# Chapter 2

# Sémantique opérationnelle et dénotationnelle d'un langage fonctionnel simple typé

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

## 2.1 Un langage fonctionnel simple typé: TFun

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

Le langage est donné ici en définissant avant tout

- la syntaxe abstraite
- le système de types

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

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

### 2.1.1 Syntaxe abstraite

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

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

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

(BOOLEENS)
B := true | false

(IDENTIFICATEURS)
Id := non specifie'

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

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

### 2.1.2 Typage

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

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

où $\Gamma$ (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 2.1.1** *On demande que chaque identificateur apparaisse une seule fois dans* $\Gamma$.

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

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

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

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

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

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

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

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

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

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

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

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

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

### 2.1.3 Sémantique opérationnelle naturelle de TFun

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

**L'appel par nom:** $\Downarrow_n$

**constantes:**

$$cst \Downarrow_n cst$$

**abstraction fonctionnelle et application:**

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

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

**conditionnel:**

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

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

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

**L'appel par valeur:** $\Downarrow_v$

**constantes:**

$$cst \Downarrow_v cst$$

**abstraction fonctionnelle et application:**

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

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

**conditionnel:**

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

### 2.1.4 Quelques résultats

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

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

*Preuve.* Par induction sur la dérivation. □

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

*Preuve.* Par induction sur les deux dérivations. □

### 2.1.5 Sémantique opérationnelle structurée (SOS) de TFun

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

**L'appel par nom:** $\Rightarrow_n$

**application:**

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

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

**application:**

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

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

**conditionnel:**

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

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

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

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

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

### 2.1.6 Quelques résultats

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

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

*Preuve.*

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

□

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

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

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

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

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

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

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

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

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

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

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

*Preuve.* Utiliser le corollaire précédent. □

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

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

## 2.2 Equivalence Observationnelle

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

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

$$\texttt{fun x:A->x} \Downarrow \texttt{fun x:A->x}$$

On a très envie de identifier `fun x:A->x` et `fun x:A->(fun y:A->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 programmes 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 programmes qui identifie exactement ces (fragments de) programmes "qui font la même chose" du point de vue d'un observateur choisi.

**Définition 2.2.1 (Contexte)** *Un contexte avec un trou, noté $C[\,]$, est un (fragment de) programme où une variable spéciale distinguée $[\,]$ apparait au plus une fois. On note $C[M]$ le (fragment de) programme obtenu en remplaçant $[\,]$ par $M$ textuellement (i.e. , sans faire les $\alpha$-conversions comme dans la substitution du $\lambda$-calcul).*

**Définition 2.2.2 (Equivalence observationnelle)** *Fixée une sémantique opérationnelle $S$, une notion de observation $O_S$, et une notion de programme, deux fragments de programmes $M$ et $N$ sont dit equivalent observationnellement, par rapport à $O_S$ et $S$, ssi:*

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

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

*Dans ce cas, on écrit*

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

**Remarque 2.2.3 (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 programme $P$ termine avec la sémantique $S$*
- *$O_S(P)$ = le programme $P$ de type de base donne comme résultat, avec la sémantique $S$, une constante donnée*

**Remarque 2.2.4** *On va voir plus avant que, par rapport aux notions d'observation $O$ precédentes, et toutes les sémantiques opérationnelles $S$ déjà vues,*

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

### 2.2.1 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 2.2.5** *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 programme $P$ (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 programme).*

- $\Downarrow_2$ **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 2.2.6** *Dans le cas de TFun, toutes les sémantiques que l'on a vu sont bien equivalentes par rapport a tous ces critéres.*

- *par rapport à la terminaison: pas de difference parce-que tout programme termine*

- *si on observe des constantes sur les types de base: là, les valeurs coincident avec les formes normales dans le lambda calcul typé simple, et donc le resultats par nom ou valeur sont les mêmes, d'où le fait que $\Downarrow_v$ et $\Downarrow_n$ genèrent la même equivalence observationnelle si on observe les types de base (et de là on obtient les trois proprietés de plus en haut).*

## 2.3 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.

### 2.3.1 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 2.3.1 (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* $\mathcal{F}_{\texttt{t2}}$

### 2.3.2 Morphismes d'interprétation

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

**T⟦⟧: Types → FTH**

- $T[\![\texttt{int}]\!] = IN$
- $T[\![\texttt{bool}]\!] = \{true, false\}$
- $T[\![\texttt{t1 -> t2}]\!] =$ l'ensemble des fonctions totales entre $T[\![\texttt{t1}]\!]$ et $T[\![\texttt{t2}]\!]$
- $T[\![Exp\texttt{[t]}]\!] = \mathrm{T}[\![\texttt{t}]\!]$
- $T[\![Id\texttt{[t]}]\!] = \mathrm{T}[\![\texttt{t}]\!]$

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

- $G[\![\Gamma]\!] = T[\![\texttt{t1}]\!] \times \cdots \times T[\![\texttt{tn}]\!]$ if $\Gamma = \texttt{x1}: Id\texttt{[t1]}, \cdots, \texttt{xn}: Id\texttt{[tn]}$

**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 $\Gamma$ 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 $a$.

