Chapitre 6

ASM est algorithmiquement complet

Nous allons, dans ce chapitre, justifier la **thèse de Yuri Gurevich** selon laquelle les ASM correspondent à un langage algorithmiquement complet, le seul connu à ce jour d'ailleurs.

Nous allons commencer par les seuls vrais algorithmes, les **algorithmes séquentiels**, appelés ainsi par opposition à des généralisations relachant tel ou tel point tels que les *algorithmes non-déterministes*, les *algorithmes parallèles* et toute une série d'autres dont le relâchement peut être utilise pour tel ou tel but.

6.1 Justification par l'expérience

Méthode.- La meilleure justification est certainement encore celle qui consiste à collecter des algorithmes, à les exprimer dans tel ou tel langage de description d'algorithmes (par exemple des langages de programmation), à s'apercevoir qu'on effectue un peu de gymnastique pour ce faire en introduisant des éléments non nécessaires, à exprimer ce même algorithme avec une ASM et à s'apercevoir alors que tout se passe bien.

Exemple 1.- Reprenons une fois de plus l'expression de l'algorithme d'Euclide permettant de calculer le pgcd en langage C :

```
int r;
while (b != 0)
    {
    r = a % b;
    a = b;
    b = r;
    }
 d = a;
```

On remarquera que l'introduction de la variable r est plus ou moins cachée par l'utilisation de l'opération "reste dans la division euclidienne" mais qu'il ne s'agit pas de sa véritable raison d'être : cette variable sert à l'échange entre a et b et on ne peut pas s'en passer par manque d'une structure de contrôle de simultanéïté.

Nous avons vu que, par contre, cette variable n'a pas besoin d'être introduite dans un programme ASM.

Pour revenir au langage C, remarquons que, par contre, le programme récursif :

```
int pgcd(int a, int b)
    {
    if (b == 0) return a;
    else return pgcd(b, a % b);
    }
```

décrit l'algorithme d'Euclide pas à pas. On introduit d'ailleurs souvent la récursivité en indiquant qu'elle permet de décrire les algorithmes mathématiques de façon plus naturelle.

Exemple 2.- Dans le cas de l'échange, par contre, on ne connaît pas de programme C qui n'introduise pas d'élément non naturel.

6.2 Un théorème sur les algorithmes séquentiels

Pour renforcer la justification de sa thèse, Yuri GUREVICH a démontré un théorème en 1999 ([Gur00]) qui montre que tout algorithme, vérifiant trois postulats naturels, peut être émulé pas à pas par une ASM.

Postulat 1.- (**Temps séquentiel**)

À un algorithme séquentiel A sont associés :
- *un ensemble $\mathcal{S}(A)$, dont les éléments sont appelés les* **états** *de A ;*
- *un sous-ensemble $\mathcal{I}(A)$ de $\mathcal{S}(A)$, dont les éléments sont appelés les* **états initiaux** *de A ;*
- *une fonction $\tau_A : \mathcal{S}(A) \longrightarrow \mathcal{S}(A)$, appelée la* **transformation** *de A (en une étape).*

Définition 1.- *Soit A un algorithme séquentiel. Un* **calcul** *de A est une suite finie ou infinie :*

$$X_0,\ X_1,\ X_2, \ldots$$

où X_0 est un état initial et $X_{i+1} = \tau_A(X_i)$ pour tout i.

Définition 2.- *Des* **algorithmes** *A et B sont* **équivalents** *si, et seulement si, $\mathcal{S}(A) = \mathcal{S}(B)$, $\mathcal{I}(A) = \mathcal{I}(B)$ et $\tau_A = \tau_A$.*

Remarque.- Des algorithmes équivalents ont les mêmes ensembles de calculs.

