4.6 Sémantique avec continuations

Dans la plupart des langages de programmation modernes il y a des mécanismes dont le but est de permettre de modifier l'ordre normale d'execution des instructions d'un programme. Cela n'a pas (ou plus) le but d'utiliser systematiquement des *goto* la où des constructions plus propres tels les boucles *while* sont appropriées (et cela depuis le travail fondamentale de Böhm et Iacopini), mais plutôt de permettre de traiter de façon efficace des situations exceptionnelles (tels les erreurs et les exceptions).

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

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

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

Sémantique

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

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

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

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

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

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

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

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

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

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

Remarque 4.6.1 *Pour les expressions de type de base, on a souvent le resultat recherché si on évalue la sémantique sur la continuation identité $\lambda x.x$. Que faut-il faire si on a des expressions de type non de base?*

Remarque 4.6.2 *Ici on a pris, informellement, $R = CPO$, mais dans les langages imperatifs, R est souvent juste l'état.*

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

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

Exemple 4.6.3 Voyons maintenant sur un exemple le développement complet de la sémantique avec continuations (on va marquer les redexes que l'on réduit en soulignant la fonction que l'on va appliquer à chaque fois).

$$
\begin{array}{rl}
\multicolumn{2}{l}{\llbracket \Gamma | - \texttt{(fun x: int->->x) 1: } Exp\texttt{[int]} \rrbracket_e =} \\
= & \lambda k_1.\llbracket \Gamma | - \texttt{fun x: int->->x: } Exp\texttt{[int -> int]} \rrbracket_e (\lambda h.\llbracket \Gamma | - \texttt{1: } Exp\texttt{[int]} \rrbracket_e (\lambda v.(hv)k_1)) \\
= & \lambda k_1.(\lambda k_2.k_2(\lambda a.\llbracket \Gamma, x : Id[int] | - \texttt{x: } Exp\texttt{[int]} \rrbracket_{(}(e,a)))(\lambda h.\llbracket \Gamma | - \texttt{1: } Exp\texttt{[int]} \rrbracket_e (\lambda v.(hv)k_1)) \\
= & \lambda k_1.\underline{(\lambda k_2.k_2(\lambda a.(\lambda k_3.k_3(a))))}(\lambda h.(\lambda k_4.k_4(1))(\lambda v.(hv)k_1)) \\
= & \lambda k_1.\underline{(\lambda h.(\lambda k_4.k_4(1))(\lambda v.(hv)k_1))}(\lambda a.(\lambda k_3.k_3(a))) \\
= & \lambda k_1.\underline{(\lambda k_4.k_4(1))}(\lambda v.((\lambda a.(\lambda k_3.k_3(a)))v)k_1) \\
= & \lambda k_1.\underline{(\lambda v.((\lambda a.(\lambda k_3.k_3(a)))v)k_1)}1 \\
= & \lambda k_1.(\underline{(\lambda a.(\lambda k_3.k_3(a)))}1)k_1 \\
= & \lambda k_1.\underline{(\lambda k_3.k_3(1))}k_1 \\
= & \lambda k_1.k_1(1)
\end{array}
$$

Donc, l'effet de `(fun x: int->->x) 1` est de passer au "reste du programme" k_1 la valeur `1`. □

4.7 Sémantique des *commandes* avec continuations

Voyons maintenant comment on peut donner la sémantique d'un langages avec commandes en utilisant la technique des continuations. Observons d'abord que, comme on veut traiter des commandes tel le `while` qui peuvent ne pas terminer, on sera obliges de choisir des domaines pointés. Ensuite, fixons un domaine R des *résultats* (qui va en réalité être tout CPO si on permet les fonctions dans le langage). Finalement, on pourra définir l'interprétation d'une expression de type `t` avec un morphisme sémantique

$$E\llbracket \_\texttt{: } Exp\texttt{[t]} \rrbracket : \text{Expression Jugements de type t} \rightarrow Env \rightarrow S \rightarrow (\llbracket \texttt{t} \rrbracket \times S \rightarrow R_\perp) \rightarrow R_\perp$$

