Automates, machines, moteurs réactifs

Gérard Huet

Notes de cours MPRI, 2008
October 15, 2008

1 Automates

1.1 Automate sur un monoïde d'actions

Soit $\mathcal{S} = \langle S, \cdot, 1\rangle$ un monoïde de support un ensemble S d'éléments appelés *actions*, muni d'une opération associative notée $\cdot$ appelée *produit* et d'un élément 1 neutre à gauche et à droite pour ce produit.

On appelle $\mathcal{S}$-automate un tuple $\langle Q, I, T, \delta\rangle$ où :

- Q est un ensemble fini d'*états*
- $I \in Q$ est l'ensemble des états *initiaux*
- $T \in Q$ est l'ensemble des états *terminaux*
- $\delta \in Q \to \wp(S \times Q)$ est la *relation de transition*, qui associe à tout état un ensemble *fini* de paires (a, q) formées d'une action a et d'un état q.

On appelle *support* de l'automate $\mathcal{A} = \langle Q, I, T, \delta\rangle$ l'ensemble fini d'actions $\Phi_{\mathcal{A}} = \{a \in S \mid \exists q, q' \in Q\ (a, q') \in \delta(q)\}$.

On appelle *inverse* de l'automate $\mathcal{A} = \langle Q, I, T, \delta\rangle$ l'automate de même support $\langle Q, T, I, \delta'\rangle$ tel que $\delta'(q') = \{(a, q) \mid (a, q') \in \delta(q)\}$, noté $\tilde{\mathcal{A}}$. On a bien sûr $\widetilde{\tilde{\mathcal{A}}} = \mathcal{A}$.

1.2 Comportement d'un automate

On appelle *parcours* de l'automate $\mathcal{A} = \langle Q, I, T, \delta\rangle$ une séquence

$$p = q_0 \stackrel{a_1}{\to} q_1 \stackrel{a_2}{\to} ...q_2 \stackrel{a_n}{\to} q_n \quad (n \geq 0)$$

avec $\forall i \geq 0\ q_i \in Q \wedge (a_{i+1}, q_{i+1}) \in \delta(q_i)$. On définit l'*action* associée au parcours p comme $act(p) = 1$ si $n = 0$, et $act(p) = a_1 \cdot ... \cdot a_n$ sinon. Le parcours est dit *acceptant* si $q_0 \in I$ et $q_n \in T$, et on note $pa(A)$ pour l'ensemble des parcours acceptants de A. On appelle *comportement* de l'automate $\mathcal{A}$ l'ensemble $|\mathcal{A}| = \{act(p) \mid p \in pa(\mathcal{A})\}$.

On dit que q_n est $\mathcal{A}$-*accessible* à *partir de* q_0, et que q est $\mathcal{A}$-*accessible* s'il est $\mathcal{A}$-accessible à partir d'un état initial de $\mathcal{A}$. On dit que q est $\mathcal{A}$-*co-accessible* s'il est $\bar{\mathcal{A}}$-accessible, et $\mathcal{A}$-*utile* s'il est $\mathcal{A}$-*accessible* et $\mathcal{A}$-*co-accessible*.

On dit que $\mathcal{A}$ est *émondé* ssi tous ses états sont utiles. Tout automate peut être réduit en un automate émondé de même comportement.

1.3 Exemples

Le monoïde des actions peut être le monoïde libre Σ^* engendré par un alphabet fini Σ. Un Σ^*-automate est alors la généralisation d'un automate fini non déterministe, où on permet d'étiqueter une transition par un mot arbitraire, et non seulement une lettre. Un cas particulier est celui des automates avec transition spontanée (ϵ-move).

On obtient les transducteurs d'un alphabet Σ dans un alphabet Σ' en considérant le monoïde (non-libre) produit des monoïdes libres Σ^* et Σ'^*.

2 Machines

2.1 Relations comme actions

Une relation entre les ensembles D et D' est un ensemble de paires de $D \times D'$. Toute relation $\rho \in \wp(D \times D')$ se currifie en une function de D dans $\wp(D')$ par une correspondance bijective. Nous écrirons $\rho : D \to D'$ pour $\rho \in \wp(D \times D')$ vue comme fonction, et nous noterons, pour tout $d \in D$, $\rho(d)$ pour $\{d' \mid (d, d') \in \rho\}$. On notera $\tilde{\rho} : D' \to D$ pour la relation inverse $\{(d', d) \mid (d, d') \in \rho\}$.

On écrit pour l'ensemble des endo-relations de D: $Rel(D) = \wp(D \times D)$. $Rel(D)$ est muni d'une structure de monoïde multiplicatif, le produit étant la composition des relations:

$$\rho \cdot \rho' = \{(d, d'') \mid \exists d' \ d' \in \rho(d) \wedge d'' \in \rho'(d')\}$$

et l'élément neutre étant la relation identité $1 = \{(d, d) \mid d \in D\}$. $Rel(D)$ est aussi muni d'une structure de monoïde additif, commutatif et idempotent, la somme $+$ étant l'union, et l'élément neutre 0 étant la relation vide. On peut également définir un itérateur de Kleene $*$ par la fermeture réflexive transitive d'une relation, ce qui permet d'interpréter les expressions rationnelles dans $Rel(D)$ vue comme une algèbre de Kleene.