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

**Remarque 2.3.2 (Projections)** *Ici, on assume que $\pi_i^n$ soit défini comme:*

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

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

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

*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.*

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

Voici quelques relations typiques entre syntaxe et sémantique:

**Lemme 2.3.3 (Substitution sémantique)**

$$E[\![\Gamma, \mathtt{x:t1} \mid - \mathtt{M}: Exp\mathtt{[t2]}]\!]_{(e, E[\![\Gamma \mid - \mathtt{N}: Exp\mathtt{[t1]}]\!]_e)} = E[\![\Gamma \mid - \mathtt{M[N/x]} : Exp\mathtt{[t2]}]\!]_e$$

*Preuve.* Longue induction sur la structure de $M$. □

**Lemme 2.3.4 (Indépendance sémantique)** *Si $e$ et $e'$ coïncident sur les variables de $\Gamma$, alors*

$$E[\![\Gamma \mid - \mathtt{M}: Exp\mathtt{[t]}]\!]_e = E[\![\Gamma \mid - \mathtt{M}: Exp\mathtt{[t]}]\!]_{e'}$$

**Notation 2.3.5** *Pour breveté, dans ce qui suit, on va souvent oublier le contexte de typage $\Gamma$, s'il n'est pas relevant dans le contexte. Aussi, on ne va pas toujours distinguer les morphismes d'interprétation: on écrira juste $[\![]\!]$.*

## 2.4 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 2.4.1** *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:*

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

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

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

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

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

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

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

*(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 2.4.2** *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 2.4.3** *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 2.4.4** *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 2.4.5 (Consistance)** *Dans TFun, si* $\Gamma| - \texttt{M}: Exp\texttt{[t]}$ *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:

$$\begin{aligned}
[\![e_1(e_2)]\!](e) &= ([\![e_1]\!](e))([\![e_2]\!](e)) \\
&= ([\![\texttt{fun}\ x : A\texttt{->}e_0]\!](e))([\![v_2]\!](e)) \\
&= ([a \mapsto [\![e_0]\!](e, a)])([\![v_2]\!](e)) \\
&= [\![e_0]\!](e, ([\![v_2]\!](e))) \\
&= [\![e_0[v_2/x]]\!](e) \\
&= [\![v_0]\!](e)
\end{aligned}$$

et on a fini. □

**Exercice 2.4.6** *Compléter la preuve, et montrer le même résultat pour* $\Downarrow_n$ .

**Lemme 2.4.7 (L'égalité dans $[\![]\!]$ est une *congruence*)**
*Si* $[\![\mathtt{M}]\!] = [\![\mathtt{N}]\!]$, *alors* $[\![\mathtt{C[M]}]\!] = [\![\mathtt{C[N]}]\!]$ *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 2.4.8** $[\![]\!]$ *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 $[\![\mathtt{M}]\!] = [\![\mathtt{N}]\!]$. Alors, pour tout contexte $C[\,]$, $[\![\mathtt{C[M]}]\!] = [\![\mathtt{C[N]}]\!]$.

Mais par la consistance, $[\![\mathtt{v}]\!] = [\![\mathtt{C[M]}]\!] = [\![\mathtt{C[N]}]\!] = [\![\mathtt{v'}]\!]$.

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

**Corollaire 2.4.9** *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 2.4.10** *Vérifier que l'équivalence suivante (règle* $\eta$*)*

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

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

**Exercice 2.4.11** *Montrer que, si l'on observe les valeurs aux types supérieurs,* $\eta$ *n'est pas valide.*

### 2.4.1 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 $[\![\mathtt{M}]\!] = n$

($[\![]\!]$ est aussi correcte par rapport à la terminaison, mais pour des raison triviales pour TFun).

