Manipulations de structures topologiques dans un langage déclaratif pour la simulation

Antoine Spicher & Olivier Michel

LaMI UMR 8042, CNRS - Université d'Évry Val d'Essonne - Genopole
523 Place des Terrasses de l'Agora, 91000 Évry
aspicher, michel @lami.univ-evry.fr

Résumé : *MGS est un langage de programmation expérimental permettant de manipuler des collections topologiques par des règles de transformation. Une collection topologique est un ensemble de valeurs muni d'une relation de voisinage. Cette approche permet d'exprimer simplement l'état d'un système dynamique et son évolution. La présentation du langage est faite à travers plusieurs exemples, dont un processus de diffusion et d'agrégation sur plusieurs quasi-variétés modélisées par des G-cartes.*

Mots-clés : collection topologique, transformation, langage de programmation déclaratif, simulation, G-carte, réécriture.

1 Introduction

L'objectif du projet MGS est de développer un langage dédié à la modélisation et la simulation de processus à structure dynamique.

Les langages dédiés fournissent au programmeur des abstractions et des notations adaptées à un domaine d'application particulier. Organisé autour d'un noyau petit et souvent fonctionnel, leur spécialisation les rendent a priori plus attractif qu'un langage généraliste et permettent une meilleure productivité en facilitant la programmation et la réutilisation.

Les processus auxquels nous nous intéressons, sont des systèmes dynamiques hautement structurés et hiérarchiquement organisés, dont la structure peut varier au cours du temps et doit alors être calculée conjointement avec l'état du système. Ce type de système est courant dans les modèles de croissance de plante, en biologie du développement, dans les modèles intégrant plusieurs échelles, etc. Les lois d'évolution sont souvent informellement décrites comme un ensemble de transformations locales agissant sur un ensemble organisé d'entités.

Afin de passer de ce type de description informelle à un modèle adapté à la simulation informatique nous proposons de représenter l'état du système dynamique par une *collection topologique* et de spécifier la fonction d'évolution par des règles locales de *transformation*. Une collection topologique est un ensemble d'éléments reliés par une relation de voisinage. Une transformation est une fonction définie par cas, les cas correspondant à l'état des sous-systèmes en interaction.

Les notions mises en œuvre dans MGS reposent sur des concepts topologiques et unifient plusieurs modèles de calcul comme la réécriture de multi-ensembles (Gamma [**?**], le calcul chimique [**?**], les P-systèmes [**?**]), la réécriture des chaînes (les L-systèmes [**?**], Snobol [**?**]) ou bien encore les automates cellulaires [**?**]. Nous nous concentrons dans cet article sur une famille de structures issue de la notion de G-carte et permettant de représenter des systèmes dont l'état correspond à un champ physique sur une variété.

La suite de cette présentation s'organise comme suit. La section **??** donne une brève présentation générale de MGS. Deux exemples illustrent cette introduction : un algorithme de tri et la simulation du déplacement collectif d'un ban de poissons. La section **??** introduit une nouvelle famille de collections topologiques fondée sur la notion de G-carte. La transformation de ce type de collection est illustrée par la simulation d'un processus de diffusion-agrégation et par un exemple plus algorithmique, le calcul d'un flot maximal sur un graphe (vu comme un complexe cellulaire de dimension 1). Pour conclure, nous indiquons l'état d'avancement de notre projet de recherche et indiquons plusieurs problèmes ouverts.

2 Présentation de MGS

MGS est un langage de programmation fonctionnel impur, dynamiquement typé et strict. Dans cette section, nous allons le présenter brièvement en nous concentrant sur les notions requises pour la compréhension des exemples de cet article. De plus, seules les différences majeurs d'avec d'autres langages fonctionnels tels que OCaml [?] seront décrites.

2.1 Les valeurs atomiques

MGS fournit les valeurs atomiques classiques (comme les entiers, les réels, les booléens, les chaînes de caractères, les symboles, *etc*) ainsi que les fonctions de bases qui permettent de les manipuler. Il est également possible de définir de nouvelles fonctions. On utilisera pour cela le mot clef `fun` comme dans `fun max(x, y) = if (x > y) then x else y fi` qui calcule le maximum de ses deux arguments. MGS permettant l'ordre supérieur, toute fonction est considérée comme une valeur (elles peuvent par conséquent êtres utilisées comme argument d'autres fonctions ou retournées comme résultat). Enfin, des paramètres optionnels peuvent également être spécifiés entre crochets : `fun succ[inc=1](x)=x+inc`. Ces paramètres sont facultatifs à l'application (`succ(0)` retourne `1`), mais peuvent également être précisés (`succ[inc=3](0)` retourne `3`).

2.2 Les collections topologiques

La principale nouveauté apportée par MGS est la manipulation d'éléments structurés par une *topologie abstraite* au moyen de *transformations* [?]. Un ensembles d'éléments organisés suivant une topologie abstraite est une *collection topologique*. Le terme « topologique » indique l'existence d'une relation de voisinage entre les éléments de la collection. Le type de la collection dépend des propriétés de cette relation. Elle permet également de définir pour toute collection la notion de *sous-collection* : une sous-collection S' d'une collection S est une sous-ensemble des éléments de S connectés entre eux par la relation de voisinage de S restreinte aux éléments de S'.

