Représentation et Manipulation de Structures Topologiques dans un Langage Fonctionnel

Antoine Spicher[1]

1: Laboratoire de Méthodes Informatiques,
523 place des terrasses de l'agora
91000 ÉVRY CEDEX, France
`aspicher@lami.univ-evry.fr`

L'objectif du projet MGS est de développer un langage de programmation dédié à la modélisation et à la simulation de phénomènes dont la structure est dynamique. Les langages dédiés, souvent déclaratifs et organisés autour d'un petit noyau, proposent de nouvelles abstractions et notations orientées vers un domaine d'application particulier, rendant la programmation et la réutilisation de code plus faciles.

Les processus auxquels nous nous intéressons sont des systèmes hautement structurés suivant une organisation hiérarchique. La spécification de tels systèmes passe souvent par la caractérisation d'un état décrit par l'association d'attributs (masse, température, charge, etc.) à une structure spatiale.

L'idée fondamentale de MGS est de permettre la spécification de ces états au moyen d'une nouvelle structure de données [GM02a] fondée sur l'organisation topologique des systèmes modélisés. D'autre part, la fonction d'évolution de ces systèmes correspond à une fonction qui transforme un état du système en un autre. La spécification de ces fonctions d'évolution est simplifiée par l'utilisation d'une définition par cas reposant sur un langage de filtre expressif. MGS peut être considéré comme un langage fonctionnel standard étendu d'une nouvelle structure de données, les *collections topologiques*, et d'un nouveau mécanisme de définition de fonction, les *transformations*. Il apparaît que les notions développées pour la simulation de systèmes dynamiques à structure dynamique (également appelés $(SD)^2$) nous conduisent vers un nouveau style de programmation permettant une description concise de plusieurs algorithmes fondamentaux sur les ensembles, les séquences, les graphes, etc.

Après une brève présentation du langage MGS, nous nous intéresserons à la proposition de Dufourd et Luther [DL02] pour la paramétrisation d'objets géométriques par le λ-calcul, que nous étendrons pour MGS. Cela nous permet de construire et de composer des objets géométriques comme un nouveau type de collection topologique impossible à définir dans les versions antérieures du langage. Enfin, nous présenterons plusieurs exemples illustrant l'utilisation des transformations pour modéliser différents processus localisés sur différents objets. Ceux-ci amènent la possibilité de manipuler des éléments de différentes dimensions tout en facilitant la spécification d'un voisinage irrégulier.

1. Présentation du langage MGS

MGS est un langage fonctionnel strict, complet, impur, dynamiquement typé. Dans cette section, nous allons uniquement nous concentrer sur les notions requises pour comprendre le reste de l'article. Nous ne décrirons que les différences majeures qui distinguent MGS d'un langage fonctionnel standard comme OCAML [Ler04].

1.1. Les valeurs atomiques.

Comme tout langage fonctionnel, `MGS` permet la manipulation, en plus des valeurs atomiques de base (comme les entiers, les flottants, les booléens, les chaînes de caractères, les symboles...) et de leurs opérations, des fonctions comme des valeurs. Le constructeur **`fun`** est utilisé pour définir de nouvelles fonctions nommées comme, par exemple, `fun max(x, y) = if (x > y) then x else y fi`. On peut également spécifier des paramètres optionnels ainsi que leur valeur par défaut au cas où la fonction serait appliquée sans précision sur la valeur de l'option. Ainsi, pour la fonction suivante, `fun succ[inc=1](x) = x + inc`, l'application `succ(0)` renvoie `1` alors que `succ[inc=3](0)` retourne `3`.

1.2. Les Collections Topologiques.

Nous allons nous intéresser ici à la propriété de `MGS` qui consiste à manipuler des entités structurées suivant une *topologie abstraite*. Un ensemble de ces entités est appelé *collection topologique*. Ici, le terme « topologique » est motivé par le fait que chaque type de collection définit une relation de voisinage (la nature de la topologie dépend des propriétés de cette relation) et introduit naturellement la notion de *sous-collection* : une sous-collection S' d'une collection S est un sous-ensemble d'éléments connectés de S, et dont l'organisation topologique hérite de celle de S.

L'utilisateur a à sa disposition un ensemble de collections prédéfinies et les moyens de définir ses propres types de collections. Bien que nous ne les présenterons pas toutes dans cet article, nous allons nous intéresser aux deux collections utilisées dans les exemples qui suivent :

- Les collections monoïdales : la relation de voisinage est définie à travers un monoïde dont le principal opérateur, dénoté par la virgule, spécifie l'insertion d'un élément. Suivant les propriétés de la virgule, on peut construire différents types de collections monoïdales. Si la virgule est associative, on construit des séquences (chaque élément a deux voisins sauf pour les extrémités) ; si la virgule est en plus commutative, on obtient des multi-ensembles (chaque élément est voisin de tous les autres). Enfin, si la virgule est en plus idempotente, on peut manipuler des ensembles.
- Les graphes de Delaunay : il s'agit de graphes non-orientés dont les sommets sont plongés dans un espace euclidien $\mathbb{R}^n$ de dimension n. La relation de voisinage est calculée à partir de plongement et de la triangulation de Delaunay : l'espace $\mathbb{R}^n$ est partitionné en cellules entourant chaque sommet du graphe et dont tout élément de la cellule est plus près de ce sommet que de n'importe quel autre sommet. Ce type de collection permet la représentation d'un voisinage répondant à la notion de plus « proche voisin ». Le plongement des éléments de la collection est spécifié par une fonction qui paramétrise le type de la collection.

Le prototype actuel de `MGS` fournit entre autre les graphes arbitraires, les graphes de Cayley (en particulier, les partitions du plan et les grilles de dimension n, circulaires ou non) et les *quasi-variétés*, nouveau type de collection topologique que nous introduirons plus loin.

Pour tout type de collection `T`, la collection vide correspondante est définie et est notée `():T`. L'opération d'*union* entre deux collections C_1 et C_2 est dénotée par la virgule : C_1, C_2. La virgule est surchargée en `MGS` et peut être utilisée pour construire n'importe quelle collection (l'ambiguïté de l'opération est levée par le type des arguments). Par exemple, `1, 1+1, 2+1, ():set` construit l'ensemble des trois éléments 1, 2 et 3, alors que `1, 1+1, 2+1, ():bag` s'évalue en un multi-ensemble contenant les mêmes éléments. Pour simplifier la notation, il n'est pas nécessaire de préciser la séquence vide `():seq` lors de la définition de séquence : l'expression `1, 2, 3` est équivalente à `1, 2, 3, ():seq`.

