Structures de contrôle, sixième cours

Théorie des effets : des monades aux effets algébriques

Xavier Leroy

2024-02-29

Collège de France, chaire de sciences du logiciel
`xavier.leroy@college-de-france.fr`

Les effets d'un programme

Tout ce qui va au delà de calculer le résultat final du programme.

Effets sur le monde extérieur :

- afficher des choses à l'écran, écrire des fichiers, ...
- communiquer sur le réseau ;
- lire des capteurs, commander des actionneurs ;
- terminer ou diverger (pour certains auteurs).

Effets sur l'état de 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, retour en arrière).

Quelle(s) théorie(s) pour rendre compte de tous ces effets ?

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)

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

Un outil pour formaliser les programmes avec effets et raisonner dessus.

Une prolifération de sémantiques dénotationnelles

Au 4e cours, nous avons aperçu différentes formes de sémantiques dénotationnelles :

$$[\![stmt]\!] : Store \rightarrow Store_\bot \qquad \text{(état mutable)}$$

$$[\![stmt]\!] : Env \rightarrow Store \rightarrow Store_\bot \qquad \text{(environnement + état)}$$

$$[\![stmt]\!] : Env \rightarrow Store \rightarrow (Store \rightarrow Res_\bot) \rightarrow Res_\bot$$

(environnement + état + goto)

La sémantique de constructions de base comme la séquence change à chaque fois qu'on ajoute un trait du langage :

$$[\![s_1; s_2]\!]\ \sigma = [\![s_2]\!]\ ([\![s_1]\!]\ \sigma)$$

$$[\![s_1; s_2]\!]\ \rho\ \sigma = [\![s_2]\!]\ \rho\ ([\![s_1]\!]\ \rho\ \sigma)$$

$$[\![s_1; s_2]\!]\ \rho\ \sigma\ k = [\![s_1]\!]\ \rho\ \sigma\ (\lambda\sigma'.\ [\![s_2]\!]\ \rho\ \sigma'\ k)$$

Une prolifération de transformations de programmes

Aux 4e et 5e cours, nous avons vu plusieurs transformations de programmes fonctionnels :

- $\mathcal{C}$, la transformation CPS (*Continuation-Passing Style*, pour expliciter la stratégie d'évaluation et pour décrire `callcc`;
- $\mathcal{C}^2$, la transformation CPS ≪à deux canons≫, pour décrire les exceptions et la gestion d'exceptions;
- $\mathcal{E}$, la transformation ERS (*Exception-Returning Style*), une autre manière de décrire les exceptions.

Similarités entre ces transformations

Pour les constantes et les λ-abstractions :

$$\begin{array}{rlrl}
\mathcal{C}(cst) &= \lambda k.\, k\ cst & \mathcal{C}(\lambda x.\, M) &= \lambda k.\, k\ (\lambda x.\, \mathcal{C}(M)) \\
\mathcal{C}^2(cst) &= \lambda k_1 k_2.\, k_1\ cst & \mathcal{C}^2(\lambda x.\, M) &= \lambda k_1 k_2.\, k_1\ (\lambda x.\, \mathcal{C}^2(M)) \\
\mathcal{E}(cst) &= V\ cst & \mathcal{E}(\lambda x.\, M) &= V\ (\lambda x.\, \mathcal{E}(M))
\end{array}$$

Dans tous les cas, on ≪renvoie≫ une valeur (cst ou $\lambda x \ldots$)
en la présentant comme un calcul trivial.

Similarités entre ces transformations

Pour la liaison `let` :

$$\mathcal{C}(\texttt{let}\ x = e_1\ \texttt{in}\ e_2) = \lambda k.\, \mathcal{C}(e_1)\ (\lambda x.\, \mathcal{C}(e_2)\ k)$$

$$\mathcal{C}^2(\texttt{let}\ x = e_1\ \texttt{in}\ e_2) = \lambda k_1 k_2.\, \mathcal{C}^2(e_1)\ (\lambda x.\, \mathcal{C}(e_2)\ k_1\ k_2)\ k_2$$

$$\mathcal{E}(\texttt{let}\ x = e_1\ \texttt{in}\ e_2) = \texttt{match}\ \mathcal{E}(e_1)\ \texttt{with}\ E\ x \to E\ x \mid V\ x \to \mathcal{E}(e_2)$$

Dans les trois transformations, on effectue le calcul e_1, on extrait la valeur résultante, on la lie à x, et on enchaîne sur le calcul de e_2.

Similarités entre ces transformations

Pour l'application de fonction :

$$\mathcal{C}(e_1\ e_2) = \lambda k.\, \mathcal{C}(e_1)\ (\lambda v_1.\, \mathcal{C}(e_2)\ (\lambda v_2.\, v_1\ v_2\ k))$$

$$\mathcal{C}^2(e_1\ e_2) = \lambda k_1.\, \lambda k_2.\, \mathcal{C}^2(e_1)\ (\lambda v_1.\, \mathcal{C}^2(e_2)\ (\lambda v_2.\, v_1\ v_2\ k_1\ k_2)\ k_2)\ k_2$$

$$\begin{aligned}\mathcal{E}(e_1\ e_2) = {} & \texttt{match}\ \mathcal{E}(e_1)\ \texttt{with}\ E\,x_1 \to E\,x_1 \mid V\,v_1 \to \\ & \texttt{match}\ \mathcal{E}(e_2)\ \texttt{with}\ E\,x_2 \to E\,x_2 \mid V\,v_2 \to v_1\ v_2\end{aligned}$$

Dans les trois transformations, on lie la valeur de e_1 à v_1, puis on lie la valeur de e_2 à v_2, puis on applique v_1 à v_2.

Le lambda-calcul computationnel

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

Pour écrire plus facilement ces sémantiques dénotationnelles et ces transformations de programmes, Moggi construit un ≪lambda-calcul computationnel≫ et ses principes d'équivalence.

Il choisit de distinguer clairement

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

≪Les valeurs sont ; les calculs font.≫ (P. B. Levy)

Un calcul produisant une valeur de type A a le type $T\,A$
(où T est un constructeur de type qui dépend des effets considérés)

Le lambda-calcul computationnel

Différents choix pour T correspondent à des sémantiques dénotationnelles / des transformations de programmes connues, pour différents effets :

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

Opérations de base sur les calculs

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

- `ret` : $A \to T\ A$ (injection)

  `ret` v est le calcul trivial qui produit la valeur v, sans effets.

- `bind` : $T\ A \to (A \to T\ B) \to T\ B$ (composition séquentielle)

  `bind` $a\ (\lambda x.b)$ effectue le calcul a, lie sa valeur résultat à x, puis effectue le calcul b, et renvoie son résultat.

