MGS: transformer des collections complexes pour la simulation en biologie

Olivier Michel, Jean-Louis Giavitto & Julien Cohen

LaMI U.M.R. 8042 du CNRS,
Université d'Évry Val d'Essone
523, place des Terrasses de l'Agora,
91025 Évry, France.
[michel,giavitto,jcohen]@lami.univ-evry.fr

1. Introduction et motivations

L'objectif du projet MGS est de développer un langage dédié à la modélisation et la simulation de processus biologique 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 rend a priori plus attractifs qu'un langage généraliste et permettent une meilleure productivité en facilitant la programmation et la réutilisation.

Les processus biologiques auxquels nous nous intéressons sont des systèmes dynamiques hautement structurés et hiérarchiquement organisés, dont la structure varie au cours du temps et doit être calculée conjointement avec l'état du système. Ce type de systèmes est courant dans les modèles de croissance de plantes, en biologie du développement, dans les modèles cellulaires intégrant plusieurs échelles, dans les mécanismes de transport de protéines et de compartimentalisation, etc.

La simulation de ce type de systèmes est particulièrement complexe car il est difficile de spécifier la structure et la dynamique du système. Les lois d'évolutions 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 calculatoire, plusieurs auteurs [Man01, FMP00] ont proposé de représenter l'état du système biologique par un terme t de la forme $t_1 + t_2 + \cdots + t_n$ où un sous-terme t_i représente une entité biologique ou un message (signal, commande, information, action, etc.) adressé à une entité. La simulation de l'évolution du système biologique se fait par réécriture du terme. La partie gauche d'une règle de réécriture $e + m \to e' + m'$ correspond à un message m et à l'entité e auquel il est destiné ; la partie droite spécifie le nouvel état e' de l'entité et (éventuellement) de nouveaux messages m'. Notons que l'interaction directe entre entités peut être spécifiée par des règles de la forme $e_1 + e_2 \to e'_1 + e'_2$; la création d'une nouvelle entité par une règle de la forme $e \to e' + e''$; etc.

L'opérateur + utilisé pour combiner les entités et les messages en un terme t est commutatif et associatif. Ainsi le terme t réalise une « solution chimique » où les entités peuvent rencontrer librement les messages qui leurs sont adressés. Les entités et les messages interagissent donc de manière non organisée, simplement parce qu'ils sont présent dans la solution chimique.

Ce mécanisme est à la base du langage Gamma [BM91] et du formalisme de la CHAM [BB89], où le terme t est interprété comme un multi-ensemble.

Le problème principal de cette approche fondée sur la réécriture de termes est l'impossibilité de représenter des *interactions spatialement organisées.* Le besoin de représenter des organisations plus complexes qu'une solution chimique motive par exemple plusieurs extensions [BH00].

Un des buts du projet `MGS` est de remédier à ce problème en permettant la représentation d'organisations complexes entre des entités variables et hétérogènes, ainsi que leur transformation par des règles locales. En remplaçant la réécriture de terme par la notion de transformation locale d'une collection topologique (voir plus bas), nous espérons dépasser les limitations des formalismes précédents dans le domaine de la simulation des systèmes biologiques.

2. Les principes du langage `MGS`

Les travaux que nous présentons ici trouvent leurs inspirations dans les travaux de A. Lindenmayer sur les L systèmes [RS92], G. Paun sur les P systèmes [Pau98], G. Berry et G. Boudol sur la CHAM [BB89] ainsi que J.-P. Banâtre et D. Le Métayer sur Gamma [BM86].

Nous appelons *collection* tout ensemble « organisé » d'éléments. Les structures de données manipulées par les langages de programmation rendent compte de différentes sortes d'organisations simples : tableaux, arbres, ensembles, multi-ensembles (ou *bags*), séquences (ou listes), termes, *etc.* Le type de collections sous-jacents dans Gamma, la CHAM et les P systèmes est le multi-ensemble, la séquence pour les L systèmes et le tableau pour les automates cellulaires (AC).

Les modélisations de systèmes dynamiques que nous voulons simuler partagent les caractéristiques suivantes :

Temps et espaces discrets. La structure de l'état est une collection discrète de valeurs : un multi-ensemble pour Gamma, une hiérarchie de membranes pour un P système, un mot pour un L système et un tableau pour un AC. Cet état évolue selon une séquence discrète d'étapes temporelles.

Localité temporelle des transformations. Le calcul d'une nouvelle valeur dans un nouvel état dépend d'un nombre fini d'étapes précédentes (généralement une seule).

Localité spatiale des transformations. Le calcul d'une nouvelle collection se fait par combinaison structurelle du résultat de calculs plus élémentaires. Chacun de ces calculs élémentaires n'implique qu'un sous-ensemble statique de la collection initiale.

« Combinaison structurelle » signifie que les résultats élémentaires sont combinés dans une nouvelle collection, indépendamment de leur valeur. « Sous-ensemble statique » rend explicite le fait que seul un sous-ensemble déterminé des éléments initiaux est utilisé pour calculer une nouvelle valeur élémentaire (ceci est mesuré par exemple par le diamètre de la règle d'évolution d'un P système, le voisinage local d'un AC, le nombre de variables dans la partie droite d'une réaction Gamma ou par le contexte d'une règle dans un L système).

La principale différence entre ces langages est l'organisation des collections traitées, ce qui n'empêche pas le mécanisme de calcul d'être identique dans tous les cas: étant donné une collection C à traiter,

1. une sous-collection A est sélectionnée dans C,
2. une nouvelle collection B est calculée à partir de A (et éventuellement de son voisinage),
3. B vient se substituer à A.

voir Fig. 1. Ces trois étapes sont appelées une *transformation.* Pour spécifier une transformation il faut préciser : comment sélectionner A, comment est calculé B et parfois les contraintes de la substitution de A par B dans C.

Ce point de vue permet d'unifier dans le même cadre formel les différents modèles de calculs cités. Pour chacun des langages il suffit de choisir la bonne organisation dans la collection utilisée.

