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 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 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 β et η. Pour voir cela, nous nous appuyons sur des résultats classiques du λ-calcul:

Notation 2.4.14 *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 2.4.15 (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 2.4.16 (Plotkin's completeness)

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

Corollaire 2.4.17 (Égalités prouvables dans $\mathcal{F}_{\mathbb{N}}$) *$[\![\,]\!]$ dans $\mathcal{F}_{\mathbb{N}}$ 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 $\Gamma| - \texttt{M: } Exp[\texttt{t}]$ 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:

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

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

- $[\![\Gamma| - \texttt{i:}\ Exp\texttt{[int]}]\!]_e = in_1(i)$ (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 Γ
- $[\![\Gamma| - \texttt{fun x: t1->M:}\ Exp\texttt{[t1->t2]}]\!]_e$ = soit $f \in ([\![\Gamma]\!] \times [\![\texttt{t1}]\!] \rightarrow [\![Exp\texttt{[t2]}]\!])$ l'interprétation $[\![\Gamma, \texttt{x:}\ Id\texttt{[t1]}| - \texttt{M:}\ Exp\texttt{[t2]}]\!]$ 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 Γ
- $[\![\Gamma|-\texttt{fun x: t1->M:}\ Exp\texttt{[t1->t2]}]\!]_e = in_1([a \mapsto f(e,a)])$
  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 : $if\ \_\ then\ e1\ else\ e2 : Bool \rightarrow C$ si $e1 : C$ et $e2 : C$, et aussi $true : Bool$, $false : Bool$

entiers

type : Nat

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

espace de fonctions

type : $A \rightarrow B$

opérations :

- application: $app(\_, \_) : ((A \rightarrow B) \times A) \rightarrow B$
- abstraction: pour toute expression e de type B qui dépend d'une variable libre $x : A$, $\lambda x.e : A \rightarrow B$ 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* $\mathcal{F}_{\mathbb{N},\{true,false\}}$.

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).

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 Caml-Light.

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)
>                             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
^^^^^^^^^^^^
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)
```

Chapitre 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 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`:

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

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