1.3. Transformations.

Les transformations constituent la seconde contribution de `MGS`. Elles sont utilisées pour spécifier la fonction d'évolution des $(\mathrm{SD})^2$. La *transformation globale* d'une collection topologique S consiste en l'*application parallèle* d'un ensemble de *transformations locales*. Celles-ci sont spécifiées par une règle r qui définit le remplacement d'une sous-collection par une autre. L'application d'une règle de réécriture de la forme $\sigma \Rightarrow f(\sigma, ...)$ sur une collection S revient à :

1. sélectionner une sous-collection s_i de S dont les éléments sont filtrés par le motif σ,
2. calculer une nouvelle sous-collection s'_i comme l'application de la fonction f à s_i et à son voisinage,
3. et spécifier la substitution de s_i par s'_i.

On remarquera que l'application des règles locales suit une stratégie *maximale parallèle*, c'est-à-dire que toutes les instances *distinctes* s_i filtrées par le motif σ sont substituées *simultanément* par les s'_i, et qu'il n'existe pas d'instance non-filtrée distincte des s_i. De plus, on considère des priorités entre les règles locales : on cherche tout d'abord à filtrer des instances pour la première règle jusqu'à ce qu'il n'y en ait plus, puis on filtre les instances de la seconde règle, et ainsi de suite pour toute les règles.

Le langage `MGS` implémente la notion de transformation de collections topologiques dans un environnement fonctionnel : les collections sont donc des nouveaux types de valeur, de même que les transformations qui ne sont que des fonctions s'appliquant sur des collections et qui sont définies à travers une syntaxe spécifique à base de règles. Les collections et les transformations peuvent être passées en argument ou retournées en résultat.

Les motifs de chemin. Le motif σ en partie gauche d'une règle filtre un ensemble d'éléments interagissant sous la forme d'une sous-collection. Or celle-ci peut se présenter sous une forme totalement arbitraire, et souvent à spécifier. En revanche, il est pratique, et en fait peu restrictif, d'énumérer séquentiellement en suivant la relation de voisinage ces éléments. Une telle séquence est appelée *chemin*.

Un chemin ***Pat*** est une séquence ou une répétition ***Rep*** de *filtres de base*. Un filtre de base ***BF*** permet de filtrer un élément. La grammaire suivante, non-exhaustive, présente cette décomposition :

$$\textit{\textbf{Pat}} ::= \textit{\textbf{Rep}} \mid \textit{\textbf{Rep}}\ \texttt{,}\ \textit{\textbf{Pat}} \qquad \textit{\textbf{Rep}} ::= \textit{\textbf{BF}} \mid \textit{\textbf{BF}}\texttt{/}exp \qquad \textit{\textbf{BF}} ::= \texttt{cte} \mid \mathrm{id} \mid \texttt{<undef>}$$

où `cte` est une valeur quelconque, id est une variable de motif et permet de nommer un des éléments filtrés, *exp* est une expression booléenne. Voici quelques explications pour une interprétation systématique des motifs :

constante : une valeur quelconque `cte` filtre un élément de la même valeur. Ainsi, 123 filtre un élément dont la valeur est 123.

éléments vides : le symbole `<undef>` filtre un élément dont la valeur n'est pas définie, c'est-à-dire une position sans valeur associée.

variable : une variable de motif a filtre exactement un élément et lui associe le nom de variable a. Cette variable peut apparaître à plusieurs reprises dans les gardes de la suite du motif et dénote la valeur de l'élément filtré.

voisinage : b**,** p est un motif filtrant un chemin qui commence par un élément filtré par b et dont la suite du chemin est filtrée par p ; le premier élément de p est voisin de b suivant la topologie de la collection.

garde : p/exp filtre un chemin filtré par p tel que l'expression booléenne exp est évaluée à `vrai`. Par exemple, le motif x, y `/` y `>` x filtre deux éléments voisins x et y tels que la valeur associée à y est supérieure à celle de x.

itération : $p*$ filtre une sous-collection composée d'une répétition de sous-collections filtrées par p.

alias : p *as* x permet de nommer un chemin filtré par p avec le nom x. Les éléments filtrés par des motifs de longueur 1 d'une règle sont tous différents. Le chemin ne peut donc pas s'auto-intersecter.

Les éléments filtrés par les filtres de base dans une règle sont distincts. Ainsi, un chemin ne peut pas présenter de cycle. Le nom d'une variable de motif ne peut être utilisé qu'une seule fois. Par exemple, le motif x,x est interdit. Néanmoins, il est possible de réécrire ce motif de la façon suivante : x,$y/y == x$.

Partie droite des règles. La partie droite d'une règle spécifie la collection à substituer à la place de la sous-collection filtrée en partie gauche. On peut voir cette substitution différemment : le chemin étant défini par une séquence d'éléments, la partie droite peut également être spécifiée par une expression s'évaluant en une séquence d'éléments. Ainsi, la substitution est donnée élément par élément : le $i^{\text{ème}}$ élément du chemin filtré est remplacé par le $i^{\text{ème}}$ élément de la partie droite. L'écriture des règles s'en trouve alors plus concise.

Pour certains types de collection, il est possible de substituer une sous-collection par une collection de forme différente. De telles collections sont dites *leibniziennes* et sont opposées aux collections *newtoniennes*. Les ensembles, les multi-ensembles, les séquences, les graphes de Delaunay sont des exemples de collections leibnizienne. En revanche, une grille à deux dimensions est un collection newtonienne : on ne peut pas substituer un sous-ensemble d'une grille par un autre sous-ensemble de forme différente sans détruire la topologie de la gille.

1.4. Quelques courts exemples.

Voici trois exemples montrant l'utilisation de transformation sur des séquences d'éléments.

1.4.1. La fonction factoriel.

Ce premier exemple permet de calculer le factoriel d'un nombre n quelconque. L'idée est de générer un multi-ensemble contenant tous les entiers compris entre 1 et n, puis de remplacer toute paire d'entiers de ce multi-ensemble par leur produit (jusqu'à ce qu'un point fixe soit atteint). Le multi-ensemble est généré par la fonction `iota` :

```
fun iota = function 0 -> ():bag
                  | n -> n, iota(n-1)
```

(le mot clef `function` est utilisé pour introduire la définition d'une fonction par cas sur des valeurs atomiques). La transformation calculant factoriel à partir du multi-ensemble s'écrit comme suit :

```
trans fact = { x,y  ⇒  x*y }
```

Le mot clef `trans` introduit la définition d'une transformation sous la forme d'un ensemble de règles. Ici, il n'y en a qu'une. L'application de cette transformation pour l'entier 5 par exemple est donnée par `fact[iter=fixpoint](iota(5))`. Le paramètre optionnel `iter` indique que la fonction `fact` doit être itérée, et sa valeur `fixpoint` qu'un point fixe doit être atteint. Le résultat obtenu est `125,():bag`, un multi-ensemble ne contenant plus que le seul entier 125, comme attendu.