MGS propose différents types de collections topologiques. Cet article n'étant pas destiné à présenter en détail les différents types de collections prédéfinies dans MGS, nous allons nous contenter de présenter deux types ci-dessous, et de compléter cette présentation lors des exemples de la sections **??** :

- Les collections *monoïdales* : il s'agit de collections dont la relation de voisinage est définie à travers un monoïde. L'opération du monoïde est l'ajout d'un éléments et est notée par une virgule. Si on considère que la virgule est associative, on construit des séquences (chaque élément a 2 voisins, sauf aux extrémités). Si la virgule est associative et commutative, on construit des multi-ensembles (tout élément est voisin de tous les autres). Et si on rajoute la propriété d'idempotence, on construit des ensembles.
- Les graphes de Delaunay : ce sont des graphes non orientés dont les sommets sont plongés dans l'espace euclidien $\mathbb{R}^n$. A partir de ce plongement, on calcule la relation de voisinage entre les éléments par une triangulation de Delaunay (qui sépare l'espace en cellules entourant un élément du graphe et dont tous les points sont plus proches de cet élément que de n'importe quel autre élément). Cette collection permet de représenter un voisinage correspondant à la notion de « plus proche voisin ».

Il est possible de définir de nouveaux types de collection à partir des collections existantes. Le prototype actuel permet par exemple de manipuler des graphes arbitraires, des graphes de Cayley (en particulier des partitions du plan et des grilles de dimension n, rebouclées ou non) ainsi que des *quasi-variétés* comme on le verra en sections **??**.

Pour toute collection `T`, la collection vide est notée `():T`. La principale opération entre deux collections C_1 et C_2 est la *réunion* de ces deux collections. Elle est dénotée par la virgule (C_1`,`C_2). Cet opérateur est surchargé et peut être utilisé pour construire des collections. Par exemple, l'expression `1,1+1,2+1,():set` construit l'ensemble contenant les entiers `1`, `2` et `3` alors que `1,1+1,2+1,():bag` définit le multi-ensemble contenant les trois mêmes éléments. La séquence

vide n'est pas nécessaire à la définition d'une séquence : `1,2,3` est équivalent à `1,2,3,():seq`. Comme indiqué plus haut, dans un ensemble ou un multi-ensemble tout élément est voisin de tous les autres (la différence est qu'on n'autorise aucune répétition dans un ensemble). Cette topologie a été choisie afin de retrouver la notion classique de réécriture de multi-ensemble.

2.3 Les tranformations

Les transformations permettent de spécifier la fonction d'évolution de l'état d'un système. Les transformations sont des fonctions définies par cas sur les sous-collections. La transformation *globale* d'une collection topologique s consiste en l'application parallèle d'un ensemble de transformations *locales*. Une transformation locale est définie par une règle r spécifiant le remplacement d'une sous-collection de s par une nouvelle collections s'. Aussi, l'application d'une règle de réécriture $\sigma \Rightarrow f(\sigma, ...)$ sur une collection s :

1. sélectionne une sous-collection s_i de s filtrée par le *motif* σ,
2. calcule une nouvelle collection s'_i comme le résultat de l'application de, f à s_i et ses voisins,
3. et définit la substitution de s_i par s'_i dans s.

On notera que la stratégie d'application maximale et parallèle des règles implique que toutes les instances de sous-collections s_i filtrées par σ soient distinctes et sans intersections. En effet, toutes ces instances sont remplacées simultanément par leur sous-collections s'_i respectives.

Les transformations `MGS` sont des fonctions comme les autres. Comme toutes fonctions, elles peuvent prendre des arguments optionnels, être données comme argument, ou retournées comme résultat d'une application.

2.3.1 Motif de chemins

Un motif σ (partie gauche de la règle de réécriture) spécifie la sous-collection à substituer. Cependant, cette collection peut avoir une forme arbitraire, ce qui rend l'application de la règle fastidieuse. Aussi, il est plus facile et tout aussi général d'énumérer séquentiellement les éléments de la sous-collection. Nous appellerons une telle énumération un *chemin* dans une collection.

Ainsi, le motif σ permettant de filtrer une sous-collection spécifie un motif de chemin filtrant une séquence d'éléments contigus dans la collection. Le motif de chemin ***`Pat`*** est une séquence ou une répétition ***`Rep`*** de filtres atomiques, un filtre atomique permettant la sélection d'un seul élément. Voici une forme simplifiée de la grammaire de motifs :

`Pat` ::= ***`Rep`*** | ***`Rep`*** `,` ***`Pat`*** | ***`Pat`*** `as` id | ***`Pat`*** `*`
`Rep` ::= ***`BF`*** | ***`BF`*** `/`*exp*
`BF` ::= `cte` | id | `<undef>`

où `cte` dénote une valeur quelconque, id est une variable du motif et permet de nommer un des éléments filtrés, et *exp* est une expression booléenne. Voici la sémantique des filtres construits sur cette grammaire :

constante : une valeur constante `cte` filtre un élément dont la valeur est `cte`. Par exemple, `123` filtre un élément dont la valeur est 123.

élément vide : le symbole `<undef>` filtre un élément dont la valeur est indéfinie. Par exemple, dans un graphe de DELAUNAY tous les sommets ne sont pas nécessairement valués. La valeur `<undef>` est alors utilisée pour étiqueter ces sommets.

variable : la variable de motif a filtre exactement un élément dont la valeur est définie. Cette variable a peut alors apparaître n'importe où ailleurs dans le reste de la règle et représente l'élément filtré.

voisin : b`,` p filtre un chemin commençant par un élément filtré par b et dont la suite du chemin est filtrée par p tel que le premier élément de ce chemin soit un des voisins de b.

garde : p/exp filtre un chemin spécifié par le motif p si l'évaluation de exp retourne la valeur booléenne "vrai". Par exemple, x`,` y `/` y `>` x filtres deux éléments voisins x et y tels que la valeur associée à y est supérieure à celle de x.

répétition : p`*` filtre une sous-collection composée de la répétition de sous-collections filtrées par p.

alias : p `as` x permet de référer au chemin filtré par p à travers le nom x.

