Programmer = démontrer?
La correspondance de Curry-Howard aujourd'hui

Cinquième cours

Peut-on changer le monde?
Programmation impérative, effets monadiques, effets algébriques

Xavier Leroy

Collège de France

2018-12-12

COLLÈGE
DE FRANCE
1530

I

Les effets en programmation et en sémantique

La programmation fonctionnelle pure

Exécuter un programme c'est calculer son résultat final (aussi appelé forme normale ou valeur).

On peut aussi observer que le programme ne termine pas (divergence). (Sauf si le système de types garantit la terminaison.)

La programmation «dans le monde réel»

L'exécution du programme a des effets sur le monde extérieur :

- afficher des choses à l'écran, écrire des fichiers, …
- communiquer sur le réseau
- lire des capteurs, commander des actionneurs.

Vision impérative, «recette de cuisine» de la programmation :

L'exécution du programme a des effets sur l'ordinateur :

- affectations de variables, de cases de tableaux ;
- allocation, modification, libération de structures de données ;
- sauter à un autre point de contrôle (exceptions, continuations, *backtracking*).

Des sémantiques pour les effets

Quelles sémantiques formelles donner aux langages avec effets ?

En particulier, quelles sémantiques dénotationnelles ?

élément de syntaxe
(expression, commande, fonction)

sémantique dénotationnelle

objet mathématique :
lambda-terme pur,
domaine de Scott,
jeu à 2 joueurs, etc.

Sémantique pour l'état mutable

Une commande `x = x+1` doit être vue comme un transformateur d'états :

$$\text{état où } \texttt{x} \text{ vaut } n \xrightarrow{\texttt{x = x+1}} \text{état où } \texttt{x} \text{ vaut } n+1$$

La dénotation d'une commande c est donc une fonction $S \to S$
de l'état $s_1 : S$ au début de l'exécution de c
vers l'état $s_2 : S$ à la fin de l'exécution de c.

Ex : la séquence $c_1; c_2$ est la composition des dénotations $[\![c_2]\!] \circ [\![c_1]\!]$.

De même, la dénotation d'une expression $e : T$ avec effets est une fonction $S \to T \times S$, état ≪avant≫ $\mapsto$ (valeur, état ≪après≫).

Remarque : la même technique de passer l'état courant en argument et résultat supplémentaires permet de programmer des algorithmes impératifs dans des langages fonctionnels purs (Haskell, Agda, Coq).

Sémantiques pour les autres effets

On peut changer la forme des résultats : pour une expression $e : T$,

- $[\![e]\!]$ est un ensemble de $T \Longrightarrow$ non-déterminisme
- $[\![e]\!]$ est une valeur de type T ou une exception $\Longrightarrow$ exceptions.

On peut ajouter une ou plusieurs continuations :

- $[\![e]\!] = \lambda k \ldots$: opérateurs de contrôle, `goto` non local ;
- $[\![e]\!] = \lambda k_{\text{succès}}.\lambda k_{\text{échec}} \ldots$: exceptions, backtracking.

Tout cela est *ad hoc* et peu modulaire : ajouter un effet change toute la sémantique. Peut-on faire plus abstrait et plus modulaire ?

II

Les monades

Les monades

Un concept métaphysique
(Platon, Leibniz, …)

Une structure en théorie des catégories
(Godement, ≪construction standard≫ ; Mac Lane)

Un outil sémantique pour décrire les langages avec effets
(Moggi, 1989)

Une technique pour programmer avec des effets dans un langage pur
(Wadler, 1991 ; la communauté Haskell)

Un outil pour programmer et raisonner sur les programmes avec effets.

Le lambda-calcul computationnel

(Eugenio Moggi, *Computational lambda-calculus and monads*, LICS 1989 ; *Notions of computations and monads*, Inf. Comput. 93(1), 1991.)

Pour modéliser la programmation avec effets, Moggi cherche à construire un lambda-calcul ≪computationnel≫ et ses principes d'équivalence.

Il choisit de distinguer clairement

- valeurs (résultats de calculs), et
- calculs (*computations*, produisant des valeurs).

Un calcul produisant une valeur de type A a un type de la forme $T\,A$.

Le lambda-calcul computationnel

Différents choix pour T correspondent à des sémantiques dénotationnelles connues pour différents effets :

Non-déterminisme :	$T\,A = \mathcal{P}(A)$	
Exceptions :	$T\,A = A + E$	(E type des exceptions)
État mutable :	$T\,A = S \to A \times S$	(S type des états)
Continuations	$T\,A = (A \to R) \to R$	(R type des résultats)

La structure de monade

Pour donner une sémantique aux langages avec effets, il faut deux opérations de base sur les calculs :

- $\texttt{ret} : A \to T\ A$ (injection)
  $\texttt{ret}\ \ v$ est le calcul trivial qui produit la valeur v, sans effets.

- $\texttt{bind} : T\ A \to (A \to T\ B) \to T\ B$ (composition séquentielle)
  $\texttt{bind}\ \ a\ (\lambda x.b)$ effectue le calcul a, lie sa valeur résultat à x, puis effectue le calcul b, et renvoie son résultat.

(Le nom ≪monade≫ est un petit abus de langage : techniquement, à notation près, $(T, \texttt{ret}, \texttt{bind})$ forme un triplet de Kleisli, équivalent à une monade en théorie des catégories.)