1.4.2. Chemin Hamiltonien.

Un graphe est une collection topologique en `MGS` et il est facile de spécifier la recherche d'un chemin hamiltonien en utilisant le programme suivant :

```
exception Not_Found, Found;;
trans H = {
    x* / size(x) = size(self) => raise Found(x)
};;
```

```
fun hamilton(g) = try
                    H(g) ; raise Not_Found
                  with Found path -> path ; ;
```

La transformation H filtre une répétition par le motif $x*$, les éléments du chemin filtré étant voisins deux à deux. Le mot clef `self` réfère à la collection passée en argument à la transformation, ici le graphe. La taille d'un graphe correspond au nombre de ses sommets. On cherche donc un chemin dont la longueur est exactement le nombre de sommets dans le graphe, sachant qu'un chemin ne peut pas s'auto-intersecter. Il s'agit bien de la définition d'un chemin hamiltonien. La partie droite de la règle lève une exception rattrapée dans la fonction `hamilton`. Le retour normal de `H` est suivi de la levée de l'exception `Not_found` dans `hamilton`. On remarquera ici l'intégration complète entre fonctions et transformations : une transformation est une fonction comme une autre.

1.4.3. Banc de poissons.

Dans cet exemple, nous cherchons à simuler le rassemblement et le déplacement d'un banc de poissons. Ce modèle est inspiré de la simulation d'un groupe d'oiseaux proposée par U. Wilensky et par le développement des comportements de déplacement de *boids* (des créatures virtuelles se déplaçant en groupe) inventés par C. Reynolds [Rey87]. L'algorithme utilisé ici est très proche du leur.

Les poissons semblent suivre la même direction ; néanmoins, ils ne sont pas guidés par un meneur particulier. En fait, chaque poisson applique le même ensemble de règles pour décider de la direction de ses mouvements. Le plus simple des modèles est constitué de trois règles : (1) pour éviter les voisins trop proches, (2) pour rejoindre le groupe quand il est trop loin, et (3) pour suivre la même direction globale que ses voisins. Ces trois règles seront détaillées plus loin. Tout d'abord, on s'intéressera à la représentation d'un poisson en `MGS`.

Représentation des poissons et du banc. Un poisson est représenté par deux valeurs : sa position et sa direction. Pour simplifier l'exemple, on ne considère qu'une représentation en deux dimensions, sachant que le passage à la troisième dimension est immédiat. On utilise un enregistrement (une structure de donnée équivalente à une `struct` en `C`) composé de trois champs : deux pour la position (`x` et `y`), et un pour la direction comme un angle par rapport à l'axe des abscisses (`theta`). Les poissons ont une vitesse uniforme qui peut augmenter si un poisson est trop loin du groupe.

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

La définition d'un type T crée automatiquement un prédicat du même nom permettant à l'utilisateur de vérifier que son argument est du type T.

La triangulation de Delaunay est un moyen facile et pratique pour générer un voisinage naturel entre les poissons. Cette collection topologique est utilisée ici pour représenter le banc. Elle est définie à partir d'une fonction retournant la position d'un poisson dans l'espace. Ainsi, on définit le type `D2` comme un graphe de Delaunay de dimension 2 et dont les éléments sont des valeurs de type `Position` :

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

La fonction anonyme qui apparaît en partie droite de la définition du type `D2`, est utilisée pour associer à chaque élément d'une collection du type `D2` une position qui sera utilisée pour calculer le voisinage de cet élément. Ici, cette fonction, appelée plongement, se contente de retourner la position qui fait partie de l'état d'un poisson.

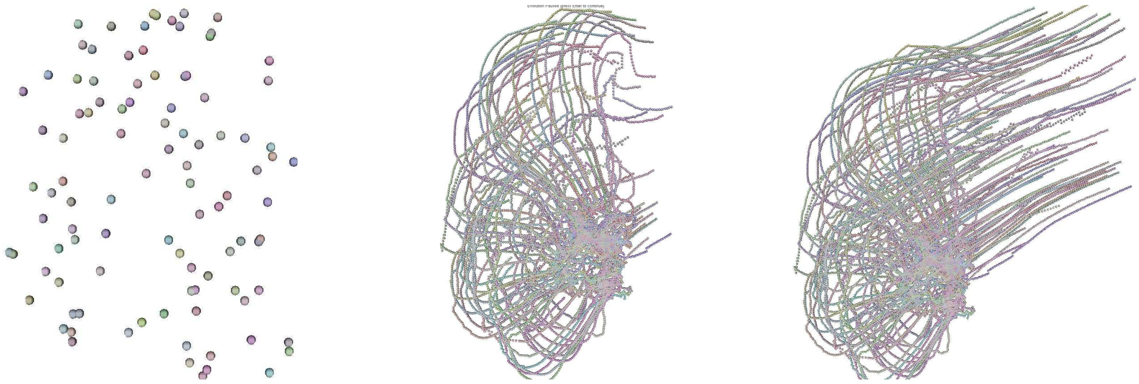


Fig. 1 – Trajectoire d'un banc de 50 poissons (le programme MGS correspondant est composé de moins de 100 lignes de code, y compris la sortie des résultats dans un fichier). À gauche : l'état initial où chaque poisson a une direction aléatoire. Au centre : la configuration après 300 itérations. À droite : après 900 itérations de la fonction d'évolution `behavior`.

Le comportement des poissons. Le modèle de base se résume en trois règles de comportement simples qui décrivent comment un poisson manœuvre suivant son entourage en modifiant sa position et sa vitesse :

1. ***Séparation*** : quand un poisson est trop proche de ses voisins, il change sa direction.
2. ***Cohésion*** : quand un poisson est trop loin de ses voisins, il essaie de rejoindre les autres en augmentant sa vitesse et en se dirigeant vers eux.
3. ***Alignement*** : quand un poisson n'est ni trop proche, ni trop loin de ses voisins, il choisit de se diriger suivant la moyenne des directions de ses voisins.