Les éléments filtrés par les motifs atomiques d'une règle sont distincts. Ainsi, un chemin ne peut pas avoir d'auto-intersection.

2.3.2 Partie droite d'une règle de réécriture

La partie droite permet de spécifier la collection qui doit remplacer la sous-collection filtrée par le motif décrit en partie gauche de la règle. Reconstruire une sous-collection dont la forme arbitraire dépend de celle qui doit être substituée n'est pas évident. C'est pourquoi on utilise la définition sous forme de séquence de la sous-collection filtrée : si la partie droite d'une règle définit une séquence, alors le i^{e} élément de cette séquence est substitué au i^{e} élément du chemin filtré. Ce point de vue permet une spécification uniforme des règles, quelque soit la topologie de la collection filtrée.

Pour certaines collections, il est possible de remplacer une sous-collection par une collection dont la forme est différente. De telles collections sont dites *leibniziennes* et sont opposées aux collections *newtoniennes*. Par exemple, les séquences, les ensembles, les multi-ensembles et les graphes de DELAUNAY sont leibniziens. Les grilles sont un exemple de collection newtonienne : on ne peut remplacer un sous-ensemble d'une grille par un sous-ensemble dont la forme diffère sans détruire la topologie de la grille.

2.4 Deux courts exemples

Voici deux exemples qui illustrent les notions introduites.

2.4.1 Le tri à bulle en `MGS`

Ce tri consiste à :

1. comparer deux éléments voisins dans une séquence et à les échanger s'ils ne sont pas dans le bon ordre,
2. itérer la première étape jusqu'à un point fixe.

Cette spécification est directe en `MGS` :

```
trans tri_bulle = {     x, y / (x > y) => y, x     }
```

Le mot clef `trans` introduit la définition d'une nouvelle transformation par un ensemble de règles. Ici, il n'y en a qu'une seule. Cette transformation peut être appliquée à une séquence `s := 4,3,2,1` jusqu'à un point fixe par `tri_bulle['iter = 'fixpoint](s)`. La valeur de l'option prédéfinie `'iter` indique que l'application de la transformation `tri_bulle` doit être itérée jusqu'à ce qu'un point fixe soit atteint. Le résultat est celui attendu : `1,2,3,4`.

2.4.2 Bans de poissons

Dans ce nouvel exemple un peu plus complet, nous allons simuler le regroupement et le déplacement d'un ban de poissons. Le modèle est inspiré de la simulation proposée par U. WILENSKY et par la notion de *boïds* proposé par C. REYNOLDS [**?**]. L'algorithme que nous allons présenter est une variante de celui qu'ils proposent.

Chaque poisson du ban semble suivre la même direction. Pourtant, aucun meneur n'impose cette direction à tout le groupe. En fait, chaque poisson suit le même ensemble de règles pour

choisir la direction de son mouvement. Le modèle le plus simple est composé de trois règles pour (1) ne pas rentrer en collision avec ses voisins, (2) se rapprocher quand on est trop éloigné du groupe et enfin (3) s'orienter dans la même direction globale que ses voisins. Elles seront présentées en détail plus loin, la première étape étant de représenter les poissons.

Représentation des poissons et du ban. Un poisson est représenté par deux grandeurs : une position et une orientation dans l'espace. Pour simplifier l'exemple, nous nous placerons en dimension 2, sachant que le passage à la dimension 3 est immédiat. Pour représenter un poisson, nous allons utiliser un enregistrement (structure de données équivalente à un `struct` en `C`) composé de trois champs : deux pour la position (`x` et `y`), et un pour l'orientation sous la forme d'un angle par rapport à l'axe des abscisses (`theta`). On supposera que tous les poissons se déplacent à la même vitesse, celle-ci pouvant augmenter si le poisson est trop loin du groupe.

```
record Position = {x, y, theta}
```

La définition d'un type T spécifie un prédicat de même nom permettant de vérifier que son argument est de type T.

L'une des façons de générer un voisinage naturel entre les poissons est d'utiliser une triangulation de Delaunay. C'est donc cette collection topologique qui va nous permettre de représenter le groupe de poissons. Elle est générée à partir d'une séquence de poissons et d'une fonction de plongement retournant la position du poisson dans l'espace. On définira ici le type `D2` qui est un graphe de Delaunay de dimension 2 prenant comme élément des valeurs de type `Position` :

```
delaunay(2) D2 =
  \e.if Position(e)
  then (e.x, e.y)
  else error("bad element type for D2 delaunay type")
  fi
```

Le comportement des poissons. Suivant leur position respective, les uns par rapport aux autres, les poissons choisissent de manoeuvrer de façon individuelle en changeant leur position, leur vitesse et leur orientation. Voyons les trois règles de mouvement :

1. ***Séparation*** : si un poisson se retrouve trop près de l'un de ses voisins, il change sa direction pour s'éloigner et éviter la collision.
2. ***Cohésion*** : si un poisson se retrouve trop loin du groupe, il se rapproche de son voisin le plus immédiat en augmentant sa vitesse.
3. ***Alignement*** : si, enfin, le poisson n'est ni trop près, ni trop loin de ses voisins, il va chercher à se diriger dans le même sens qu'eux en choisissant la direction moyenne de ses voisins.