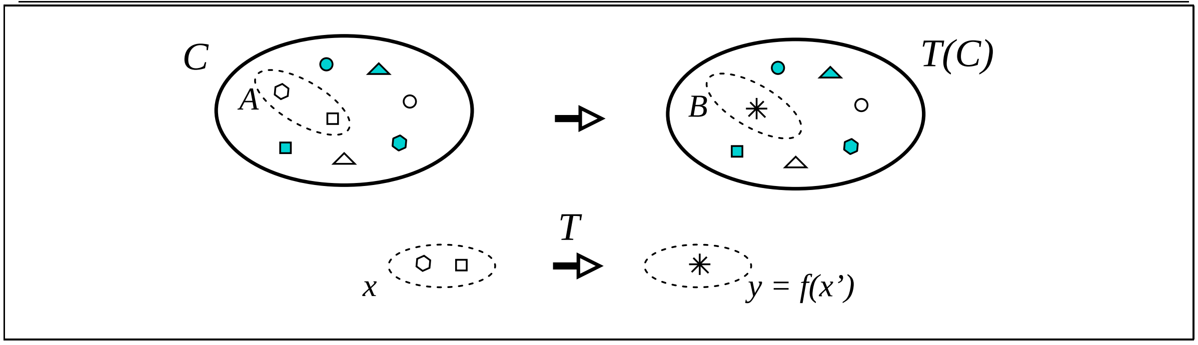


FIG. 1 – Transformation d'une collection. C est une collection. Une règle T spécifie qu'une sous-collection A dans C doit être remplacée par la collection B calculée à partir de A. La collection calculée dans la partie droite de la règle T dépend de la sous-collection filtrée dans sa partie gauche et éventuellement de son voisinage.

2.1. Collections

Ces considérations nous poussent à voir une collection comme un ensemble de positions relatives à une topologie définie par une relation de voisinage entre ses éléments. Afin de souligner l'importance de l'organisation des collections nous les appellerons *collections topologiques*.

Dans une séquence par exemple, on peut définir le voisinage d'un élément comme l'élément précédent et le suivant. Nous noterons « , » la relation de voisinage. x,y signifie que y est un voisin de x.

Nous dirons qu'un sous-ensemble S d'éléments d'une collection $\mathcal{C}$ est *connecté* lorsque le quotient de S par la clôture transitive de « , » est réduite à un seul élément. Ceci nous permet de définir la sous-séquence comme un sous-ensemble connecté d'une séquence. Ceci revient à dire que dans une séquence, les sous-séquences sont ses intervalles. On pourrait aussi choisir de contraindre plus fortement la notion de séquence en n'acceptant que les préfixes de la séquence comme sous-séquence. La seule contrainte constante est qu'une sous-collection est un sous-ensemble connecté d'une collection.

Cette approche topologique formalisant la notion de collection fait partie d'un long travail de recherche [GMS96] détaillé notamment dans [Gia00] qui étudie la notion de sous-collection et dans [GM01a] qui décrit un outil générique pour la définition de relations de voisinage. Les notions topologiques nécessaires à la définition de voisinages dans les séquences ou dans d'autres structures simples sont modestes. Néanmoins des notions plus complexes sont nécessaires pour la modélisation de certains processus biologiques [GM01b]. Par ailleurs l'approche topologique permet de traiter différents problèmes de façon uniforme.

3. Un survol de MGS

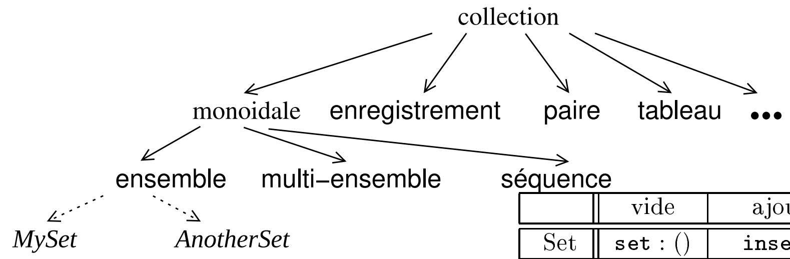
MGS est un langage fonctionnel dynamiquement typé qui permet la manipulation de collections topologiques et leurs transformations. Les collections sont justes de nouvelles sortes de valeurs et les transformations sont des fonctions définies par des *règles*. MGS est un langage applicatif : les opérateurs utilisent des valeurs pour en calculer de nouvelles mais n'agissent pas par effet de bord.

Le typage est dit dynamique car il n'est pas vérifié statiquement et les erreurs de type sont détectées à l'exécution. L'ensemble des types possibles dans MGS est très riche, adapté au filtrage (*pattern matching*) et exploitable dans les règles de transformation.

Dans cette section nous donnons un aperçu de la construction de types de collections, de la définition et de l'utilisation des transformations.

3.1. Types de collections

En plus des types de base (entiers, flottants, chaînes de caractères, λ-expressions, *etc.*) `MGS` manipule des enregistrements et certaines collections de base. Les éléments d'une collection sont des valeurs qui peuvent être de base, des enregistrements ou des collections quelconques et ne sont pas forcément du même type (collections hétérogènes). Les valeurs des champs d'un enregistrement sont quelconques, formant ainsi des objets complexes comme dans [BNTW95]. Les collections sont typées et il existe une relation de sous-typage entre leurs types. La figure 2 montre la hiérarchie des types de plusieurs sortes de collections.



	vide	ajout	singleton	fusion
Set	`set` : ()	`insert`	`single_set`(x)	`union`
Bag	`bag` : ()	`increment`	`single_bag`(x)	`sum`
Seq	`seq` : ()	`::`	`single_seq`(x)	`@`
(*)		,		,

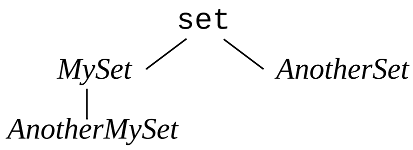
Fig. 2 – Relation de sous-typage des types collection. *MySet* et *AnotherSet* sont des types collection définis par l'utilisateur, *cf.* ci-dessous. Les types `collection` et `monoïdale` ne correspondent pas à des structures de données concrètes mais à des prédicats, *cf.* ci-dessous. Formellement, un enregistrement est un ensemble de paires (*nom-du-champ, valeur-du-champ*) mais son utilisation nécessite des opérateurs particuliers. La table à droite décrit les principales opérations pour les collections monoïdales. La ligne (*) décrit une syntaxe surchargée.