Les lois des monades

$$\texttt{bind}\ (\texttt{ret}\ v)\, f = f\ v \qquad \text{(neutre gauche)}$$

$$\texttt{bind}\ a\ \texttt{ret} = a \qquad \text{(neutre droit)}$$

$$\texttt{bind}\ (\texttt{bind}\ a\, f)\, g = \texttt{bind}\ a\ (\lambda x.\ \texttt{bind}\ (f\ x)\, g) \qquad \text{(associativité)}$$

Autre présentation des monades

Au sens de la théorie des catégories, une monade est un triplet (T, η, μ) avec :

$$\eta : A \to T\,A \qquad \mu : T\,(T\,A) \to T\,A \qquad T(f) : T\,A \to T\,B \text{ si } f : A \to B$$

Les deux présentations sont reliées en prenant $\texttt{ret} = \eta$ et

$$\begin{aligned} \texttt{bind}\ \ a\,f &= \mu(T(f)\,a) \\ \mu\,a &= \texttt{bind}\,a\,(\lambda y.\,y) \\ T(f) &= \lambda a.\,\texttt{bind}\ \ a\,(\lambda x.\,\texttt{ret}(f\,x)) \end{aligned}$$

Exemple de monade : le non-déterminisme

$$T\,A = \mathcal{P}(A)$$

$$\texttt{ret}\ v = \{v\}$$

$$\texttt{bind}\ a\,f = \bigcup_{x \in a} f\,x$$

Opérations spécifiques au non-déterminisme :

$$\texttt{fail} = \emptyset$$

$$\texttt{choose}\ a\,b = a \cup b$$

Exemple de monade : les exceptions

$$T\,A = A + E \qquad (E \text{ = type des valeurs d'exceptions})$$

$$\texttt{ret}\ v = \texttt{inj}_1(v)$$

$$\texttt{bind}\ (\texttt{inj}_1(v))\,f = f\,v$$

$$\texttt{bind}\ (\texttt{inj}_2(e))\,f = \texttt{inj}_2(e) \qquad \text{(propagation de l'exception)}$$

Opérations spécifiques aux exceptions :

$$\texttt{raise}\,e = \texttt{inj}_2(e)$$

$$\texttt{try}\ a\ \texttt{with}\ x \to b = \texttt{match}\ a\ \texttt{with}\ \texttt{inj}_1(x) \to \texttt{inj}_1(x) \mid \texttt{inj}_2(x) \to b$$

Exemple de monade : l'état mutable

$$T\,A = S \to A \times S \qquad (S = \text{type des états})$$

$$\texttt{ret}\ \ v = \lambda s.\,(v, s)$$

$$\texttt{bind}\ \ a\,f = \lambda s_1.\,\texttt{let}\ \ (x, s_2) = a\ s_1\ \ \texttt{in}\ \ f\ x\ s_2$$

Opérations spécifiques : (ℓ = identifiants de référénces)

$$\texttt{get}\ \ell = \lambda s.\,(s(\ell), s)$$

$$\texttt{set}\ \ell\ v = \lambda s.\,(\texttt{()}, s\{\ell \leftarrow v\})$$

Exemple de monade : les continuations

$$T\ A = (A \to R) \to R \qquad (R \text{ = type du résultat final})$$

$$\texttt{ret}\ \ v = \lambda k.\ k\ v$$

$$\texttt{bind}\ \ a\ f = \lambda k.\ a\ (\lambda x.\ f\ x\ k)$$

Opérateurs de contrôle :

$$\texttt{callcc}\ \ f = \lambda k.\ f\ (\lambda v.\lambda k'.\ k\ v)\ k$$

$$\mathcal{C}\ f = \lambda k.\ f\ (\lambda v.\lambda k'.\ k\ v)\ (\lambda x.x)$$

Des monades qui combinent plusieurs effets

État + exceptions : $T\,A = S \to (A + E) \times S$

État + continuations : $T\,A = S \to (A \to S \to R) \to R$

Continuations + exceptions : $T\,A = ((A + E) \to R) \to R$
ou $T\,A = (A \to R) \to (E \to R) \to R$

Exercice : écrire `ret` et `bind` pour ces 4 monades.

Voir aussi : les transformateurs de monades, une approche plus systématique pour combiner les effets.

Et encore d'autres monades

Environnement (*reader monad*) : $T\,A = Env \to A$

$\texttt{ret}\ v = \lambda e.\, v$

$\texttt{bind}\ a\, f = \lambda e.\, f\,(a\, e)\, e$

Journal (*writer monad*) : $T\,A = A \times \texttt{string}$

$\texttt{ret}\ v = (v, \texttt{""})$

$\texttt{bind}\ a\, f = \texttt{let}\ (x, s_1) = a\ \texttt{in let}\ (y, s_2) = f\, x\ \texttt{in}\ (y, s_1.s_2)$

Distributions : $T\,A = \mathcal{P}(A \times \mathbb{I})$ (= non-déterminisme + probabilités)

$\texttt{ret}\ v = \{(v, 1)\}$

$\texttt{bind}\ a\, f = \{(y, p_1 \times p_2) \mid (x, p_1) \in a,\ (y, p_2) \in f\, x\}$

$\texttt{choose}\ p\, a\, b = \{(a, p); (b, 1-p)\}$

Espérance : $T\,A = (A \to \mathbb{I}) \to \mathbb{I}$ (= continuations + probabilités)

$\texttt{ret}\ v = \lambda\mu.\, \mu\, v$

$\texttt{bind}\ a\, f = \lambda\mu.\, a\,(\lambda x.\, f\, x\, \mu)$

$\texttt{choose}\ p\, a\, b = \lambda\mu.\, p \times (a\, \mu) + (1-p) \times (b\, \mu)$

Le lambda-calcul computationnel

$$
\begin{array}{lll}
M, N ::= x \mid \lambda x.\, M \mid M\ N & & \text{lambda-calcul} \\
\quad \mid \ldots & & \text{produits, sommes, types inductifs} \\
\quad \mid \texttt{val}\ M & & \text{calcul trivial} \\
\quad \mid \texttt{let}\ x \Leftarrow M\ \texttt{in}\ N & & \text{séquencement de 2 calculs} \\
\quad \mid \ldots & & \text{opérations propres à la monade}
\end{array}
$$