Donc, l'interprétation sémantique d'une expression sera une fonction qui nous donne un résultat des que l'on lui fournit l'état courant et une continuation $k : (X \times S \rightarrow R_\perp)$ qui représente le reste du programme (cette fois, le reste du programme n'a pas seulement besoin d'un valeur $v \in \llbracket \texttt{t} \rrbracket$ comme dans le cas fonctionnel pur, mais aussi d'un état $s \in S$).

Une fois cela fixé, on peut donner la sémantique du langage comme suit:

- $\llbracket \Gamma | - \texttt{i: } Exp\texttt{[int]} \rrbracket_e = \lambda s \in S.\lambda k.k(i,s)$ (pour `i` constante entière)
- $\llbracket \Gamma | - \texttt{true: } Exp\texttt{[bool]} \rrbracket_e = \lambda s \in S.\lambda k.k(true,s)$,

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

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

- $[\![\Gamma| - \texttt{fun x: t1->M: } Exp\texttt{[t1->t2]}]\!]_e = \lambda s \in S.\lambda k.k(\lambda a.f(e, a), s)$
  où $f \in ([\![\Gamma]\!] \times [\![\texttt{t1}]\!] \to [\![Exp\texttt{[t2]}]\!])$ est l'interprétation $[\![\Gamma, \texttt{x: } Id\texttt{[t1]}| - \texttt{M: } Exp\texttt{[t2]}]\!]$

- $[\![\Gamma| - \texttt{M N: } Exp\texttt{[t]}]\!]_e =$
  $\lambda s \in S.\lambda k.[\![\Gamma| - \texttt{M: } Exp\texttt{[t1->t]}]\!](e)(s)(\lambda(f, s').[\![\Gamma| - \texttt{N: } Exp\texttt{[t1]}]\!](e)(\lambda(v, s'').(fv)s''))$.

- $[\![\Gamma| - \texttt{c1; c2: } Exp\texttt{[unit]}]\!]_e =$
  $\lambda s\in S.\lambda k.[\![\Gamma| - \texttt{c1: } Exp\texttt{[unit]}]\!](e)(s)(\lambda(v, s').[\![\Gamma| - \texttt{c2: } Exp\texttt{[unit]}]\!](e)(s')(k))$

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

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

Remarque 4.7.1 (Notation) *Dans la literature, une fonction dans* $(X \times S \to R_\perp)$ *est appelée souvent une continuation d'expression, vu qu'elle représente bien ce qui reste à faire après une expression: on prend l'état modifié et un valeur (celui produit par l'expression).*
Par contre, on trouvera aussi des continuations de commandes, qui sont des fonctions dans $(S \to R_\perp)$*, qui correspondent à ce qui reste à faire après un commande: il n'y a pas de valeurs produits... En effet, dans notre cas, on a tout fait avec des continuations d'expressions, en faisant produire aux commandes des valeurs inintéressants (c'est* $\perp_{\mathbb{1}}$*) que l'on jette ensuite systématiquement. Mais on peut aussi distinguer la catégorie syntaxique des commandes et les traiter à l'aide de continuations de commandes explicitement avec une fonction sémantique spéciale*

$$C[\![\texttt{cmd}]\!] : \to Env \to S \to (S \to R_\perp) \to R_\perp$$

Finalement, on trouvera souvent aussi la notation Ans (de l'anglais Answer, réponse) pour le domaine R des résultats.