La transformation suivante implémente le comportement des poissons. On ne décrira ici que la règle concernant l'alignement des poissons, les autres suivant le même principe :

```
trans behavior = {
  separation = ... => ... ;
  cohesion   = ... => x... ;
  alignment  =
    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(noise),
                  y = a.y + speed*sin(dir) + random(noise),
                  theta = dir}
         end
}
```

Pour la règle `alignment`, on commence par sommer les directions des voisins pour en calculer la moyenne `dir`. La fonction `neighborsfold` est appliquée sur l'ensemble des voisins du poisson `a`. Le nombre de voisins est calculé de la même façon. Finalement, on met à jour l'enregistrement `a` : le poisson bouge et suit maintenant la nouvelle direction `dir`. On introduit également un peu de bruit pour rendre la simulation plus réaliste. La figure 1 montre trois étapes de cette simulation.

2. Construction de complexes cellulaires dans un langage fonctionnel

2.1. Au delà des graphes : les complexes cellulaires.

La relation de voisinage utilisée dans les deux exemples précédents peut être décrite par un *graphe de voisinage* : les positions en sont les sommets, et deux positions sont reliées par un arc si elles sont voisines dans la collection. Par exemple, les séquences sont des graphes linéaires et les ensembles sont des graphes complets. En fait, toute collection topologique peut être représentée sous la forme d'un graphe.

Cependant, cette structure de graphe de voisinage apparaît parfois trop limitée. Pour illustrer cela, nous allons nous intéresser à une loi mise en jeu en électrostatique, la loi de Gauss : elle relie la charge électrique contenue dans un volume fermé à la valeur du flux électrique à sa surface, sous la forme d'une équation différentielle. On remarque ici que la charge électrique est une quantité tridimensionnelle, alors que le flux électrique est bidimensionnel. De plus, cette loi peut s'appliquer sur un volume dont la forme est arbitraire et donc difficilement représentable. Le lecteur intéressé par ce problème peut se référer aux travaux d'E. Tonti [Ton74] qui montre que l'expression de nombreuses lois physiques est intimement liée à la dimension des objets qu'elles caractérisent.

Ces applications nous amènent à considérer qu'une collection topologique correspond à un espace décoré par des valeurs : une valeur est associée à chaque partie élémentaire de cet espace. On retrouve là une notion analogue à la notion de champ de données [Lis93]. Les champs de données utilisés en parallélisme (data-field) correspondent à une généralisation des régions arbitraires de $\mathbb{Z}^2$ et généralisent la notion de tableau. La notion de collection topologique permet d'aller plus loin en considérant des espaces plus structurés et/ou plus complexes que $\mathbb{Z}^n$. Il faut considérer tous les éléments de n'importe quelle dimension, ainsi que leurs interactions dans la description des systèmes, et la possibilité de leur associer une valeur.

Des collections topologiques arbitraires vues comme des complexes cellulaires. Le développement d'une notion générale de collection topologique peut s'appuyer sur les outils développés dans le cadre de la *topologie algébrique combinatoire* pour construire et manipuler de manière discrète des espaces complexes.

Un *complexe cellulaire* est un ensemble d'objets de différentes dimensions appelés *cellules topologiques de dimension* k, ou plus simplement k*-cellule*. Une k-cellule permet de représenter de façon abstraite un espace de dimension k : les 0-cellules sont des points, les 1-cellules des courbes, les 2-cellules des surfaces, les 3-cellules des volumes, etc. Pour représenter un domaine arbitrairement compliqué, il va être partitionné en des espaces plus simples de dimension différente, et chaque partition va être représentée par une k-cellule où k est la dimension de la partition représentée. Les cellules sont collées les unes aux autres suivant l'organisation de la partition définissant ainsi la *relation d'incidence* : une k-cellule c_1 est incidente à une $(k+1)$-cellule c_2 si $c_1 \subset \partial c_2$, où ∂c_2 dénote le bord de c_2 et est une relation arbitraire qui associe à toute cellule de dimension k un ensemble (éventuellement vide) de cellules de dimension $k-1$.

On peut utiliser la relation de bord entre cellules ∂ pour définir une relation de voisinage qui permettra de construire une collection topologique : *deux* k*-cellules sont voisines si elles partagent au moins une même* $(k-1)$*-cellule ou si elles sont incidentes à une même* $(k+1)$*-cellule.* On peut également aller au-delà des dimensions $(k-1)$ et $(k+1)$ en généralisant ce voisinage à la dimension p : *deux* k*-cellules sont* p*-voisines si elles sont incluses dans le bord d'une même* p*-cellule (quand* $p > k$*) ou si leurs bords ont en commun une* p*-cellule (pour* $p < k$*).* On définit finalement une collection topologique comme une fonction qui associe une valeur aux cellules d'un complexe cellulaire, dans lequel on peut utiliser le p*-voisinage* pour se déplacer.

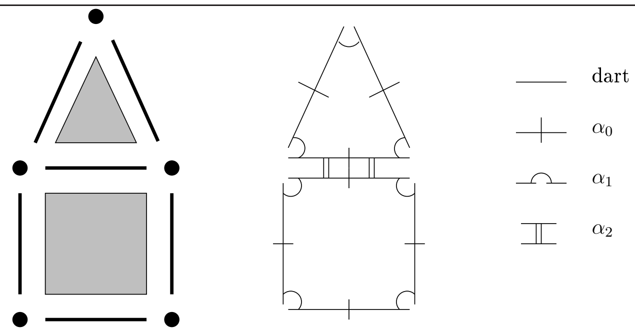


FIG. 2 – Vue d'un complexe cellule en terme de brins et d'involutions.

Cette définition des collections topologiques inclut les exemples précédents correspondant à des graphes. En effet, ces collections topologiques sont en fait des complexes cellulaires de dimension 1, où seules étaient valuées les cellules de dimension 0 (les positions), et dans laquelle on se déplace en suivant la notion de 1-voisinage, c'est-à-dire lorsque deux 0-cellules sont liées par une 1-cellule.

2.2. Les Cartes Généralisées.

L'implémentation des complexes cellulaires n'est pas immédiate et dépend de l'utilisation qu'on souhaite en faire et du type ou des propriétés des objets qu'on souhaite représenter. Il existe plusieurs familles de complexes cellulaires dont, par exemple, les *complexes simpliciaux*. Il s'agit de complexes cellulaires où les cellules sont des *simplexes*, c'est-à-dire l'enveloppe convexe de $n+1$ sommets : un triangle est un simplexe de dimension 2. Pour `MGS`, nous avons choisi de représenter les complexes par une structure de données appelée *cartes généralisées* (ou encore G-carte) [Lie91]. Les objets que l'on peut décrire par les G-cartes sont les *quasi-variétés* [Lie94].