La structure de monade

Pour définir `ret` et `bind`, Moggi se place dans une monade de la théorie des catégories, c.à.d. un triplet (T, η, μ) avec :

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

satisfaisant certaines lois.

On peut alors définir le triplet de Kleisli $(T, \texttt{ret}, \texttt{bind})$ par :

$$\texttt{ret}\ v \stackrel{def}{=} \eta(v)$$
$$\texttt{bind}\ a\ f \stackrel{def}{=} \mu(T(f)\ a)$$

(De nos jours, les informaticiens préfèrent définir directement le triplet de Kleisli, et l'appelent «monade» par abus de langage.)

Les lois des monades (triplets de Kleisli)

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

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

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

Exemple de monade : le non-déterminisme

$$T\,A = \mathcal{P}(A) \qquad \text{(ou } \mathit{List}(A)\text{)}$$

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

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

Opérations spécifiques au non-déterminisme :

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

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

Exemple de monade : les exceptions

$$T\,A = V \ \texttt{of}\ A \mid E \ \texttt{of}\ Exn \qquad (\approx A + Exn)$$

$$\begin{aligned}
\texttt{ret}\ v &= V\,v \\
\texttt{bind}\ (V\,v)\,f &= f\,v \\
\texttt{bind}\ (E\,e)\,f &= E\,e \qquad \text{(propagation de l'exception)}
\end{aligned}$$

Opérations spécifiques aux exceptions :

$$\begin{aligned}
\texttt{raise}\,e &= E\,e \\
\texttt{try}\ a\ \texttt{with}\ x \to b &= \texttt{match}\ a\ \texttt{with}\ V\,v \to V\,v \mid E\,x \to b
\end{aligned}$$

Exemple de monade : l'état mutable

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

$$\begin{aligned} \texttt{ret}\ \ v &= \lambda s.\,(v, s) \\ \texttt{bind}\ \ a\,f &= \lambda s_1.\,\texttt{let}\,(x, s_2) = a\,s_1\ \texttt{in}\,f\,x\,s_2 \end{aligned}$$

Opérations spécifiques : (ℓ = identifiant de référence)

$$\begin{aligned} \texttt{get}\ \ell &= \lambda s.\,(s(\ell), s) \\ \texttt{set}\ \ell\ v &= \lambda s.\,((), s\{\ell \leftarrow v\}) \end{aligned}$$

Exemple de monade : les continuations

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

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

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

Opérateur de contrôle :

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

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 :

$$\begin{aligned} & T\,A = ((A + E) \to R) \to R \\ \text{ou} \quad & T\,A = (A \to R) \to (E \to R) \to R \end{aligned}$$

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

Un lambda-calcul computationnel

≪Les valeurs sont; les calculs font.≫

Valeurs :

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

Calculs :

$M, N ::= v_1\ v_2$	application
$\mid \mathtt{if}\ v\ \mathtt{then}\ M\ \mathtt{else}\ N$	conditionnelle
$\mid \mathtt{val}\ v$	calcul trivial
$\mid \mathtt{do}\ x \Leftarrow M\ \mathtt{in}\ N$	séquencement de 2 calculs
$\mid \ldots$	opérations spécifiques

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

Les lois du lambda-calcul computationnel

L'appel de fonction :

$$(\lambda x.\, M)\; v = M\{x \leftarrow v\}$$

Les trois lois monadiques :

$$\begin{aligned} \mathtt{do}\; x \Leftarrow \mathtt{val}\; v\; \mathtt{in}\; M &= M\{x \leftarrow v\} \\ \mathtt{do}\; x \Leftarrow M\; \mathtt{in}\; \mathtt{val}\; x &= M \\ \mathtt{do}\; x \Leftarrow (\mathtt{do}\; y \Leftarrow M\; \mathtt{in}\; N)\; \mathtt{in}\; P &= \mathtt{do}\; y \Leftarrow M\; \mathtt{in}\; (\mathtt{do}\; x \Leftarrow N\; \mathtt{in}\; P) \end{aligned}$$

La transformation monadique

Transforme un langage fonctionnel impur avec effets implicites en lambda-calcul computationnel avec effets monadiques explicites.

$$
\begin{aligned}
\mathcal{M}(cst) &= \texttt{val}\ cst \\
\mathcal{M}(\lambda x.\, e) &= \texttt{val}\ (\lambda x.\, \mathcal{M}(e)) \\
\mathcal{M}(x) &= \texttt{val}\ x \\
\mathcal{M}(\texttt{let}\ x = e_1\ \ \texttt{in}\ \ e_2) &= \texttt{do}\ x \Leftarrow \mathcal{M}(e_1)\ \texttt{in}\ \mathcal{M}(e_2) \\
\mathcal{M}(e_1\ e_2) &= \texttt{do}\ f \Leftarrow \mathcal{M}(e_1)\ \texttt{in}\ \texttt{do}\ v \Leftarrow \mathcal{M}(e_2)\ \texttt{in}\ f\ v \\
\mathcal{M}(\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3) &= \texttt{do}\ b \Leftarrow \mathcal{M}(e_1)\ \texttt{in} \\
&\qquad \texttt{if}\ b\ \texttt{then}\ \mathcal{M}(e_2)\ \texttt{else}\ \mathcal{M}(e_3)
\end{aligned}
$$

Combinée avec la monade appropriée, on retrouve les transformations CPS, ERS, CPS à deux canons, etc.

Programmer directement en style monadique

(Notations `do` en Haskell, `let*` en OCaml.)

On peut écrire du code utilisable dans toutes les monades, p.ex. un itérateur `map` monadique :

```
let (let*) = bind
let rec mmap (f: 'a -> 'b t) (l: 'a list) : 'b list t =
  match l with
  | [] -> ret []
  | h :: t ->
      let* h' = f h in let* l' = mmap f l in ret (h' :: l')
```

(`let*` $x = a$ `in` b s'expanse en `bind` a (`fun` $x \rightarrow b$) .)

Programmer directement en style monadique

Dans la monade de non-déterminisme, voici toutes les manières d'insérer un élément `x` dans une liste `l`.

```
let rec insert (x: 'a) (l: 'a list) : 'a list t =
  choose (ret (x :: l))
         (match l with
            | [] -> fail
            | h :: t -> let* t' = insert x t in ret (h :: t'))
```

Voici toutes les permutations de la liste `l` :

```
let rec permut (l: 'a list) : 'a list t =
  match l with
    | [] -> ret []
    | h :: t -> let* t' = permut t in insert h t'
```

Monades libres et arbres d’interaction

Exécuter un programme monadique sans exécuter les effets

Prenons comme effets l'état mutable et le non-déterminisme.