Notation 4.7.2 *On va utiliser les abréviations suivantes:*

- $Ec = X \times S \to R_\perp$

- $Cc = S \to R_\perp$

4.8 Sémantique des sauts

La technique des continuations est suffisamment puissante pour pouvoir traiter aisément aussi les sauts non structurés, tels ceux permis par les `goto` dans `C` ou `Pascal`. Nous verrons cela sur une extension de notre langage précédent qui rajoute les expressions suivantes:

syntaxe de TFun$_{goto}$ on étende celle de la section précédente comme suit:

- **catégories syntaxiques** on a une nouvelle catégorie syntaxique *Etiquettes* de *noms d'étiquettes*
- **expressions** on a des nouvelles expressions `L: M`, `block M` et `goto L` où `L` et un nom d'étiquettes
- **typage** on rajoute les règles

$$\frac{\mathtt{L} \in \mathit{Etiquettes}}{\Gamma| - \mathtt{goto\ L:}\ \mathit{Exp}[\mathtt{unit}]}$$

$$\frac{\Gamma| - \mathtt{M:}\ \mathit{Exp}[\mathtt{t}]}{\Gamma| - \mathtt{block\ M:}\ \mathit{Exp}[\mathtt{t}]}$$

$$\frac{\mathtt{L} \in \mathit{Etiquettes} \qquad \Gamma| - \mathtt{M:}\ \mathit{Exp}[\mathtt{t}]}{\Gamma| - \mathtt{L:\ M:}\ \mathit{Exp}[\mathtt{t}]}$$

- **domaines** On rajoute un domaine *Et* plat de noms d'étiquettes avec égalité continue

L'intuition étant que `block M` délimite un bloque où l'on peut effectuer des sauts vers des étiquettes dont la portée est limitée à `M`, tandis que `goto L` continue l'exécution à l'endroit du programme où l'étiquette `L` dans la portée de la commande est définie, et `L: M` déclare une étiquette associée à la commande `M`.

Observons tout d'abord que pour traiter des sauts, il faut:

- définir la *portée* des étiquettes, ce qui nous permets de savoir vers où on peut sauter
  Cela se fait avec une fonction auxiliaire $J[\![]\!]$
- calculer les *continuations* associées aux étiquettes
  Cela se fait en partie dans $J[\![]\!]$ et en partie dans $E[\![]\!]$ en utilisant $J[\![]\!]$ et les points fixes, et dans notre cas, vu que l'on ne permettra pas de sauter *dans* des expressions,on utilisera juste des continuations de commandes dans Cc

- mémoriser les continuations associées aux étiquettes dans un environnement d'etiquettes: cela nous force à rajouter un paramètre supplémentaire

$$g \in (Et \rightarrow Cc)_\perp \quad = \quad EtEnv$$

  à $E[\![]\!]$ pour cet environnement d'etiquettes.

On va donc travailler avec les fonctions

$$\begin{array}{rcl} J[\![\_\!:\ Exp[\mathtt{t}]]\!] & : & \text{Exp. Judg.}\rightarrow Env\rightarrow EtEnv\rightarrow Ec\rightarrow EtEnv \\ E[\![\_\!:\ Exp[\mathtt{t}]]\!] & : & \text{Exp. Judg.}\rightarrow Env\rightarrow EtEnv\rightarrow S\rightarrow Ec\rightarrow R_\perp \end{array}$$

4.8.1 La fonction $J[\![]\!]$

C'est elle qui nous dit vers où on peut sauter: si $J[\![\mathtt{M}]\!] = \emptyset$, on ne peut pas sauter *dans* la commande `M`, vu que aucune étiquette faisant mention de quelque chose apparaissant dans `M` sera visible hors de `M`.

En même temps, on commence à calculer des continuations associées aux commandes (on ne note pas, pour breveté, les nécessaires et $lift$ nécessaires pour opérer sur $ETEnv$).