Voici une brève présentation de la structure de G-carte. Soit $\mathcal{C}$ un complexe cellulaire. Soit N, la plus grande dimension des cellules du complexe, N est appelé la dimension de $\mathcal{C}$. On considère le *graphe d'incidence* $\mathcal{G}$ de $\mathcal{C}$, c'est-à-dire le graphe relatant la relation d'incidence entre les cellules de $\mathcal{C}$. On appelle *tuple*, la séquence de cellules $(c_N, c_{N-1}, \ldots, c_1, c_0)$ représentant un chemin dans le graphe d'incidence où c_i dénote une cellule de dimension i. On dira que deux tuples sont *i-adjacents* s'ils partagent les mêmes cellules, à l'exception de la dimension i où les cellules peuvent être différentes.

La G-carte de dimension N représentant $\mathcal{C}$ est un couple $(\mathcal{B}, (\alpha_0, \ldots, \alpha_N))$. $\mathcal{B}$ dénote un ensemble d'éléments appelés *brins*, chaque brin étant une vue abstraite d'un des tuples de $\mathcal{C}$. Le second membre du couple est un ensemble de $N+1$ involutions α_i $(0 \leq i \leq N)$. En fait, pour tout couple (b, b') de brins de $\mathcal{B}$, on aura $d' = \alpha_i(d)$ si les deux tuples représentés par b et b' sont i-adjacents (Cf. figure 2). Pour que les objets représentés par des G-cartes respectent les propriétés des *quasi-variétés*, les brins doivent également respecter l'équation suivante :

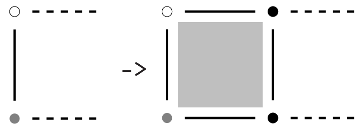
$$\forall i, j, \quad 0 \leq i < i+2 \leq j \leq N, \; \alpha_i \circ \alpha_j \text{ est une involution.}$$

2.3. G-cartes et λ-termes.

L'intégration des G-cartes dans `MGS` est similaire à la paramétrisation d'objets géométriques par le λ-calcul suggérée dans [DL02]. Dufourd et Luther y présentent une intégration des G-cartes dans un λ-calcul typé qui facilite énormément la composition des objets géométriques. Les G-cartes et leurs opérations deviennent alors des expressions fonctionnelles qui s'intègrent parfaitement dans un langage déclaratif tel que MGS. Cette utilisation du λ-calcul est intéressante car elle permet une large paramétrisation dans la construction d'objets topologiques. En effet, ils proposent tout d'abord une description de base et du premier ordre des G-cartes, pour ensuite construire des opérations plus complexes sur ces fondements. Finalement, ils ajoutent à leur prototype un côté impur pour le partage, le clonage et la réutilisation d'objets déjà définis.

La grande expressivité de ces expressions fonctionnelles autorise une spécification concise et facile d'opérations complexes sur les G-cartes par la composition de fonctions et l'utilisation de la récursivité. Le programme[1] suivant présente comment construire en MGS une ligne de n carrés (vus comme des 2-cellules à 4 côtés).

```
let add_square = function
```



```
let rec line = function  0  ->
```

```
                      |  n  ->  add_square(line(n-1))
```

La fonction `add_square` prend en argument un complexe cellulaire dont un des côtés est distingué des autres (les cercles gris et vide dénotent le bord de cet arc, les lignes pointillées représentent la partie restante du complexe cellulaire). Elle retourne alors le même complexe (les pointillés ne sont pas modifiés) auquel sera ajouté un nouveau carré construit à partir du côté distingué ; un autre arc est alors distingué. La fonction `line` utilise `add_square` et la récursivité pour construire une ligne de n carrés à partir d'un arc distingué originel (pour $n = 0$).

2.4. Intégration des complexes cellulaires dans MGS par implémentation des G-cartes.

Pour manipuler des complexes cellulaires en MGS, le langage a été étendu avec l'intégration des G-cartes suivant les idées proposées par Dufourd *et al.* Cependant, nous avons souhaité cacher au programmeur l'aspect « brins et involutions » des G-cartes qui ne s'intègre pas dans l'idée de collection topologique telle qu'elle a été présentée précédemment. On n'a donc fait *remonter* dans le langage que des primitives fonctionnelles manipulant la notion de cellule (notion non explicite dans la structure de G-carte), masquant ainsi les mécanismes internes concernant les brins. Finalement, nous avons adapté le filtrage des transformations pour manipuler ces nouvelles collections.

La nouvelle structure de données est appelée *QMF*, une référence au terme anglais *Quasi-ManiFold* pour quasi-variété. Ces collections sont représentées par un triplet (G, N, h) où G est une G-carte, N la dimension de G, et h une fonction de valuation des cellules de G. Dans la pratique, h est implémentée sous la forme d'une table de hachage. Ainsi, une QMF est un nouveau type de collection paramétrisé par une G-carte. Une cellule n'ayant pas de valeur associée prend la valeur spéciale `<undef>`.

MGS fournit plusieurs opérations pour construire des quasi-variétés : produits, extrusion, contraction, fusion de deux cellules, etc. Au niveau le plus bas, toutes les opérations proposées dans le noyau de l'outil MOKA[2], un logiciel interactif de CAO, sont disponibles dans MGS (on peut par exemple sauvegarder ou charger des G-cartes dans le format d'échange de MOKA). En fait, l'intégration des G-cartes dans MGS est une interface avec la bibliothèque de manipulation de G-cartes de MOKA.