Mieux encore, en définissant les opérateurs d'implicature ou semi-complément:

$$\rho \to \sigma = \{(v, w) \mid \forall u \ u\rho v \Rightarrow u\sigma w\}$$

$$\sigma \leftarrow \rho = \{(u, w) \mid \forall v \ w\rho v \Rightarrow u\sigma v\}$$

on peut voir $Rel(D)$ comme une *algèbre d'actions* au sens de Pratt [6]. Rappelons que les algèbres d'actions forment une variété (c'est-à-dire sont axiomatisables équationnellement) au contraire des algèbres de Kleene, qui ne forment qu'une quasi-variété (c'est-à-dire nécessitent pour leur axiomatisaton des équations conditionnelles) [5]. Les opérations de semi-complément peuvent être vues comme des interpolants permettant la complétude du raisonnement équationnel. Elles ont aussi l'avantage de formaliser une notion logique d'implication entre relations, utile pour décrire des transductions conditionnelles. Les algèbres d'action semblent donc un compromis intéressant entre les algèbres de Kleene servant à interpréter les expressions rationnelles et le modèle relationnel de la théorie des ensembles.

On peut ainsi calibrer la théorie axiomatique des actions pour accommoder plus ou moins des propriétés des relations. Par exemple, se donner une opération involutive d'inverse, permettant d'accommoder à la fois l'inverse d'une relation $\tilde{\rho}$ et l'image-miroir d'un mot sur un alphabet fini. Néanmoins, voir la difficulté dans Pratt.

Enfin, on peut compléter les algèbres d'actions par une opération limite autorisant des unions infinies. Dans l'algèbre des relations, ceci correspond à associer à une famille R de relations la relation union :

$$U(R) = \bigcup_{\rho \in R} \rho$$

Les algèbres relationnelles sont ainsi des *algèbres d'actions continues* - ce sont des ordres partiels complets au sens de Scott, la relation d'approximation étant la partie, et la continuité la compacité.

Bien sûr, $Rel(D)$ possède de nombreuses autres propriétés, et par exemple l'ordre partiel défini par $a \leq b$ ssi $a + b = b$ lui donne une structure d'algèbre Booléenne complète. Mais les machines que nous considérons n'ont pas besoin d'opérer sur des algèbres d'actions aussi riches. Par exemple, si la somme de deux actions n'est pas idempotente, nous pouvons remplacer la notion de relation comme ensemble de paires par une notion plus fine de relation représentée par un multi-ensemble de paires, ou même par une séquence si l'on oublie aussi la commutativité.

2.2 Relations calculables

Nous allons maintenant nous restreindre à des relations calculables, et pour ce faire d'abord se restreindre à des espaces de données (D ci-dessus) récursivement énumérables représentés par des flux calculables. Un flux sur un espace de données D (anglais *stream*) est soit *Vide* soit une paire *Stream(d,f)* composée d'un élément d de D et d'une fonction partielle f de $U \rightarrow D$, où U est le type de données trivial réduit à un élément noté (). Ces fonctions partielles permettent le calcul progressif d'un sous ensemble récursivement énumérable de D potentiellement infini. La valeur *Vide* permet de distinguer les ensembles finis.

On définit, pour tout flux ϕ sur le domaine D, l'ensemble $\overline{\phi}$ comme suit:

$$\overline{Vide} = \emptyset$$

$$\overline{Stream(d, f)} = \{d\} \cup \overline{f()}$$

On dit que le flux ϕ est *progressif* si soit $\phi = Vide$, soit $\phi = Stream(d, f)$, avec f une fonction totale telle que $f()$ est un flux progressif. Autrement dit, toutes les fonctions intervenant dans le calcul de l'ensemble $\overline{\phi}$ terminent. Les flux progressifs permettent de définir des ensembles calculables (i.e. récursivement énumérables) dont la vacuité est décidable.

On appelle *domaine de calcul* un ensemble énuméré par un flux progressif.

On appelle *relation calculable* sur le domaine de calcul D une endo-relation de D définie par une fonction totale ρ associant à tout $d \in D$ un flux progressif $\rho(d)$ de valeurs de D. Autrement dit, $(x, y) \in \rho$ ssi $y \in \overline{\rho(x)}$.

2.3 Machines d'Eilenberg

Une machine d'Eilenberg calculant sur le domaine de calcul D est définie comme un $\mathcal{S}$-automate, où le monoïde d'actions $\mathcal{S}$ est pris comme l'ensemble des relations calculables sur D muni de la composition et de l'identité [1].

Le comportement d'une telle machine $\mathcal{M}$ est un ensemble $|\mathcal{M}|$ de relations calculables - obtenues par composition des relations étiquetant un parcours acceptant. On peut énumérer ces parcours pour obtenir un flux de relations.

La *relation caractéristique* d'une machine $\mathcal{M}$ est l'union de ses comportements:

$$||\mathcal{M}|| = \bigcup_{\rho \in |\mathcal{M}|} \rho$$

Les relations caractéristiques interprètent relationnellement le langage reconnu par l'automate.

2.4 Machines d'Eilenberg modulaires

On obtient la notion de machine modulaire en étiquetant les transitions de l'automate par les relations caractéristiques des machines paramètres d'une machine ainsi définie comme un foncteur. En termes de langages formels, on instancie la description d'une machine syntaxique définie sur un alphabet Σ de relations formelles par un morphisme de renommage (ang. relabeling). Chaque paramètre formel est instancié par la relation caractéristique de la machine correspondante.