- $J[\![\Gamma| - \mathtt{i}\!:\ Exp[\mathtt{int}]]\!](e)(g)(k) = \emptyset$ (pour `i` constante entière)
- $J[\![\Gamma| - \mathtt{true}\!:\ Exp[\mathtt{bool}]]\!](e)(g)(k) = \emptyset$,
- $J[\![\Gamma| - \mathtt{false}\!:\ Exp[\mathtt{bool}]]\!](e)(g)(k) = \emptyset$,
- $J[\![\Gamma| - \mathtt{id}\!:\ Exp[\mathtt{t}]]\!](e)(g)(k) = \emptyset$,
- $J[\![\Gamma| - \mathtt{fun\ x\!:\ t1\text{->}M}\!:\ Exp[\mathtt{t1\text{->}t2}]]\!](e)(g)(k) = \emptyset$ (on ne saute pas dans une fonction)
- $J[\![\Gamma| - \mathtt{M\ N}\!:\ Exp[\mathtt{t}]]\!](e)(g)(k) = \emptyset$ (on ne saute pas dans une application de fonction)
- $J[\![\Gamma| - \mathtt{c1;\ c2}\!:\ Exp[\mathtt{unit}]]\!](e)(g)(k) =$
  $J[\![\mathtt{c1}]\!](e)(g)(\lambda s.E[\![\mathtt{c2}]\!](e)(g)(s)(k))[J[\![\mathtt{c2}]\!](e)(g)(k)]$
  Ici, l'environnement d'etiquettes pour `c2` vient modifier celui pour `c1` (c'est ce que la notation _[_] utilisée ici signifie), et cela donne priorité à la *dernière* étiquette au cas où il y en aie plusieurs du même nom.
- $J[\![\Gamma| - \mathtt{while\ b\ do\ c}\!:\ Exp[\mathtt{unit}]]\!](e)(g)(k) =$
  $J[\![\Gamma| - \mathtt{c}\!:\ Exp[\mathtt{unit}]]\!](e)(g)(\lambda s.E[\![\mathtt{while\ b\ do\ c}]\!](e)(g)(s)(k))$
  les étiquettes d'un while sont celles de son corps (on ne saute pas dans les

expressions, donc `b` n'est pas utilisé), mais la continuation doit prendre en compte le fait que la boucle doit être exécutée avant de pouvoir terminer le programme.

- $J[\![\Gamma| - \texttt{L: M: } Exp\texttt{[t]}]\!](e)(g)(k) =$
  $J[\![\Gamma| - \texttt{M: } Exp\texttt{[t]}]\!](e)(g)(k)[L \mapsto \lambda s.E[\![\texttt{M}]\!](e)(g)(s)(k)]$ les étiquettes de `L: M` sont celles de `M`, *plus* `L` qui est associée à `M`.

- $J[\![\Gamma| - \texttt{goto L: } Exp\texttt{[unit]}]\!](e)(g)(k) = \emptyset$

4.8.2 La fonction $E[\![]\!]$

Ici, on doit prendre en compte le paramètre auxiliaire $g \in EtEnv$, qui sert seulement pour les sauts. donc, on ne fait que le passer dans les vieilles expressions.

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

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

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

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

- $[\![\Gamma| - \texttt{fun x: t1->M: } Exp\texttt{[t1->t2]}]\!]_e = \lambda g.\lambda s \in S.\lambda k.k(\lambda a.f(e, a)(g), s)$
  où $f \in ([\![\Gamma]\!] \times [\![\texttt{t1}]\!] \to [\![Exp\texttt{[t2]}]\!])$ est l'interprétation $[\![\Gamma, \texttt{x: } Id\texttt{[t1]}| - \texttt{M: } Exp\texttt{[t2]}]\!]$

- $[\![\Gamma| - \texttt{M N: } Exp\texttt{[t]}]\!]_e = \lambda g.\lambda s \in S.\lambda k.$
  $[\![\Gamma| - \texttt{M: } Exp\texttt{[t1->t]}]\!](e)(s)(\lambda(f, s').[\![\Gamma| - \texttt{N: } Exp\texttt{[t1]}]\!](e)(g)(\lambda(v, s'').(fv)s''))$.

