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

**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**

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

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

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 $\lambda a.f(e,a) = \bot \in [\![\mathtt{t1}]\!] \rightarrow [\![\mathtt{t2}]\!]_\bot$, cette fonction indéfinie partout ne peut pas être confondue avec $\bot \in ([\![\mathtt{t1}]\!] \rightarrow [\![\mathtt{t2}]\!]_\bot)_\bot$, 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**

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

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

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

**E$[\![]\!]$: Expression Judgements → CPO**

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

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

### 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

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

pour lequel on peut donner la sémantique operationnelle comme suit:

**CBV: on rajoute les règles**

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

$$\frac{e_1 \Downarrow_v \mathtt{f\ where\ rec\ f(x)} = \mathtt{e_0} \qquad e_2 \Downarrow_v v_2 \qquad e_0[v_2/x; \mathtt{f\ where\ rec\ f(x)} = \mathtt{e_0}/f] \Downarrow_v v_0}{e_1(e_2) \Downarrow_v v_0}$$

**CBN: on rajoute les règles**

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

$$\frac{e_1 \Downarrow_n \mathtt{f\ where\ rec\ f(x)} = \mathtt{e_0} \qquad e_0[e_2/x; \mathtt{f\ where\ rec\ f(x)} = \mathtt{e_0}/f] \Downarrow_n v}{e_1(e_2) \Downarrow_n v}$$

et la sémantique denotationnelle comme suit:

**CBV**

$$\begin{aligned}&[\![\Gamma | - \mathtt{f\ where\ rec\ f(x)\ =\ M:}\ Exp[\mathtt{t1\text{-}\!>t2}]]\!]_e = \\ &\quad up(fix(\lambda h.\lambda a.[\![\Gamma, \mathtt{f:}\ Id[\mathtt{t1\text{-}\!>t2}], \mathtt{x:}\ Id[\mathtt{t1}] | - \mathtt{M:}\ Exp[\mathtt{t2}]]\!](e, h, a)))\end{aligned}$$

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

**CBN**

$$\begin{aligned}&[\![\Gamma | - \mathtt{f\ where\ rec\ f(x)\ =\ M:}\ Exp[\mathtt{t1\text{-}\!>t2}]]\!]_e = \\ &\quad fix(\lambda h.up(\lambda a.[\![\Gamma, \mathtt{f:}\ Id[\mathtt{t1\text{-}\!>t2}], \mathtt{x:}\ Id[\mathtt{t1}] | - \mathtt{M:}\ Exp[\mathtt{t2}]]\!](e, h, a))))\end{aligned}$$

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

### 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.

# Chapitre 4

# Déclarations, état, erreurs, exceptions, commandes, nondeterminisme, parallélisme

Dans ce chapitre, nous allons introduire quelques unes des techniques fondamentales utilisées en sémantique dénotationnelle pour modéliser les différents aspects des langages de programmation, et cela en utilisant le métalangage que l'on a dérivé dans le chapitre précédent pour les CPO's.
On va notamment s'intéresser ici à la modélisation de tous ces aspects *impures* que l'on rencontre dans les langages impératifs ordinaires.

## 4.1 Les déclarations

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

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

peut être vu comme une abbreviation de

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

et aussi, pour les déclarations simultanés, on peu traiter

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

comme une abréviation de

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

ce qui est bien différent de

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

qui est vu comme une abréviation de

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

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

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

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

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

**typage** on rajoute pour tout type $t$ la règle

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

## Sémantique

**appel par valeur**

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

**appel par nom**

$$\begin{aligned}
&[\![\Gamma| - \texttt{let x: t1 = M1 in M2: } Exp\texttt{[t]}]\!]_e = \\
&\quad [\![\Gamma, x : Id[t1]| - \texttt{M2: } Exp\texttt{[t]}]\!](e, [\![\Gamma| - \texttt{M1: } Exp\texttt{[t1]}]\!]_e)
\end{aligned}$$

**Exercice 4.1.1** *Donner la sémantique d'une construction pour la déclaration simultanée:*
`let id1: t1 = M1 and id2: t2 = M2 in M`

## 4.2 TFun$_{err}$ Un langage avec erreurs

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

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

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

- **catégories syntaxiques** on a une nouvelle catégorie syntaxique *Erreurs* de *noms d'erreurs*
- **expressions** on a des nouvelles expressions `error ern`, où `ern` et un nom d'erreur
- **typage** on rajoute pour tout type $t$ la règle

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

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

## Sémantique

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

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

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

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

Où *Err* est un domaine standard défini comme suit:

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

Voici la sémantique des types (pour le langage sans non-terminaison) :

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

et des nouvelles catégories syntaxiques et expressions:

- noms d'erreur:

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

- génération d'un erreur

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

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

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

$$\begin{array}{l}\texttt{case(f(e),}\lambda\ \texttt{f'}\in[\![\texttt{t1->t2}]\!]\texttt{.}\\ \qquad\qquad\qquad\texttt{case(g(e),}\lambda\ \texttt{v'}\in[\![\texttt{t1}]\!]\texttt{. (f' v')}\\ \qquad\qquad\qquad\qquad\qquad\texttt{,}\lambda\ \texttt{n'}\in\ \texttt{Err. } in_2(n'))\\ \qquad\quad\texttt{,}\lambda\ \texttt{n}\in\ \texttt{Err. } in_2(n))\end{array}$$