Valeurs :

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

Calculs :

$$\begin{aligned} M ::= {} & v_1\, v_2 \mid \texttt{if}\ v\ \texttt{then}\ M\ \texttt{else}\ N \\ \mid {} & \texttt{val}\ \ v \mid \texttt{do}\ x \Leftarrow M_1\ \texttt{in}\ M_2 \\ \mid {} & \texttt{get}\ \ell \mid \texttt{set}\ \ell\ v & \text{état mutable} \\ \mid {} & \texttt{choose}\ M_1\ M_2 \mid \texttt{fail} & \text{non-déterminisme} \end{aligned}$$

Peut-on évaluer les `do` et les appels de fonctions, tout en laissant les effets non interprétés ?

Exécuter un programme monadique sans exécuter les effets

On définit un type des résultats intermédiaires d'évaluation qui représente tous les enchaînements possibles des effets du programme.

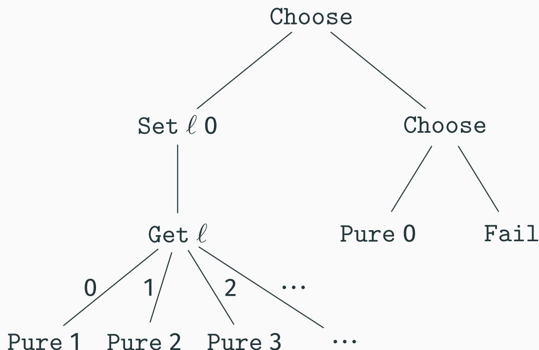
$$\begin{array}{rcl} R\,A & = & \texttt{Pure} : A \to R\,A \\ & \mid & \texttt{Get} : Loc \to (Val \to R\,A) \to R\,A \\ & \mid & \texttt{Set} : Loc \to Val \to R\,A \to R\,A \\ & \mid & \texttt{Choose} : R\,A \to R\,A \to R\,A \\ & \mid & \texttt{Fail} : R\,A \end{array}$$

Une représentation arborescente des effets

Programme :

$$\texttt{choose}\ (\texttt{do}\ _ \Leftarrow \texttt{set}\ \ell\ 0\ \texttt{in}\ \texttt{do}\ x \Leftarrow \texttt{get}\ \ell\ \texttt{in}\ \texttt{val}\ (x+1))$$
$$(\texttt{choose}\ (\texttt{val}\ 0)\ \texttt{fail})$$

Résultat intermédiaire :



La monade des résultats intermédiaires

$$
\begin{aligned}
R\,A \quad = \quad & \mathtt{Pure} : A \to R\,A \\
\mid \quad & \mathtt{Get} : \mathit{Loc} \to (\mathit{Val} \to R\,A) \to R\,A \\
\mid \quad & \mathtt{Set} : \mathit{Loc} \to \mathit{Val} \to R\,A \to R\,A \\
\mid \quad & \mathtt{Choose} : R\,A \to R\,A \to R\,A \\
\mid \quad & \mathtt{Fail} : R\,A
\end{aligned}
$$

Ce type forme une monade, avec $\mathtt{ret} \stackrel{def}{=} \mathtt{Pure}$ et $\mathtt{bind}$ défini par :

$$
\begin{aligned}
\mathtt{bind}\,(\mathtt{Pure}\ v)\ f &= f\ v \\
\mathtt{bind}\,(\mathtt{Get}\ \ell\ k)\ f &= \mathtt{Get}\ \ell\ (\lambda v.\ \mathtt{bind}\,(k\ \ell)\ f) \\
\mathtt{bind}\,(\mathtt{Set}\ \ell\ v\ R)\ f &= \mathtt{Set}\ \ell\ v\ (\mathtt{bind}\ R\ f) \\
\mathtt{bind}\,(\mathtt{Choose}\ R_1\ R_2)\ f &= \mathtt{Choose}\ (\mathtt{bind}\ R_1\ f)\ (\mathtt{bind}\ R_2\ f) \\
\mathtt{bind}\ \mathtt{Fail}\ f &= \mathtt{Fail}
\end{aligned}
$$

Sémantique dénotationnelle des programmes monadiques

À l'aide de cette monade des résultats, on peut calculer le résultat intermédiaire $[\![M]\!]$ d'un calcul monadique M.

$$
\begin{aligned}
[\![v_1\ v_2]\!] &= [\![v_1]\!]_v\ [\![v_2]\!]_v \\
[\![\texttt{if}\ v\ \texttt{then}\ M_1\ \texttt{else}\ M_2]\!] &= \texttt{if}\ [\![v]\!]_v\ \texttt{then}\ [\![M_1]\!]\ \texttt{else}\ [\![M_2]\!] \\
[\![\texttt{val}\ v]\!] &= \texttt{Pure}\ [\![v]\!]_v \\
[\![\texttt{do}\ x \Leftarrow M_1\ \texttt{in}\ M_2]\!] &= \texttt{bind}\ [\![M_1]\!]\ (\lambda x.\ [\![M_2]\!]) \\
[\![\texttt{get}\ \ell]\!] &= \texttt{Get}\ \ell\ (\lambda v.\ \texttt{Pure}\ v) \\
[\![\texttt{set}\ \ell\ v]\!] &= \texttt{Set}\ \ell\ [\![v]\!]_v\ (\texttt{Pure}\ ()) \\
[\![\texttt{choose}\ M_1\ M_2]\!] &= \texttt{Choose}\ [\![M_1]\!]\ [\![M_2]\!] \\
[\![\texttt{fail}]\!] &= \texttt{Fail}
\end{aligned}
$$

Avec $[\![cst]\!]_v = cst$, $[\![x]\!]_v = x$, $[\![\lambda x.\, M]\!]_v = \lambda x.\ [\![M]\!]$.

Interpréter les effets

Finalement, on peut interpréter les effets (fonction `run`) par un parcours de type *fold* de l'arbre de résultats R.

Avec retour en arrière de l'état mémoire aux points de choix : `run` est de type $R\ A \rightarrow \mathit{Store} \rightarrow \mathit{Set}\ A$ et on prend

$$\begin{aligned}
\mathtt{run}\ (\mathtt{Pure}\ v)\ s &= \{v\} \\
\mathtt{run}\ (\mathtt{Get}\ \ell\ k)\ s &= \mathtt{run}\ (k\ (s\ \ell))\ s \\
\mathtt{run}\ (\mathtt{Set}\ \ell\ v\ R)\ s &= \mathtt{run}\ R\ (s\{\ell \leftarrow v\}) \\
\mathtt{run}\ \mathtt{Fail}\ s &= \emptyset \\
\mathtt{run}\ (\mathtt{Choose}\ R_1\ R_2)\ s &= \mathtt{run}\ R_1\ s \cup \mathtt{run}\ R_2\ s
\end{aligned}$$

Interpréter les effets

Finalement, on peut interpréter les effets (fonction `run`) par un parcours de type *fold* de l'arbre de résultats R.