Pour une monade $(T, \texttt{ret}, \texttt{bind})$ donnée, la sémantique s'obtient en interprétant $\texttt{val}\ M$ par $\texttt{ret}\ M$ et $\texttt{let}\ x \Leftarrow M\ \texttt{in}\ N$ par $\texttt{bind}\ M\ (\lambda x.\, N)$.
Équivalences :

$$(\lambda x.\, M)\ N = M\{x \leftarrow N\} \qquad (\beta)$$

$$\lambda x.\, M\ x = M \qquad (\eta)$$

$$\texttt{let}\ x \Leftarrow \texttt{val}\ M\ \texttt{in}\ N = N\{x \leftarrow M\}$$

$$\texttt{let}\ x \Leftarrow M\ \texttt{in}\ \texttt{val}\ x = M$$

$$\texttt{let}\ x \Leftarrow (\texttt{let}\ y \Leftarrow M\ \texttt{in}\ N)\ \texttt{in}\ P = \texttt{let}\ y \Leftarrow M\ \texttt{in}\ \texttt{let}\ x \Leftarrow N\ \texttt{in}\ P$$

Exemple de programme

Dans la monade de non-déterminisme.

Toutes les manière d'insérer un élément `x` dans une liste `l` :

```
let rec insert x l =
  choose (val (x :: l))
         (match l with
            | [] -> fail
            | h :: t -> let t' ⇐ insert x t in val (h :: t'))
```

Toutes les permutations de la liste `l` :

```
let rec permut l =
  match l with
    | [] -> val []
    | h :: t -> let t' ⇐ permut t in insert h t'
```

La transformation monadique

Transforme un langage fonctionnel impur avec effets implicites (Caml, Scheme, etc) en lambda-calcul computationnel avec effets monadiques.

Rend explicites les effets monadiques et la stratégie d'évaluation.

Appel par valeur

$$
\begin{aligned}
[\![cst]\!]_v &= \texttt{val}\ \ cst \\
[\![\lambda x.\, M]\!]_v &= \texttt{val}(\lambda x.\, [\![M]\!]_v) \\
[\![x]\!]_v &= \texttt{val}\ \ x \\
[\![M\ N]\!]_v &= \texttt{let}\ \ f \Leftarrow [\![M]\!]_v\ \ \texttt{in} \\
&\qquad \texttt{let}\ \ a \Leftarrow [\![N]\!]_v\ \ \texttt{in}\ \ f\ a
\end{aligned}
$$

)

Remarque : transformation CPS = transformation monadique + monade des continuations.

La transformation monadique

Transforme un langage fonctionnel impur avec effets implicites (Caml, Scheme, etc) en lambda-calcul computationnel avec effets monadiques.

Rend explicites les effets monadiques et la stratégie d'évaluation.

Appel par valeur

$$
\begin{aligned}
[\![cst]\!]_v &= \texttt{val}\ \ cst \\
[\![\lambda x.\, M]\!]_v &= \texttt{val}(\lambda x.\, [\![M]\!]_v) \\
[\![x]\!]_v &= \texttt{val}\ \ x \\
[\![M\ N]\!]_v &= \texttt{let}\ \ f \Leftarrow [\![M]\!]_v\ \ \texttt{in} \\
&\qquad \texttt{let}\ \ a \Leftarrow [\![N]\!]_v\ \ \texttt{in}\ \ f\ a
\end{aligned}
$$

Appel par nom

$$
\begin{aligned}
[\![cst]\!]_n &= \texttt{val}\ \ cst \\
[\![\lambda x.\, M]\!]_n &= \texttt{val}(\lambda x.\, [\![M]\!]_n) \\
[\![x]\!]_n &= x \\
[\![M\ N]\!]_n &= \texttt{let}\ \ f \Leftarrow [\![M]\!]_n\ \ \texttt{in} \\
&\qquad f\ [\![N]\!]_n
\end{aligned}
$$

Remarque : transformation CPS = transformation monadique + monade des continuations.

La transformation monadique

Effet sur les types :

$$[\![A]\!] = T\, A^*$$

$$\iota^* = \iota$$

$$(A \to B)^* = \begin{cases} A^* \to [\![B]\!] & \text{(appel par valeur)} \\ [\![A]\!] \to [\![B]\!] & \text{(appel par nom)} \end{cases}$$

III

La logique derrière les monades

Curry-Howard pour les monades

Dans l'esprit de Curry-Howard : que signifient les monades et la transformation monadique, vues comme des propositions et des transformations de propositions et de démonstrations ?

- Pour des monades spécifiques (continuations, exceptions) : une interprétation ≪logique≫ intéressante.
- En général : un lien avec les logiques modales.

Monade des continuations et logique classique

Comme vu au cours précédent :

Traduction monadique en appel par nom
pour la monade des continuations $T\,A = (A \to R) \to R = \neg_R \neg_R A$
$\Rightarrow$ traduction négative relative
de la logique classique dans la logique minimale.

$$[\![A]\!]_R = \neg_R \neg_R A \qquad [\![P \Rightarrow Q]\!]_R = \neg_R \neg_R ([\![P]\!]_R \Rightarrow [\![Q]\!]_R)$$
$$[\![P \wedge Q]\!]_R = \neg_R \neg_R ([\![P]\!]_R \wedge [\![Q]\!]_R) \qquad [\![P \vee Q]\!]_R = \neg_R \neg_R ([\![P]\!]_R \vee [\![Q]\!]_R)$$
$$[\![\forall x.\, P]\!]_R = \neg_R \neg_R \forall x.\, [\![P]\!]_R \qquad [\![\exists x.\, P]\!]_R = \neg_R \neg_R \exists x.\, [\![P]\!]_R$$

L'opération `callcc` de la monade correspond à la loi de Clavius, et l'opération $\mathcal{C}$ à l'élimination de la double négation.