On explicite uniquement le code de la règle de réécriture correspondant à l'alignement des poissons pour ne pas surcharger inutilement cet exemple. En effet, les autres règles sont gérées de la même façon. La transformation correspondant au comportement des poissons peut s'écrire :

```
trans behavior = {
  separation = ... => ... ;
  cohesion   = ... => ... ;
  alignement =
    a => begin
           let phi = neighborsfold(add_theta, 0, a)
           and nb = neighborsfold(nb_neighbors, 0, a) in
           let dir = phi / nb in
             a + {x = a.x + speed*cos(dir) + random(bruit),
                  y = a.y + speed*sin(dir) + random(bruit),
                  theta = dir}
         end
}
```

Pour la règle d'alignement, on commence par calculer la somme des orientations `theta` des voisins. Pour cela, on utilise la primitive `neighborsfold` qui n'est autre que l'itérateur `fold` des langages fonctionnels appliqué sur la liste des voisins du poisson `a`. On somme ensuite le nombre de voisins pour finalement calculer la direction moyenne `dir` des poissons voisins. On met alors à jour la position du poisson qui se déplace dans cette nouvelle direction (on notera un léger bruit ajouté pour rendre plus vivante la simulation), puis on fixe le champ `theta` à `dir`. La figure **??** illustre trois étape de ce processus.

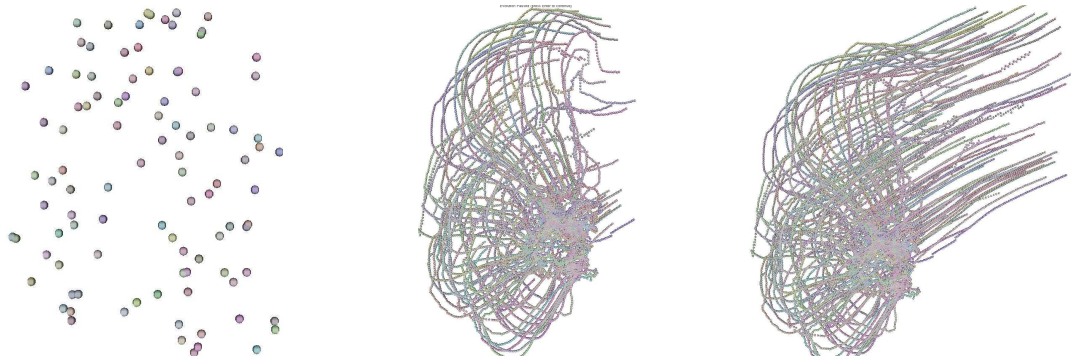

FIG. 1 – Trajectoire d'un ban de 50 poissons. Sont représentées : la situation initiale où chaque poisson à une direction aléatoire, la configuration après 300 itérations et après 900 itérations de la fonction d'évolution `behavior`.

3 Filtrage sur des complexes cellulaires

Dans cette section, nous allons voir que la notion de collection topologique, telle qu'elle a été précédemment décrite, est limitée. Nous présenterons alors un nouveau point de vue sur ces collections en s'appuyant sur des concepts de topologie algébrique combinatoire. Enfin, nous présenterons un exemple de transformation sur ce nouveau type de collection.

3.1 Intérêt des complexes cellulaires

3.1.1 Au delà des graphes

L'ensemble des collections décrites jusqu'ici peut être vue de façon générique comme un graphe dont les sommets représentent les éléments (c'est-à-dire à la fois leur position et leur valeur), et dont les arcs relient deux sommets s'ils sont voisins suivant la topologie de la collection. Les séquences sont juste des graphes linéaires, les graphes de DELAUNAY construisent des graphes à partir de points de $\mathbb{R}^n$. Ces collections ne permettent pas la représentation de topologies plus arbitraires.

Cette remarque motive le développement de topologies plus riches. Par exemple, nous voulons pouvoir définir des maillages irréguliers, mais également représenter des espaces hétérogènes. Prenons par exemple la modélisation des lois électrostatiques. Elles dépendent (dans les cas les plus généraux) de la géométrie du système et les valeurs doivent être associées à des dimensions : on parle par exemple de densité volumique de charge (dimension 3) alors que le flux du champ électrique est calculé sur une surface (dimension 2). On remarquera également l'existence de lois qui relient ces valeurs et donc ces dimensions (la loi de GAUSS pour notre exemple). Pour plus de détails, le lecteur intéressé se reportera aux travaux d'E. TONTI [**?**].

Ainsi, la description de la structure d'un système (c'est-à-dire les interactions entre ces sous-systèmes) peuvent faire apparaître des simplexes de dimension supérieure à 1. Dans le cas général, tous les sous-systèmes de n'importe quelle dimension ainsi que leurs interactions doivent apparaître dans la description du système, et il doit être possible d'y associer des valeurs.

3.1.2 Les collections topologiques vues comme des complexes cellulaires

Le cadre le plus adapté au développement de collections topologiques offrant assez de souplesse pour représenter des éléments de dimensions différentes, est la topologie algébrique combinatoire.

Une collection topologique est alors un *complexe cellulaire* : c'est une collection d'objets de dimensions différentes appelés *k-cell* (ou cellule de dimension k). Les 0-cells sont des sommets, les 1-cells des arcs, les 2-cells des faces, les 3-cells des volumes, *etc*. Une k-cell c est incidente à une $(k-1)$-cell c' si $c' \subset \partial c$, où ∂c dénote le bord de c. On peut utiliser la relation ∂ pour définir une relation de voisinage dans une collection topologique : *deux k-cells sont voisines si elles partagent une* $(k-1)$*-cell incidente ou si elles sont incidentes à une même* $(k+1)$*-cell.* Cette définition est consistante avec celle proposée plus haut.