Ceci nous amène donc à définir les machines d'Eilenberg syntaxiques relativement à un alphabet Σ , comme des Σ^*-automates particuliers, où les actions étiquetant les transitions sont réduites à un élément de Σ. Notons que ce sont exactement les automates finis non-déterministes traditionnels. On appelle donc foncteur d'Eilenberg, ou machine d'Eilenberg modulaire, la paire (Σ, M) où M est une machine syntaxique relativement à Σ. Si on se donne un contexte d'instanciation comme une table $\overline{N}$ associant une machine d'Eilenberg N_a à toute action $a \in \Sigma$, alors on peut définir la machine instance du foncteur par cette application:

$$App(\Sigma, M)\ \overline{N}$$

En effet, il suffit d'instancier la relation de transition de M en y remplaçant toute paire (a, q) par la paire $(||N_a||, q)$. Si N_a est une machine de domaine de calcul D_a, la machine ainsi obtenue est une machine de domaine de calcul $\bigcup_{a \in \Sigma} D_a$. En particulier, si les N_a ont toutes le même domaine de calcul D, la machine composée a aussi ce domaine de calcul.

2.5 Interfaces

Ce que nous avons défini jusqu'à présent est le *noyau* d'une machine d'Eilenberg, définissant ses données et son contrôle. La description se complète par la donnée d'une *interface*, composée d'un domaine d'input D_-, d'un domaine d'output

D_+, d'une relation d'input ϕ_- et d'une relation d'output ϕ_+. La machine M complétée par cette interface I définit une relation $\phi(M, I) : D_- \to D_+$ par composition :

$$\phi(M, I) = \phi_- \circ ||\mathcal{M}|| \circ \phi_+$$

2.6 Calculs d'une machine d'Eilenberg

L'idée est de remplacer l'état d'un automate muni d'une bande de calcul par une *cellule* (d, q) où d est une donnée du domaine de calcul D de la machine, et q un état de son espace d'états Q. La cellule est dite *initiale* (resp. *finale*) si q est un état initial (resp. terminal).

Une *étape de calcul* est une transition avec calcul sur la donnée :

$$(d, q) \xrightarrow{\rho} (d', q')$$

où $(\rho, q') \in \delta(q)$ et $d' \in \rho(d)$. Notons qu'il y a deux choix non-déterministes successifs; le premier est sur le contrôle, le deuxième est sur la donnée. Le premier choix est déterministe si l'automate sous-jacent est déterministe, le deuxième choix l'est si la relation sélectionnée est fonctionnelle. Une cellule (d, q) est dite *bloquante* si pour tout $(\rho, q') \in \delta(q)$ on a $\rho(d) = \emptyset$.

Un *calcul* est une suite d'étapes de calcul issue d'une cellule initiale. On définit l'ensemble des données *permises* à l'état q comme le sous-ensemble $D(q)$ des élements $d \in D$ tels qu'il existe un calcul se terminant en (d, q). L'ensemble des données $D(\mathcal{M})$ calculables par une machine $\mathcal{M}$ est l'union des $D(q)$ pour q état terminal. C'est l'image de D par la relation caractéristique $||\mathcal{M}||$.

Donnons maintenant quelques définitions permettant de limiter la puissance du formalisme vers des machines plus finitistes.

La relation $\rho : D \to D'$ est dite *localement finie* si pour tout $d \in D$ l'ensemble $\rho(d)$ est fini. La machine M est dite *localement finie* si toute relation $\rho \in \Phi_M$ est localement finie. La machine M est dite *nœthérienne* si tous ses calculs sont finis.

Remarquons qu'une machine est nœthérienne si son domaine de calcul D est un ordre bien fondé par la relation d'ordre $>$ engendrée par :

$$d > d' \;\Leftarrow\; \exists \rho \in \Phi(M)\ d' \in \rho(d)$$

En effet, s'il existe une suite de calcul infinie alors il existe une sous-suite infinie passant par le même état q. Par contre, la réciproque n'est pas vraie, car une machine peut terminer pour une raison dépendant essentiellement de son contrôle.

Enfin, la machine M est dite *finie* si elle est localement finie et nœthérienne.

On dit qu'une machine est *localement déterministe* ("sequential" in Eilenberg [1]) ssi pour toute cellule (d, q) apparaissant dans un calcul il existe au plus une transition de calcul issue de cette cellule, c'est-à-dire si $\delta(q)$ est un ensemble de paires $\{(\rho_1, q_1), (\rho_2, q_2), ...(\rho_n, q_n)\}$ tel que pour au plus un $1 \leq k \leq n$ $\rho_k(d)$ est non vide, et que si ce k existe alors $\rho_k(d)$ est singleton. Cette condition exige que

d'une part la relation de transition de l'automate sous-jacent soit une fonction partielle, et que d'autre part les relations issues d'un état q soient des fonctions partielles sur $D(q)$. Remarquez qu'une machine localement déterministe peut néanmoins engendrer plusieurs solutions, puisqu'une cellule terminale n'est pas forcément bloquante.

2.7 Exemples

Traductions On traduit un formalisme calculatoire, et un problème à résoudre vis-à-vis de ce formalisme, par l'intermédiaire d'une traduction, comme suit.

Une *traduction* (eng. relabeling) entre un $\mathcal{S}_1$-automate $\mathcal{A}_1$ et un $\mathcal{S}_2$-automate $\mathcal{A}_2$ est donnée par un morphisme de monoïde $\alpha : S_1 \to S_2$ étendu en ré-étiquetage des transitions de $\mathcal{A}_1$ en transitions de $\mathcal{A}_2$. Il est facile de montrer que $|\mathcal{A}_2| = \alpha(|\mathcal{A}_1|)$.