Collections monoïdales. `MGS` manipule plusieurs types de collections topologiques. Nous présentons ici plus particulièrement les types ensembles, multi-ensembles et séquences. Ces types de collections sont appelées *monoïdales* car leurs valeurs peuvent êtres construites comme les éléments d'un monoïde avec l'opérateur *ajout* « , » : une séquence correspond à une opération d'ajout sans propriétés particulières (mis à part l'associativité bien sûr) ; les multi-ensembles sont obtenus avec un opérateur d'ajout commutatif et enfin, les ensembles nécessitent un opérateur d'ajout à la fois commutatif et idempotent. L'opérateur d'ajout muni de ses propriétés induit une topologie sur la collection et la relation de voisinage : on dira que x est voisin de y dans une collection monoïdale C si C peut s'écrire U,x,y,V ou U,y,x,V où U et V sont des sous collections de C. Par exemple considérons l'ensemble $\{a,b,c\}$ (notation mathématique). Le constructeur "," associé aux ensembles est associatif et commutatif et l'ensemble considéré peut d'écrire en `MGS` $a,b,c,$`set` : (). L'associativité et la commutativité nous permettent de dire que cet ensemble peut s'écrire $a,c,b,$`set` : () et donc a et c sont voisins.

Il existe un grand nombre d'opérations génériques pour tous les types collections, basés sur l'algèbre fonctionnelle développé par exemple dans [BNTW95]. On ne les décrira pas ici dans la mesure ou elles ne sont pas pertinentes pour notre propos. La table dans la figure 2 décrit les principales opérations utilisées pour la récursion structurelle.

Sous-types définis par l'utilisateur. Il est souvent nécessaire de distinguer plusieurs collections de la même *sorte* (par exemple, des multi-ensembles imbriqués dans un autre multi-ensemble). Plusieurs solution existent pour permettre cette distinction. Ainsi, dans le formalisme des P systèmes, chaque multi-ensemble est étiqueté par un entier unique pour le référencer sans ambiguïté. Nous choisissons de distinguer entre les collections d'une même sorte par des *types*. À ce niveau, le type d'une collection doit être vu comme une étiquette qui ne change pas la structure de la collection. Les types sont organisés selon une relation de sous-typage. La relation de sous-typage hiérarchise les types en un ensemble partiellement ordonné. C'est l'élément maximal de cette hiérarchie qui constitue la sorte de la collection. Voici un exemple de déclarations de types collection :

$$\begin{array}{rcl} \texttt{collection}\ \mathit{MySet} & = & \texttt{set}; \\ \texttt{collection}\ \mathit{AnotherSet} & = & \texttt{set}; \\ \texttt{collection}\ \mathit{AnotherMySet} & = & \mathit{MySet}; \end{array}$$



Ces trois déclarations définissent une hiérarchie de trois types. Le type *AnotherMySet* est un sous-type de *MySet* lui-même sous-type de `set`. Le type `set` est un type pré-défini et correspond à une sorte de collection (les autres types pré-définis sont `seq` pour les séquences, `bag` pour les multi-ensembles, `gbf` pour les *group based fields*[1] qui généralisent les tableaux et enfin les enregistrements). Le type *AnotherSet* est aussi un sous-type de `set` mais il n'est pas comparable à *MySet*.

Un type introduit lors d'une déclaration de type peut être utilisé dans un filtre (*cf.* section 3.3) ou comme prédicat pour tester si une valeur est d'un type donné. Un type de collection monoïdale peut aussi être utilisé dans la construction d'une collection par l'énumération de ses éléments :

$$1, 1+1, 2+1, 2*2, \mathit{MySet} : ()$$

est une expression s'évaluant en un ensemble de quatre entiers : 1, 2, 3 et 4. La sorte de collection est un ensemble et son type est *MySet*. En fait, l'expression « $\mathit{Myset} : ()$ » dénote une valeur de type *MySet* vide et « , » est l'opérateur d'ajout surchargé : x,X crée une nouvelle collection avec l'élément x ajouté aux éléments de la collection X ; et l'expression X,Y créé une nouvelle collection avec les éléments des deux collections X et Y.

Le type d'une collection est pris en compte pour de nombreuses opérations sur les collections. Par exemple, l'*ajout* de deux collections de type A et B donne une collection de type C correspondant à l'ancêtre commun de A et de B (sur l'exemple précédent, `set` est l'ancêtre commun de *MySet* et *AnotherSet*). Autre exemple, *MySet* est l'ancêtre commun de *AnotherMySet* et de lui-même.

3.2. Enregistrements

Dans MGS un enregistrement est une sorte de collection particulière. Un enregistrement associe des valeurs à des noms appelés *champs*. Ces valeurs peuvent être de n'importe quel type, y compris des enregistrements ou des collections. L'accès aux valeurs des champs se fait par la notation pointée : l'expression $\{a = 1, b = \texttt{"red"}\}.b$ s'évalue en `"red"`.

On peut fusionner des enregistrements avec l'opérateur $+$. L'expression $r_1 + r_2$ donne un nouvel enregistrement r possédant les champs de r_1 et de r_2. Si un champ a est présent dans r_2 alors $r.a$ vaut $r_2.a$, sinon $r.a$ vaut $r_1.a$. Cette fusion est asymétrique [Rém92].

La déclaration d'un type enregistrement se fait ainsi :

$$\begin{array}{rcl} \texttt{state}\ R & = & \{a\}; \\ \texttt{state}\ S & = & \{b, {\sim}c\} + R; \\ \texttt{state}\ T & = & S + \{a = 1, d : \texttt{string}\}; \end{array}$$

(`state` est le mot-clé qui introduit la définition d'un type dans MGS.) La première déclaration spécifie le type d'enregistrement R défini par la présence d'un champ a. Les types peuvent être utilisés comme prédicats, ainsi

$$R(\{a = 2, x = 3\}) \qquad \text{ou} \qquad \{a = 2, x = 3\} : R \qquad \text{(notation équivalente)}$$

1. Les champs de données basés sur des groupes (*group based fields*) sont détaillés dans [GM01a].

a pour valeur *vrai* car un champ a est bien présent dans l'enregistrement.