Avec un état mémoire qui persiste aux points de choix : `run` est de type $R\ A \to \mathit{Store} \to \mathit{Set}\ A \times \mathit{Store}$ et on prend

$$\begin{aligned}
\mathtt{run}\ (\mathtt{Pure}\ v)\ s &= (\{v\}, s) \\
\mathtt{run}\ (\mathtt{Get}\ \ell\ k)\ s &= \mathtt{run}\ (k\ (s\ \ell))\ s \\
\mathtt{run}\ (\mathtt{Set}\ \ell\ v\ R)\ s &= \mathtt{run}\ R\ (s\{\ell \leftarrow v\}) \\
\mathtt{run}\ \mathtt{Fail}\ s &= (\emptyset, s) \\
\mathtt{run}\ (\mathtt{Choose}\ R_1\ R_2)\ s &= (V_1 \cup V_2, s_2) \\
&\qquad \text{avec } \mathtt{run}\ R_1\ s = (V_1, s_1) \text{ et } \mathtt{run}\ R_2\ s_1 = (V_2, s_2)
\end{aligned}$$

La monade libre (*free monad*)

Le type $R\,A$ est une instance d'une construction catégorique plus générale : la monade libre.

$$\begin{array}{rcl} R\,A & = & \mathtt{Pure} : A \to R\,A \\ & \mid & \mathtt{Op} : F\,(R\,A) \to R\,A \end{array}$$

où $F : \mathit{Type} \to \mathit{Type}$ est un foncteur : il vient avec une opération

$$\mathtt{fmap} : \forall A, B,\ (A \to B) \to (F\,A \to F\,B)$$

On retrouve l'exemple précédent avec F défini par

$$\begin{array}{ll} F\,X = \mathtt{Get} : \mathit{Loc} \to (\mathit{Val} \to X) \to F\,X & \mid \mathtt{Set} : \mathit{Loc} \to \mathit{Val} \to X \to F\,X \\ \quad \mid\ \mathtt{Choose} : X \to X \to F\,X & \mid \mathtt{Fail} : F\,X \end{array}$$

Exercice : définir `fmap`.

La monade libre (*free monad*)

$$
\begin{array}{rcl}
R\,A & = & \texttt{Pure} : A \to R\,A \\
 & | & \texttt{Op} : F\,(R\,A) \to R\,A
\end{array}
$$

Cette présentation «fonctorielle» permet de définir `ret` et `bind` de manière générique :

$$
\begin{array}{rcl}
\texttt{ret}\; v & = & \texttt{Pure}\; v \\
\texttt{bind}\;(\texttt{Pure}\; v)\; f & = & f\; v \\
\texttt{bind}\;(\texttt{Op}\; \varphi)\; f & = & \texttt{Op}\;(\texttt{fmap}\;(\lambda x.\, \texttt{bind}\; x\; f)\; \varphi)
\end{array}
$$

La monade «plus libre» *(freer monad)*

(O. Kiselyov, H. Ishii, *Freer Monads, More Extensible Effects*, 2015.)

Une autre construction générique du type des résultats intermédiaires d'exécution.

$$\begin{array}{rcl} R\,A & = & \mathtt{Pure} : A \to R\,A \\ & \mid & \mathtt{Op} : \forall B, \mathit{Eff}\ B \to (B \to R\,A) \to R\,A \end{array}$$

Le type $\mathit{Eff}\ B$ est le type des effets qui produisent un résultat de type B. Chaque effet est un constructeur de Eff.

Si $\varphi : \mathit{Eff}\ B$, les sous-arbres de $\mathtt{Op}(\varphi, k)$ sont $k\ b$ pour $b : B$.
Il y a autant de sous-arbres que d'éléments dans B.

Déclarer et typer des effets

Pour l'état mutable et le non-déterminisme :

$\texttt{Get} : \mathit{Loc} \to \mathit{Eff}\ \mathit{Val}$	(autant de sous-arbres que de valeurs)
$\texttt{Set} : \mathit{Loc} \to \mathit{Val} \to \mathit{Eff}\ \texttt{unit}$	(un sous-arbre)
$\texttt{Fail} : \mathit{Eff}\ \texttt{empty}$	(aucun sous-arbre)
$\texttt{Flip} : \mathit{Eff}\ \texttt{bool}$	(deux sous-arbres)

On code l'opération `choose` avec l'effet `Flip` :

$$\texttt{choose}\ R_1\ R_2 \stackrel{\mathit{def}}{=} \texttt{Op}(\texttt{Flip}, \lambda b.\ \texttt{if}\ b\ \texttt{then}\ R_1\ \texttt{else}\ R_2)$$

La monade «plus libre» *(freer monad)*

$$
\begin{array}{rcl}
R\,A & = & \mathtt{Pure} : A \to R\,A \\
 & | & \mathtt{Op} : \forall B, \mathit{Eff}\ B \to (B \to R\,A) \to R\,A
\end{array}
$$

Cette présentation indexée (par le type B) permet aussi de définir `ret` et `bind` de manière générique :

$$
\begin{array}{rcl}
\mathtt{ret}\ v & = & \mathtt{Pure}\ v \\
\mathtt{bind}\ (\mathtt{Pure}\ v)\ f & = & f\ v \\
\mathtt{bind}\ (\mathtt{Op}\ \varphi\ k)\ f & = & \mathtt{Op}\ (\varphi, \lambda x.\ \mathtt{bind}\ (k\ x)\ f)
\end{array}
$$

(Note : plus besoin de foncteur ni de `fmap`.)