Monade des exceptions et ex falso quodlibet

Traduction monadique en appel par nom
pour la monade des exceptions $T\,A = A + E$
$\Rightarrow$ une traduction de la logique intuitionniste dans la logique minimale.

$$[\![\bot]\!] = E \qquad [\![A]\!] = A \vee E \text{ si } A \text{ atomique}$$
$$[\![P \Rightarrow Q]\!] = ([\![P]\!] \Rightarrow [\![Q]\!]) \vee E$$
$$[\![P \wedge Q]\!] = ([\![P]\!] \wedge [\![Q]\!]) \vee E \qquad [\![P \vee Q]\!] = ([\![P]\!] \vee [\![Q]\!]) \vee E$$
$$[\![\forall x.\, P]\!] = (\forall x.\, [\![P]\!]) \vee E \qquad [\![\exists x.\, P]\!] = (\exists x.\, [\![P]\!]) \vee E$$

La règle $\bot \Rightarrow P$, *ex falso quod libet*, devient dérivable après traduction :
$E \Rightarrow \cdots \vee E$, et correspond à l'opération `raise` des exceptions.

Monade = modalité ?

$$\texttt{ret} : A \to T\,A$$
$$\texttt{bind} : T\,A \to (A \to T\,B) \to T\,B$$

Les types des opérations `ret` et `bind` d'une monade peuvent faire penser à des règles de logique modale, le constructeur de type T étant vu comme une modalité.

Logiques modales

Qualifier les propositions logiques par des modalités qui spécifient des qualités du vrai.

Par exemple, suivant Aristote, on peut distinguer les vérités nécessaires $\Box P$ des vérités contingentes A et des vérités possibles $\Diamond P$.

Les modalités $\Box$ et $\Diamond$ sont, en logique classique, reliées par

$$\Box\neg P \Leftrightarrow \neg\Diamond P \qquad \Diamond\neg P \Leftrightarrow \neg\Box P$$

On peut les lire de différentes manières :

- Doxastique : $\Box$ = nécessairement, $\Diamond$ = possiblement.
- Temporelle : $\Box$ = pour toujours, $\Diamond$ = un jour.
- Géographique : $\Box$ = partout, $\Diamond$ = quelque part.

On peut considérer d'autres modalités, p.ex. ≪connu par l'agent i≫ dans les logiques épistémiques.

Logiques modales

De nombreuses axiomatisations différentes, suivant le sens que l'on souhaite donner aux modalités.

Exemple : dans la logique modale S4, les règles pour $\Box$ sont :

$$\Box P \text{ si } P \text{ est une tautologie classique} \quad \text{(N)}$$
$$\Box(P \Rightarrow Q) \Rightarrow (\Box P \Rightarrow \Box Q) \quad \text{(K)}$$
$$\Box P \Rightarrow P \quad \text{(T)}$$
$$\Box P \Rightarrow \Box\Box P \quad \text{(4)}$$

Les règles pour $\Diamond$ s'obtiennent en définissant $\Diamond P \stackrel{def}{=} \neg\Box\neg P$.

Monade = modalité ?

$$\texttt{ret} : A \to T\,A$$
$$\texttt{bind} : T\,A \to (A \to T\,B) \to T\,B$$

Le type de `ret` peut se lire $A \Rightarrow \Diamond A$, suggérant que T est la modalité $\Diamond$, «possiblement».

Mais le type de `bind` est logiquement faux : ✘ $\Diamond A \Rightarrow (A \Rightarrow \Diamond B) \Rightarrow \Diamond B$.

Symétriquement, si T est compris comme la modalité $\Box$, «nécessairement», le type de `bind` est valide mais pas celui de `ret` : ✘ $A \Rightarrow \Box A$.

La modalité relaxée $\bigcirc$ (*lax modality*)

(Mendler, 1991 ; Fairtlough and Mendler, 1997, 2003)

Introduite par Mendler dans le contexte de la vérification formelle de circuits, la modalité $\bigcirc P$ peut se lire comme $\ll P$ est vraie sous certaines conditions$\gg$, ou encore comme $C \Rightarrow P$ pour une condition C implicite.

Elle est caractérisée par les axiomes

$$P \Rightarrow \bigcirc P \qquad \text{(I)}$$

$$\bigcirc \bigcirc P \Rightarrow \bigcirc P \qquad \text{(M)}$$

$$(P \Rightarrow Q) \Rightarrow (\bigcirc P \Rightarrow \bigcirc Q) \qquad \text{(Ext)}$$

$$\bigcirc P \wedge \bigcirc Q \Rightarrow \bigcirc (P \wedge Q) \qquad \text{(S)}$$

Monade = modalité relaxée

(Benton, Bierman, de Paiva, JFP(8), 1998)

Le constructeur de types T d'une monade correspond à la modalité relaxée $\bigcirc$. Les axiomes de la modalité sont réalisés par des termes du lambda-calcul computationnel.

$$\begin{aligned}
\texttt{val} &: P \Rightarrow \bigcirc P \\
\lambda x.\, \texttt{let}\ y \Leftarrow x\ \texttt{in}\ y &: \bigcirc \bigcirc P \Rightarrow \bigcirc P \\
\lambda f.\lambda x.\, \texttt{let}\ v \Leftarrow x\ \texttt{in}\ \texttt{val}(f\ v) &: (P \Rightarrow Q) \Rightarrow (\bigcirc P \Rightarrow \bigcirc Q) \\
\lambda x.\, \texttt{let}\ v_1 \Leftarrow \pi_1(x)\ \texttt{in}& \\
\texttt{let}\ v_2 \Leftarrow \pi_2(x)\ \texttt{in}\ &: \bigcirc P \wedge \bigcirc Q \Rightarrow \bigcirc(P \wedge Q) \\
\texttt{val}(v_1, v_2)&
\end{aligned}$$