3.2 G-cartes et quasi-variétés

Il existe plusieurs spécialisations de la notion de complexe cellulaire. On connaît par exemple les complexes simpliciaux dont les cellules sont toutes des simplexes ; un simplexe de dimension n est l'enveloppe convexe des $n+1$ sommets. Un triangle est un simplexe de dimension 2. Pour `MGS`, le choix de la représentation des simplexes s'est porté sur les *cartes généralisées*, ou G-cartes dans le cadre d'une collaboration avec le laboratoire IRCOM-SIC de l'Université de Poitiers. Les objets topologiques pouvant être construits à partir des G-cartes sont les *quasi-variétés* [**?**].

Voici une brève description des G-cartes. Considérons le graphe d'incidence des cellules, c'est-à-dire le graphe dont les sommets sont les cellules et où deux sommets a et b sont reliés par un arc orienté si a est incident à b. Soit N la dimension du domaine, c'est-à-dire la plus grande dimension que peut avoir une cellule. On appellera *tuple* le N-uplet $(c_N, c_{N-1}, \ldots, c_1, c_0)$ correspondant à un chemin dans le graphe d'incidence. Enfin, on dira que deux tuples sont *i-adjacents* s'ils partagent les mêmes cellules, à l'exception des cellules de dimension i qui peuvent différer.

Une G-carte de dimension N est un couple dont le premier élément est un ensemble de *brins* et dont le second est un ensemble de $N+1$ involutions notées α_i $(0 \leq i \leq N)$. Un brin est en fait une représentation abstraite d'un tuple, et pour une paire (d, d') de brins donnés, on a $d' = \alpha_i(d)$ si les deux tuples correspondant sont i-adjacents. Enfin, pour que la G-carte soit correcte, il faut en plus vérifier l'équation :

$$\forall i, j, \quad 0 \leq i < i+2 \leq j \leq N, \ \alpha_i \circ \alpha_j \text{ est une involution.}$$

3.3 Intégration des G-cartes dans `MGS`

Un nouveau type de collection topologique basée sur les G-cartes a été développé ; on l'appelera *QMF* en référence au terme anglais *Quasi-Manifolds* pour *quasi-variétés*. Ces collections sont représentées par un triplet (G, N, h) tel que G est une G-carte, N est la dimension de la G-carte et enfin, h est une table de hachage qui permet d'associer une valeur à chaque cellule de G. Une QMF est donc un type de collection paramétré par une G-carte. Une cellule n'ayant pas de valeur définie sera associée à `<undef>`.

`MGS` offre plusieurs opérations de constructions de G-cartes : produits, extrusion, contraction, fusion de deux cellules, *etc* Plus généralement, toutes les opérations disponibles dans le noyau du modeleur `MOKA`[1] sont disponibles en `MGS`. `MGS` permet aussi le chargement de G-cartes éditées

[1] `http://www.sic.sp2mi.univ-poitiers.fr/moka/`

avec MOKA, un logiciel interactif de CAO. On notera par ailleurs l'utilisation directe dans MGS de la bibliothèque de manipulation des G-cartes de MOKA.

La représentation des cellules topologiques à partir des brins et des involutions de la G-carte n'est pas immédiate. En effet, une cellule c est l'ensemble des brins correspondant aux tuples contenant c. Cependant, il est possible de parcourir cet ensemble en utilisant une *orbite d'involutions*. Une orbite est un ensemble d'involutions noté $< \alpha_{i_1}, \alpha_{i_2}, \alpha_{i_k} >$; en appliquant cette orbite à un brin d de la G-carte, on obtient l'ensemble des brins images de d par l'ensemble des compositions des involutions de l'orbite en question. Ainsi, si d est un brin d'une cellule c de dimension i de la G-carte, on peut calculer l'ensemble des brins appartenant à c. Pour cela, il suffit de ne pas emprunter l'involution α_i, qui nous ferait sortir de la cellule c. L'orbite associée à la i-cell c est donc $< \alpha_0, \alpha_1, \ldots, \alpha_{(i-1)}, \alpha_{(i+1)}, \ldots, \alpha_N >$. Ainsi, pour représenter une cellule, il suffit de connaître l'un de ses brins et sa dimension. Ce couple ne peut malheureusement pas être utilisé tel quel dans MGS pour identifier une cellule dans la G-carte, car il n'est pas unique : il existe autant de représentations d'une cellule qu'il y a de brins qui la composent. Cela pose un problème pour la gestion de la table de hachage de la QMF qui nécessite des clefs uniques. Nous avons donc choisi d'utiliser un brin distingué représentant les cellules ; ce brin est celui dont l'adresse mémoire est la plus petite dans l'ensemble des brins composant la cellule. Il nous a donc fallu adapter notre utilisation de la bibliothèque de G-cartes de MOKA en conséquence pour gérer la normalisation de la représentation d'une cellule.

