Effets et gestionnaires d'effets en OCaml 5
Deuxième partie: Un peu de théorie

Xavier Leroy

Journées Francophones des Langages Applicatifs, 2023-02-02

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

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. (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, …

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.

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

La dénotation d'une expression $e : T$ avec effets est une fonction $S \to T \times S$, état «avant» $\mapsto$ (valeur, état «après»).

Remarque : permet aussi 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 un 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 ?

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 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 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 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)

Opérations de base sur les calculs

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.

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

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

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

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

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)

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

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

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 :

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

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*) : $T\ A = Env \to A$

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

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

$\texttt{getenv} = \lambda e.\ e$

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

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

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

$\texttt{log}\ s = (\texttt{tt}, s)$

Des monades pour la programmation probabiliste

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

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

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

$\texttt{choose}\ p\,a\,b = \{(x, p \times p_1) \mid (x, p_1) \in a\}$
$\qquad\qquad \cup \{(y, (1-p) \times p_2) \mid (y, p_2) \in b\}$

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

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

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 $\texttt{bind}\ \ M\ (\lambda x.\, N)$.

Les lois du lambda-calcul computationnel

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

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

$$\texttt{let}\ x \Leftarrow \texttt{val}\ M\ \texttt{in}\ N = N\{x \leftarrow M\} \qquad (\beta\text{-mon})$$

$$\texttt{let}\ x \Leftarrow M\ \texttt{in}\ \texttt{val}\ x = M \qquad (\eta\text{-mon})$$

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

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

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

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

$$[\![cst]\!]_v = \texttt{val}\ cst$$

$$[\![\lambda x.\, M]\!]_v = \texttt{val}(\lambda x.\, [\![M]\!]_v)$$

$$[\![x]\!]_v = \texttt{val}\ x$$

$$[\![M\ N]\!]_v = \texttt{let}\ f \Leftarrow [\![M]\!]_v\ \texttt{in}\ \texttt{let}\ a \Leftarrow [\![N]\!]_v\ \texttt{in}\ f\ a$$

Appel par nom

$$[\![cst]\!]_n = \texttt{val}\ cst$$

$$[\![\lambda x.\, M]\!]_n = \texttt{val}(\lambda x.\, [\![M]\!]_n)$$

$$[\![x]\!]_n = x$$

$$[\![M\ N]\!]_n = \texttt{let}\ f \Leftarrow [\![M]\!]_n\ \texttt{in}\ f\ [\![N]\!]_n$$

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, de la génération des effets par les opérations de base de la monade ? 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

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).

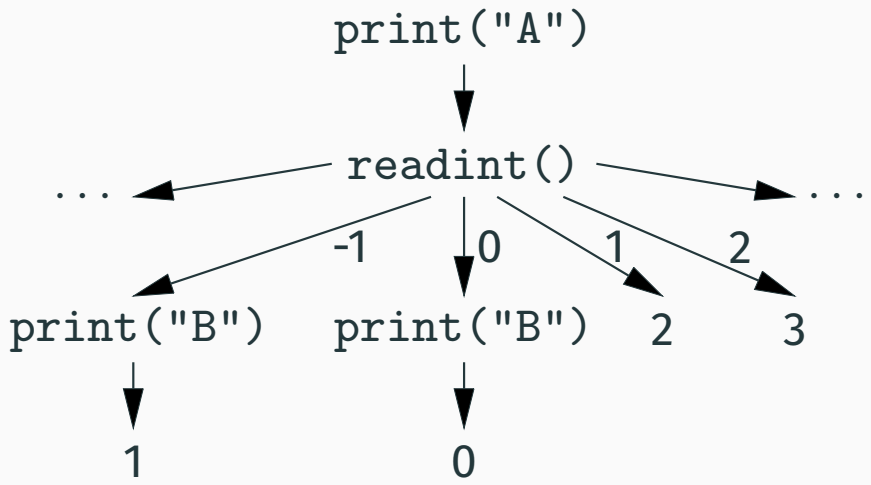
Équivalence sémantique : $op(\vec{v}, y.\, M) = \texttt{let}\ y \Leftarrow op(\vec{v}\,)\ \texttt{in}\ M$.