Un autre codage modal

(Pfenning et Davies, MSCS(11), 2001)

On peut aussi coder les types d'un langage monadique avec les modalités $\Box$ et $\Diamond$ usuelles :

$$\begin{aligned} [\![\iota]\!] &= \iota \text{ pour un type de base } \iota \\ [\![A \to B]\!] &= \Box[\![A]\!] \Rightarrow [\![B]\!] \\ [\![T\ A]\!] &= \Diamond\Box[\![A]\!] \end{aligned}$$

Intuitions en termes de logique temporelle :

- une valeur de type A est stable vis-à-vis d'effets futurs $\Longrightarrow \Box[\![A]\!]$, «pour toujours A» ;
- un calcul de type A produira, après des effets, une valeur de type A $\Longrightarrow \Diamond\Box[\![A]\!]$, «un jour, pour toujours A».

IV

Des monades pour faire de la logique

Types dépendants, pré-conditions, post-conditions

Dans un langage à types dépendants (comme Agda, Coq, ou F*), on peut écrire des types très précis, comme par exemple

$\forall x : A.\ P(x) \to B$	fonction prenant un $x : A$ et une preuve de $P(x)$
$\{y : B \mid Q(y)\}$	paire d'un $y : B$ et d'une preuve de $Q(y)$
$\forall x : A.\ P(x) \to \{y : B \mid Q(x, y)\}$	fonction $A \to B$ respectant la précondition P et la postcondition Q

Exemple : la division Euclidienne.

```
div: ∀(a b: nat), b > 0 -> { q | ∃r, a = b * q + r ∧ 0 <= r < b }
```

Monade d'état : invariants, évolutions monotones

$$T\ A = S \to A \times S$$

On peut garantir un invariant $Inv : S \to \texttt{Prop}$ sur les états en remplaçant S par un type sous-ensemble S_{Inv} :

$$T\ A = S_{Inv} \to A \times S_{Inv} \quad \text{avec} \quad S_{Inv} = \{s : S \mid Inv\ s\}$$

On peut aussi garantir que l'état évolue de manière monotone vis-à-vis d'un ordre $Ord : S \to S \to \texttt{Prop}$:

$$T\ A = \forall (s : S),\ A \times \{s' : S \mid Ord\ s\ s'\}$$

Un état monotone : le temps

Supposons que l'état est juste une estampille temporelle (*timestamp*). On peut assurer que les calculs ne ≪reviennent pas en arrière dans le temps≫ avec la monade

$$T\ A = \forall (t : Z),\ A \times \{t' : Z \mid t \leq t'\}$$

Un calcul $c : T\ A$ dans cette monade garantit automatiquement que $c\ t_1 = (v, t_2) \Rightarrow t_2 \geq t_1$.

Cela aide beaucoup à établir des propriétés d'unicité des estampilles :

```
let t1 ⇐ timestamp in
let x ⇐ f ... in
let t2 ⇐ timestamp in
(t1, x, t2)
```

Quelle que soit l'action de `f`, on sait que `t1` $<$ `t2` et donc `t1` $\neq$ `t2`.

Un état monotone : le temps

Les opérations de la monade sont plus complexes et font intervenir des termes de preuve :

```
Definition T(A: Type) := forall (t: Z), A *  t' : Z | t <= t' .

Definition ret (A: Type) (a: A) : T A :=
  fun (t: Z) => (a, exist _ t (Z.le_refl t)).

Definition bind (A B: Type) (a: T A) (f: A -> T B) : T B :=
  fun (t1: Z) =>
    let '(x, exist _ t2 p12) := a t1 in
    let '(y, exist _ t3 p23) := f x t2 in
    (y, exist _ t3 (Z.le_trans t1 t2 t3 p12 p23)).

Definition timestamp : T Z :=
  fun (t: Z) => (t, exist _ (Z.succ t) (Z.le_succ_diag_r t)).
```

HTT : la théorie des types de Hoare

(*Hoare Type Theory*. Nanevski et al, ICFP 2008, POPL 2010.)

Au lieu de fixer à l'avance des propriétés attendues d'un état (*Inv*) ou de deux états (*Ord*), on peut aussi paramétrer la monade d'état par une précondition P et une postcondition Q quelconques.

$$pre \stackrel{def}{=} S \to \texttt{Prop}$$

$$post\ A \stackrel{def}{=} A \to S \to S \to \texttt{Prop}$$

$$ST\ :\ pre \to \forall (A : \texttt{Type}), post\ A \to \texttt{Type}$$

$$ST\ P\ A\ Q \stackrel{def}{=} \forall (s_1 : S),\ P\ s_1 \to \{(a, s_2) : A \times S \mid Q\ a\ s_1\ s_2\}$$

Un calcul $c : ST\ P\ A\ Q$ est l'équivalent fonctionnel monadique d'une commande c satisfaisant le triplet de Hoare $\{P\}\ c\ \{Q\}$:
évalué dans un état initial s_1 satisfaisant P, le calcul c produit une valeur a et un état final s_2 satisfaisant Q.