Interpréter les effets dans la monade «plus libre»

Par un *fold* générique sur le type des résultats :

$$\begin{aligned} \mathtt{run} &: (A \to B) \to (\forall C, \mathit{Eff}\ C \to (C \to B) \to B) \to R\ A \to B \\ \mathtt{run}\ f\ g\ (\mathtt{Pure}\ v) &= f\ v \\ \mathtt{run}\ f\ g\ (\mathtt{Op}\ \varphi\ k) &= g\ \varphi\ (\lambda x.\ \mathtt{run}\ f\ g\ (k\ x)) \end{aligned}$$

Pour le non-déterminisme avec retour en arrière de l'état mémoire, on prendrait

$$\begin{aligned} f &: A \to \mathit{Store} \to \mathit{Set}\ A \\ f\ x\ s &= \{x\} \\ g &: \mathit{Eff}\ B \to (B \to \mathit{Store} \to \mathit{Set}\ A) \to \mathit{Store} \to \mathit{Set}\ A \end{aligned}$$

$$g\ (\mathtt{Get}\ \ell)\ k\ s = k\ (s\ \ell)\ s \qquad g\ (\mathtt{Set}\ \ell\ v)\ k\ s = k\ ()\ s\{\ell \leftarrow v\}$$

$$g\ \mathtt{Flip}\ k\ s = k\ \mathtt{false}\ s \cup k\ \mathtt{true}\ s \qquad g\ \mathtt{Fail}\ k\ s = \emptyset$$

Interpréter les effets dans la monade ≪plus libre≫

Par un *fold* générique sur le type des résultats :

$$\begin{aligned} \mathtt{run} &: (A \to B) \to (\forall C, \mathit{Eff}\ C \to (C \to B) \to B) \to R\ A \to B \\ \mathtt{run}\ f\ g\ (\mathtt{Pure}\ v) &= f\ v \\ \mathtt{run}\ f\ g\ (\mathtt{Op}\ \varphi\ k) &= g\ \varphi\ (\lambda x.\ \mathtt{run}\ f\ g\ (k\ x)) \end{aligned}$$

On remarque une inversion de contrôle : ce n'est plus le programme qui appelle les opérations `get`, `set`, … de la monade ; c'est l'implémentation de ces opérations (la fonction g) qui évalue le programme ≪à la demande≫ via la continuation k.

Les arbres d'interaction

(Xia, Zakowski, *et al*, *Interaction Trees*, POPL 2020).

Une version coinductive du type des résultats intermédiaires qui permet de rendre compte des calculs qui divergent :

$$\begin{array}{rcl} R\,A & = & \texttt{Pure} : A \to R\,A \\ & | & \texttt{Op} : \forall B, \mathit{Eff}\ B \to (B \to R\,A) \to R\,A \\ & | & \texttt{Tau} : R\,A \to R\,A \end{array}$$

`Tau` matérialise une étape de calcul sans effet observable.

L'arbre infini $\bot \stackrel{def}{=} \texttt{Tau}\ \bot = \texttt{Tau}(\texttt{Tau}(\texttt{Tau}(\ldots)))$ représente un calcul qui diverge sans effet observable.

L'arbre infini $x \stackrel{def}{=} \texttt{Op}(\texttt{Flip}, \lambda b.\ \texttt{if}\ b\ \texttt{then}\ \texttt{Pure}\ \ 0\ \texttt{else}\ x)$ représente `let rec f () = choose 0 (f ())`.

Rappels sur les structures algébriques

Structures algébriques

Une structure algébrique =

- un ensemble (ou un type) appelé le support de la structure ;
- des opérations sur cet ensemble ;
- des équations (lois) satisfaites par ces opérations.

Exemple : un monoïde est $(T, \varepsilon, \cdot)$ avec

$\varepsilon : T$	élément neutre
$\cdot : T \to T \to T$	composition
$\varepsilon \cdot x = x$	neutre à gauche
$x \cdot \varepsilon = x$	neutre à droite
$(x \cdot y) \cdot z = x \cdot (y \cdot z)$	associativité

Structures algébriques

Une structure algébrique =

- un ensemble (ou un type) appelé le support de la structure ;
- des opérations sur cet ensemble ;
- des équations (lois) satisfaites par ces opérations.

Exemple : un groupe est $(T, 0, +, -)$ avec

$0 : T$	élément neutre
$+ : T \to T \to T$	composition
$- : T \to T$	inverse
$0 + x = x + 0 = x$	neutre à gauche, à droite
$(x + y) + z = x + (y + z)$	associativité
$(-x) + x = x + (-x) = 0$	inverse à gauche, à droite

Théories et modèles

Une théorie :

la signature des opérateurs (noms et types)

+ les équations.

Un modèle de la théorie : une définition du support et des opérateurs qui satisfait les équations.

Exemples de modèles de la théorie des monoïdes
(ou juste ≪de monoïdes≫) :

$$(\mathbb{N}, 0, +) \qquad (\mathbb{R}, 1, \times) \qquad (T \to T, id, \circ)$$

Exemples de modèles de la théorie des groupes
(ou juste ≪de groupes≫) :

$$(\mathbb{Z}, 0, +, -) \qquad (\mathbb{R}^*, 1, \times, {}^{-1})$$

Les types abstraits algébriques

Un type abstrait algébrique est la spécification d'une structure de données persistante via une signature et des équations.

(→ cours 2022–2023, 1re séance)

Exemple : les piles

$$\mathtt{empty} : S \qquad \mathtt{push} : E \to S \to S \qquad \mathtt{top} : S \to E \quad \mathtt{pop} : S \to S$$

$$\mathtt{top}(\mathtt{push}\; v\; s) = v \qquad \mathtt{pop}(\mathtt{push}\; v\; s) = s$$

Pour en faire une file (*FIFO*), on ajoute une opération :

$$\mathtt{add} : S \to E \to S$$

$$\mathtt{add}\ \mathtt{empty}\ v = \mathtt{push}\ v\ \mathtt{empty} \qquad \mathtt{add}\ (\mathtt{push}\ w\ s)\ v = \mathtt{push}\ w\ (\mathtt{add}\ s\ v)$$

Le monoïde libre

Étant donné un ensemble (un «alphabet») A,
le monoïde libre sur A est $(A^*, \varepsilon, \cdot)$, avec

- support : A^* l'ensemble des listes finies de A («mots sur A») comme $a_1a_2\dots a_n$;
- élément neutre ε : la liste vide ;
- opération de composition $\cdot$: la concaténation de listes.

Exemple : avec $A = \{1, \dots, 9\}$,

$$1 \cdot (23 \cdot 456) = (1 \cdot 23) \cdot 456 = 123456$$

Le monoïde libre

Le monoïde libre sur A est ≪le plus simple≫ ou ≪le moins contraint≫ des monoïdes dont le support contient A.

En effet, si $(B, 0, +)$ est un monoïde, avec $A \subseteq B$, on peut définir une fonction $\Phi : A^* \to B$ par

$$\Phi(a_1 \ldots a_n) = 0 + a_1 + \cdots + a_n$$

(C'est un *fold* de ≪$+$≫ sur la liste $a_1 \ldots a_n$.)