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

## Erreurs et nonterminaison

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

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

et pour les nouvelles catégories syntaxiques et expressions:

- noms d'erreur:

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

- génération d'un erreur

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

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

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

```
let [h] = f(e)
in case(h,λ f'∈⟦t1->t2⟧.let [v] = g(e)
                         in case(v,λ v'∈⟦t1⟧. f'(v')
                                  ,λ n'∈ Err. in₂(n'))
         ,λ n∈ Err. in₂(n))
```

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

- et pour la recursion

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

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

**Exercice 4.2.1 (CBN)** *Adapter la sémantique de TFun$_{rec}$ en CBN aux erreurs, analoguement à ce que l'on vient de faire pour la CBV ici.*

**Remarque 4.2.2 (Manque de modularité)** *Ici on voit bien que, même si l'approche dénotationnel direct classique est assez élégant et concis si on doit donner la sémantique d'un langage figé, il est par contre très décevant quand il s'agit de modifier une sémantique déjà donnée. Il faudra faire mieux que cela si on veut pouvoir étudier la sémantique de façon modulaire. En effet, en se basant sur des notions catégoriques, Eugenio Moggi à ouvert la voie à un approche de la sémantique dénotationnelle qui suit les lignes du génie logiciel.*

## 4.3 TFun$_{exc}$ Un langage avec exceptions

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

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

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

**typage** on rajoute pour tout type $t$ la règle

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

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

### Sémantique

On rajoute à la sémantique de $\text{TFun}_{err}$ la clause suivante pour `handle(ern,M,N)`:

- $[\![\Gamma| - \texttt{handle(ern,M,N):}\ Exp\texttt{[t]}]\!]_e =$

$$
\begin{array}{l}
\texttt{let [m] = f(e)} \\
\texttt{in case(m,}\lambda\ \texttt{m'}\in[\![\texttt{t}]\!]\texttt{.up(}in_1(m')\texttt{)}, \\
\qquad\qquad \lambda\ \texttt{ern'}\in\ \texttt{Err. if ern' = ern} \\
\qquad\qquad\qquad\qquad\quad \texttt{then g(e)} \\
\qquad\qquad\qquad\qquad\quad \texttt{else up(}in_2(ern')\texttt{))}
\end{array}
$$

## 4.4 $\text{TFun}_{ref}$ Un langage avec état

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

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

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

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

$$
\frac{\Gamma| - \texttt{M:}\ Exp\texttt{[t]} \quad \texttt{t}\ \text{mémorisable}}{\Gamma| - \texttt{ref M:}\ Exp\texttt{[t ref]}}
\qquad
\frac{\Gamma| - \texttt{M:}\ Exp\texttt{[t ref]}}{\Gamma| - \texttt{! M:}\ Exp\texttt{[t]}}
$$

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

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

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

## Sémantique

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

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

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

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

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

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

**Définition 4.4.1 (Manipulation de la mémoire (store) S)** *On définit les suivantes fonctions de manipulation de la mémoire (store):*

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

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

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

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

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

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

**Remarque 4.4.2** *Le domaine Sv est en general une somme de domaines, et alors dans get et alloc-init on doit rajouter le necessaire $in_D$ pour indiquer que v est dans le domaine D.*

Voici comment changer la sémantique des types

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

et des nouvelles expressions:

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

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

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

$$\begin{array}{l} E[\![\Gamma| - \texttt{! M: } Exp\texttt{[int]}]\!]\_e \;=\; \lambda\ s \in\ State. \\ \quad \texttt{let (l,s') = } E[\![\Gamma| - \texttt{M: } Exp\texttt{[int ref]}]\!](e)(s) \\ \quad \texttt{in (get(s')(l),s')} \end{array}$$

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

$$\begin{array}{l} E[\![\Gamma| - \mathtt{id:\ =M:}\ Exp\mathtt{[unit]}]\!]_e = \lambda s \in State. \\ \quad \mathtt{let}\ (v, s') = E[\![\Gamma| - \mathtt{M:}\ Exp\mathtt{[t]}]\!](e)(s) \\ \quad \mathtt{in}\ (\bot_{\mathbb{1}}, \mathtt{set}(s')(v)(\pi_i(e))) \\ (\mathtt{id}\text{ le } i\text{-ème identificateur dans } \Gamma) \end{array}$$

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

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

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

### 4.4.1 État et nonterminaison ensemble

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

$$E[\![]\!] : \text{Expression Judgements} \to Env \to State \to (CPO \times State)_\bot$$

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

- $T[\![Exp\mathtt{[t]}]\!] = \text{State} \to (\text{T}[\![\mathtt{t}]\!] \times \text{State})_\bot$