Typage des opérations

On peut donner des types horriblement précis aux opérations de la monade d'état :

$$\texttt{ret} : \forall (A : \texttt{Type})(v : A), ST\ (\lambda s_1.\top)\ A\ (\lambda x, s_1, s_2.\ s_2 = s_1 \wedge x = v)$$

$$\begin{aligned}\texttt{get} : \ & \forall (A : \texttt{Type})(l : loc\ A), \\ & ST\ (\lambda s_1.\ \texttt{valid}\ \ l\ s_1)\ A\ (\lambda x, s_1, s_2.\ s_2 = s_1 \wedge x = get\ l\ s_1)\end{aligned}$$

$$\begin{aligned}\texttt{set} : \ & \forall (A : \texttt{Type})(l : loc\ A)(v : A), \\ & ST\ (\lambda s_1.\ \texttt{valid}\ \ l\ s_1)\ \texttt{unit}\ (\lambda x, s_1, s_2.\ s_2 = set\ l\ v\ s_1 \wedge x = \texttt{tt})\end{aligned}$$

$$\begin{aligned}\texttt{bind} : \ & \forall (A\ B : \texttt{Type})(P_1 : pre)(Q_1 : post\ A)(P_2 : A \to pre)(Q_2 : A \to post\ B), \\ & ST\ P_1\ A\ Q_1 \to (\forall (a : A), ST\ (P_2\ a)\ B\ (Q_2\ a)) \to ST\ P\ B\ Q\end{aligned}$$

avec $P = \lambda s_1.\ P_1\ s_1 \wedge \forall a, s_2.\ Q_1\ a\ s_1\ s_2 \Rightarrow P_2\ s_2$
et $Q = \lambda b, s_1, s_3.\ \exists a, s_2.\ Q_1\ a\ s_1\ s_2 \wedge Q_2\ a\ b\ s_2\ s_3$.

Plus faibles préconditions et transformateurs de prédicats

Depuis Dijkstra (1975), on sait que pour toute commande c et postcondition Q, il existe une plus faible précondition P telle que $\{P\}\ c\ \{Q\}$.

On peut la calculer en fonction de c et Q : $P = wp(c, Q)$ (*weakest precondition*).

Autrement dit, le comportement de la commande c est entièrement caractérisé par le transformateur de prédicats $Q \mapsto wp(c, Q)$, c'est-à-dire une fonction W : *postcondition* $\mapsto$ *plus faible précondition*.

La monade de Dijkstra

(Swamy et al, PLDI 2013, POPL 2016)

Une monade d'état *ST A W* qui décrit les calculs produisant des valeurs de type *A* et qui satisfont le transformateur de prédicats *W*

$$pre \stackrel{def}{=} S \to \texttt{Prop}$$

$$post\ A \stackrel{def}{=} A \to S \to \texttt{Prop}$$

$$wptransf\ A \stackrel{def}{=} post\ A \to pre$$

$$ST\ :\ \forall (A : \texttt{Type}), wptransf\ A \to \texttt{Type}$$

$$ST\ A\ W \stackrel{def}{=} \forall (Q : post\ A)(s_1 : S),\ W\ Q\ s_1 \to \{(a, s_2) : A \times S \mid Q\ a\ s_1\ s_2\}$$

Typage des opérations de la monade de Dijkstra

Les types des opérations de la monade de Dijkstra sont un peu plus simples que dans la monade HTT, et se prêtent mieux à l'inférence par unification.

$$\texttt{ret} : \forall (A : \texttt{Type})(x : A), ST\ A\ (\lambda Q.\ Q\ x)$$

$$\begin{aligned}\texttt{get} : \ & \forall (A : \texttt{Type})(l : loc\ A), \\ & ST\ A\ (\lambda Q.\lambda s.\ \texttt{valid}\ l\ s \wedge Q\ (\texttt{get}\ l\ s)\ s)\end{aligned}$$

$$\begin{aligned}\texttt{set} : \ & \forall (A : \texttt{Type})(l : loc\ A)(v : A), \\ & ST\ \texttt{unit}\ (\lambda Q.\lambda s.\ \texttt{valid}\ l\ s \wedge Q\ \texttt{tt}\ (\texttt{set}\ l\ v\ s))\end{aligned}$$

$$\begin{aligned}\texttt{bind} : \ & \forall (A\ B : \texttt{Type})(W_1 : wptransf\ A)(W_2 : A \to wptransf\ B), \\ & ST\ A\ W_1 \to (\forall (a : A), ST\ B\ (W_2\ a)) \to ST\ B\ (\lambda Q.\ W_1\ (\lambda a.\ W_2\ a\ Q))\end{aligned}$$

De plus, l'approche ≪monade de Dijkstra≫ s'étend à d'autres effets (partialité, exceptions) et à leurs combinaisons $\Longrightarrow$ le langage F*.

V

Effets algébriques et gestionnaires d'effets

D'où viennent les effets ?

Le lambda-calcul computationnel de Moggi, et plus généralement l'approche monadique, rend compte de la propagation et de l'enchaînement des effets de manière *générique* (indépendamment du type d'effets considéré).

Peut-on rendre compte, de manière générique également, des opérations de base qui créent les effets ? Par exemple,

- Entrées-sorties : `print`, `read`
- Exceptions : `raise`
- État mutable : `set`, `get`
- Non-déterminisme : `choose`, `fail`.

Plotkin et Power (2003) proposent une vision algébrique de ces opérations qui créent les effets.

Structures algébriques

En mathématiques, une structure algébrique est un ensemble muni d'opérations qui satisfont des identités (équations).

Exemple : un groupe est un ensemble G avec trois opérations : une constante 1, une opération binaire $\cdot$, une opération unaire $^{-1}$, satisfaisant les identités

$$(x \cdot y) \cdot z = x \cdot (y \cdot z)$$
$$1 \cdot x = x = x \cdot 1$$
$$x \cdot x^{-1} = 1 = x^{-1} \cdot x$$

Types abstraits algébriques

En informatique, un type abstrait algébrique est un type abstrait (= nom de type + opérations) spécifié par des équations portant sur les opérations.