Cette fonction est un morphisme de $(A^*, \varepsilon, \cdot)$ dans $(B, 0, +)$, car elle commute avec les opérations des monoïdes :

$$\Phi(\varepsilon) = 0 \qquad \Phi(\ell_1 \cdot \ell_2) = \Phi(\ell_1) + \Phi(\ell_2)$$

Modèles libres

Soit T une théorie algébrique et X un ensemble.

Un T-modèle libre engendré par X est un T-modèle M et une fonction $f : X \to supp(M)$ tels que :

Pour tout autre T-modèle M' et fonction $f' : X \to supp(M')$, il existe un unique morphisme $\Phi : M \to M'$ tel que le diagramme suivant commute :

$$\begin{array}{ccc} X & \xrightarrow{f} & supp(M) \\ & \searrow^{f'} & \downarrow \Phi \\ & & supp(M') \end{array}$$

Les monades vues comme des structures algébriques

On peut voir une monade comme une structure algébrique ayant comme opérations $\texttt{ret}$, $\texttt{bind}$, et $\texttt{op}(F)$ pour chaque constructeur F du type *Eff*, avec comme signatures :

$$
\begin{aligned}
\texttt{ret} &: A \to T\ A \\
\texttt{bind} &: T\ A \to (A \to T\ B) \to T\ B \\
\texttt{op}(F) &: P \to (B \to T\ A) \to T\ A \quad \text{si } F : P \to \mathit{Eff}\ B
\end{aligned}
$$

et comme équations les trois lois monadiques, plus éventuellement d'autres lois portant sur $\texttt{op}(F)$.

Les monades libres sont libres

La monade libre et la monade plus libre sont bien des monades libres engendrées par les constructeurs du type *Eff*.

On le vérifie dans le cas de la monade plus libre :

$$\begin{aligned} R\,A &= \mathtt{Pure} : A \to R\,A \\ &\mid \mathtt{Op} : \forall B, \mathit{Eff}\ B \to (B \to R\,A) \to R\,A \end{aligned}$$

Avec les définitions suivantes, on a bien une monade avec la signature attendue :

$$\begin{aligned} \mathtt{ret}\ x &= \mathtt{Pure}\ x \\ \mathtt{bind}\ (\mathtt{Pure}\ x)\ f &= f\ x \\ \mathtt{bind}\ (\mathtt{Op}(\varphi, k))\ f &= \mathtt{Op}(\varphi, \lambda x.\ \mathtt{bind}\ (k\ x)\ f) \\ \mathtt{op}(F) &= \lambda x.\ \mathtt{Op}(F\ x, \lambda y.\ \mathtt{Pure}\ y) \end{aligned}$$

Les monades libres sont libres

Soit $M = (T, \mathtt{ret}_M, \mathtt{bind}_M, \mathtt{op}_M(F))$ une autre monade avec la signature attendue. On définit un morphisme Φ de la monade plus libre dans M par

$$\begin{aligned} \Phi &: R\,A \to T\,A \\ \Phi(\mathtt{Pure}\ v) &= \mathtt{ret}_M\ v \\ \Phi(\mathtt{Op}\ (F\,x, k)) &= \mathtt{bind}_M\ (\mathtt{op}_M(F)\ x)\ (\lambda y.\ \Phi(k\ y)) \end{aligned}$$

Ce morphisme commute avec les opérations `ret` et `bind`.

$$\Phi(\mathtt{bind}\ (\mathtt{Pure}\ v)\ f) = \Phi(f\ v) = \mathtt{bind}_M\ (\Phi(\mathtt{Pure}\ v))\ (\lambda y.\ \Phi(f\ y)) \quad \text{(1re loi)}$$

$$\begin{aligned} \Phi(\mathtt{bind}\ (\mathtt{Op}(F\,x, k))\ f) &= \Phi(\mathtt{Op}(F\,x, \lambda y.\ \mathtt{bind}\ (k\ y)\ f)) \\ &= \mathtt{bind}_M\ (\mathtt{op}_M(F)\ x)\ (\lambda y.\ \Phi(\mathtt{bind}\ (k\ y)\ f)) \\ \text{(3e loi)} \quad &= \mathtt{bind}_M\ (\mathtt{bind}_M\ (\mathtt{op}_M(F)\ x)\ (\lambda y.\ \Phi(k\ y)))\ (\lambda z.\ \Phi(f\ z)) \\ &= \mathtt{bind}_M\ (\Phi(\mathtt{Op}(F\,x, k)))\ (\lambda z.\ \Phi(f\ z)) \end{aligned}$$

Les effets algébriques

Une vision algébrique des effets

Le lambda-calcul computationnel de Moggi, et plus généralement l'approche monadique, spécifie la propagation et l'enchaînement des effets de manière générique.

Comment spécifier la génération des effets par les opérations de base de la monade ? (`set`, `get`, `choose`, `fail`, …)

Plotkin et Power (2003) proposent de spécifier ces opérations par des équations, obtenant ainsi une structure algébrique pour les effets.

Un lambda-calcul computationnel avec effets

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

Calculs : $M, N ::= v\ v'$ — application

$\mid \texttt{if}\ v\ \texttt{then}\ M\ \texttt{else}\ N$ — conditionnelle

$\mid \texttt{val}\ v$ — calcul trivial

$\mid \texttt{do}\ x \Leftarrow M\ \texttt{in}\ N$ — séquencement de 2 calculs

$\mid F(\vec{v}; y.\, M)$ — opération avec effet

Le terme $F(v_1 \ldots v_n; y.\, M)$ 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 : $F(\vec{v}\,) \stackrel{def}{=} F(\vec{v}; y.\, \texttt{val}(y))$ (continuation triviale).

Les lois du lambda-calcul computationnel avec effets

Les mêmes lois que celles du lambda-calcul computationnel :

$$(\lambda x.\, M)\; v = M\{x \leftarrow v\}$$

$$\mathtt{do}\; x \Leftarrow \mathtt{val}\; v\; \mathtt{in}\; M = M\{x \leftarrow x\}$$

$$\mathtt{do}\; x \Leftarrow M\; \mathtt{in}\; \mathtt{val}\; x = M$$

$$\mathtt{do}\; x \Leftarrow (\mathtt{do}\; y \Leftarrow M\; \mathtt{in}\; N)\; \mathtt{in}\; P = \mathtt{do}\; y \Leftarrow M\; \mathtt{in}\; \mathtt{do}\; x \Leftarrow N\; \mathtt{in}\; P$$

Plus : commutation entre `do` et opérations avec effets :

$$\mathtt{do}\; x \Leftarrow F(\vec{v};\, y.\, M)\; \mathtt{in}\; N = F(\vec{v};\, y.\, \mathtt{do}\; x \Leftarrow M\; \mathtt{in}\; N)$$

Plus : lois spécifiques à certains effets.