**Exercice 2.4.12** *Montrez que* $[\![]\!]$ *n'est pas correcte si on observe des valeurs qui ne sont pas de base.*

### 2.4.2 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 2.4.13** *Prouvez que L'égalité $\beta$ 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 $\eta$:*

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

*et retrouvez le résultat de l'exercice 2.4.10.*

### 2.4.3 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 $\beta$ et $\eta$. Pour voir cela, nous nous appuyons sur des résultats classiques du $\lambda$-calcul:

**Notation 2.4.14** *Nous disons que $M = N$ est prouvable via $\beta\eta$ si $M$ et $N$ sont égaux modulo $\alpha$-conversion dans la théorie engendrée par $\beta$ et $\eta$ et la fermeture par contexte.*

**Théorème 2.4.15 (Friedman's completeness)** *Si $X$ est un ensemble infini, alors:*

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

**Théorème 2.4.16 (Plotkin's completeness)**

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

**Corollaire 2.4.17 (Égalités prouvables dans $\mathcal{F}_{IN}$)** $[\![]\!]$ *dans $\mathcal{F}_{IN}$ identifie seulement les programmes qui sont égaux modulo $\beta\eta$.*

**Remarque 2.4.18** *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.*

## 2.5 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

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

**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 2.5.1**

- *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):*

$$\texttt{(fun x:t->n)loop} \Downarrow_n n \textit{ mais } \texttt{(fun x:t->n)loop} \not\Downarrow_v n$$

**Exercice 2.5.2** *En programmant un peu, montrez que observer les types de base ou la terminaison sur le types de base sont équivalents.*

**Exercice 2.5.3** *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* $\Gamma| -$ `M:`$Exp$`[t]` *pour un quelque type* `t`*)*

### 2.5.1 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:

### 2.5.2 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:

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

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:

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

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

### 2.5.3 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:

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

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

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

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

**Remarque 2.5.4 (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!)*

### 2.5.4 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].

## 2.6 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.

### 2.6.1 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 2.6.1 (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 2.6.2** *Montrez qu'il n'est pas possible de capturer avec un métalangage toutes les fonctions disponibles dans* $\mathcal{F}_{I\!N,\{true,false\}}$.

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).