La représentation des cellules à partir des brins et des involutions d'une G-carte n'est pas

[1]Une version codée en MGS de ce programme est disponible à l'url suivante http://mgs.lami.univ-evry.fr/ImageGallery/EXEMPLES/GmapFct/functional_gmap.mgs

[2]Une présentation du projet MOKA est disponible à l'url suivante : http://www.sic.sp2mi.univ-poitiers.fr/moka/

immédiate. En effet, une cellule c correspond à l'ensemble des brins dont le tuple associé contient c. Néanmoins, il est possible d'énumérer les brins de cet ensemble au moyen d'une orbite d'un ensemble d'involutions dénotée par $< \alpha_{i_1}, \alpha_{i_2}, \ldots \alpha_{i_k} >$. En appliquant et en composant l'ensemble des k involutions sur un seul brin, on obtient l'orbite associée à ce brin. Ainsi, si d est un brin d'une cellule c de dimension i, on peut retrouver l'ensemble des brins de c en utilisant l'orbite $< \alpha_1, ..., \alpha_{(i-1)}, \alpha_{(i+1)}, \ldots, \alpha_N >$, c'est-à-dire l'ensemble de toutes les involutions, excepté α_i qui pourrait entraîner le passage dans une autre cellule de dimension i voisine. Ainsi, pour représenter une cellule, il suffit d'avoir un de ses brins et sa dimension. Ce couple ne peut cependant pas être utilisé directement pour représenter les cellules dans `MGS`, puisqu'il n'est pas unique (il y a autant de représentations d'une cellule que de brins qui la composent). Or, cela pose problème pour la gestion des clefs de la table de hachage h de la QMF qui doivent être uniques. Nous avons choisi d'utiliser un brin *normalisé* de la cellule pour la représenter. Il s'agit du brin dont l'adresse mémoire est la plus petite. Notre utilisation de la bibliothèque proposée par `MOKA` est donc adaptée pour supporter cette normalisation des cellules.

La dernière étape pour compléter l'intégration des G-cartes dans `MGS` est l'écriture d'une interface entre l'interpréteur `MGS` programmé en `OCAML` et la bibliothèque de manipulation de G-cartes écrite en `C++`. Pour cela, nous utilisons un type semi-abstrait [Ler04, pp234] pour représenter les brins et les G-cartes du côté `OCAML`. Ces types correspondent à des valeurs du tas `OCAML` qui sont personnalisées par des fonctions de comparaison, de finalisation et de hachage *ad-hoc*. On peut ainsi utiliser la plupart des fonctions de base sur ces valeurs et résoudre efficacement les problèmes issus du glaneur de cellules d'`OCAML`.

3. Exemples

Dans cette section, nous allons présenter deux exemples de programmes utilisant des collections QMF et montrant comment on peut, grâce aux transformations, animer les valeurs associées à différentes dimensions.

3.1. Modélisation du processus de DLA.

Dans ce premier exemple, on cherche à modéliser un processus de *diffusion limitée par agrégation*, ou DLA [WS81]. Son principe est simple : un ensemble de particules diffuse de façon aléatoire dans un espace donné. À l'état initial, une des particules est fixée. Lorsqu'une particule mobile rencontre une particule fixe, elles s'agrègent et se fixent. Pour simplifier ce modèle, nous considérons qu'elles restent collées ensemble définitivement et qu'il n'y a pas de formation d'agrégat de particules mobiles.

Ce processus de diffusion est en général considéré par le biais d'un simple automate cellulaire animé par une fonction de mise à jour asynchrone, ou d'un automate de gaz sur réseau et d'un ensemble de règles d'évolution plus élaborées [TM87]. Il est très facile de suivre ce point de vue en `MGS` sur des espaces homogènes (grille carrée, hexagonale, etc.) [Gia03], mais nous allons ici nous intéresser à des espaces plus arbitraires représentés avec des QMF.

La fonction d'évolution de la DLA en `MGS`. La transformation décrivant le comportement des particules est très simple : nous allons utiliser deux symboles `‘mobile` et `‘fixed` pour représenter respectivement les particules mobiles et fixes (les symboles sont des constantes définies par un nom commençant par une apostrophe inversée). Les particules suivent deux règles d'évolution ; il y a donc deux règles dans la transformation :

1. la première concerne l'agrégation : si une particule mobile est voisine d'une particule fixe, elle devient fixe (sans changer de position) ;

2. la seconde représente la diffusion : si une particule mobile est voisine d'une place vide (une position décorée par `<undef>`), elle quitte sa position courante qui devient vide, pour occuper la place libre qui lui est voisine.

On notera que l'ordre des règles a son importance car la première est prioritaire sur la seconde. On écrit :

```
trans dla = {
    ‘mobile, ‘fixed   =>  ‘fixed, ‘fixed ;
    ‘mobile, <undef>  =>  <undef>, ‘mobile
}
```

État initial du système. La façon la plus élégante d'initialiser la collection représentant l'espace où diffusent les particules, est de définir une transformation qui construit l'état initial. Pour cela, nous considérons qu'aucune position n'a de valeur associée, c'est-à-dire qu'elles sont toutes décorées par la valeur spéciale `<undef>`. On commence par créer la particule fixe et on distribue ensuite de façon aléatoire les particules mobiles partout ailleurs dans la collection :

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

La première règle est appliquée une seule fois car dès la première occurrence trouvée, la garde `flag` devient fausse. La seconde règle s'applique avec une probabilité de 0.25.

Application sur différentes topologies. Sur la figure 3, on peut voir différents types d'objets, construits avec des G-Cartes. En fait, le changement de topologie n'affecte en rien la transformation qui peut être appliquée aussi bien sur des QMF, comme nous allons le voir, que sur des grilles ou sur des multi-ensembles. On voit apparaître ici le caractère *polytypique* des transformations. Un système d'inférence de type polytypique a été développé pour `MGS` [Coh03, Coh04].

Dans les exemples présentés ici, les collections sont de dimension 2 et les valeurs (représentant les particules) sont associées aux 2-cellules. En ce qui concerne le voisinage, nous avons arbitrairement choisi d'utiliser le 1-voisinage, c'est-à-dire que deux 2-cellules sont voisines si elles ont un côté (de dimension 1) en commun. L'avantage des QMF sur d'autres types de collections topologiques plus homogènes, est la large palette d'objets représentables, où les cellules peuvent présenter un voisinage hétérogène : ici, on remarquera que sur la sphère, les cellules situées aux pôles ont 3 voisines, alors que les autres en ont 4. Sur le haut de la figure, les différents objets sont présentés sous leur forme de G-carte. En revanche, sur la bas de la figure, on ne représente uniquement que les 2-cellules portant une valeur lorsqu'un point fixe est atteint.

3.2. Problème du flot maximum

Un graphe est un complexe cellulaire de dimension 1 dont les sommets sont des 0-cellules et les arcs des 1-cellules. Nous allons profiter de cette propriété pour représenter un graphe par une QMF en `MGS`, l'avantage étant la possibilité de valuer les arcs. Cette section illustre la grande expressivité des transformations dans l'écriture d'algorithmes compliqués. De plus, l'implémentation décrite ci-dessous présente une alternative à la programmation classique de cet algorithme. Cependant, aucune comparaison n'est donnée ici en ce qui concerne son efficacité.