Notez que

$$App(\Sigma, M)\ \overline{N}$$

est une traduction de M, α étant défini dans ce cas comme l'extension unique de $\overline{N}$ en un morphisme du monoïde libre Σ^* en le monoïde des relations sur son domaine de calcul.

Automates finis non-déterministes Un automate fini non déterministe (NFA) sur l'alphabet Σ est un Σ^*-automate $\mathcal{A}$ tel que $\Phi_{\mathcal{A}} \subset \Sigma$. C'est ce que nous avons appelé plus haut une machine d'Eilenberg syntaxique. Son comportement $|\mathcal{A}|$ est un langage rationnel sur Σ^*. On traduit un NFA (Q, I, T, δ) en une machine d'Eilenberg $\mathcal{M}$ qui résout le problème du mot (appartenance) comme suit. $D = \Sigma^*$. La traduction α est définie par $\alpha(a) = L_a^{-1} =_{def} \{(a \cdot w, w) \mid w \in \Sigma^*$. On a : $||\mathcal{M}|| = \{(w \cdot w', w') \mid w \in |\mathcal{A}|,\ w' \in \Sigma^*\}$ $\mathcal{M}$ est une machine finie. En effet, $\alpha(a)$ est une fonction partielle, donc on a la finitude locale. Une étape de calcul est de la forme $(d, q) \xrightarrow{\rho} (d', q')$ avec $|d'| = |d| - 1$ et donc tous les calculs sont finis.

On complète ce noyau par une interface $D_- = \Sigma^*$, $\phi_- = Id_{\Sigma^*}$, $D_+ = B = \{0, 1\}$, $\phi_+(w) = 1$ *ssi* $w = \epsilon$. On a bien $\rho(w) = 1$ ssi $w \in |\mathcal{A}|$.

Notez que si l'automate est déterministe alors la machine associée l'est aussi.

Transducteurs rationnels Soient Σ et Γ deux alphabets finis. Un transducteur $\Sigma \Rightarrow \Gamma$ est un $\mathcal{M}$-automate avec $M = \Sigma^* \times \Gamma^*$ (monoïde non libre, produit terme à terme). On prend $\Phi_{\mathcal{A}} \subset (\Sigma \times \epsilon) \cup (\epsilon \times \Gamma)$. $|\mathcal{A}|$ définit une *relation rationnelle*. On donne des traductions en $\mathcal{M}$-machines pour résoudre 3 problèmes:

1. Appartenance. Donné $(w, w') \in M$. Décider si $(w, w') \in |\mathcal{A}|$.
2. Synthèse. Donné $w \in \Sigma^*$. Calculer l'image $|\mathcal{A}|(w) \subset \Gamma^*$.
3. Analyse. Donné $w \in \Gamma^*$. Calculer l'image inverse $|\mathcal{A}^{-1}|(w) \subset \Sigma^*$

Appartenance. La traduction α est définie par $\alpha(\sigma, \epsilon) = L_\sigma^{-1} \times Id_{\Gamma^*}$ et $\alpha(\epsilon, \gamma) = Id_{\Sigma^*} \times L_\gamma^{-1}$. Comme pour les automates on a une machine finie, puisque

le long d'une transition le couple des longueurs décroît. On prend comme interface $D_- = \Sigma^* \times \Gamma^*$, $\phi_- = Id_{\Sigma^*} \times \Gamma^*$, $D_+ = B$, $\phi_+(w, w') = 1$ *ssi* $w = w' = \epsilon$.

Synthèse. La traduction α est définie par $\alpha(\sigma, \epsilon) = L_\sigma^{-1} \times Id_{\Gamma^*}$ et $\alpha(\epsilon, \gamma) = Id_{\Sigma^*} \times R_\gamma$, avec $R_\gamma =_{def} \{(w, w \cdot \gamma) \mid w \in \Gamma^*\}$. On complète par l'interface $D_- = \Sigma^*$, $\phi_- = \{(w, (w, \epsilon)) \mid w \in \Sigma^*\}$, $\phi_+ = \{(\epsilon, w'), w') \mid w' \in \Gamma^*\}$. On a $|\mathcal{A}| = \phi_- \circ ||\mathcal{M}|| \circ \phi_+$. Une telle machine est localement finie trivialement. Mais la machine n'est pas forcément Nœthérienne, il peut y avoir des cycles de transitions étiquetés avec des actions (ϵ, w). C'est d'ailleurs la seule manière d'avoir des calculs infinis, et la machine est nœthérienne si et seulement si il n'y a pas de tels cycles, si et seulement si l'ensemble $|\mathcal{A}|(w)$ est fini pour tout $w \in \Sigma^*$ [7].

Analyse. Symétrique de la synthèse, en remplaçant L_σ^{-1} par R_σ et R_γ par L_γ^{-1}.