Postulat 2.- (**États abstraits**)

Soit A un algorithme séquentiel :
- *Les états de A sont des structures logiques du premier ordre.*
- *Tous les états de A ont la même signature.*
- *La transformation τ_A ne change pas l'ensemble de base des états.*
- *Les ensembles $\mathcal{S}(A)$ et $\mathcal{I}(A)$ sont clos par isomorphisme. De plus si les états X et Y sont isomorphes alors il en est de même de $\tau_A(X)$ et de $\tau_A(Y)$.*

Définition 3.- *Soient $\mathcal{L}$ un vocabulaire d'ASM et $\mathcal{A}$ et $\mathcal{B}$ des $\mathcal{L}$-structures de même ensemble de base A. On note $\Delta(\mathcal{A}, \mathcal{B})$ l'ensemble de modifications $(f, \overline{a}, b)$ avec :*
- *f élément de $\mathcal{L}$;*
- *$\overline{a}$ élément de A^n, si n est l'arité de f ;*
- *$b = \overline{f}^{\mathcal{B}}(\overline{a}) \neq \overline{f}^{\mathcal{A}}(\overline{a})$.*

Remarque.- Cet ensemble de modifications peut être fini ou infini.

Définition 4.- *Soient $\mathcal{L}$ un vocabulaire d'ASM, $\mathcal{A}$ un $\mathcal{L}$-état et A un $\mathcal{L}$-algorithme. On note $\Delta(A, \mathcal{A})$ l'ensemble de modifications $\Delta(\mathcal{A}, \tau_A(\mathcal{A}))$.*

Définition 5.- *Soient $\mathcal{L}$ un vocabulaire d'ASM, $\mathcal{A}$ et $\mathcal{B}$ des $\mathcal{L}$-structures et T un ensemble de $\mathcal{L}$-termes. On dit que $\mathcal{A}$ et $\mathcal{B}$)* **coïncident sur l'ensemble de termes** *T si, et seulement si :*

$$\forall t \in T\ \ \overline{t}^{\mathcal{B}} = \overline{t}^{\mathcal{A}}.$$

Postulat 3.- (**Exploration bornée**)

Soit A un algorithme séquentiel de vocabulaire $\mathcal{L}$. Il existe un ensemble fini T de $\mathcal{L}$-termes tels que pour tous $\mathcal{L}$-états $\mathcal{A}$ et $\mathcal{B}$ coïncidant sur T, on ait :

$$\Delta(A,\ \mathcal{A}) = \Delta(A,\ \mathcal{B}).$$

Remarques.-

1. Ce postulat signifie intuitivement que l'algorithme A n'examine que la partie définie par l'ensemble de termes T de l'état donné.
2. Exiger que l'ensemble des modifications soit fini n'est pas suffisant. Considérons par exemple l'algorithme suivant sur les graphes qui vérifie si un graphe donné possède des points isolés :

   `if` $\forall x\ \exists y\ Edge(x,\ y)$ `then Ouput := false`
   `else Output := true`

   L'algorithme ne possède qu'une seule modification (booléenne) mais explore le graphe dans son intégralité en une seule étape. On ne peut pas dire qu'il s'agisse d'un algorithme séquentiel.

Définition 6.- *Un* **algorithme séquentiel** *est un objet A qui vérifie les trois postulats de temps séquentiel, d'états abstraits et d'exploration bornée.*

Théorème.- *Pour tout algorithme séquentiel A, il existe une machine à états abstraits séquentiel qui lui est équivalente.*

Démonstration.- Nous renvoyons à [Gur00], pp. 16–20 pour la démonstration de ce théorème.

6.3 Autres types d'algorithmes

Les algorithmes séquentiels sont les seuls vrais algorithmes. On a cependant introduit d'autres types d'entités, portant également le nom d'*algorithmes* pour simplifier, pour lesquels on relâche telle ou telle contrainte qui portait sur les algorithmes séquentiels. On peut ainsi parler :

- d'**algorithmes non déterministes** pour lesquels l'action à entreprendre, un état étant donné, n'est pas déterminée mais est l'une des actions d'un ensemble fini d'actions. On sait, par exemple, qu'il est plus aisé d'utiliser, dans une première étape, les automates finis non déterministes pour concevoir les compilateurs.
- d'**algorithmes parallèles** pour lesquels on peut entreprendre plusieurs actions en même temps (sur des microprocesseurs différents) dans l'espoir de gagner en temps.
- d'**algorithmes multi-agents**.