L'algorithme résolvant le problème du flot maximum requiert la valuation des 0-cellules et des 1-cellules. Il peut être écrit en `MGS` avec deux transformations agissant chacune sur une dimension différente.

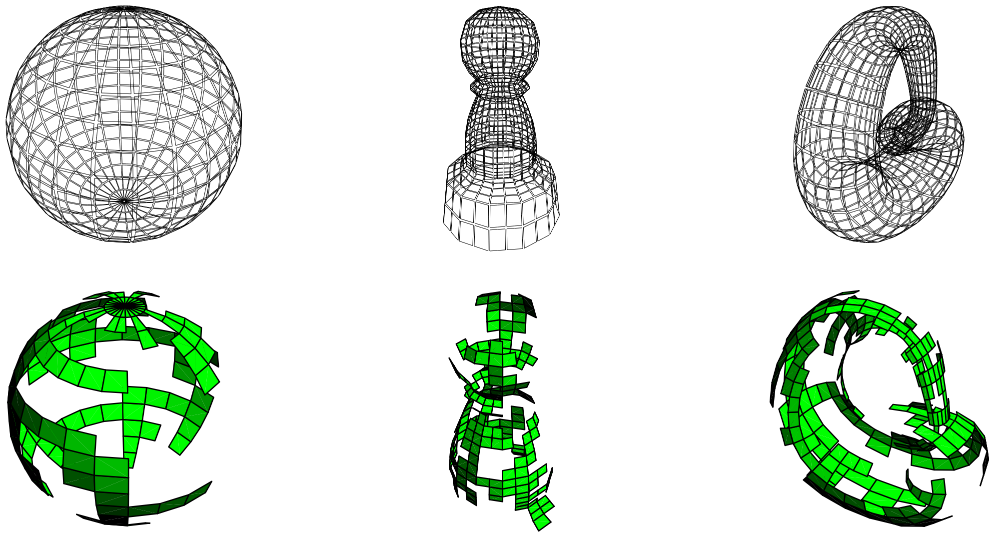


FIG. 3 – Le processus de DLA sur des objets complexes. En haut, différents objets sont représentés : de gauche à droite : une sphère avec 18 méridiens et 24 parallèles, un pion d'un jeu d'échec et une bouteille de Klein. La ligne du bas présente les mêmes objets dans l'état final du processus de DLA.

Algorithme de Ford-Fulkerson. On cherche à représenter un réseau de communication entre différents sites. Les liens entre les sites sont des canaux dont le débit est limité par une capacité maximale. On appellera *source* et *target* deux des sites du réseau tels que *source* est la source du réseau d'où provient l'ensemble des données qui circulent dans le réseau (aucun flot n'entre dans *source*), et *target* est la cible du réseau où l'ensemble des données doit arriver. Le problème qu'on se pose est : quel est le débit maximal que le réseau peut supporter, c'est-à-dire le débit maximal que *source* peut relâcher, que *target* peut supporter, mais sans saturer les canaux du réseau ?

Le réseau est représenté par un graphe ; les sites sont les nœuds et les canaux des arcs orientés. L'algorithme de Ford-Fulkerson [FF56] résout ce problème en calculant l'une des solutions à partir d'un état initial *admissible* du réseau. Un flot est admissible si :

– pour tout site i, différent de *source* et de *target*, le débit entrant dans i est égal au débit sortant de i,
– pour tout canal c entre deux sites, le débit est positif et inférieur à la capacité de c.

Un flot admissible trivial est le flot nul : aucune donnée ne circule dans le réseau. L'algorithme suit deux étapes :

1. le marquage : on suppose que le flot sortant de *source* est légèrement augmenté. Cette augmentation est dénotée par la marque `'plus` associée au sommet *source*. On essaie alors d'étendre l'augmentation à tout le réseau : soit c un arc orienté des sites i à j. Si i est marqué, j non-marqué et c non-saturé, on marque j par `'plus`. À l'inverse, si j est marqué, i non-marqué et que le débit sur c est non nul, i est marqué par `'minus`. On itère ce marquage sur l'ensemble du réseau jusqu'à atteindre un point fixe.
2. Augmentation du flot : si *target* est marqué, il existe un chemin entre *source* et *target* dont les débits des arcs peuvent être modifiés pour augmenter le flot global du réseau. Ce chemin est une *chaîne augmentante*. Après avoir calculé la valeur de l'augmentation, on met à jour le réseau et on réitère les deux étapes. Si *target* n'est pas marqué, le flot est maximal.

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

Définition des types. Les valeurs associées aux cellules sont typées. Un site a un nom et peut être marqué ou non. On utilisera un enregistrement pour les représenter :

```
record site     = {name : string}
record marked   = site + {mark}
record unmarked = site + {~mark}
```

Un enregistrement `marked` contient tous les champs d'un enregistrement de type `site`, avec en plus le champ `mark`. En revanche, un enregistrement `unmarked` contient les champs d'un `site`, mais ne doit pas avoir de champ `mark`. Le champ `mark` prendra sa valeur dans l'ensemble des symboles `{‘plus, ‘minus}`.

Les valeurs des arcs sont également des enregistrements avec deux champs (pour la valeur du débit et sa capacité maximale) :

```
record channel = {flow : int, capacity : int}
```

Étape de marquage. On commence par programmer une transformation qui initialise le marquage, et une autre pour appliquer les règles de marquage décrites précédemment :

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

trans marking = {
    i : marked, j : unmarked / (edge(i,j).flow < edge(i,j).capacity)
    => i, j+{mark = ‘plus};
    i : unmarked, j : marked / (edge(i,j).flow > 0)
    => i+{mark = ‘minus}, j
}
```

La fonction `source` est un prédicat qui vérifie que son argument est *source* (il existe un prédicat équivalent `target` pour *target*). La fonction `edge` prend deux sommets comme paramètres et retourne la valeur de l'arc qui les relie. On doit également ajouter un test d'orientation des arcs. Ici, nous ne l'avons pas introduit pour simplifier le code. Les éléments filtrés sont de type `site`. Les déplacements sont donc faits en suivant les cellules de dimension 1. L'addition entre deux enregistrements `r + r'` calcule la fusion asymétrique : le résultat est un enregistrement contenant tous les champs de `r` et `r'` avec une priorité pour les champs de `r'` quand il y a une collision.

Augmentation du flot. La chaîne augmentante peut facilement être spécifiée avec une itération :