Machines oracles Soit un ensemble D arbitraire, et un prédicat P sur D. On considère la relation ρ sur D définie comme la restriction de l'identité aux éléments vérifiant P: $\rho(d) = \{d\}$ si (d), $\rho(d) = \emptyset$ sinon. On définit de manière canonique la machine d'Eilenberg dont l'automate sous-jacent $\mathcal{A}$ a deux états $Q = \{0, 1\}$ avec $I = \{0\}$ et $T = \{1\}$, la fonction de transition δ étant définie par $\delta(0) = \{(\rho, 1)\}$ et $\delta(0) = \emptyset$. Cette machine est une machine finie localement déterministe, qui sait décider en un pas de calcul de l'appartenance à P. Notre restriction des machines d'Eilenberg à des relations calculables limite ces oracles à des prédicats récursifs, mais d'une complexité arbitraire. Plus généralement, nos machines d'Eilenberg savent énumérer des ensembles récursivement énumérables arbitraires, et ont donc la puissance des machines de Turing.

3 Moteurs réactifs terminaux

On peut simuler les exécutions d'une machine d'Eilenberg finie en adaptant la notion de *moteur réactif* de la bibliothèque Zen [2–4, 7].

3.1 Implémentation des relations calculables par projection en des générateurs de flux

L'idée est de "Currifier" une relation binaire sur D (classiquement un sous-ensemble de $D \times D$) en une fonction de D dans une partie de D énumérée par une séquence d'éléments de D représentée par un flôt calculable progressivement.

```
type stream 'data =
  [ Void
  | Stream of 'data and delay 'data
  ]
and delay 'data = unit → stream 'data; (* frozen stream *)

type relation 'data = 'data → stream 'data;
```

3.2 Noyau de machines d'Eilenberg calculables

On axiomatise le graphe de transitions de l'automate sous-jacent par une liste d'adjacence. Voici l'interface ML définissant le noyau de machines d'Eilenberg calculables (EMK=Eilenberg machines kernel).

```
module type EMK = sig
 type data;
 type state;
 type generator;
 value transition: state → list (generator × state);
 value initial: list state;
 value accept: state → bool;
 value semantics : generator → relation data;
end;
```

3.3 Moteur réactif simulant une machine en profondeur d'abord

Nous donnons maintenant un simulateur de machines d'Eilenberg, fonctionnant en profondeur d'abord. L'avantage en est que les appels récursifs sont terminaux, et sont compilés en de simples boucles, ne nécessitant pas d'espace d'empilage. L'inconvénient est que le simulateur peut ne pas terminer et donc ne pas énumérer toutes les solutions.

```
module Engine (Machine: EMK) = struct
open Machine;

type choice = list (generator × state);

(* We stack backtrack choice points in a resumption *)
type backtrack =
 [ React of data and state
 | Choose of data and choice and delay data and state
 ]
and resumption = list backtrack;

(* The 3 internal loops of the depth-first search reactive engine *)

(* react: data → state → resumption → stream data *)
value rec react d q res =
 let ch = transition q in
 (* we need to compute [choose d ch res] but first
    we deliver data [d] to the stream of solutions when [q] is accepting *)
 if accept q
   then Stream d (fun () → choose d ch res) (* Solution d found *)
   else choose d ch res
```

```
(* choose: data → choice → resumption → stream data *)
and choose d ch res =
 match ch with
 [ [] → resume res
 | [ (g, q') :: rest ] → match semantics g d with
                [ Void → choose d rest res
                | Stream d' del → react d' q' [ Choose d rest del q' :: res ]
                ]
 ]

(* The scheduler which backtracks in depth-first exploration *)
(* resume: resumption → stream data *)
and resume res =
 match res with
 [ [] → Void
 | [ React d q :: rest ] → react d q rest
 | [ Choose d ch del q' :: rest ] →
   match del () with (* we unfreeze the delayed stream of solutions *)
   [ Void → choose d ch rest (* finally we look for next pending choice *)
   | Stream d' del' → react d' q' [ Choose d ch del' q' :: rest ]
   ]
 ]
;

(* Note that the recursions are just loops, since the recursive calls are ter

(* Simulating the characteristic relation: relation data *)
value simulation d =
 let rec init_res l acc =
  match l with
  [ [] → acc
  | [ q :: rest ] → init_res rest [ React d q :: acc ]
  ] in
 resume (init_res initial [])
;

end; (* module Engine *)
```

Ce moteur, inspiré du moteur réactif de la bibliothèque Zen [2,3], a été généralisé aux machines d'Eilenberg finies par Benoît Razet [7].

La fonction de simulation étant de type (relation data), on peut la ré-injecter comme sémantique d'une machine d'Eilenberg composée, ce qui ouvre la voie à la construction modulaire de machines complexes. Ceci est expliqué dans [4], et une implémentation concrète en est mise en œuvre dans le segmenteur de Sanskrit disponible à `http://sanskrit.inria.fr`.

3.4 Certification du moteur réactif et preuve de complétude

Benoît Razet a montré comment utiliser l'assistant de preuves Coq pour axiomatiser les machines d'Eilenberg finies, faire la preuve de complétude du moteur réactif terminal, extraire le programme ML automatiquement à partir de la preuve formelle, et ainsi certifier la méthodologie. Ce développement est présenté en [8].

4 Un moteur réactif général, dirigé par une stratégie

Le moteur réactif terminal est bien adapté aux machines finies, puisqu'il énumère complètement tous les calculs, avec un bon comportement algorithmique. Mais il ne se prête pas à la simulation de machines non finies. Notamment, lorsqu'il existe un générateur g s'interprétant dans la sémantique par une relation ρ telle que pour une donnée d il existe une infinité de valeurs d' images de d par ρ, alors si la machine arrive dans une configuration de cellule (d, q) telle qu'il existe un q' avec $(\rho, q') \in \delta(q)$, alors la simulation de cette transition provoquera un bouclage de la fonction `choose`, par parcours du flux de données d' correspondant. Et même si les relations intervenant dans le calcul sont toutes fonctionnelles, il existe le risque de boucler en déroulant un calcul infini sans laisser leur chance aux autres calculs potentiels. Ce phénomène est bien connu des programmeurs Prolog, dont l'évaluateur standard procède en profondeur d'abord, et n'est donc pas complet pour l'énumération de toutes les preuves possibles.