Exemple : les tableaux fonctionnels (opérations `get`, `set`)

$$\texttt{get}\ i\ (\texttt{set}\ i\ v\ t) = v$$
$$\texttt{get}\ i\ (\texttt{set}\ j\ v\ t) = \texttt{get}\ i\ t \quad \text{si } i \neq j$$

Exemple : les piles (opérations `empty`, `push`, `pop`, `top`)

$$\texttt{top}\ (\texttt{push}\ v\ s) = v$$
$$\texttt{pop}\ (\texttt{push}\ v\ s) = s$$

Effets algébriques

(Plotkin, Power, Pretnar, et al ; 2003–)

Valeurs : $v ::= x \mid cst \mid \lambda x.\, M$

Calculs :

$M, N ::=$	$\texttt{val}\ v$	calcul trivial
	$\mid \texttt{let}\ x \Leftarrow M\ \texttt{in}\ N$	séquencement de 2 calculs
	$\mid v\, v'$	application
	$\mid op(\vec{v}; y.\, M)$	opération avec effet

Le terme $op(v_1 \ldots v_n; y.\, M)$ représente une opération qui produit un effet. Les valeurs v_i sont les paramètres de cette opération. L'opération produit une valeur résultat qui est liée à y dans la continuation M.

Notation : $op(\vec{v}) \stackrel{def}{=} op(\vec{v}; y.\, \texttt{val}(y))$ (continuation triviale).

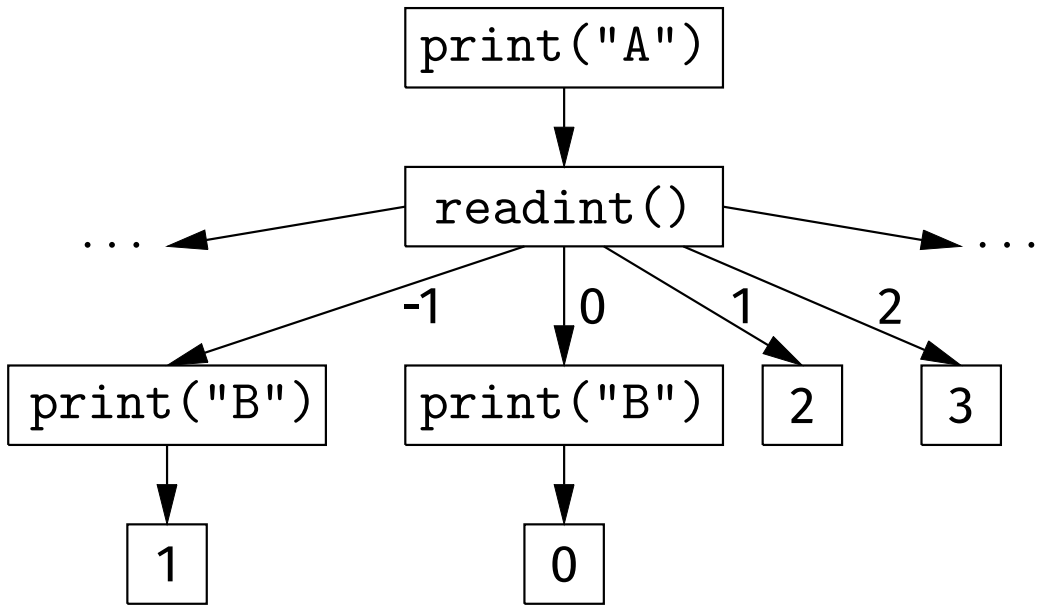
Sémantiquement on a l'équivalence $op(\vec{v}, y.\, M) = \texttt{let}\ y \Leftarrow op(\vec{v})\ \texttt{in}\ M$.

Exemple : entrées-sorties

(Pretnar, *An introduction to algebraic effects and handlers*, MFPS 2015)

Opérations : `print` qui prend une chaîne et `readint` qui renvoie un entier.

```
let _ ⇐ print("A") in
let n ⇐ readint() in
if n <= 0 then
  (let _ ⇐ print("B")
   in val (-n))
else
  val (n+1)
```



Sémantique intuitive : un arbre d'actions avec des opérations aux noeuds et des valeurs (ou $\bot$) aux feuilles.

Équations sur les effets

Les effets d'entrée-sortie sont ≪libres≫ : après une sortie, toutes les entrées restent possibles. Ce n'est pas le cas pour d'autres effets. Pour l'état mutable (opérations $\mathtt{get}$ et $\mathtt{set}$ sur des références ℓ), on a au moins les équations suivantes :

$$\mathtt{set}(\ell, v;\ \_.\, \mathtt{get}(\ell; z.\, M)) = \mathtt{set}(\ell, v;\ \_.\, M\{z \leftarrow v\})$$
$$\mathtt{set}(\ell, v;\ \_.\, \mathtt{get}(\ell'; z.\, M)) = \mathtt{get}(\ell'; z.\, \mathtt{set}(\ell, v; \_.\, M)) \quad \text{si } \ell' \neq \ell$$

et pour être complet, on peut ajouter

$$\mathtt{get}(\ell; y.\, \mathtt{get}(\ell; z.\, M)) = \mathtt{get}(\ell; y.\, M\{z \leftarrow y\}) \qquad \text{(double lecture)}$$
$$\mathtt{get}(\ell; y.\, \mathtt{set}(\ell, y;\ \_.\, M)) = M \qquad \text{(lire puis réécrire)}$$
$$\mathtt{set}(\ell, v_1; \_.\, \mathtt{set}(\ell, v_2; \_.\, M)) = \mathtt{set}(\ell, v_2; \_.\, M) \qquad \text{(double écriture)}$$
$$\mathtt{get}(\ell; y.\, \mathtt{get}(\ell'; z.\, M)) = \mathtt{get}(\ell'; z.\, \mathtt{get}(\ell; y.\, M)) \quad \text{si } \ell' \neq \ell$$
$$\mathtt{set}(\ell, v; y.\, \mathtt{set}(\ell', v'; z.\, M)) = \mathtt{set}(\ell', v'; z.\, \mathtt{set}(\ell, v; y.\, M)) \quad \text{si } \ell' \neq \ell$$

Gérer les effets

Pour les entrées-sorties ou l'état mutable, on peut imaginer que les effets sont exécutés par le système d'exploitation ou l'environnement d'exécution du langage.