La seconde déclaration définit le type S ; un enregistrement de ce type doit contenir les champs de R, un champ b et pas de champ c. L'opérateur + entre types émule une sorte d'héritage.

La définition de T est une spécialisation de S où a doit avoir pour valeur 1 et où un champ d du type *string* doit être présent.

3.3. Motifs, règles et transformations

Une transformation est un ensemble de règles noté :

$$\texttt{trans}\ T = \{\ \ldots\quad \mathit{regle};\quad \ldots\ \}$$

Lorsque la transformation est réduite à une seule règle on peut ne pas écrire les accolades. Une règle a la forme suivante :

$$\mathit{motif}\ \ \texttt{=>}\ \ \mathit{expression}$$

où *motif* « filtre » une sous-collection A de la collection C à laquelle la transformation est appliquée. A sera remplacée dans C par une collection B calculée par *expression.* Il existe d'autres sortes de règles que nous verrons plus loin.

3.3.1. Motifs

Nous présentons les expressions de motifs qui ont un sens générique, c'est-à-dire qu'ils peuvent filtrer n'importe quelle sorte de collection. La grammaire des expressions de motif est :

$$Pat ::= x \mid \{\ldots\} \mid p,p' \mid p+ \mid p* \mid p : P \mid p/exp \mid p\ \texttt{as}\ x \mid (p)$$

où p,p' sont des motifs, x décrit des variables de motifs, P est un prédicat et exp est une expression s'évaluant en un booléen. Les explication qui suivent donnent une sémantique informelle de ces motifs :

variable : une variable de motif x filtre exactement un élément de la collection. La variable x peut apparaître n'importe où dans le reste de la règle.

motif d'enregistrement : $\{\ldots\}$ est utilisé pour filtrer un élément qui est un enregistrement. Le contenu des accolades peut être utilisé pour filtrer un enregistrement devant (ou ne devant pas) contenir un champ (éventuellement contraint par une certain valeur de champ ou un type). Par exemple, $\{a, b : \texttt{string}, c = 3, \tilde{}d\}$ est un motif qui filtre un enregistrement contenant les champs a, b de type `string` et c de valeur 3, mais sans champ d.

voisin : p,p' est un motif qui filtre deux collections connectées p et p'. Par exemple, x,y filtre deux éléments connectés (*i.e*, x doit être voisin de y). La relation de connectivité dépend de la sorte de collection.

répétition : le motif $p+$ (resp. $p*$) filtre une sous-collection non vide d'éléments filtrés par p (resp. une sous-collection pouvant être vide).

liaison : une liaison p `as` x donne le nom x à la collection filtrée par p. Ce nom peut être utilisé partout ailleurs dans la règle. Par exemple le motif (x,x) `as` y filtre deux éléments connectés de même valeur et y fait référence à cette sous-collection dans le reste de la règle.

garde: p/exp filtre la collection filtrée par p vérifiant *exp.* Le motif $p : P$ est du sucre syntaxique pour $((p\ \texttt{as}\ x)/P(x))$ où x est une variable fraîche. Par exemple, x : *MySet* filtre un élément de type *MySet.* Un autre exemple est : $y\ /\ y > 3$ filtre un élément y pourvu que y soit supérieur 3.

Voici un exemple complet. Le motif

$$(x : int/x < 3)+\ \ \texttt{as}\ \ S\quad /\quad (\texttt{card}(S) < 5)\ \ \&\ \ (\texttt{fold}[+](S) > 10)$$

sélectionne une sous-collection S d'entiers inférieurs à 3, tel que le cardinal de S soit inférieur à 5 et que la somme des éléments de la collection soit supérieur à 10. Si ce motif est utilisé contre une séquence (resp. un ensemble, un multi-ensemble), S dénote une sous-séquence (resp. un sous-ensemble, un sous-multi-ensemble).

Certaines constructions de motifs n'existent que pour une sorte de collection. Par exemple, la construction « $\hat{}, x$ » est utilisée pour sélectionner un élément, dans une séquence, ne possédant pas de voisin gauche. Un tel motif n'a pas de sens quand la transformation est appliquée par exemple à un ensemble ; une erreur signale alors l'inconsistance du filtrage. Les opérateurs *left* et *right* sont un autre exemple de constructions particulières. Elles peuvent êtres utilisées comme gardes dans un motif (ou dans la partie droite d'une règle) pour faire référence à l'élément à la droite, ou à la gauche d'une sous-séquence filtrée. Ces constructions dépendent de la topologie de la collection et nous développons actuellement une spécification générique et systématique de ces opérateurs, spécification basée sur la notion de bord.

3.3.2. Règles

Une transformation est un ensemble de règles. Lorsqu'une transformation est appliquée à une collection, la stratégie est d'appliquer en parallèle le plus de règles possible. Une règle peut être appliquée quand son motif « filtre » une sous-collection. On peut affiner la façon dont les règles sont appliquées dans une transformation.

Règles exclusives et inclusives. Une *règle exclusive* monopolise la sous-collection filtrée *i.e.* cette dernière ne peut avoir d'intersection avec une sous-collection filtrée par une autre règle.

Les *règles inclusives* n'ont pas cette contrainte. Elles sont principalement utilisées pour modifier des parties différentes d'une valeur comme un enregistrement.

Le concept de règle inclusive peut sembler *ad-hoc*. C'est en fait un moyen très puissant de réduire significativement l'explosion combinatoire de la spécification des comportements.