Les lois pour l'état mutable

Les propriétés «de bonne variable» (lecture après écriture) :

$$\mathtt{set}(\ell, v;\ _.\, \mathtt{get}(\ell; z.\, M)) = \mathtt{set}(\ell, v;\ _.\, M\{z \leftarrow v\})$$

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

Autres commutations entre accès à des adresses différentes :

$$\mathtt{get}(\ell; y.\, \mathtt{get}(\ell'; z.\, M)) = \mathtt{get}(\ell'; z.\, \mathtt{get}(\ell; y.\, M))$$

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

Autres commutations entre accès à la même adresse :

$$\mathtt{get}(\ell; y.\, \mathtt{get}(\ell; z.\, M)) = \mathtt{get}(\ell; y.\, M\{z \leftarrow y\}) \quad \text{(double lecture)}$$

$$\mathtt{get}(\ell; y.\, \mathtt{set}(\ell, y;\ _.\, M)) = M \quad \text{(lire puis réécrire)}$$

$$\mathtt{set}(\ell, v_1; _.\, \mathtt{set}(\ell, v_2; _.\, M)) = \mathtt{set}(\ell, v_2; _.\, M) \quad \text{(double écriture)}$$

Les lois pour le non-déterminisme

Pour l'échec :

$$\texttt{Fail}(;k) = \texttt{Fail}(;k') = \texttt{Fail}() \qquad \text{(propagation)}$$

Pour le choix non-déterministe :

$$\begin{aligned}
\texttt{choose}\ M\ M &= M && \text{(idempotent)}\\
\texttt{choose}\ M\ N &= \texttt{choose}\ N\ M && \text{(commutatif)}\\
\texttt{choose}\ (\texttt{choose}\ M\ N)\ P &= \texttt{choose}\ M\ (\texttt{choose}\ N\ P) && \text{(associatif)}\\
\texttt{choose}\ \texttt{Fail}()\ M &= \texttt{choose}\ M\ \texttt{Fail}() = M && \text{(neutre)}
\end{aligned}$$

Moins naturel à exprimer avec le codage

$$\texttt{choose}\ M\ N = \texttt{Flip}(; \lambda b.\ \texttt{if}\ b\ \texttt{then}\ M\ \texttt{else}\ N)$$

Une sémantique pour le lambda-calcul computationnel avec effets

À chaque calcul on associe un arbre d'interaction / un terme de la monade plus libre.

$$\begin{aligned}
[\![v_1\ v_2]\!] &= [\![v_1]\!]_v\ [\![v_2]\!]_v \quad \text{ou} \quad \texttt{Tau}([\![v_1]\!]_v\ [\![v_2]\!]_v) \\
[\![\texttt{val}\ v]\!] &= \texttt{Pure}\ [\![v]\!]_v \\
[\![\texttt{do}\ x \Leftarrow M_1\ \texttt{in}\ M_2]\!] &= \texttt{bind}\ [\![M_1]\!]\ (\lambda x.\ [\![M_2]\!]) \\
[\![F(\vec{v}; y.\, M)]\!] &= \texttt{Op}\ (F\ \vec{v})\ (\lambda y.\ [\![M]\!])
\end{aligned}$$

On peut ensuite interpréter les effets par le *fold* approprié :

$$\begin{aligned}
\texttt{fold} &: (A \to B) \to (\forall C, \mathit{Eff}\ C \to (C \to B) \to B) \to R\ A \to B \\
\texttt{fold}\ f\ g\ (\texttt{Pure}\ v) &= f\ v \\
\texttt{fold}\ f\ g\ (\texttt{Op}\ \varphi\ k) &= g\ \varphi\ (\lambda x.\ \texttt{fold}\ f\ g\ (k\ x))
\end{aligned}$$

Interpréter les effets par composition de gestionnaires

Un *fold* peut reconstruire un arbre d'interaction au lieu de produire le résultat final de l'exécution. Cela lui permet de gérer seulement certains effets, et de réémettre tous les autres.

Exemple : un gestionnaire pour les effets `Get` et `Set`.

$$\begin{aligned}
\mathtt{state} : R\,A \to \mathit{Store} \to R\,A &= \mathtt{fold}\, f_{state}\, g_{state} \\
f_{state}\, v &= \lambda s.\, \mathtt{Pure}\, v \\
g_{state}\, (\mathtt{Get}\ \ell)\, k &= \lambda s.\, k\, (s\, \ell)\, s \\
g_{state}\, (\mathtt{Set}\ \ell\ v)\, k &= \lambda s.\, k\, ()\, s\{\ell \leftarrow v\} \\
g_{state}\, \varphi\, k &= \lambda s.\, \mathtt{Op}(\varphi, \lambda x.\, k\, x\, s) \quad \text{pour tout autre } \varphi
\end{aligned}$$

Interpréter les effets par composition de gestionnaires

Exemple : un gestionnaire pour les effets `Flip` et `Fail`.

$$
\begin{aligned}
\texttt{nondet} : R\,A \to R\,(Set\,A) &= \texttt{fold}\, f_{nondet}\, g_{nondet} \\
f_{nondet}\, v &= \texttt{Pure}\, \{v\} \\
g_{nondet}\, \texttt{Fail}\, k &= \texttt{Pure}\, \emptyset \\
g_{nondet}\, \texttt{Flip}\, k &= \texttt{bind}\, (k\, \texttt{true})\, (\lambda x_1. \\
&\quad\; \texttt{bind}\, (k\, \texttt{false})\, (\lambda x_2. \\
&\quad\; \texttt{Pure}\, (x_1 \cup x_2))) \\
g_{nondet}\, \varphi\, k &= \texttt{Op}(\varphi, k) \quad \text{pour tout autre } \varphi
\end{aligned}
$$

Interpréter les effets par composition de gestionnaires

La composition `nondet` $($`state` $t\ s_0)$ donne la sémantique de retour en arrière de l'état mémoire aux points de choix.

La composition `state` $($`nondet` $t)\ s_0$ donne la sémantique avec persistance de l'état mémoire.

Si l'arbre t ne contient pas d'autres effets que `Get`, `Set`, `Fail` et `Flip`, les deux compositions produisent un arbre trivial `Pure` v où v est la valeur finale du programme.

Implémenter correctement les équations

Les équations qui portent sur `bind` sont automatiquement satisfaites par la sémantique à base d'arbres d'interaction.

Les autres équations doivent être vérifiées par les gestionnaires qui interprètent les effets.