Yuri Gurevich a considéré le plus grand nombre possible de ces généralisations. Il a donné une variante des ASM (séquentielles) pour les cas considérés et, le plus souvent, il est arrivé à démontrer un théorème analogue à celui de la section 6.2. On se réfèrera à sa liste annotée des articles pour plus de renseignements :

`http://research.microsoft.com/~gurevich/annotated.html`

Nous allons, à titre d'exemple, considérer l'opérateur de choix pour traiter le cas des algorithmes non déterministes.

6.3.1 Complément de AsmL

6.3.1.1 L'opérateur de choix

En AsmL, on introduit le non déterminisme grâce à l'**opérateur de choix**, dont la syntaxe est la suivante :

choose *lien instruction*

où la façon de lier une ou plusieurs variables se fait comme pour la définition d'un ensemble par une propriété caractéristique.

6.3.1.2 Exemple : décomposition d'une permutation en cycles disjoints

Introduction.- L'un des premiers algorithmes rencontrés dans le cours d'algèbre supérieure est la décomposition d'une permutation (c'est-à-dire d'une bijection du sous-ensemble [1,n] de $\mathbb{N}$ dans lui-même) en cycles disjoints.

Considérons, par exemple la permutation :

```
1  2  3  4  5  6
5  6  4  3  2  1
```

On considère le premier élément 1, et on commence la décomposition de la façon suivante : $(1$.

On regarde l'image de 1, ici 5 et et on écrit : $(1, 5$.

On continue ainsi de suite jusqu'à ce qu'on trouve le premier élément du cycle, c'est-à-dire 1 : $(1, 5, 2$, $(1, 5, 2, 6$.

L'image de 6 étant 1, on a trouvé le premier cycle. On met une parenthèse fermante, on ouvre une autre parenthèse et on place le premier élément non considéré : $(1, 5, 2, 6)(3$.

L'image de 3 est 4, on a donc : $(1, 5, 2, 6)(3, 4$. L'image de 4 est 3, on a donc terminé un second cycle : $(1, 5, 2, 6)(3, 4)$.

Il n'y a plus d'élément non considéré dans la permutation. On a donc terminé notre décomposition en cycles disjoints.

Bien entendu on peut remplacer la phrase "*on place le premier élément non considéré*" par "*on place un élément non considéré*". Ceci n'a pas d'importance en mathématiques. Par contre, en faisant ainsi, on casse le caractère *déterministe* de l'algorithme. Intéressons-nous à cet algorithme non déterministe.

Implémentation de l'algorithme en langage C.- La décomposition d'une permutation en cycles, que ce soit dans le cas déterministe ou non déterministe, n'est jamais un des premiers exemples considérés en langage C. Il s'agit, en effet d'un vrai défi (*challenge* en anglais). Par opposition, nous allons voir que cet algorithme s'écrit très facilement et naturellement en ASM.

Implémentation en AsmL.- Écrivons un programme AsmL qui affiche à l'écran la décomposition en cycles de la permutation ci-dessus :

```
// permutation.asml

var s as Map of Integer to Integer = {1 -> 5, 2 -> 6, 3 -> 4,
                                      4 -> 3, 5 -> 2, 6 -> 1}
var R as Set of Integer = {1, 2, 3, 4, 5, 6}

Main()
  var j as Integer
  step while Size(R) <> 0
    choose i in R
      step
        Write("(" + i)
        j := s(i)
        remove i from R
      step while j <> i
        Write(", " + j)
        j := s(j)
        remove j from R
      step
        Write(")")
  step
    WriteLine("")
```

Si on effectue plusieurs exécutions, on obtiendra des résultats différents, par exemple :

```
E:>permutation
(6, 1, 5, 2)(4, 3)

E:>
```

Exercice 1.- *Écrire un programme AsmL déterministe pour le problème de la décomposition d'une permutation en cycles.*

[*On pourra définir une méthode :*

`min(E as Set of Integer) as int`

pour remplacer le choix.]

Exercice 2.- *Mettre ces deux programmes (non déterministe et déterministe) sous forme normale.*