La dernière étape de l'intégration des G-cartes est l'interfaçage entre l'interpréteur MGS, écrit en OCAML, et la bibliothèque de G-cartes, écrite en C++. Nous avons choisi d'utiliser un type opaque (ou *représentation abstraite*) [?, pp234] des brins et des G-cartes côté OCAML. Ces types abstraits sont en fait des blocs du tas OCAML encapsulant simplement les adresses des objets de la bibliothèque C++. Les blocs sont personnalisés avec des fonctions de finalisation, de comparaison et de hachage *ad-hoc*. On peut ainsi utiliser la plupart des fonctions de base sur ces valeurs et gérer efficacement les problèmes issus du glaneur de cellules d'OCAML.

3.4 Exemple de manipulation des G-cartes avec MGS : le DLA

Nous présentons dans cette section un exemple de programme MGS impliquant les QMF. Il s'agit de l'*aggrégation limitée par diffusion*, ou DLA. C'est un modèle de croissance étudié par deux physiciens, T.A. Witten et L.M. Sander, dans les années 80 [?]. Le principe est le suivant : un ensemble de particules diffusent de façon aléatoire dans un domaine spatial donné. À l'état initial, une des particules est fixée. Puis, lorsqu'une particule mobile en rencontre une fixée, elle s'agrège à celle-ci et se fixe.

Ce processus est habituellement modélisé sous la forme d'un gaz sur réseau, une variante des automates cellulaires [?]. Ceci pourrait être facilement fait en MGS [?]. Mais nous allons ici implémenter ce modèle sur des G-cartes et non sur les domaines réguliers utilisés habituellement avec les automates cellulaires.

Fonction d'évolution du système. La transformation décrivant le processus de DLA est le suivant : on utilise deux symboles `‘red` et `‘green` pour représenter respectivement les particules mobiles et fixes. Deux règle vont définir la transformation :

1. si une particule mobile est voisine d'une particule fixe, elle se fixe,
2. si une particule mobile est voisine d'une place vide (dont la valeur associée est `<undef>`), elle quitte sa position courante pour occuper la place vide (sa position courante devient alors vide).

On remarquera que l'ordre d'application des règles est important car la première est prioritaire sur la seconde. On obtient ainsi :

```
trans dla = {
    ‘red, ‘green   =>  ‘green, ‘green
    ‘red, <undef>  =>  <undef>, ‘red
}
```

État initial du système. La solution la plus élégante pour associer des valeurs aux cellules d'une G-carte est de déclarer, quand cela est possible, une transformation qui construit cet état initial. On suppose donc pour cette transformation d'initialisation qu'aucune valeur n'est pour l'instant associé aux cellules. Il faut ensuite créer une première cellule fixe, puis répartir de façon aléatoire dans le reste du domaine les particules mobiles :

```
trans init[flag = true] = {
    <undef> / flag            =>  (flag := false ; ‘green) ;
    <undef> / (random(4)==0)  =>  ‘red
}
```

La première règle ne s'applique qu'une seule fois (car dès la première occurrence trouvée la garde `flag` est fausse) ; la seconde règle s'applique avec une probabilité égale à 0.25.

Application sur différentes topologies. La figure **??** montre l'application des transformations `init` puis `dla` sur trois *supports* différents. On notera que les transformations sont totalement indépendantes du support sur lesquels elles s'appliquent ; et même mieux, ces transformations sont également valables pour tous les autres types de collection topologique.

Dans ces exemples, les collections topologiques sont de dimension 2 et les valeurs (`‘red` et `‘green`) sont associées aux 2-cells, deux 2-cells étant voisines si elles partagent une arête. Sur le haut de la figure, seules les 2-cells figurent et présentent les topologies des objets (certaines cellules ont 4 voisins et d'autres 3, comme aux pôles de la sphère). La rangée du bas présente les mêmes objets en ne faisant figurer que les cellules valuées. Il s'agit des états finaux, la transformation `dla` ayant atteint un point fixe.