Voici un exemple illustrant l'utilisation des règles inclusives : supposons que nous voulions traiter des enregistrements contenant entre autre un champ x et un champ y. Les règles

$$\{x \texttt{ as } v\} \quad \texttt{+=>} \quad \{x = v + 1\} \qquad \text{et} \qquad \{y \texttt{ as } v\} \quad \texttt{+=>} \quad \{y = 2 * v\}$$

sont déclarée inclusives (symbole +=>) et filtrent respectivement un enregistrement avec un champ x et un enregistrement avec un champ y. Elles peuvent donc toutes les deux s'appliquer à l'enregistrement $\{x = 2\ ,\ y = 3\}$ par exemple. Une règle inclusive de la forme r+=> r' où r filtre un enregistrement et où r' donne un enregistrement, remplace par $R + r'$ un enregistrement R filtré. Ainsi si l'on applique les deux règles ci-dessus à $\{x = 2,\ y = 3,\ z = 0\}$ on obtient $\{x = 3, y = 6,\ z = 0\}$. Le résultat est calculé ainsi :

$$\begin{aligned} &\Big(\{x = 2, y = 3, z = 0\} + \{x = 2 + 1\}\Big) + \{y = 2 * 3\} \\ \text{ou bien} \quad &\Big(\{x = 2, y = 3, z = 0\} + \{y = 2 * 3\}\Big) + \{x = 2 + 1\} \end{aligned}$$

et est indépendant de l'ordre d'application des règles. Ceci est possible car les règles agissent sur des parties différentes de l'enregistrement.

Priorité. Les règles exclusives sont appliquées avant les règles inclusives. Par ailleurs ont peut associer une priorité aux règles pour spécifier leur précédence dans chacune des classes de règles.

Variables locales et règles conditionnelles. MGS n'est pas un langage purement fonctionnel, des variables locales « impératives » peuvent être associées à une transformation et modifiées par ses règles. De telles variables peuvent être utilisées pour exprimer une condition à l'application d'une règle, par exemple la transformation

$$\texttt{trans}\ T[a=0] \quad = \{...; \quad R = \quad x \ =\!\{\ \texttt{on}\ a<5\,\}\!=\!> \quad (a := a+1;\ 2*x); \quad ...\}$$

définit une règle R applicable au plus cinq fois (à chaque nouvelle application de la transformation). Le point-virgule dans $(a := a+1;\ 2*x)$ dénote le séquencement des calculs. Dans cette règle a est mis à 0 lorsque T est appliquée puis est incrémentée à chaque application de R.

3.4. Gérer l'application des transformations

Une transformation est considérée comme une simple fonction et est donc une valeur comme une autre : elle peut être passée comme argument ou retournée par une fonction. Ceci permet de séquencer ou de composer les transformations simplement.

L'expression $T(c)$ dénote l'application d'une étape de transformation à la collection c. Les étapes de transformations peuvent être itérées de diverses manières :

$T[n]\,(c)$	dénote l'application de n étapes de transformation à c
$T[\texttt{fixpoint}]\,(c)$	dénote l'application de T jusqu'à atteindre un point fixe
$T[\texttt{fixrule}]\,(c)$	dénote l'application de T jusqu'à ce qu'aucune règle ne s'applique

Il existe deux autres stratégies d'application des transformations : le *mode stochastique* où l'ordre d'application des règles exclusives est choisi au hasard et où la probabilité pour qu'une règle soit choisie est proportionnelle à sa priorité, et le *mode asynchrone* où une seule règle exclusive est appliquée à chaque étape de transformation.

4. Exemples de programmes MGS

Tri d'une séquence (ou d'une liste). Un tri de la forme *tri à bulle* est simple à implémenter :

$$\texttt{trans}\ Sort \quad = \quad (x,y\ /\ y<x) \ \texttt{=>}\ y,x;$$

(Ce n'est pas véritablement un tri à bulle : dans le vrai tri à bulle l'élément sélectionné « migre » en une seule fois alors que notre algorithme implémente des échanges de valeurs en tout lieu de la séquence.)

Crible d'Eratosthene sur un ensemble. Le but est de calculer les nombres premiers inférieurs à N. Pour cela on génère l'ensemble des entiers entre 2 et N puis on remplace chaque couple (x,y) où x divise y par le singleton x.

$$\begin{aligned}
\texttt{trans}\ Generate &= \{val=x\}\ \texttt{=>}\ x,\{val=x+1\};\\
\texttt{trans}\ Succeed &= \{val=x\}\ \texttt{=>}\ x;\\
\texttt{trans}\ Eliminate &= (x,y\ /\ y\ \texttt{mod}\ x=0)\ \texttt{=>}\ x;
\end{aligned}$$

Lors de la génération par la règle *Generate* de l'ensemble d'entiers sur lequel le crible a lieu, l'entier le plus grand est encapsulé dans un enregistrement afin d'être distingué des autres valeurs. Une fois l'ensemble créé la valeur $\{val=x\}$ redevient l'entier x via *Succeed*.

Après avoir déclaré ces transformation, l'expression suivante donne le résultat attendu :

$$\mathit{Eliminate}[\texttt{fixrule}]\Big(\mathit{Succeed}\big(\mathit{Generate}[N](\{2,\mathit{true}\},\texttt{set} : ())\big)\Big)$$

Nous avons choisi cet algorithme de calcul des nombres premiers pour montrer que certains problèmes se résolvent en quelques lignes, mais nous aurions traité la question différemment si l'objectif avait été les performances du programme.

Enveloppe convexe. Étant donné un ensemble P de points du plan nous cherchons à calculer le plus petit polygone convexe contenant tous ces points. On peut montrer que chaque sommet de ce polygone est dans P. Le programme MGS qui calcule ce polygone consiste à itérer la transformation suivante : sélectionner un point X dans P et trois points U, V et W dans P puis éliminer X s'il est dans le triangle (U,V,W) (et distinct de ses trois sommets).

On définit l'enregistrement *Point* comme ayant un champ x et un champ y :

$$\texttt{state}\ \ \mathit{Point} = \{x,y\}$$

On définit ensuite une fonction *inside* prenant quatre points en arguments et indiquant si le premier est dans le triangle formé par les trois autres. Cette fonction est détaillée dans [GM01b] et utilise essentiellement des calculs simples sur les flottants.