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

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

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

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

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

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

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

$$\begin{array}{l} E[\![\Gamma| - \texttt{id: =M: } Exp\texttt{[unit]}]\!]_e = \lambda s \in State. \\ \quad \texttt{let } [(v, s')] = E[\![\Gamma| - \texttt{M: } Exp\texttt{[t]}]\!](e)(s) \\ \quad \texttt{in } up((\bot_{\mathbb{1}}, \texttt{set}(s')(v)(\pi_i(e)))) \\ \quad (\texttt{id} \text{ le } i\text{-ème identificateur dans } \Gamma) \end{array}$$

- $[\![\Gamma| - \texttt{i: } Exp\texttt{[int]}]\!]_e = \lambda s.up((i, s))$ (pour `i` constante entière)

- $[\![\Gamma| - \texttt{true: } Exp\texttt{[bool]}]\!]_e = \lambda s.up((true, s))$,

- $[\![\Gamma| - \texttt{false: } Exp\texttt{[bool]}]\!]_e = \lambda s.up((false, s))$,

- $[\![\Gamma| - \texttt{loop: } Exp\texttt{[t]}]\!]_e = \lambda s.(\bot \in ([\![\texttt{t}]\!] \times S)_\bot) \equiv \bot \in [\![Exp\texttt{[t]}]\!]$,

- $[\![\Gamma| - \texttt{id: } Exp\texttt{[t]}]\!]_e = \lambda s.up((\pi_i(e), s))$ (`id` le $i$-ème identificateur dans $\Gamma$)

- $[\![\Gamma| - \texttt{fun x: t1->M: } Exp\texttt{[t1->t2]}]\!]_e = \lambda s.up((\lambda a.\lambda s'.f(e, a)(s'), s))$
  où $f \in ([\![\Gamma]\!] \times [\![\texttt{t1}]\!] \rightarrow [\![Exp\texttt{[t2]}]\!])$ est l'interprétation $[\![\Gamma, \texttt{x: } Id\texttt{[t1]}| - \texttt{M: } Exp\texttt{[t2]}]\!]$.
  Au fait, $\lambda s'.f(e, a)(s') = f(e, a)$, mais on a donné la version plus redondante pour clarté.

- $[\![\Gamma| - \texttt{M N: } Exp\texttt{[t]}]\!]_e =$
  $\lambda s.\ \texttt{let } [(f', s')] = f(e)(s)\ \texttt{in let } [(v, s'')] = g(e)(s')\ \texttt{in } f'(v)(s'')$

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

- et pour la recursion

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

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

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

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

**Exercice 4.4.3 (CBN)** *Adapter la sémantique de TFun$_{rec}$ en CBN aux références, analoguement à ce que l'on vient de faire pour la CBV ici.*

**Exercice 4.4.4 (Mémoire et S)** *On a dans notre approche fait une assumption très simpliste sur les états $s : Loc \rightarrow Sv$: en effet, on a supposé qu'ils sachent renvoyer toujours un valeur pour toute location, même une location qui n'a pas été encore initialisée, ce qui n'est raisonnable que si toute location contient le même type de valeur. Modifiez la définition des domaines sémantiques pour que l'on puisse renvoyer un erreur sur les cases non initialisées, et modifiez la sémantique du langage en consequence. Discutez l'importance des modifications nécéssaires.*

**Remarque 4.4.5 (Manque de modularité)** *Même remarque que pour les erreurs.*

## 4.5 Un langage avec commandes

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

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

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

**typage** on rajoute les règles

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

pour typer la séquentialisation de deux commandes

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

pour typer une boucle *while*

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

pour typer une boucle *repeat*

## Sémantique

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

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

- $[\![\Gamma|-\texttt{c1; c2: }Exp\texttt{[unit]}]\!]_e = \lambda s\in S.$

```
let [(v,s')] = [[Γ|- c1: Exp[unit]]](e)(s)
in [[Γ|- c2: Exp[unit]]](e)(s')
```

- $[\![\Gamma|-\texttt{while b do c: }Exp\texttt{[unit]}]\!]_e =$

```
fix(λ h∈ S→(𝟙× S)⊥.λ s∈ S.
    let [(v,s')] = [[Γ|- b: Exp[bool]]](e)(s)
    in if v then let [(v',s'')] = [[Γ|- c: Exp[unit]]](e)(s')
                 in h(s'')
            else (⊥𝟙,s'))
```

- $[\![\Gamma|-\texttt{repeat c until b: }Exp\texttt{[unit]}]\!]_e =$

```
fix(λ h∈ S→(𝟙× S)⊥.λ s∈ S.
    let [(v,s')] = [[Γ|- c: Exp[unit]]](e)(s)
    in let [(v',s'')] = [[Γ|- b: Exp[bool]]](e)(s')
       in  if v' then (⊥𝟙,s''))
                 else h(s'')
```

**Exercice 4.5.1** *Montrez que, pour tout commande* `c` *et expression booléenne* `b` *on peut prouver l'egalité suivante.*

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

**Exercice 4.5.2 (Sémantique opérationnelle)** *Reflechissez à une sémantique opérationnelle pour TFunref avec commandes, dans le but d'arriver à prouver la correction de la sémantique dénotationnelle que l'on vient de donner par rapport à l'equivalence observationnelle induite par l'observation des valeurs de base.*