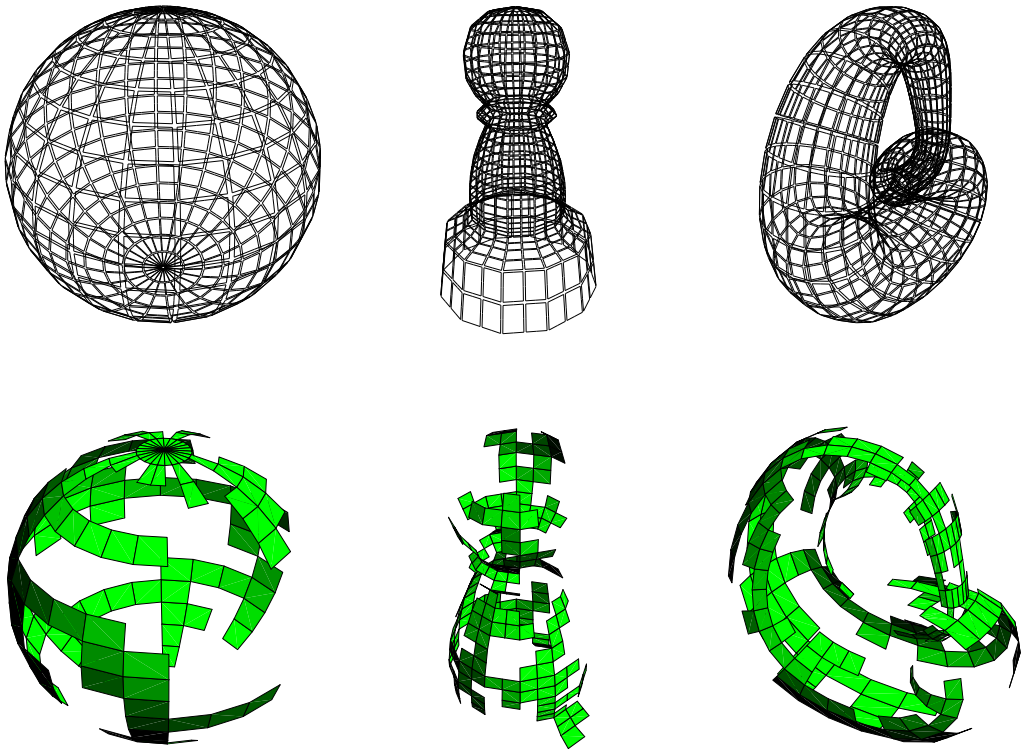

Fig. 2 – DLA sur des supports complexes. La première rangée présente les différents supports : de gauche à droite, une sphere à 18 parallèles et 24 méridiens ; un pion de jeu d'échec et une bouteille de Klein. Sur la ligne du bas, les états finaux du processus de DLA sont présentés pour chacun des supports.

3.5 Calcul d'un flot maximal sur un graphe

Un graphe est un complexe cellulaire de dimension 1. Nous tirons parti de cette propriété pour manipuler les graphes à travers les QMF en `MGS`. L'objectif de cette section est d'illustrer la puissance expressive des transformations sur l'exemple d'un traitement complexe.

L'algorithme de calcul de flot maximal sur un graphe requiert la valuation des 0-cells et des 1-cells, et peut être programmé en `MGS` sous la forme de deux transformations s'appliquant chacune sur des cellules de dimensions différentes.

L'algorithme de Ford-Fulkerson. On cherche à représenter un réseau de communication entre différents sites. Les liens raccordant deux sites sont des canaux où le débit est limité par une capacité maximale. On se donne deux sites de ce réseau, l'un source s (aucun flot n'y entre), l'autre cible t (aucun flot n'en sort). La question qu'on se pose est alors : quel est le débit maximal que s peut relâcher dans le réseau jusqu'à atteindre t, mais sans saturer les canaux du réseau ?

Le réseau est naturellement représenté par un graphe dont les sommets correspondent aux sites, et dont les arcs orientés sont les canaux entre les sites. L'algorithme de Ford-Fulkerson [?] retourne une des solutions possibles en partant d'un état admissible du réseau. Le flot est admissible si :
– pour tout sommet i (à l'exception de s et t), le flot entrant dans i est égale au flot sortant,
– pour tout canal t entre deux sites, le débit est positif mais est inférieur à la capacité de t.
Le flot admissible trivial est le flot nul pour lequel rien ne circule dans réseau. L'algorithme procède en deux étapes :

1. Marquage : on suppose qu'on augmente légèrement le débit sortant de s. Cela sera dénoté par la marque `'plus` associé au sommet s. Ensuite, on va chercher à propager cette augmentation de la façon suivante : soit un arc c reliant le site i à j. Si i est marqué, que j est non-marqué et que c n'est pas encore saturé, on peut marquer j par `'plus`. Si par contre j est marqué, i est non-marqué, et que le débit de c est non-nul, on marque i par `'moins`. On effectue ce marquage jusqu'à un point fixe.
2. Augmentation du flot : si t est marqué, on peut trouver un chemin entre s et t dont les arcs peuvent voir leur débit changer, créant ainsi une augmentation du flot total du réseau. Ce chemin est une *chaîne augmentante*. On calcul alors cette augmentation, on met à jour le réseau, et on recommence. Si t n'est pas marqué, le flot est maximal.

On remarquera que la première étape modifie les valeurs des sommets, alors que la seconde agit sur les arcs.

Définition des types. Nous allons typer les valeurs associées aux cellules. Un site porte un nom et peut-être soit marqué, soit non-marqué. On utilisera un enregistrement `MGS` pour les représenter :

```
record site      = {nom : string}
record marqué    = site + {marque}
record nonMarqué = site + {~marque}
```

Un enregistrement `marqué` contient tous les champs d'un enregistrement de type `site` plus un champ `marque`. Un enregistrement `nonMarqué` contient tous les champs d'un enregistrement de type `site` et ne doit pas contenir de champ de type `marque`. Le champ `marque` prendra ses valeurs dans l'ensemble de symboles {`'plus, 'moins`}.

Les valeurs des arcs seront également des enregistrements à deux champs (pour le débit et la capacité) :

```
record canal = {débit : int, capacité : int}
```

Étape de marquage. Elle est programmée par une première transformation qui initialise le marquage, puis une seconde qui applique les règles précédemment décrites :

```
trans init = {
    s / source(s) => s + {marque = ‘plus} ;
    x             => x + {~marque}
}

trans marquage = {
    i :marqué, j :nonMarqué / (edge(i,j).débit < edge(i,j).débit)
    => i, j+{marque = ‘plus} ;
    i :nonMarqué, j :marqué / (edge(i,j).débit > 0)
    => i+{marque = ‘moins}, j
}
```

La fonction `source` est un prédicat vérifiant que son argument est bien s (on utilisera également le prédicat `cible` pour t) La fonction `edge` prend deux sommets et renvoie la valeur de l'arc qui les relie. Il faut également rajouter un test d'orientation de l'arc que nous n'avons pas mis pour simplifier le code. Les éléments filtrés étant du type `site`, les déplacements se font suivant les cellules de dimension 1. La somme de deux enregistrements `r + r'` calcule la fusion asymétrique : le résultat contient tous les champs des enregistrements `r` et `r'` avec une priorité donnée à `r'` si il y a collision des champs.