Ceci étant fait, la transformation suivante permet de résoudre le problème posé :

$$\texttt{trans}\ \ \mathit{Convex} = \ \ X,U,V,W/\ \mathit{inside}(X,U,V,W)\ \texttt{=>}\ U,V,W;;$$

Par exemple, nous cherchons l'enveloppe convexe de points se situants dans un carré délimité par (0,0) et (1,1) (les quatres coins compris) :

$\mathit{Convex}[\texttt{fixrule}]($ $(\{x = 0, y = 0\}$,$\{x = 0.2, y = 0.1\}$,$\{x = 0.5, y = 0.7\}$,$\{x = 1, y = 0\}$,
$\{x = 0.1, y = 0.2\}$,$\{x = 1, y = 1\}$,$\{x = 0.2, y = 0.4\}$,$\{x = 0.4, y = 0.6\}$,
$\{x = 0, y = 1\}$,$\texttt{set} : ())$) ; ;
donne le résultat attendu :

$$\{x = 0, y = 0\},\ \{x = 0, y = 1\},\ \{x = 1, y = 0\},\ \{x = 1, y = 1\},\ () : \texttt{set}$$

Processus de diffusion/réaction de Turing.

Le mathématicien Alan Turing souhaitait savoir si la forme des taches sur le pelage d'animaux pouvaient avoir une origine autre que génétique. Son modèle montre que des taches *réalistes* peuvent être produites comme le résultat d'un processus ne faisant pas intervenir de matériel génétique. Il a proposé un modèle de réaction chimique pour expliquer la formation de motifs, comme le résultat de concentrations de certaines substances chimiques, dans un milieu initialement homogène. Turing appelle ces agents chimiques des *morphogènes* et le modèle de réaction chimique, le modèle de *diffusion–réaction.* La forme générale des équations de Turing est :

$$\frac{\partial x}{\partial t} = F(x,y) + D_x \nabla^2 x \qquad\qquad \frac{\partial y}{\partial t} = G(x,y) + D_y \nabla^2 y$$

où les équations indiquent que le changement de concentration pour un morphogène x à un instant t dépend d'une fonction F des concentrations locales des morphogènes x et y et d'un processus de diffusion de x. La constante D_x exprime la vitesse de diffusion du morphogène x et $\nabla^2 x$ est une mesure de la variation de la concentration de x par rapport à ses voisins. Depuis l'introduction de ces idées, de nombreux systèmes ont été étudiés afin de mettre en évidence l'action de morphogènes dans un processus de diffusion–réaction lors de l'apparition de motifs.

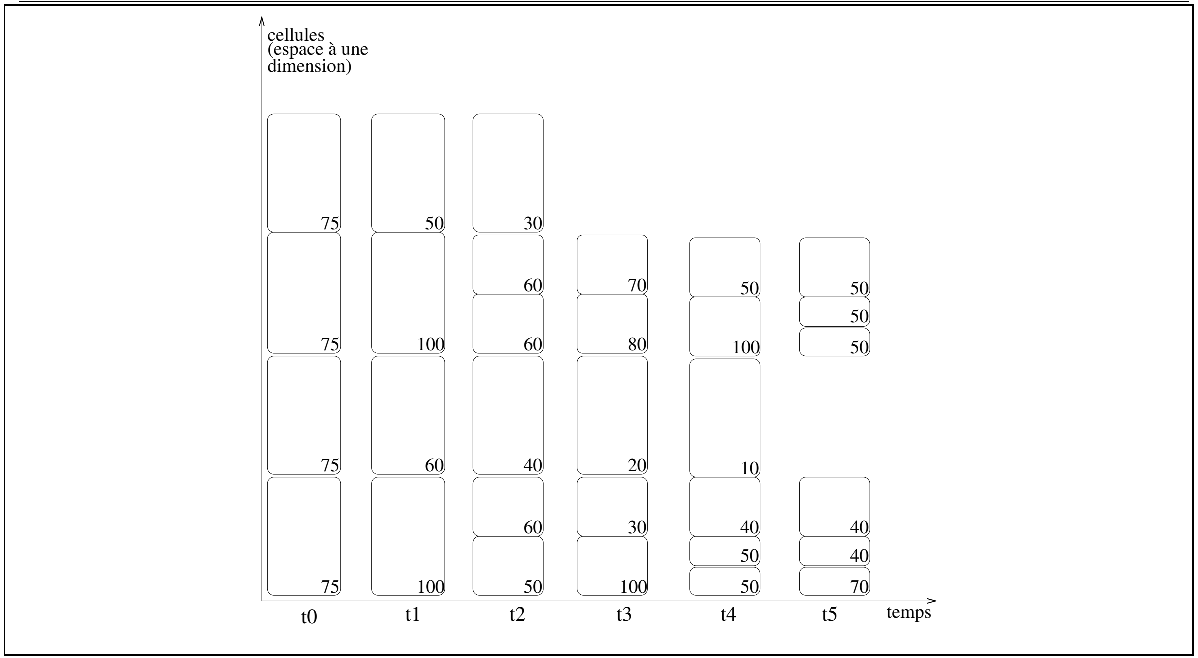


FIG. 3 – Division cellulaire et quantité de morphogène x au cours du temps. L'état initial est une barre de quatre cellules avec une quantité de x identique. Dans cet exemple la quantité totale de x est constante mais le morphogène va s'accumuler en deux points. Lorque x atteint 100 dans une cellule celle-ci se divise et lorsque x atteint 0 elle meurt. Les valeurs sont choisies à titre explicatif et ne sont pas issues d'une simulation.

Dans notre exemple, la quantité de morphogène x sert à déclencher la *division cellulaire*. Chaque cellule correspond dans le graphique ci-dessous à un bloc unique dont la hauteur est proportionelle à la quantité du morphogène x. Sur la représentation graphique, quand une cellule se divise en deux la largeur du bloc est divisée par deux ; ainsi, toutes les cellules d'une même lignée sont dans la même bande (axe dirigé vers le lecteur, qui représente aussi l'écoulement du temps). L'image correspond à la simulation de l'évolution d'une population initiale de 18 cellules sur 180 pas de temps élémentaires.