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 les équations suivantes :

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

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

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

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

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

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

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

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).

On peut donner 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) :

$$\texttt{try}\ a\ \texttt{with}\ x\ k \to \texttt{if}\ \ldots\ \texttt{then}\ k\ 0\ \texttt{else}\ b$$

(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
	$\mid v\, v'$	application
	$\mid op(\vec{v}; y.\, M)$	opération avec effet
	$\mid \texttt{with}\ H\ \texttt{handle}\ M$	gestionnaire d'effets
Gestionnaires :	$H ::= \{\ \texttt{val}(x) \to M_{val};$	
	$\quad op_1(\vec{x}; k) \to M_1;$	
	$\quad \ldots$	
	$\quad op_n(\vec{x}; k) \to M_n\ \}$	

Les gestionnaires d'effets

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

Sémantique intuitive :

- si M évalue $op_i(\vec{v}; y.\, N)$, le cas M_i est évalué avec $\vec{x} = \vec{v}$ et
  $k = \lambda y.\, N$ (*shallow handler*) ou
  $k = \lambda y.\, \texttt{with}\{\ldots\}\ \texttt{handle}\ \ N$ (*deep handler*)
- si M termine sur la valeur v, le cas M_{val} est évalué avec $x = v$.

Sémantique opérationnelle (de type *deep handling*)

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

$$\texttt{let}\ x \Leftarrow \texttt{val}\ v\ \texttt{in}\ N \to N\{x \leftarrow v\}$$

$$\texttt{let}\ x \Leftarrow \texttt{op}(v; y.M)\ \texttt{in}\ N \to \texttt{op}(v, y.\texttt{let}\ x \Leftarrow M\ \texttt{in}\ N)$$

$$\texttt{let}\ x \Leftarrow M\ \texttt{in}\ N \to \texttt{let}\ x \Leftarrow M'\ \texttt{in}\ N \qquad \text{si } M \to M'$$

En notant $H = \{\texttt{val}(x) \to M_{val}\ ; \ldots ; op_i(\vec{x}; k) \to M_i\ ; \ldots\}$:

$$\texttt{with}\ H\ \texttt{handle}\ M \to \texttt{with}\ H\ \texttt{handle}\ M' \qquad \text{si } M \to M'$$

$$\texttt{with}\ H\ \texttt{handle}\ (\texttt{val}\ v) \to M_{val}\{x \leftarrow v\}$$

$$\texttt{with}\ H\ \texttt{handle}\ op_i(\vec{v}; y.M) \to M_i\{\vec{x} \leftarrow \vec{v}, k \leftarrow \lambda y.\ \texttt{with}\ H\ \texttt{handle}\ M\}$$

$$\texttt{with}\ H\ \texttt{handle}\ op(\vec{v}; y.M) \to op(\vec{v}, y.\ \texttt{with}\ H\ \texttt{handle}\ M)$$

$$\text{si } op \notin \{op_1, \ldots, op_n\}$$

Exemples de gestionnaires d'effets

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

Gestion d'exceptions :

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

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) }
 handle a) s0
```

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

(Exercice : implémenter et tester ≪état mutable≫ en OCaml 5.)

Exemples de gestionnaires d'effets

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

(Les exemples suivants utilisent les continuations `k` plusieurs fois et ne peuvent donc pas s'exprimer en OCaml 5.)

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) }
```

Continuations (délimitées par le `with`) :

```
with { val(x) → val(x);
       callcc(f; k) → k (f k) }
```

Typage statique des effets

Systèmes de types et d'effets

Une série de travaux remontant aux années 1980 qui vise à refléter dans les types et à vérifier statiquement

- non seulement la forme de la valeur qui est calculée,
- mais aussi les effets possiblement produits par son calcul (exceptions, affectations, non-terminaison, …)

Exemple simple : les clauses `throws` en Java

```
public static void writeToFile() throws IOException {
  ...
}
```