### 2.6.2 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.

### 2.6.3 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                                      (* identificateurs *)
#
#and  Exp       = INT of int | BOOL of bool                   (* les expressions *)
#              | 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 booleens *)
#         | 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)
>                              ^^^
```

# Chapter 3

# Sémantique dénotationnelle de la recursion

> Dans ce chapitre, nous allons d'abord mettre en evidence les limitations du simple modèle $Set$, qui est inadequat pour representer les constructions recursives generiques disponibles dans les langages de programmation reels, et ensuite presenter la solution traditionnelle a ce probleme, basee sur les domaines de Scott. Sans pretension de traiter le sujet de fac,on comple'te, on va rappeler ici les notions indispensable sur les ordres partiels complets et leurs propietes topologiques. On va aussi montrer comment le metalangage extrait de $Set$ au chapitre precedent peut etre tout aussi bien reinterprete dans $CPO$, ce qui nous permets de recuperer la semantique donnee auparavant, tout en rajoutant les constructions necessaires a traiter la recursion.

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 3.0.3 (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.*

## 3.1 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.

## 3.2 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 $\beta$-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`:

$$\begin{array}{lll} [\![\texttt{while B do C}]\!](\mathrm{s}) = & \text{if } [\![\texttt{B}]\!](\mathrm{s}) = \text{true} \\ & \text{then } [\![\texttt{while B do C}]\!]([\![\texttt{C}]\!](\mathrm{s})) \\ & \text{else s} \end{array}$$

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:

$$\begin{array}{lll} [\![\texttt{while B do C}]\!](\mathrm{s}) = & \mathrm{f} & \\ & \text{where rec f} = & \text{if } [\![\texttt{B}]\!](\mathrm{s}) = \text{true} \\ & & \text{then f}([\![\texttt{C}]\!](\mathrm{s})) \\ & & \text{else s} \end{array}$$

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

## 3.3 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[1].
Cependant, si le langage n'est pas typé[2], 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 $E$ 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.

---

[1] 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.

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

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).

## 3.4 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 3.4.1 (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 3.4.2** *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 3.4.3 (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 3.4.4 (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 3.4.5 (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 3.4.6 (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 3.4.7 (Treilli complet)** *Un treilli $D$ est dit* complet *si tout sous-ensemble $S$ de $D$ admet sup et inf.*

**Remarque 3.4.8** *Tout treilli complet a maximum $\top = sup(D)$ et minimum $\bot = sup(\{\})$.*

**Théorème 3.4.9 (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:

$$\begin{array}{lcl}
\forall x \in X.x \leq a & \Rightarrow & \text{def. sup} \\
\forall x \in X.f(x) \leq f(a) & \Rightarrow & \text{monotonie} \\
\forall x \in X.x \leq f(x) \leq f(a) & \Rightarrow & \text{def. X} \\
a \leq f(a) & \Rightarrow & \text{def. sup} \\
f(a) \leq f(f(a)) & \Rightarrow & \text{monotonie} \\
f(a) \in X & \Rightarrow & \text{def. X} \\
f(a) \leq a & \Rightarrow & \text{def. sup} \\
f(a) = a & &
\end{array}$$

□

**Remarque 3.4.10** *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 3.4.11 (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 3.4.12** *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.*

## 3.5 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 3.5.1 (CPO)** *Un ordre partiel $(D, \leq)$ est dit* complet *si pour toute chaîne $C$ dans $D$ existe $sup(C) \in D$.*

**Définition 3.5.2 (CPO pointé)** *Un Cpo $D$ est dit* pointé *s'il existe dans $D$ un élément minimum, noté $\perp$ (lire "bottom").*

**Définition 3.5.3 (fonction stricte)** *Une fonction $f : A \to B$ entre deux cpos pointés $A$ et $B$ est dite* stricte *si $f(\perp) = \perp$.*

**Notation 3.5.4** *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 3.5.5 (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 3.5.6** *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 3.5.7 (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 3.5.8 (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 3.5.9 (Continuité par composantes)** *Une fonction de $n$ arguments est continue si et seulement si elle l'est par composantes.*

**Théorème 3.5.10 (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 3.5.11 (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 3.5.12** *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 3.5.13 (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 3.5.14** `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 3.5.15 (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 $h(fix(f)) = fix(g)$.*

### 3.5.1 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 3.5.16 (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 3.5.17 (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 3.5.18 (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 $a = b$.*

### 3.5.2 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 3.5.19 (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 3.5.20 (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 3.5.21 (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 3.5.22** *La topologie de Scott est bien une topologie.*

*Preuve.* exercice □

**Définition 3.5.23 (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 3.5.24 (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). □

## 3.6 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.

### 3.6.1 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.

### 3.6.2 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):

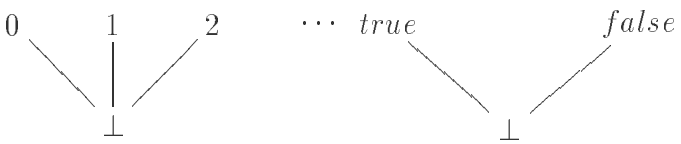
l'inclusion $up : A \to A_\perp$

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

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

### 3.6.3 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 $\perp$.

### 3.6.4 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 3.6.1 (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 3.6.2 (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* $b \in B$*).*

### 3.6.5 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 3.6.3 (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 3.6.4 (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.*

### 3.6.6 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 3.6.5 (continuité)** *La fonction case et les fonctions $in_1, in_2$ sont continues.*

**Remarque 3.6.6 (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

## 3.6.7 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):

- $nil : D^*$
- $hd : D^* \to D_\perp$
- $tl : D \to D^*$
- $cons : D \times D^* \to D^*$

## 3.6.8 One point CPO

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

## 3.6.9 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 3.6.7 (Continuité de *strict*)** *La fonction* strict$:(C \to D) \to (C \circ\!\!\to D)$ *définie comme*

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

*est continue.*

## Isomorphismes

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

**Définition 3.6.8 (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.

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

### 3.6.10 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:$(\mathbf{A} \to \mathbf{B}) \times \mathbf{A} \to \mathbf{B}$** est la fonction $app(f, a) = f(a)$

**$\lambda$x$\in$ A.e:(A$\to$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$\to$B$_\perp$)$\to$(A$_\perp\to$B$_\perp$)** est l'injection entre (A$\to$B$_\perp$) et (A$_\perp\to$B$_\perp$)

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

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

etc.

**Définition 3.6.9 (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 3.6.10 (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 $\pi_i$

**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**

$f[l \mapsto e] = (\lambda l'.\text{if } l = l' \text{ then } e \text{ else } f(l))$
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 3.6.11 (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 3.6.12 (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 3.6.13** *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*

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

## 3.7 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 $\text{TFun}_{loop}$ 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.

### 3.7.1 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)
- $T[\![Exp\texttt{[t]}]\!] = \text{T}[\![\texttt{t}]\!]_\perp$
- $T[\![Id\texttt{[t]}]\!] = \text{T}[\![\texttt{t}]\!]$

**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**

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

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

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 $\lambda a.f(e,a) = \bot \in [\![\texttt{t1}]\!] \to [\![\texttt{t2}]\!]_\bot$, cette fonction indéfinie partout ne peut pas être confondue avec $\bot \in ([\![\texttt{t1}]\!] \to [\![\texttt{t2}]\!]_\bot)_\bot$, qui n'est *pas* un valuer fonctionnel. Dans les chapitres ce qui suivent, on pourra, pour brevité, utiliser ces abréviations.

### 3.7.2 Appel par nom

**T[[]]: Types → CPO**

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

**G[[]]: Environnements → 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 → CPO**

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

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

### 3.7.3 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 $\mathrm{TFun}_{rec}$ de TFun avec des fonctions recursives:

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

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

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**

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

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**

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

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

### 3.7.4 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.