La figure 3 illustre sommairement le principe de la réaction modélisée. La figure 4 montre le résultat d'une simulation par le programme `MGS` détaillé ci-dessous.

Le morceau de programme `MGS` suivant sert à initialiser la structure de données :

```
trans init =
c => {
x = 4.0,
y = 3.5 + (random(1.0) − 0.5),
beta = 12.0 + (random(0.05 ∗ 2.0) − 0.05),
} ;;
rsp := 1.0/16.0 ;;
diff1 := 0.25 ;;
diff2 := 0.0625 ;;
NbCell := 18 ;;
segment0 := init[1](gen(NbCell,() : seq)) ;;
```

La transformation *init* est appliquée à une séquence de 18 éléments générée par *gen*, ce qui donne une séquence de 18 enregistrements dotés d'un champs x, y et *beta*. Afin de faciliter l'obtention du résultat visuel de la figure 4 nous avons ajouté un champs *taille* dans notre programme mais n'étant

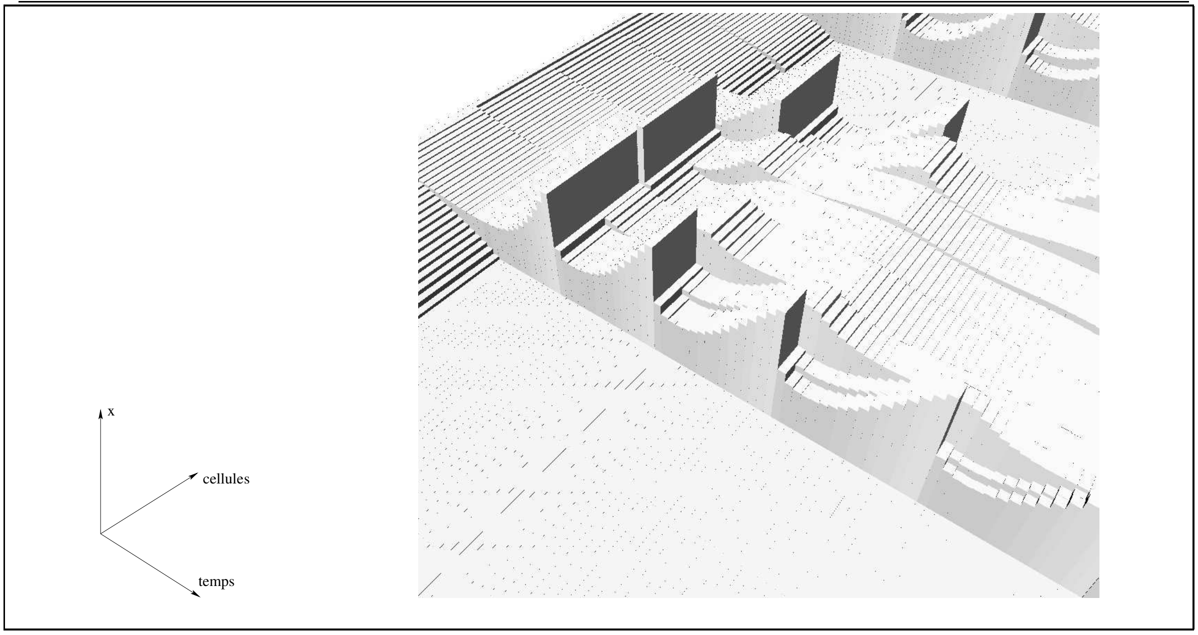


FIG. 4 – Extrait de la visualisation des résultats. Cette figure est une vue de la représentation graphique des résultats sous la forme d'objet 3D.

pas nécessaire à la simulation proprement dite nous en ferons abstraction.

Les deux fonctions suivantes calculent les variations de x et y et seront utilisées dans la transformation principale :

`fun` $dx(x,y,beta,lx,rx) = (1.0/16.0) * (y * x - x - beta) + diff2 * (lx + rx - 2.0 * x)$; ;
`fun` $dy(x,y,ly,ry) = (1.0/16.0) * (16.0 - y * x) + diff1 * (ly + ry - 2.0 * y)$; ;

La transformation principale est composée d'une règle calculant les divisions cellulaires, d'une règle calculant la diffusion du morphogène dans la barre et de deux autres règles calculant la diffusion aux bornes de la barre.

`trans` $Turing = \{$
$Split =$ $(c/c.x > 8)$ =>
$\{\ x = c.x/2\ ,\ y = c.y/2\ ,\ beta = c.beta\ \}\ ,\ \{\ x = c.x/2\ ,\ y = c.y/2\ ,\ beta = c.beta\ \}$;

$evol =$ $(c/($`left` $c)\&($`right` $c))$ +=>
$\{\ x = Max(0.0, c.x + dx(c.x, c.y, c.beta,($`left` $c).x,($`right` $c).x))$,
$y = c.y + dy(c.x, c.y,($`left` $c).y,($`right` $c).y)\ \}$;

$evol\_right =$ $(c/\tilde{}$`left` $c)$ +=>
$\{\ x = Max(0.0, c.x + dx(c.x, c.y, c.beta, 0,($`right` $c).x))\ ,\ y = c.y + dy(c.x, c.y, 0,($`right` $c).y)\ \}$;

$evol\_left =$ $(c/\tilde{}$`right` $c)$ +=>
$\{\ x = Max(0.0, c.x + dx(c.x, c.y, c.beta,($`left` $c).x, 0))\ ,\ y = c.y + dy(c.x, c.y,($`left` $c).y, 0)\ \}$;
$\}$; ;

La fonction *evol* applique la transformation le nombre d'étapes voulues :

```
fun evol(barre,t,tmax) =
if (t < tmax)   then evol(Turing[iter = 1](barre),t + 1,tmax)
else barre
fi ;
```

Développement d'un organisme uni-dimensionnel à la façon des L systèmes.
Nous reprenons ici un exemple paradigmatique de formalisation d'un processus de croissance en L système. Les états du développement d'un organisme filamentaire «uni-dimensionnel» sont donnés à travers la définition d'un L système [Lin68] que nous implémenterons ensuite en MGS. Les règles décrivant l'évolution de l'organisme permettent à ses cellules de rester dans le même état, de changer d'état, de se diviser en plusieurs cellules ou bien de disparaître.