- $[\![\Gamma| - \texttt{c1; c2: } Exp\texttt{[unit]}]\!]_e = \lambda g.\lambda s \in S.\lambda k.$
  $[\![\Gamma| - \texttt{c1: } Exp\texttt{[unit]}]\!](e)(g)(s)(\lambda(v, s').[\![\Gamma| - \texttt{c2: } Exp\texttt{[unit]}]\!](e)(g)(s')(k))$

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

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

Et on l'utilise pour les sauts:

- $[\![\Gamma|-\texttt{goto L: } \mathit{Exp}\texttt{[unit]}]\!]_e = \lambda g.\lambda s \in S.\lambda k.(g(L))(s)$
  pour le `goto L`, c'est facile: on regarde dans l'environnement d'etiquettes quelle est la continuation associée à `L` et on continue par là en lui passant l'état courant.

- $[\![\Gamma|-\texttt{L: M: } \mathit{Exp}\texttt{[t]}]\!]_e = E[\![\texttt{M}]\!]_e$
  une étiquette ne change pas la sémantique d'un commande: ell'est ignorée

- $[\![\Gamma|-\texttt{block M: } \mathit{Exp}\texttt{[t]}]\!]_e = \lambda g.\lambda s \in S.\lambda k.E[\![\texttt{M}]\!](e)(g')(s)(k)$
  où $g' = J[\![\texttt{M}]\!](e)(g[g'])(k)$ (ou aussi $g' = fix(\lambda g''.J[\![\texttt{M}]\!](e)(g[g''])(k)))$
  Donc, l'environnement d'etiquettes qui est mis en place par un `block M` est *récursif*!

Exemple 4.8.1 Pour se convaincre de la nécessité de la récursion dans la définition, on peut d'abord remarquer que avec les `goto`'s on peut bien créer des boucles, comme dans:

```
block (L: goto L)
```

Ensuite, calculons la sémantique de ce programme qui boucle:

$$
\begin{aligned}
&E[\![\texttt{block (L: goto L)}]\!](e)(g)(s)(k) = \\
&= E[\![\texttt{L: goto L}]\!](e)(g')(s)(k) \text{ où } g' = J[\![\texttt{L: goto L}]\!](e)(g[g'])(k) \\
&= E[\![\texttt{goto L}]\!](e)(g')(s)(k) \text{ où } g' = J[\![\texttt{L: goto L}]\!](e)(g[g'])(k) \\
&= g'(L)(s) \text{ où } g' = J[\![\texttt{L: goto L}]\!](e)(g[g'])(k) \\
&= g'(L)(s) \text{ où } g' = (J[\![\texttt{goto L}]\!](e)(g[g'])(k))[L \mapsto \lambda s.E[\![\texttt{goto L}]\!](e)(g[g'])(k)] \\
&= g'(L)(s) \text{ où } g' = [L \mapsto \lambda s.E[\![\texttt{goto L}]\!](e)(g[g'])(k)] (\text{parce-que } J[\![\texttt{goto L}]\!] = \emptyset) \\
&= E[\![\texttt{goto L}]\!](e)(g[g'])(k)] \text{ où } g' = [L \mapsto \lambda s.E[\![\texttt{goto L}]\!](e)(g[g'])(k)]
\end{aligned}
$$

Ici, on a bien un programme qui boucle: la récursion implicite dans la boucle `block (L: goto L)` a été poussée *dans* l'environnement d'etiquettes, qui est devenu, lui récursif.