Un système de types et d'effets simples

Type de valeur : $\tau, \sigma ::= \texttt{int} \mid \texttt{bool} \mid \tau \to \kappa$

Type de calcul : $\kappa ::= \tau / \varphi$

Type d'effet : $\varphi ::= \{op_1, \ldots, op_n\}$ (ensemble d'opérations)

Typage des valeurs :

$$\Gamma \vdash x : \Gamma(x) \qquad \frac{\Gamma, x : \tau \vdash M : \kappa}{\Gamma \vdash \lambda x.M : \tau \to \kappa}$$

Typage des calculs :

$$\frac{\Gamma \vdash v : \tau}{\Gamma \vdash \texttt{ret}\ \ v : \tau / \varphi} \qquad \frac{\Gamma \vdash v : \tau \to \kappa \quad \Gamma \vdash v' : \tau}{\Gamma \vdash v\, v' : \kappa}$$

Un système de types et d'effets simples

Typage des calculs :

$$\frac{\Gamma \vdash M : \sigma/\varphi \qquad \Gamma, x : \sigma \vdash M' : \tau/\varphi}{\Gamma \vdash \texttt{let } x \Leftarrow M \texttt{ in } M' : \tau/\varphi}$$

$$\frac{op : \sigma_{op} \to \tau_{op} \qquad \Gamma \vdash v : \sigma_{op} \qquad \Gamma, y : \tau_{op} \vdash M : \tau/\varphi \qquad op \in \varphi}{\Gamma \vdash op(v, y.M) : \tau/\varphi}$$

$$\frac{\begin{array}{c}\Gamma \vdash M : \tau/\varphi \qquad op : \sigma_{op} \to \tau_{op} \qquad \Gamma, x : \tau \vdash M_{val} : \tau'/\varphi' \\ \Gamma, x : \sigma_{op}, k : \tau_{op} \to \tau'/\varphi' \vdash M_{op} : \tau'/\varphi' \qquad \varphi \setminus \{op\} \subseteq \varphi'\end{array}}{\Gamma \vdash \texttt{with } \{\texttt{ret}(x) \to M_{val}; \texttt{op}(x, k) \to M_{op}\} \texttt{ handle } M : \tau'/\varphi'}$$

(On suppose que chaque opération est déclarée globalement avec un type unique $op : \sigma_{op} \to \tau_{op}$.)

Vers un typage polymorphe des effets

On voudrait bien que

- `List.map` f l ait les mêmes effets que f
- `with` $\{\texttt{Exit}(\_,\_) \to 0\}$ `handle` f 0
  ait les mêmes effets que f moins l'effet `Exit`.

Une possibilité : au lieu d'ensembles d'effets, utiliser des rangées (comme pour les objets et les variants polymorphes d'OCaml) :

Type d'effet : $\varphi ::= \emptyset \mid op; \varphi \mid \rho$ (variable de rangée)

Le genre de typages qu'on espère :

$$\texttt{List.map} : \forall \alpha\beta\rho, (\alpha \to \beta/\rho) \to \alpha\ \texttt{list} \to \beta\ \texttt{list}/\rho$$

$$\lambda f.\ \texttt{with}\ \{\texttt{Exit}(\_,\_) \to 0\}\ \texttt{handle}\ f\ 0 : \forall \rho, (\texttt{int} \to \texttt{int}/(\texttt{Exit}; \rho)) \to \texttt{int}/\rho$$

Autres difficultés

Prendre en compte la génération dynamique des effets.

(Tout comme pour les exceptions, plusieurs effets différents peuvent avoir le même nom. Voir P. Emílio de Vilhena et F. Pottier, *A type system for effect handlers and dynamic labels*, ESOP 2023.)

Garder les types lisibles et pas trop difficiles à écrire dans les interfaces de modules.

(Voir p.ex. le langage Koka de D. Leijen et al, `https://koka-lang.github.io/`.)

Conclusion

Amusez-vous bien
avec plein d'effets
en style direct!