Comment faire pour permettre au programme de gérer (≪exécuter≫) lui-même les effets qu'il déclenche ?

La gestion des exceptions

`raise`(e) peut être vu comme un opérateur qui produit l'effet «exception e». Il peut être géré par la construction

$$\texttt{try } a \texttt{ with } x \to b$$

qui intercepte les exceptions levées par a et évalue alors b (le gestionnaire d'exceptions).

Certains langages (Common Lisp, Dylan) donnent au gestionnaire la possibilité de reprendre le calcul au point où l'exception a été levée. On peut modéliser cela par un paramètre k du gestionnaire, qui est lié à la continuation de l'expression `raise`(e) :

$$\texttt{try } a \texttt{ with } (x,k) \to \texttt{if ... then } k\ \texttt{0 else } b$$

(reprise avec la valeur 0 pour le raise) (arrêt sur la valeur b)

Les gestionnaires d'effets

Valeurs :	v	$::= x \mid cst \mid \lambda x.\, M$	
Calculs :	M, N	$::= \texttt{val}\ v$	calcul trivial
		$\mid \texttt{let}\ x \Leftarrow M\ \texttt{in}\ N$	séquencement de 2 calculs
		$\mid v\, v'$	application
		$\mid op(\vec{v}; y.\, M)$	opération avec effet
		$\mid \texttt{with}\ H\ \texttt{handle}\ M$	gestionnaire d'effets
Gestionnaires :	H	$::= \{\ \texttt{val}(x) \to M_{val};$	
		$\quad op_1(\vec{x}; k) \to M_1;$	
		$\quad \ldots$	
		$\quad op_n(\vec{x}; k) \to M_n\ \}$	

Dans `with` H `handle` M,

- si M évalue $op_i(\vec{v}; y.\, N)$, le cas M_i est évalué avec $\vec{x} = \vec{v}$ et $k = \lambda y.\, N$;
- si M évalue `val` v, le cas M_{val} est évalué avec $x = v$.

Exemples de gestionnaires d'effets

Gestion d'exceptions :

```
with { val(x) → val(x);
       raise(e; k) → if ... then k 0 else b }
handle a
```

Inverser l'ordre des impressions faites par a :

```
with { val(x) → val(x);
       print(s; k) → let _ ⇐ k() in print(s) }
handle a
```

Collecter les impressions dans une chaîne de caractères :

```
with { val(x) → val(x, "");
       print(s; k) → let (x, acc) ⇐ k()
                     in val (x, concat s acc) }
```

(Le type des calculs est changé : de A en $A \times \texttt{string}$.)

Exemples de gestionnaires d'effets

Non-déterminisme par *backtracking :* (`choose()` est un effet qui renvoie `true` ou `false` de manière non-déterministe)

```
with { val(x) → val(x);
       choose(_; k) → with { fail(_; k') → k false }
                      handle k true }
```

État mutable :

```
with { val(x) → λs. (x, s);
       get(l; k) → λs. (k (lookup l s)) s;
       set(l, v; k) → λs. (k ()) (update l v s) }
```

(Le type des calculs est changé : de A en $S \to A \times S$.)

Pour aller plus loin sur les effets algébriques

Typage statique qui garde trace des effets, p.ex.

Types de valeurs : $A ::= \iota \mid A_1 \times A_2 \mid A \to C \mid C_1 \Rightarrow C_2$ (type de gestionnaire)

Types de calculs : $C ::= A!\{op_1, \ldots, op_n\}$

Langages et implémentations :

- Eff `https://www.eff-lang.org`
- Frank `https://github.com/frank-lang/frank`
- Multicore OCaml
  `https://github.com/ocamllabs/ocaml-multicore/wiki`

VI

En guise de conclusion

Effets monadiques, effets algébriques

Un succès pour l'approche ≪catégorique≫ des langages de programmation.

> *The view that "category theory comes, logically, before the λ-calculus" led us to consider a categorical semantics of computations first, rather than to modify directly the rules of $\beta\eta$-conversion to get a correct calculus.*
>
> *(E. Moggi, Notions of Computations and Monads, 1991)*

Pas un succès pour l'approche ≪Curry-Howard≫ : les liens avec la logique mathématique sont faibles.

VII

Bibliographie

Bibliographie

Programmer avec des monades :

- *All About Monads*, `https://wiki.haskell.org/All_About_Monads`

Programmer et démontrer avec des monades de Dijkstra :

- *Verified programming in F**, `https://www.fstar-lang.org/tutorial/`

Effets algébriques et gestionnaires d'effets :

- Matija Pretnar, *An Introduction to Algebraic Effects and Handlers*, tutoriel, MFPS 2015, `https://www.eff-lang.org/handlers-tutorial.pdf`
- Andrej Bauer, *Algebraic effects and handlers*, cours à l'école d'été OPLSS 2018, `https://www.cs.uoregon.edu/research/summerschool/summer18/lectures/bauer_notes.pdf`