Augmentation du graphe. La recherche de la chaîne augmentante est simple à décrire par une itération :

```
trans augmentante = {
    (c :canal)* as C / source(sommet_i(hd(C))) && cible(sommet_j(last(C)))
    => augmente(C)
}
```

Les fonctions `sommet_i` et `sommet_j` retournent les sommets de l'arc passé en paramètre. La fonction `augmente` prend le chemin d'arcs en argument et l'augmente. Cela met à jour le flot du réseau. Si la règle ne filtre pas, le flot courant est maximal. Pour appliquer l'algorithme sur un graphe g, il suffit d'écrire :

```
fun ff(x) = augmentante(marquage[‘iter=‘fixpoint](init(x)))
ff[‘iter=‘fixpoint](g)
```

4 Conclusion et perspectives

Cet article n'aborde qu'une partie des notions développées en `MGS` pour faciliter la modélisation et la simulation de systèmes dynamiques complexes. L'approche a été appliquée avec succès à de nombreux exemples biologique comme la croissance d'une tumeur, la simulation d'un nid de fourmis, la différentiation en hétérocyste des cellules de l'*Anabaena Catenula*... ainsi qu'à des exemples de nature plus algorithmique : calcul des nombres premiers, divers algorithmes de tri, recherche d'un chemin hamiltonien, calcul de l'enveloppe convexe d'un ensemble de points, *etc*. Ces exemples, ainsi que la version actuelle de l'interprète `MGS`, sont librement disponibles à partir de la page `http://mgs.lami.univ-evry.fr`.

D'un point de vue formel, la notion de collection topologique s'apparente à la notion de *chaîne* et la notion de transformation correspond à certaines *cochaînes*, concepts développés en algèbre homologique [**?**]. L'intégration des G-cartes dans `MGS` est très semblable à la paramétrisation d'objet géométrique via le λ-calcul proposée dans [**?**]. Cependant, une collection topologique décore chaque cellule par une valeur manipulée à travers la notion de transformation. Du point de vue des langages fonctionnels, une transformation étend la notion de réécriture de termes ou de fonction définie par cas (filtrage) à des structure de données non algébriques.

Nous étudions actuellement plusieurs extensions du langage de motif et la complexité des différents algorithmes de filtrage nécessaires. Nous travaillons aussi sur le typage des collections topologiques, la compilation du langage et diverses applications nécessitant de changer dynamiquement la topologie des collections.

Remerciements

Les auteurs remercient vivement Yves Bertrand, Pascal Lienhardt, Agnès Arnoux et tous les membres du projet MOKA pour leur assistance amicale et leur conseils. Leurs remerciements vont aussi à Jean-Louis Giavitto, Julien Cohen, Florent Jacquemard, Fréderic Gruau et Franck Delaplace pour leurs nombreuses questions, leur remarques constructives et leurs encouragements. Ce travail est soutenu par le CNRS, le GDR ALP, IMPG, l'Université d'Évry et GENOPOLE-Évry.

Références

[BB92] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96 :217–248, 1992.

[BFM01] Jean-Pierre Banâtre, Pascal Fradet, and Daniel Le Métayer. Gamma and the chemical reaction model : Fifteen years after. *Lecture Notes in Computer Science*, 2235 :17–? ?, 2001.

[DL02] Jean-François Dufourd and Sven Luther. Parametrizing geometric objects using λ-calculus. In *Proceedings of the 18th spring conference on Computer graphics*, pages 185–194. ACM Press, 2002.

[FF56] Lester R. Ford, Jr. and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8 :399–404, 1956.

[GH91] Ralph E. Griswold and David R. Hanson. String processing languages. Technical Report TR-306-91, Princeton University, Computer Science Department, February 1991.

[Gia03] J.-L. Giavitto. Invited talk : Topological collections, transformations and their application to the modeling and the simulation of dynamical systems. In *Rewriting Technics and Applications (RTA'03)*, volume LNCS 2706 of *LNCS*, pages 208 – 233, Valencia, June 2003. Springer.

[GM02] J.-L. Giavitto and O. Michel. The topological structures of membrane computing. *Fundamenta Informaticae*, 49 :107–129, 2002.

[Ler03] Xavier Leroy *et. al.* *The Objective Caml system release 3.07.* INRIA, http ://www.ocaml.org, september 2003.

[Lie94] Pascal Lienhardt. N-dimensional generalized combinatorial maps and cellular quasi-manifolds. *Journal on Computational Geometry and Applications*, 4(3), 1994.

[Pau00] Gheorghe Paun. Computing with membranes. *Journal of Computer and System Sciences*, 1(61) :108–143, 2000.

[Rey87] Craig W. Reynolds. Flocks, herds, and schools : A distributed behavioral model. In *SIGGRAPH*, volume 21(4) of *Computer Graphics*, pages 25–34, Anaheim, California, July 1987. ACM.

[RS92] Grzegorz Rozenberg and Arto Salomaa. *Lindenmayer Systems.* Springer, Berlin, 1992.

[TM87] T. Toffoli and N. Margolus. *Cellular automata machines : a new environment for modeling.* MIT Press, Cambridge, 1987.

[Ton74] Enzo Tonti. The algebraic-topological structure of physical theories. In P. G. Glockner and M. C. sing, editors, *Symmetry, similarity and group theoretic methods in mechanics*, pages 441–467, Calgary, Canada, August 1974.

[WS81] T.A. Witten and L. M. Sander. Diffusion limited aggregation, a kinetic critical phenomena. *Physical Review Letters*, 47 :1400–1403, 1981.