```
trans augmenting_path = {
    (c : channel)* as C / source(vertex_i(hd(C)))
                       && cible(vertex_j(last(C)))
      => increase(C)
}
```

Les fonctions `vertex_i` et `vertex_j` retournent les sommets d'un arc. La fonction `increase` calcule l'augmentation d'un chemin d'arcs du réseau, et retourne le même chemin mais augmenté. On peut alors mettre à jour le flot. Si aucun chemin ne peut être filtré par la transformation, le flot est maximal. En fait, l'application de l'algorithme sur un graphe g est donnée par :

```
fun ff(x) = augmenting_path(marking[iter=fixpoint](init(x)))
ff[iter=fixpoint](g)
```

4. Conclusion et Perspectives

Cet article ne présente qu'une partie des notions introduites dans MGS qui peuvent être utilisées pour la simulation de systèmes complexes. Cette approche a été largement appliquée à différents phénomènes aussi bien biologiques (la croissance d'une tumeur, le comportement des fourmis, la différenciation en hétérocyste des cellules d'*Anabaena*, etc.), qu'algorithmiques (la génération des nombres premiers, des algorithmes de tri, la recherche d'un chemin hamiltonien, etc.). Ces exemples, ainsi que les sources de l'implémentation du langage, sont disponibles à l'url `http://mgs.lami.univ-evry.fr`.

La modélisation de $(SD)^2$ en considérant la structure de leurs interactions internes (entre sous-systèmes), fait partie d'un projet de recherche à long terme [Gia03]. Le point de vue topologique présenté ici fournit une description unifiée de différents modèles de calcul : les systèmes de Lindenmayer [Lin68] correspondent à des transformations de séquences, la CHAM, Gamma et les systèmes de Paun [BB90, BFM01, Pau01] peuvent être émulés par des transformations sur des multi-ensembles, et les automates cellulaires classiques [TM87] peuvent être implémentés sur des topologies régulières telles que les collections GBF (pour *Group Based Fields*, collections utilisant des champs basés sur un groupe, et généralisant la notion de tableau) [GMC02].

D'un point de vue formel, le concept de collection topologique est en relation avec celui de *chaîne topologique*, et le concept de transformation correspond à l'idée de *cochaînes*. Ces deux notions sont développées en algèbre homologique [GM02b]. L'intégration des G-cartes dans MGS est très proche de la paramétrisation d'objets géométriques via le λ-calcul suggéré dans [DL02]. Une collection topologique décore chaque cellule d'une valeur qui peut être animée par une transformation. Du point de vue des langages fonctionnels, une transformation étend le concept de réécriture de terme ou de fonction définie par cas (filtrage) sur des structures de données non-algébriques.

Les perspectives de ce travail sont nombreuses. Mais pour cela, il faut aller au-delà de l'animation du plongement de la collection topologique (*i.e.*, les valeurs impliquées) et pouvoir transformer la structure de l'espace sous-jacent à la collection. Par rapport au langage, l'étude du concept de collection topologique doit être poursuivi avec une étude plus fine des sortes de transformations. Plusieurs sortes de restrictions peuvent en effet être ajoutées aux transformations, amenant à identifier différents types de langages pour les motifs ou les règles. La complexité du filtrage pour ces types de motifs doit être étudiée. Enfin, nous nous intéressons également à la compilation efficace de programmes MGS.

Remerciements.

L'auteur souhaiterait remercier l'équipe du projet MOKA de l'Univ. de Poitiers, O. Michel, J.-L. Giavitto et J. Cohen du LaMI, F. Jacquemard de l'INRIA/LSV-Cachan et les membres du groupe « Simulation and epigenesis » de Génopole pour leurs apports techniques, les discussions stimulantes et les motivations biologiques. Ce projet est supporté en partie par le CNRS, le GDR ALP, IMPG, l'Université d'Évry et Génopole/Évry.

Références

[BB90] G. Berry and G. Boudol. The chemical abstract machine. In *Conf. Record 17th ACM Symp. on Principles of Programmming Languages, POPL'90, San Francisco, CA, USA, 17–19 Jan. 1990*, pages 81–94. ACM Press, New York, 1990.

[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 :17–44, 2001.

[Coh03] Julien Cohen. Typing rule-based transformations over topological collections. In Jean-Louis Giavitto and Pierre-Étienne Moreau, editors, *4th International Workshop on Rule-Based Programming (RULE'03)*, pages 50–66, 2003.

[Coh04] Julien Cohen. Typage fort et typage souple des collections topologiques et des transformations. In Valérie Ménissier-Morain, editor, *Journées Francophones des Langages Applicatifs (JFLA 2004)*, pages 37–54. INRIA, 2004.

[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.

[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.

[GM02a] J.-L. Giavitto and O. Michel. Data structure as topological spaces. In *Proceedings of the 3nd International Conference on Unconventional Models of Computation UMC02*, volume 2509, pages 137–150, Himeji, Japan, October 2002. Lecture Notes in Computer Science.

[GM02b] J.-L. Giavitto and O. Michel. The topological structures of membrane computing. *Fundamenta Informaticae*, 49 :107–129, 2002.

[GMC02] J.-L. Giavitto, O. Michel, and J. Cohen. Pattern-matching and rewriting rules for group indexed data structures. *ACM SIGPLAN Notices*, 37(12) :76–87, December 2002.

[Ler04] X. Leroy. The Objective Caml system, revision 3.07, 2004. Software and documentation available on the web at http ://pauillac.inria.fr/ocaml/.

[Lie91] P. Lienhardt. Topological models for boundary representation : a comparison with n-dimensional generalized maps. *Computer-Aided Design*, 23(1) :59–82, 1991.

[Lie94] P. Lienhardt. N-dimensional generalized combinatorial maps and cellular quasi-manifolds. *Journal on Computational Geometry and Applications*, 4(3) :275–324, 1994.

[Lin68] A. Lindenmayer. Mathematical models for cellular interaction in development, Parts I and II. *Journal of Theoretical Biology*, 18 :280–315, 1968.

[Lis93] B. Lisper. On the relation between functional and data-parallel programming languages. In *Proc. of the 6th. Int. Conf. on Functional Languages and Computer Architectures.* ACM, ACM Press, June 1993.

[Pau01] G. Paun. From cells to computers : Computing with membranes (P systems). *Biosystems*, 59(3) :139–158, March 2001.

[Rey87] Craig W. Reynolds. Flocks, herds, and schools : A distributed behavioral model. In Maureen C. Stone, editor, *Computer Graphics (SIGGRAPH '87 Proceedings)*, volume 21, pages 25–34, July 1987.

[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 phenomenon. *Phys. Rev. Lett.*, 47 :1400–1403, 1981.