Les lois pour l'état mutable

Après passage aux arbres d'interaction et simplification par le gestionnaire `state`, les 7 lois pour l'état mutable sont impliquées par les 5 égalités suivantes (les 2 lois `get-get` sont triviales) :

$$
\begin{aligned}
s\{\ell \leftarrow v\}\ \ell &= v \\
s\{\ell \leftarrow v\}\ \ell' &= s\ \ell' \quad \text{si } \ell' \neq \ell \\
s\{\ell \leftarrow v\}\{\ell \leftarrow v'\} &= s\{\ell \leftarrow v'\} \\
s\{\ell \leftarrow v\}\{\ell' \leftarrow v'\} &= s\{\ell' \leftarrow v'\}\{\ell \leftarrow v\} \quad \text{si } \ell' \neq \ell \\
s\{\ell \leftarrow s\ \ell\} &= s
\end{aligned}
$$

Exercice : montrer que `nondet` vérifie les lois pour le non-déterminisme.

Programmer ses gestionnaires d'effets

On ajoute au lambda-calcul computationnel une construction pour définir des gestionnaires d'effets dans le langage lui-même.

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

Calculs :
$$
\begin{aligned}
M, N ::=\ & v\ v' \\
\mid\ & \texttt{if}\ v\ \texttt{then}\ M\ \texttt{else}\ N \\
\mid\ & \texttt{val}\ v \\
\mid\ & \texttt{do}\ x \Leftarrow M\ \texttt{in}\ N \\
\mid\ & F(\vec{v}; y.\, M) && \text{opération avec effet} \\
\mid\ & \texttt{with}\ H\ \texttt{handle}\ M && \text{gestion des effets}
\end{aligned}
$$

Gestionnaires :
$$
\begin{aligned}
H ::= \{\ & \texttt{val}(x) \to M_{val}; \\
& F_1(\vec{x}; k) \to M_1; \\
& \dots \\
& F_n(\vec{x}; k) \to M_n\ \}
\end{aligned}
$$

Sémantique intuitive des gestionnaires d'effets

$$\texttt{with}\ \{\texttt{val}(x) \to M_{val}\ ; \ldots ; F_i(\vec{x}; k) \to M_i\ ; \ldots\}\ \texttt{handle}\ M$$

Si M termine sur la valeur v, le cas M_{val} est évalué avec $x = v$.

Si M lance l'effet $F_i(\vec{v}; y.\, N)$, le cas M_i est évalué avec $\vec{x} = \vec{v}$ et
$k = \lambda y.\, N$ ou $k = \lambda y.\, \texttt{with}\ \{\ldots\}\ \texttt{handle}\ N$.
(gestionnaire superficiel) *(gestionnaire profond)*

Si M lance un autre effet $F(\vec{v}; y.\, N)$, avec $F \notin \{F_1, \ldots, F_n\}$, on relance l'effet $F(\vec{v}; y.\, N)$ ou l'effet $F(\vec{v}; y.\, \texttt{with}\ \{\ldots\}\ \texttt{handle}\ N)$.
(gestionnaire superficiel) *(gestionnaire profond)*

Sémantique dénotationnelle des gestionnaires d'effets

La dénotation $[\![H]\!]$ d'un gestionnaire d'effet est un transformateur arbre d'interaction $\to$ arbre d'interaction, de sorte que

$$[\![\texttt{with}\ H\ \texttt{handle}\ M]\!] = [\![H]\!]\ [\![M]\!]$$

Ce transformateur est un *fold* dans le cas d'un gestionnaire profond et une analyse de cas pour un gestionnaire superficiel :

$$[\![H]\!] = \begin{cases} \texttt{fold}\ [\![H]\!]_{ret}\ [\![H]\!]_{eff} & (\text{cas} \ll\text{profond}\gg) \\ \texttt{case}\ [\![H]\!]_{ret}\ [\![H]\!]_{eff} & (\text{cas} \ll\text{superficiel}\gg) \end{cases}$$

`fold` et `case` sont définis par

$$\texttt{fold}\ f\ g\ (\texttt{Pure}\ v) = f\ v \qquad \texttt{case}\ f\ g\ (\texttt{Pure}\ v) = f\ v$$

$$\texttt{fold}\ f\ g\ (\texttt{Op}(\varphi, k)) = g\ \varphi\ (\lambda x.\ \texttt{fold}\ f\ g\ (k\ x)) \qquad \texttt{case}\ f\ g\ (\texttt{Op}(\varphi, k)) = g\ \varphi\ k$$

Sémantique dénotationnelle des gestionnaires d'effets

$$H = \{\mathtt{val}(x) \to M_{val} \,;\, F_1(\vec{x}; k) \to M_1 \,;\, \ldots \,;\, F_n(\vec{x}; k) \to M_n\}$$

On définit les sémantiques pour le retour normal et pour le retour sur effet comme suit :

$$\begin{aligned} [\![H]\!]_{ret}\; x &= [\![M_{val}]\!] \\ [\![H]\!]_{eff}\; (F_i\; \vec{x}\,)\; k &= [\![M_i]\!] \\ [\![H]\!]_{eff}\; (F\; \vec{x}\,)\; k &= \mathtt{Op}\; (F\; \vec{x}\,)\; k \quad \text{si } F \notin \{F_1, \ldots, F_n\} \end{aligned}$$

Point d’étape

Point d'étape

Deux étapes vers une théorie générale des effets dans les langages de programmation.

- Les monades :
  - «Mettent de l'ordre» dans les sémantiques dénotationnelles et les transformations de programmes fonctionnels.
  - La programmation en style monadique généralise la programmation en CPS et permet d'utiliser des effets non supportés par le langage de programmation.
- Les effets algébriques :
  - Spécification des effets par des équations.
  - Implémentation par des gestionnaires d'effets, qui sont des *fold* ou des *case* sur les arbres d'interaction.
  - Les gestionnaires peuvent être définis dans le langage de programmation.

Bibliographie

Bibliographie

Une introduction à la programmation monadique :

- Philip Wadler : *Monads for functional programming*. 1995. http://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/baastad.pdf

Une introduction aux effets et aux gestionnaires d'effets :

- Matija Pretnar : *An Introduction to Algebraic Effects and Handlers*, ENTCS 319, 2015. https://doi.org/10.1016/j.entcs.2015.12.003

La monade plus libre :

- Oleg Kiselyov, Hiromi Ishii : *Freer monads, more extensible effects*. Haskell 2015 : 94-105. https://doi.org/10.1145/2804302.2804319

Le point de vue algébrique :

- Andrej Bauer : *What is algebraic about algebraic effects and handlers ?* 2018. https://arxiv.org/abs/1807.05923