Notons que dans le cas d'une machine finie, pour une donnée d il n'existe qu'un nombre fini de valeurs d' images de d par ρ. On pourrait donc représenter cet ensemble de valeurs par une liste, plutôt que par un flôt. Mais alors on ne pourrait pas obtenir la modularité, la relation caractéristique d'une machine étant exprimée par le flôt des solutions à partir d'une donnée initiale. Cette architecture permet de faire des calculs par nécessité, et d'arrêter les calculs dès qu'une valeur de sortie obéit à un critère d'arrêt.

On définit maintenant un moteur réactif général, dirigé par une stratégie, permettant de sélectionner la succession des calculs d'une manière équitable. L'auteur en est Benoît Razet.

L'idée principale est de remplacer la continuation (*resumption*) implémentée comme liste de valeurs de backtrack par une pile, gérée par un module paramètre *Resumption* de la stratégie représentée par un foncteur. La signature de ce module paramètre est composée des opérations d'une pile abstraite.

```
open Eilenberg;

module Engine (Machine: EMK) = struct
open Machine;

type choice = list (generator × state);

(* Now we separate the control choices and the data relation choices *)
```

```
type backtrack =
 [ React of data and state
 | Choose of data and choice
 | Relate of stream data and state
 ]
;

(* Now resumption is an abstract data type, given in module Resumption
   argument to the Strategy functor, generalizing a backtrack stack *)

module Strategy (* resumption management *)
 (Resumption : sig
   type resumption;
   value empty: resumption;
   value pop: resumption → option (backtrack × resumption);
   value push: backtrack → resumption → resumption;
 end) =
 struct

open Resumption;

(* The generic reactive engine, using an exploration strategy *)
(* react: data → state → resumption → stream data *)
value rec react d q res =
 let ch = transition q in
 if accept q
   then Stream d (fun () → resume (push (Choose d ch) res)) (* Solution d fou
   else resume (push (Choose d ch) res)

(* choose: data → choice → resumption → stream data *)
and choose d ch res =
 match ch with
 [ [] → resume res
 | [ (g, q') :: rest ] →
   let res' = push (Choose d rest) res in
   relate (semantics g d) q' res'
 ]

(* relate: stream data → state → resumption → stream data *)
and relate str q res =
 match str with
 [ Void → resume res
 | Stream d del → let str = del () in
                  resume (push (React d q) (push (Relate str q) res))
 ]
```

```
(* resume: resumption → stream data *)
and resume res =
 match pop res with
 [ None → Void
 | Some (b, rest) →
   match b with
   [ React d q → react d q rest
   | Choose d ch → choose d ch rest
   | Relate str q → relate str q rest
   ]
 ]
;

(* characteristic_relation: relation data *)
value simulation d =
 let rec init_res l acc =
   match l with
   [ [] → acc
   | [ q :: rest ] → init_res rest (push (React d q) acc)
   ] in
 resume (init_res initial empty)
;

end; (* module Strategy *)
```

4.1 Quelques stratégies typiques

Nous donnons dans cette section quelques implémentations de stratégies usuelles. On retrouve le moteur terminal avec la stratégie Terminal. On donne ensuite une stratégie appropriée aux machines séquentielles, et une stratégie équitable avec une pile à priorité garantissant que tout point de backtrack enregistré sera ultimement traité - pourvu que les sémantiques soient totales, c'est-à-dire que pour tout générateur g, la fonction $semantics(g)$ soit une fonction totale sur le domaine des valeurs d calculées.

```
(* Terminal strategy module (adequate for Finite Eilenberg Machines) *)
module Terminal = struct
 type resumption = list backtrack;
 value empty = [];
 value push b res = [ b :: res ];
 value pop res =
   match res with
   [ [] → None
   | [ b :: rest ] → Some (b,rest)
   ];
```

```
end; (* module Terminal *)

(* Sequential module for sequential machines *)
module Sequential = struct
 type resumption = list backtrack;
 value empty = [];
 value push b res =
   match b with
   [ React _ _ → [ b :: res ]
   | Choose _ _ → [ b ] (* cut : the list contains only one element *)
   | Relate _ _ → res (* no other delay *)
   ];
 value pop res =
   match res with
   [ [] → None
   | [ b :: rest ] → Some (b,rest)
   ];
end; (* module Sequential *)

(* A module for general Eilenberg machines with fair simulation *)
(* Fairness is insured by Razet' alternation weaver *)
module Fair = struct
 type resumption = (list backtrack × list backtrack);
 value empty = ([],[]);
 value push b res =
   let (left,right) = res in
   (left, [ b :: right ])
 ;
 value pop res =
   let (left,right) = res in
   match left with
   [ [] → match right with
          [ [] → None
          | [ r :: rrest ] → Some (r, (rrest,[]))
          ]
   | [ l :: lrest ] → Some (l, (lrest,right))
   ]
 ;
end; (* module Fair *)


module FEM = Strategy Terminal;  (* Similar to Eilenberg.Engine *)
module Sequential_Engine = Strategy Sequential; (* The sequential engine *)
module Complete_Engine = Strategy Fair; (* The fair engine *)
```

```
end; (* module Engine *)
```

Nous avons maintenant fini de définir le moteur réactif général, et ses stratégies typiques Terminal, Sequential et Fair.

