBN)

Et voici la forme du type de donnée abstrait pour les différents Exp[t] que l'on a vu auparavant :

$[\![Exp\texttt{[t]}]\!]$	$T(t)$	Val	$let\ x \Leftarrow c\ in\ e$
$[\![\texttt{t}]\!]$	t	$\lambda i.i$	$(\lambda x.e)c$
$[\![\texttt{t}]\!]_\perp$	$t_\perp$	$\lambda i.up(i)$	$(lift(\lambda x.e))c$
$[\![\texttt{t}]\!] + Err$	$t + Err$	$\lambda i.in_1(i)$	$case(c, \lambda x.e, \lambda err.in_2(err))$
$S \to ([\![\texttt{t}]\!] \times S)$	$S \to (t \times S)$	$\lambda i.\lambda s.(i, s)$	$\lambda s.let(v, s') = c(s)\ in\ (\lambda x.e)(v)(s')$
$([\![\texttt{t}]\!] \to R) \to R$	$(t \to R) \to R$	$\lambda i.\lambda k.k(i)$	$\lambda k.c(\lambda x.e(k))$

Il est maintenant un simple exercice de verifier que la sémantique donnée à l'aide de val et $Lift$ de façon uniforme s'instancie correctement sur les sémantiques données directement, une fois que l'on choisie le correcte type de donnée associé à Exp[t].

3.20 Axiomes des monades

Sur notre type de donnée abstrait (qui revient en effet à une *monade*), on va demander aussi que les suivantes egalités soient valides :

$$\begin{aligned} Lift(f)(val(x)) &= f(x) \\ Lift(val)(c) &= c \\ Lift(g)(Lift(f)(c)) &= Lift(\lambda x.Lift(g)(fx))(c) \end{aligned}$$

En terme de notre construction abregée let x⇐ c in e, cela peut aussi s'écrire :

$$\begin{aligned} let\ y \Leftarrow val(x)\ in\ f(y) &= f(x) \\ let\ y \Leftarrow c\ in\ val(y) &= c \\ let\ y_2 \Leftarrow (let\ y_1 \Leftarrow e_1\ in\ e_2)\ in\ e_3 &= let\ y_1 \Leftarrow e_1\ in\ let\ y_2 \Leftarrow e_2\ in\ e_3 \end{aligned}$$

Ces axiomes sont motivés par l'interpretation des programmes dans une categorie dite de Kleisli, et permettent de prouver des proprietés telles

$$[\![e_1 + (e_2 + e_3)]\!] = [\![(e_1 + e_2) + e_3]\!]$$

sans specifier la monade (et donc, pour toute sémantique !).
Bien sûr, cela vient du fait que $[\![e_1 + e_2]\!]$ s'écrit comme

$$let\ x \Leftarrow [\![e_1]\!]\ in\ let\ y \Leftarrow [\![e_2]\!]\ in\ val(x + y)$$

3.21 Quelques références

Le sujet est assez récent, mais il à déjà intéressé beaucoup de monde. On pourra trouver des très complets détails sur la théorie sous-jacente aux monades dans [Mog91], et une introduction plutôt orientée à la construction d'interprètes modulaires dans [Ham92, Wad92a, Wad92b, Wad93].

Bibliographie

[AG92] A.Arnold and Irène Guessarian. *Mathèmatiques pour l'Informatique*. Masson, Paris, 1992.

[Cur93] Pierre-Louis Curien. *Categorical combinators, sequential algorithms and functional programming*. Progress in Theoretical Computer Science. Birkha¨user, 1993. second edition.

[DP90] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Orders*. Cambridge Mathematical Textbooks. Cambridge University Press, 1990.

[DS95] Daniel J. Dougherty and Ramesh Subrahmanyam. Equality between functionals in the presence of coproducts. In *Proceedings, Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 282–291, San Diego, California, 26–29 June 1995. IEEE Computer Society Press.

[Fri75] Harvey Friedman. Equality between functionals. In R. Parikh, editor, *?*, pages 22–37, 1975.

[GLT90] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Cambridge University Press, 1990.

[GS92] Carl Gunter and Dana Scott. *Handbook of Theoretical Computer Science*, chapter Semantic Domains, pages 633–674. Elsevier, 1992.

[Gun92] Carl A. Gunter. *Semantics of programming languages : structures and techniques*. The MIT Press, 1992.

[Ham92] K. Hammond. Efficient type inference using monads. In C.K. Holst R. Heldal and P.L. Wadler, editors, *Functional Programming, Glasgow 1991 : Proceedings of the 1991 Workshop, Portree, UK*, pages 146–157, Berlin, DE, 1992. Springer-Verlag.

[Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1) :55–92, July 1991.

[PS82] Gordon D. Plotkin and M.B. Smyth. The category theoretic solution of recursive domain equations. *SIAM Journal of Computing*, 11(4) :761–783, November 1982.

[SAM94] R. Jagadeesan S. Abramsky and P. Malacaria. Full abstraction for pcf. In *Theoretical Aspect of Computer Science (TACS 94)*, LNCS, Sendai, 1994.

[Sch86] David A. Schmidt. *Denotational semantics : a methodology for language development*. Brown Publisher, Iowa, 1986.

[Sto77] Joseph E. Stoy. *Denotational semantics : the Scott - Strachey approach to programming language theory*. MIT Press, 1977.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5 :285–309, 1955.

[Wad92a] P.L. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2 :461–493, 1992.

[Wad92b] P.L. Wadler. The essence of functional programming. In *Proceedings of the 19th Annual ACM Symposium on Principles of Programming Languages, Albuquerque, NM*, pages 1–14, 1992.

[Wad93] P.L. Wadler. Monads and functional programming. In M. Broy, editor, *Marktoberdorf International Summer School on Program Design Calculi*. Springer-Verlag, New York, NY, 1993.

[Win93] Glynn Winskel. *The Formal semantics of programming languages : an introduction*. The MIT Press, 1993.