Programmation Avancée, Cours 4

Les Monades

David Baelde

ENS Paris-Saclay

12 février 2021

La pureté

Dans un langage pur (e.g. λ-calcul simplement typé) on peut interpréter :

- les programmes comme des valeurs (résultats) ;
- les fonctions comme des fonctions totales entre des valeurs.

$$\begin{aligned} [\![\mathsf{nat}]\!] &\stackrel{\text{def}}{=} \mathbb{N} \\ [\![A \to B]\!] &\stackrel{\text{def}}{=} [\![A]\!] \to [\![B]\!] \end{aligned}$$

Cela facilite l'étude théorique, le raisonnement équationnel, l'analyse statique, la compilation optimisante, etc.

La pureté peut aussi faciliter des techniques de programmation, e.g. le backtracking.

Les effets

Les effets sont ce qui nous éloigne de ce point de vue :

- Distinction programme / valeur retournée,
  importance de l'ordre d'évaluation.
- On n'a plus des fonctions totales entre ensembles de valeurs.

Exemples

- États mutables
- Entrées-sorties
- Continuations, exceptions : contrôle non-local
- Non-terminaison
- Non-déterminisme, traçage, programmation quantique, etc.

Un problème concret

Rappelon la règle du **let** dans le typage de ML :

$$\frac{\Gamma \vdash u : \sigma \quad \Gamma, x : \forall \vec{\alpha}.\sigma \vdash v : \tau}{\Gamma \vdash \textbf{let } x = u \textbf{ in } v : \tau} \qquad \text{où} \qquad \vec{\alpha} \subseteq \textbf{ftv}(\sigma) \setminus \textbf{ftv}(\Gamma)$$

Intuitivement, cette règle est correcte car on peut assigner les variables α différemment pour les différentes occurrences de u dans $v[x := u]$.

Cela ne tient pas la route en présence d'effets...
en fait, cette règle est incorrecte dès qu'on rajoute les références.

Encoder les impuretés

Pourquoi ?

- Pour définir une sémantique dans un langage mathématique pur.
- Pour compiler un langage impur vers un langage pur,
  et en faciliter le traitement.
- Parce que son langage de programmation favori
  n'est pas doté de tel ou tel effet.

Comment ?

Commençons par deux exemples. . .

Exemple : un unique compteur avec **incr** et **get**

$(\lambda f.f(f3))(\mathbf{incr};(\lambda x.\mathbf{incr};(\mathbf{get}+x)))$ renvoie 8 si compteur initialisé à 0

L'idée

Transformer un programme $\Gamma \vdash P : T$ en $[\![\Gamma]\!] \vdash [\![P]\!] : \mathbf{int} \to \mathbf{int} \times [\![T]\!]$ avec

$$[\![B]\!] \stackrel{\text{def}}{=} B \text{ pour un type de base} \qquad [\![A \times B]\!] \stackrel{\text{def}}{=} [\![A]\!] \times [\![B]\!]$$

$$[\![A \to B]\!] \stackrel{\text{def}}{=} \mathbf{int} \times [\![A]\!] \to \mathbf{int} \times [\![B]\!]$$

Une transformation (évaluation de gauche à droite)

$$[\![\mathbf{get}]\!] \stackrel{\text{def}}{=} \lambda c.\langle c, c\rangle \qquad [\![\mathbf{incr}]\!] \stackrel{\text{def}}{=} \lambda c.\langle c+1, \langle\rangle\rangle \qquad [\![n]\!] \stackrel{\text{def}}{=} \lambda c.\langle c, n\rangle$$

$$[\![\langle u, v\rangle]\!] \stackrel{\text{def}}{=} \lambda c.\ \mathbf{let}\ \langle c', u'\rangle = [\![u]\!]c\ \mathbf{in}\ \mathbf{let}\ \langle c'', v'\rangle = [\![v]\!]c'\ \mathbf{in}\ \langle c'', \langle u', v'\rangle\rangle$$

$$[\![x]\!] \stackrel{\text{def}}{=} \lambda c.\langle c, x\rangle \qquad [\![\lambda x.u]\!] \stackrel{\text{def}}{=} \lambda c.\langle c, \lambda\langle c', x\rangle.([\![u]\!]c')\rangle$$

$$[\![uv]\!] \stackrel{\text{def}}{=} \lambda c.\ \mathbf{let}\ \langle c', u'\rangle = [\![u]\!]c\ \mathbf{in}\ \ \mathbf{let}\ \langle c'', v'\rangle = [\![v]\!]c'\ \mathbf{in}\ \ u'\langle c'', v'\rangle$$

Exemple : **failwith** et **try-with**

type $A + B =$ **left of** $A \mid$ **right of** B

L'idée

Transformer un programme $\Gamma \vdash P : T$ en $[\![\Gamma]\!] \vdash [\![P]\!] : [\![T]\!] +$ **string** avec $[\![B]\!] \stackrel{\text{def}}{=} B$ pour un type de base et $[\![A \to B]\!] \stackrel{\text{def}}{=} \ldots$

La transformation

$[\![n]\!] \stackrel{\text{def}}{=}$ **left** n $\quad [\![x]\!] \stackrel{\text{def}}{=}$ **left** x $\quad [\![\lambda x.u]\!] \stackrel{\text{def}}{=}$ **left** $\lambda x.[\![u]\!]$

$[\![$**failwith** $s]\!] \stackrel{\text{def}}{=}$ **right** s

$[\![$**try** u **with** $s \mapsto v]\!] \stackrel{\text{def}}{=}$ **match** $[\![u]\!]$ **with left** $u' \mapsto$ **left** $u' \mid$ **right** $s \mapsto [\![v]\!]$

$[\![uv]\!] \stackrel{\text{def}}{=}$ **match** $[\![u]\!], [\![v]\!]$ **with left** u', **left** $v' \mapsto u'v'$

$\quad \mid$ **right** $s, \_ \mid \_,$ **right** $s \mapsto$ **right** s

On continue ?