4.2 Exemple

Donnons un exemple d'utilisation, pour engendrer le langage régulier `(a(bc)*ba)*`.

```
(* A toy transition graph for language L=(a(bc)*ba)* *)
value graph = fun
 [ 1 → [ ('a',2) ]
 | 2 → [ ('b',3) ]
 | 3 → [ ('a',1) ; ('c',2) ]
 | _ → assert False
 ]
;
value delay_eos = fun () → Void;
value unit_stream x = Stream x delay_eos;

module AutoGen = struct
 type data = list char;
 type state = int;
 type generator = char;
 value transition = graph;
 value initial = [ 1 ];
 value accept i = (i = 1);
 value semantics c tape = unit_stream [ c :: tape ]
 ;
end; (* AutoGen *)

module AutoGen2 = struct
 type data = (list char × int); (* string with credit bound *)
 type state = int;
 type generator = char;
 value transition = graph;
 value initial = [ 1 ];
 value accept i = (i = 1);
 value semantics c (tape, n) =
     if n <= 0 then Void
     else unit_stream ([ c :: tape ], n-1)
 ;
end; (* AutoGen2 *)

module WordGen = Engine AutoGen;
module WordGen2 = Engine AutoGen2;
```

```
(* Service functions on character streams for testing *)

(* print char list *)
value print_cl l =
 let rec aux l =
   match l with
   [ [] → ()
   | [ c :: rest ] → let () = print_char c in aux rest
   ] in
 do { aux l; print_string "\n" }
;
value iter_stream f str =
 let rec aux str =
   match str with
   [ Void → ()
   | Stream v del → let () = f v in aux (del ())
   ] in
 aux str
;

value print_cl2 (tape, _ ) = print_cl tape
;
value cut str n =
 let rec aux i str =
   if i ≥ n then Void
   else
     match str with
     [ Void → Void
     | Stream v del → Stream v (fun () → aux (i+1) (del ()))
     ] in
 aux 0 str
;

(* Caution: it generates the mirror image of words in L *)
print_string "First 10 words in ~L generated depth-first";
iter_stream print_cl (cut (WordGen.FEM.simulation []) 10);

print_string "First 10 words in ~L generated depth-first sequentially";
iter_stream print_cl (cut (WordGen.Sequential_Engine.simulation []) 10);

print_string "All words in ~L of length bounded by 12";
iter_stream print_cl2 (WordGen2.FEM.simulation ([],12));

print_string "First 50 words in ~L in a complete enumeration";
iter_stream print_cl (cut (WordGen.Complete_Engine.simulation []) 50);
```

5 Machines progressives

Nous avons présenté une stratégie équitable *Fair* permettant de garantir une exploration complète de l'espace de recherche des calculs. Une première remarque est qu'il existe de nombreuses stratégies équitables, garantissant que tout point de choix (backtrack) empilé dans la simulation est ultimement dépilé et traité. On peut alors prouver la complétude de la simulation, pourvu que tous les calculs sur les données terminent, c'est à dire que pour toute valeur d atteinte par un calcul, et pour tout générateur g, la fonction *semantics g d* termine. Autrement dit, la relation ρ_g associée à g est calculable progressivement, au sens où il existe une fonction récursive (totale) énumérant l'ensemble $\rho_g(d)$. Le domaine de cette fonction est un segment initial de $\mathcal{N}$, possiblement infini. Ceci est une restriction à la calculabilité, car dans le cas général $\rho_g(d)$ sera seulement récursivement énumérable, et donc le co-domaine d'une fonction partielle récursive, mais en général on ne saura pas décider si cet ensemble est vide.

Ceci nous amène à définir la notion de *machine d'Eilenberg progressive*, dont la relation caractéristique est calculable progressivement. Les machines d'Eilenberg progressives peuvent être composées modulairement, et simulées complètement par toute stratégie équitable. Alors qu'une machine peut ne pas être progressive, par exemple par une boucle dans le contrôle ne produisant pas de valeur en un état acceptant.

La notion clé est de remplacer les flôts synchrones par des flôts asynchrones, qui vont entrelacer les calculs pour ne pas s'obstiner à effectuer un calcul possiblement non terminant.

5.1 Flôts asynchrones

```
type astream 'data =
[ Void
| Tick of delay 'data
| Astream of 'data and delay 'data
]
and delay 'data = unit → astream 'data; (* frozen astream *)

type relation 'data = 'data → astream 'data;
```

5.2 Machines asynchrones

La signature d'un noyau de machine ne change pas, sauf que les relations sont maintenant asynchrones:

```
module type EMK = sig
type data;
type state;
type generator;
value transition: state → list (generator × state);
value initial: list state;
```

```
value accept: state → bool;
value semantics : generator → relation data;
end;
```

5.3 Moteur réactif asynchrone avec stratégie

Maintenant, la stratégie doit signaler si le calcul doit se poursuivre ou faire une pause.

```
type signal 'cont =
 [ Wait of 'cont
 | Go of 'cont
 ]
```