Si par contre on avait donnée la définition naïve

$$[\![\Gamma|-\texttt{block M: } \mathit{Exp}\texttt{[t]}]\!]_e = \lambda g.\lambda s \in S.\lambda k.E[\![\texttt{M}]\!](e)(g')(s)(k) \text{ où } g' = J[\![\texttt{M}]\!](e)(g[g'])(k)$$

on se serait retrouvés avec

$$
\begin{aligned}
&E[\![\texttt{block (L: goto L)}]\!](e)(g)(s)(k) = \\
&= E[\![\texttt{L: goto L}]\!](e)(g')(s)(k) \text{ où } g' = J[\![\texttt{L: goto L}]\!](e)(g)(k)
\end{aligned}
$$

$$
\begin{aligned}
&= E[\![\texttt{goto L}]\!](e)(g')(s)(k) \text{ où } g' = J[\![\texttt{L: goto L}]\!](e)(g)(k)\\
&= g'(L)(s) \text{ où } g' = J[\![\texttt{L: goto L}]\!](e)(g)(k)\\
&= g'(L)(s) \text{ où } g' = (J[\![\texttt{goto L}]\!](e)(g)(k))[L \mapsto \lambda s.E[\![\texttt{goto L}]\!](e)(g)(k)]\\
&= g'(L)(s) \text{ où } g' = [L \mapsto \lambda s.E[\![\texttt{goto L}]\!](e)(g)(k)](\text{parce-que } J[\![\texttt{goto L}]\!] = \emptyset)\\
&= E[\![\texttt{goto L}]\!](e)(g)(k)\\
&= g(L)(s)
\end{aligned}
$$

et la, on peut sortir de la boucle si `L` était déjà définie ailleurs! □

4.9 Les exceptions avec les continuations

Avec le continuations, il y a un moyen tout à fait simple de traiter les exceptions: il suffit d'utiliser encore une fois un environnement $ErrEnv = Error \rightarrow (Cc + \mathbb{1})$, mais en stoquant des noms d'erreurs plutôt que des étiquettes. Ensuite, `error ern` continuera sur la continuation associée à `ern`, si disponible, tandis que `handle(ern,M,N)` est chargé de mettre en place les bonnes continuations dans l'environnement.

On aura alors:

- $[\![\Gamma| - \texttt{handle(ern,M,N):}\ Exp\texttt{[t]}]\!]_e = \lambda g.\lambda s.\lambda k.$
  $[\![\Gamma| - \texttt{M:}\ Exp\texttt{[t]}]\!](e)(g[ern \mapsto [\![\Gamma| - \texttt{N:}\ Exp\texttt{[t]}]\!](e)(g)(s)(k)])(s)(k)$
- $[\![\Gamma| - \texttt{error ern:}\ Exp\texttt{[t]}]\!]_e = \lambda g.\lambda s.\lambda k.g(ern)$

Pour être plus précis, il faudrait utiliser in_1, in_2 et tester dans la sémantique de `error ern` si la valeur de $g(ern)$ est bien définie... sinon, on renvoie directement l'erreur ern au toplevel.

4.10 Quelques lectures

L'approche à la sémantique dénotationnelle basé sur les continuations est exposé très en détail dans [Sch86], qui est un livre pas trop vieux sur le sujet, mais un approche plus rapide sans pretensions mathèmatiques est aussi donné dans [Gor79], qui reste encore très lisible aujourd'hui.

Chapitre 5

De la composition de Kleisli à la sémantique via monades

Dans ce chapitre, nous allons decouvrir que la façon de donner la sémantique que l'on a utilisé jusque là peut se voir comme une manipulation de morphismes dans la catégorie de Kleisli. Cela est équivalent à travailler avec des foncteurs particuliers qu'on appelle *monades*, et qui nous permettrons de resoudre le problème de manque de modularité rencontré au dernier chapitre. On peut penser à ça comme à des types de donnée abstraits qui encapsulent les specificités de chaque notion de computation, et on pourra donc réaliser des extensions en redeffinissant juste le type de donné abstrait, i.e. les monades.

5.1 Une pause de reflexion

Maintenant que l'on a vu comment traiter a peu près tous les aspects des langages de programmation (sequentiels), nous pouvons marquer une pause de reflexion et regarder la tête que nos définitions sémantiques ont pris. On a déjà vu qu'il y avait un manque de modularité, mais voyons cela mieux maintenant sur deux cas significatifs: comment change la sémantique d'une *constante*, et comment change la sémantique d'une application de fonction.