Non. Systématisons plutôt ce qui peut l'être...

Comment ?

Si l'on exclut les opérations spécifiques (e.g. **incr**, **failwith**)
on a eu besoin de définir, pour chaque transformation :

- le type $F(A)$ des programmes renvoyant des valeurs de type A ;
- comment les valeurs de A s'injectent dans $F(A)$;
- comment les programmes s'enchaînent.

Dans les deux cas on avait $[\![P]\!] : F([\![A]\!])$ et $[\![A \to B]\!] = [\![A]\!] \to F([\![B]\!])$.

Les monades

Définition

Une monade est donnée par un type paramétré $F(\_)$ et deux opérations :

- **return** : $A \to F(A)$ (pour tout A)
- **bind** : $F(A) \to (A \to F(B)) \to F(B)$

tel que **bind** (**return** u) $v = vu$, **bind** u **return** $= u$
et **bind** (**bind** u v) $v' =$ **bind** u $(\lambda x.$ **bind** (vx) $v')$ pour $x \notin \mathrm{fv}(v, v')$.

Transformation monadique $P : T \rightsquigarrow [\![P]\!] : F([\![T]\!])$

$$[\![n]\!] \stackrel{\text{def}}{=} \textbf{return}\ n \qquad [\![x]\!] \stackrel{\text{def}}{=} \textbf{return}\ x \qquad [\![\lambda x.u]\!] \stackrel{\text{def}}{=} \textbf{return}\ (\lambda x.[\![u]\!])$$

$$[\![uv]\!] \stackrel{\text{def}}{=} \textbf{bind}\ [\![u]\!]\ (\lambda u'.\ \textbf{bind}\ [\![v]\!]\ (\lambda v'.\ u'v')) \quad \text{(de gauche à droite)}$$

$$[\![uv]\!] \stackrel{\text{def}}{=} \textbf{bind}\ [\![v]\!]\ (\lambda v'.\ \textbf{bind}\ [\![u]\!]\ (\lambda u'.\ u'v')) \quad \text{(de droite à gauche)}$$

$$[\![\textbf{let}\ x = u\ \textbf{in}\ v]\!] \stackrel{\text{def}}{=} \textbf{bind}\ [\![u]\!]\ (\lambda x.[\![v]\!])$$

Des monades à gogo

Quelques monades utiles :

- Environnement : $F(A) = \mathbf{env} \to A$
- Journalisation : $F(A) = A \times \mathbf{string}$
- Continuations : $F(A) = (A \to R) \to R$ où R est le type des résultats
- Non-déterminisme, probabilités, etc.

Une monade peu utile en pratique :

- Partialité : $F(A) = A$ **option**

Langages de programmation

Les monades sont une technique utile pour programmer.
On peut mélanger programmation pure et monadique, mixer les monades, etc. et dévier un peu de la transformation de programmes systématique.
Par exemple, on peut écrire **bind** u $\big($**fun** $x \to$ **return** $(x + 3)\big)$.

En Haskell

- Toutes les impuretés sont encapsulées via des monades.
- Compilation efficace de la monade d'entrée-sortie.
- Programmation monadique abstraite via les type classes.
- Allons voir ça dans le code : `haskell/*hs`

En OCaml

- Langage impur, le compilateur n'utilise pas les monades.
- La notion abstraite de monade s'exprime via les modules.
- Sucre syntaxique pour faciliter la programmation monadique.
- Allons voir ça dans le code : `ocaml/*ml`

Pour conclure

À retenir

Une structure abstraite pour encoder diverses notions de calcul :

- $T(A)$ type des calculs retournant des valeurs de type A
- **return** : $A \to T(A)$ injecte les valeurs dans les calculs
- **bind** : $T(A) \to (A \to T(B)) \to T(B)$ permet de composer les calculs

Questions en suspens

- Quel rapport avec Leibniz ? la théorie des catégories ?
- Peut-on composer des monades ? les transformer ?
- Comment traiter les opérations spécifiques à une monade.
- ...

Pour aller plus loin


Xavier Leroy, *Peut-on changer le monde ? Programmation impérative, effets monadiques, effets algébriques*, cours au collège de France [vidéo, PDF].


Learn You a Haskell for Great Good, chapitres 11–13 [VO ou VF].


Eugenio Moggi, *Computational Lambda-Calculus and Monads*, LICS 1989 [PDF].


Philip Wadler, *Comprehending Monads*, LISP and Functional Programming 1990 & Mathematical Structures in Computer Science 1992 [PDF].