Voici maintenant le moteur asynchrone.

```
module Engine (Machine: EMK) = struct
open Machine;

type choice = list (generator × state);

(* We separate the control choices and the data relation choices *)
type backtrack =
[ React of data and state
| Choose of data and choice
| Relate of astream data and state
]
;

module Strategy (* resumption management *)
(Resumption : sig
  type resumption;
  value empty: resumption;
  value pop: resumption → option (signal (backtrack × resumption));
  value push: backtrack → resumption → resumption;
end) =
struct

open Resumption;

(* The generic reactive engine, using an exploration strategy *)
(* react: data → state → resumption → astream data *)
value rec react d q res =
let ch = transition q in
if accept q
  then Astream d (fun () → resume (push (Choose d ch) res)) (* Solution d fou
  else resume (push (Choose d ch) res)
```

```
(* choose: data → choice → resumption → astream data *)
and choose d ch res =
match ch with
[ [] → resume res
| [ (g, q') :: rest ] →
  let res' = push (Choose d rest) res in
  relate (semantics g d) q' res'
]

(* relate: astream data → state → resumption → astream data *)
and relate str q res =
match str with
[ Void → resume res
| Tick del → let str = del () in
             resume (push (Relate str q) res)
| Astream d del → let str = del () in
                  resume (push (React d q) (push (Relate str q) res))
]

(* resume: resumption → astream data *)
and resume res =
match pop res with
[ None → Void
| Some c →
  let dispatch b rest =
    match b with
    [ React d q → react d q rest
    | Choose d ch → choose d ch rest
    | Relate str q → relate str q rest
    ] in
  match c with
  [ Wait (b,rest) → Tick (fun () → dispatch b rest)
  | Go (b,rest) → dispatch b rest
  ]
]
;

(* characteristic_relation: relation data *)
value simulation d =
let rec init_res l acc =
  match l with
  [ [] → acc
  | [ q :: rest ] → init_res rest (push (React d q) acc)
  ] in
```

```
resume (init_res initial empty)
;

end; (* module Strategy *)
```

5.4 Les stratégies

```
(* Terminal depth first strategy module (adequate for Finite Eilenberg Machin
module Terminal = struct
type resumption = list backtrack;
value empty = [];
value push b res = [ b :: res ];
value pop res =
  match res with
  [ [] → None
  | [ b :: rest ] → Some (Go (b,rest))
  ];
end; (* module Terminal *)

(* Sequential module for sequential machines *)
module Sequential = struct
type resumption = list backtrack;
value empty = [];
value push b res =
  match b with
  [ React _ _ → [ b :: res ]
  | Choose _ _ → [ b ] (* cut : the list contains only one element *)
  | Relate _ _ → res (* no other delay *)
  ];
value pop res =
  match res with
  [ [] → None
  | [ b :: rest ] → Some (Go (b,rest))
  ];
end; (* module Sequential *)

(* A module for general Eilenberg machines with fair simulation *)
(* Fairness is insured by Razet' alternation weaver *)
(* This strategy scales to modularity because it is a "productive" function
  in the sense of the coinductive construction of Coq *)
module Fair = struct
type resumption = (list backtrack × list backtrack);
value empty = ([],[]);
value push b res =
  let (left,right) = res in
```

```
  (left, [ b :: right ])
;
value pop res =
  let (left,right) = res in
  match left with
  [ [] → match right with
         [ [] → None
         | [ r :: rrest ] → Some (Wait (r, (rrest,[])))
         ]
  | [ l :: lrest ] → Some (Wait (l, (lrest,right)))
  ]
;
end; (* module Fair *)


module FEM = Strategy Terminal;  (* Similar to Eilenberg.Engine *)
module Sequential_Engine = Strategy Sequential; (* The sequential engine *)
module Complete_Engine = Strategy Fair; (* The fair engine *)

end; (* module Engine *)
```

References

1. S. Eilenberg. *Automata, Languages, and Machines, volume A*. Academic Press, 1974.
2. G. Huet. The Zen computational linguistics toolkit: Lexicon structures and morphology computations using a modular functional programming language. In *Tutorial, Language Engineering Conference LEC'2002*, 2002.
3. G. Huet. A functional toolkit for morphological and phonological processing, application to a Sanskrit tagger. *J. Functional Programming*, 15,4:573–614, 2005. `http://yquem.inria.fr/~huet/PUBLIC/tagger.pdf`.
4. G. Huet and B. Razet. The reactive engine for modular transducers. In K. Futatsugi, J.-P. Jouannaud, and J. Meseguer, editors, *Algebra, Meaning and Computation, Essays Dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*, pages 355–374. Springer-Verlag LNCS vol. 4060, 2006. `http://yquem.inria.fr/~huet/PUBLIC/engine.pdf`
5. D. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Information and Computation*, 110:366–390, 1994.
6. V. Pratt. Action logic and pure induction. In *Workshop on Logics in Artificial Intelligence*. Springer-Verlag LNCS vol. 478, 1991.
7. B. Razet. Finite Eilenberg machines. In O. Ibarra and B. Ravikumar, editors, *Proceedings of CIIA 2008*, pages 242–251. Springer-Verlag LNCS vol. 5148, 2008. `http://gallium.inria.fr/~razet/fem.pdf`
8. B. Razet. Simulating finite Eilenberg machines with a reactive engine. In *Proceedings of MSFP 2008*. Electric Notes in Theoretical Computer Science, 2008. `http://gallium.inria.fr/~razet/PDF/razet_msfp08.pdf`