5.1.1 Sémantique d'une constante entière

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

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

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

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

5.1.2 Sémantique de l'application de fonction

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

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

$$Lift : (\llbracket \mathtt{t} \rrbracket \to \llbracket Exp\mathtt{[t']} \rrbracket) \to (\llbracket Exp\mathtt{[t]} \rrbracket \to \llbracket Exp\mathtt{[t']} \rrbracket)$$

5.1.3 La sémantique modulaire

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

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

Notation 5.1.1 *On écrira* `let x ⇐ c in e` *pour* $(Lift(\lambda x.e))c$*.*

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

- $\llbracket \mathtt{Expt} \rrbracket = T(\llbracket \mathtt{t} \rrbracket)$
- $\llbracket \Gamma | - \mathtt{i:}\ Exp\mathtt{[int]} \rrbracket_e = val(i)$
- $\llbracket \Gamma | - \mathtt{M\ N:}\ Exp\mathtt{[t]} \rrbracket_e =$
  $let\ f \Leftarrow \llbracket \Gamma | - \mathtt{M:}\ Exp\mathtt{[t->t1]} \rrbracket\ in\ (let\ v \Leftarrow (\llbracket \Gamma | - \mathtt{N:}\ Exp\mathtt{[t1]} \rrbracket)\ in\ f v)$ (CBV)
- $\llbracket \Gamma | - \mathtt{M\ N:}\ Exp\mathtt{[t]} \rrbracket_e =$
  $let\ f \Leftarrow \llbracket \Gamma | - \mathtt{M:}\ Exp\mathtt{[t->t1]} \rrbracket\ in\ f(\llbracket \Gamma | - \mathtt{N:}\ Exp\mathtt{[t1]} \rrbracket)$ (CBN)

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

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

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

5.2 Axiomes des monades

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

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

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

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

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

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

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

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

5.3 Quelques références

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

Bibliographie

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

[Ame83] American national standard institute. *The programming language ADA: reference manual*, 1983. ANSI-MIL-STD-1815A-1983.

[Chu32] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2): 346–366, 1932.

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

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

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

[Fre79] Gottlob Frege. Begriffsschrift, eine der arithmetischen nachgebildete formelsprache des reinen für das reinene dansken, 1879. Reprinted in [vH67], pagg. 1–82.

[Fre93] Gottlob Frege. *Grundgesetze der Arithmetik, begriffsschriftlich abgeleitet*, volume 1. Pohle, Jena, 1893.

[Fre03] Gottlob Frege. *Grundgesetze der Arithmetik, begriffsschriftlich abgeleitet*, volume 2. Pohle, Jena, 1903.

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

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

[Gor79] Michael J. Gordon. *The denotational description of programming languages.* Springer-Verlag, 1979.

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

[Gun92a] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. MIT Press, 1992.

[Gun92b] Carl A. Gunter. *Semantics of programming languages : structures and techniques*. The MIT Press, 1992.

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

[JW72] Kathleen Jensen and Niklaus Wirth. *PASCAL user manual and report : ISO PASCAL standard*. 1972.

[KR35] S. C. Kleene and J.B. Rosser. The inconsistency of certain formal theories. *Annals of Mathematics*, 36(2): 630–636, 1935.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. Prentice-Hall software series. Prentice Hall, 1978.

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

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.

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

[RW10] Bertrand Russel and Alfred North Whitehead. *Principia Mathematica*. Cambridge University Press, 1910. Cambridge, England.

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

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

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

[Str86] Bjarne Stroustrup. *The C++ programming language*. Addison-Wesley series in Computer Science. Addison-Wesley, 1986.

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

[vH67] Jean van Heijenoort, editor. *From Frege to Gödel, a source book in mathematical logic.* Harvard University Press, 1967.

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

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

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

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

Index