Pour chaque cellule, deux états a et b sont possibles. L'état a correspond à une division cellulaire imminente, alors que l'état b correspond à un état d'attente durant une étape de division. Les règles de production et les quatre premières dérivations sont :

$$
\begin{array}{llcll}
\omega & : b_r & \qquad & t_0 & : b_r \\
p_1 & : a_r \rightarrow a_l\, b_r & & t_1 & : a_r \\
p_2 & : a_l \rightarrow b_l\, a_r & & t_2 & : a_l\, b_r \\
p_3 & : b_r \rightarrow a_r & & t_3 & : b_l\, a_r\, a_r \\
p_4 & : b_l \rightarrow a_l & & t_4 & : a_l\, a_l\, b_r\, a_l\, b_r
\end{array}
$$

Les suffixes l et r indiquent la polarité des cellules. Un arbre de dérivation du processus de croissance est illustré par la figure 5 (empruntée à [LJ92]). Les règles de changement de polarité de cet exemple sont très proches de celles de la bactérie bleue-verte *Anabaena catenula* [MW72, KL87].

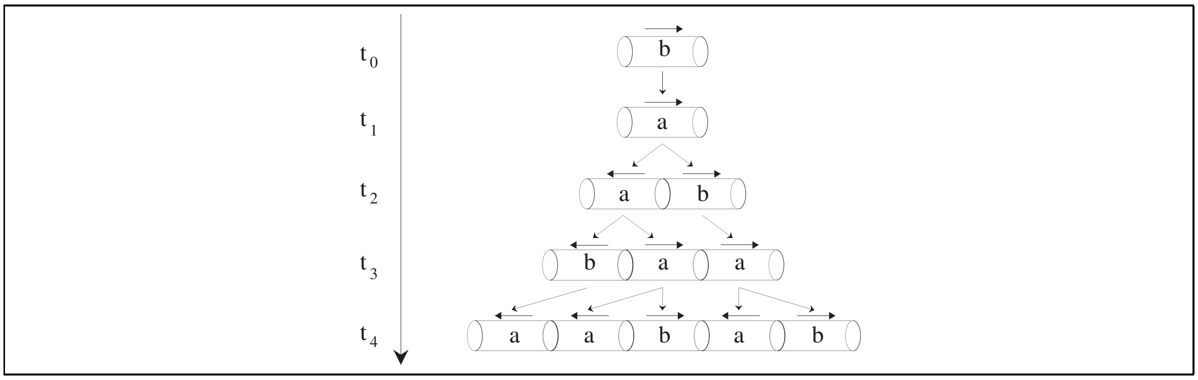


FIG. 5 – Les quatre premières dérivations du processus de croissance de la bactérie *Anabaena catenula* modélisé par un L système (la polarité des cellules est indiquée avec une flèche).

Une implémentation des règles de production en MGS est immédiate si l'on choisit la séquence comme topologie :

```
trans Ana = {
"ar" => "al" , "br" ;
"al" => "bl" , "ar" ;
"br" => "ar" ;
"bl" => "al" ;
};;
```

La ligne MGS suivante calcule les quatre premières étapes du développement :

```
Ana[4]( "br" , seq : () );;
```

L'application en parallèle des règles de la transformation *Ana* fait que l'on obtient bien le résultat spécifié par le L système.

5. Conclusion et travaux futurs

Le rapport technique [GM01b] décrit plus en détail la formalisation topologique des collections, les transformations et décrit de nombreux exemples de programmes MGS (la tokenisation d'une séquence de lettres, le calcul du segment de somme maximale, le déplacement d'un jeton sur un anneau, etc.).

Dans l'implémentation actuelle de MGS, seuls les ensembles, les multi-ensembles et les séquences sont pris en compte par l'interprète [2]. Les éléments peuvent êtres de n'importe quel type, permettant toutes les imbrications possibles. L'implémentation des tableaux et des champs de données basés sur des groupes (GBF, qui généralisent les tableaux fonctionnels, *cf.* [GMS96, GM01a]) est en cours . Nous commençons aussi l'étude d'une implémentation générique de chaînes topologiques complexes, une formalisation adaptée à nos collections topologiques, en utilisant des G-cartes [Lie91] pour représenter n'importe quelle structure de voisinage. Par ailleurs le problème du filtrage efficace est un effort de recherche à long terme. Pour certaines classes de filtres et pour certaines topologies il existe des techniques efficaces et bien maîtrisées comme dans l'atelier ELAN [BKKMR] par exemple pour le filtrage AC ; ce n'est plus le cas pour des topologies arbitraires. Un de nos objectifs est donc de déterminer des classes utiles et expressives de filtres admettant une compilation du filtrage efficace.

En ce qui concerne le langage, l'étude des concepts concernant les collections topologiques doivent continuer avec une étude plus précise des sortes de transformations. De nombreuses restrictions peuvent être mises sur les transformations donnant lieu à de nombreux types de langages de motifs et de règles. La complexité du filtrage de tels motifs doit être étudiée plus avant. Nous voulons aussi développer un système de types prenant en compte les collections imbriquées, suivant la voie définie par [Ble93]. Enfin, nous aimerions déterminer si la construction d'espaces topologiques par des transformations peut être caractérisée par un système de types non standards. L'efficacité de la compilation de MGS est un effort de recherche à long terme.

Les applications possibles, ouvertes par ces travaux préliminaires, sont nombreuses. Nous nous intéressons particulièrement à la simulation de changements topologiques qui ont lieu dans les premières étapes du développement de l'embryon. C'est un exemple de formation et de fusion de tissus qui nécessite une topologie complexe, bien au delà de ce qu'il est possible de faire en utilisant de simples structures de données classiques. Une autre application motivante est le cas des réseaux d'interactions bio-chimiques, pour lesquels l'extension de systèmes de réécriture ont déjà été proposés, voir par exemple [FMP00, Man01].

2. Actuellement, deux versions de l'interprète MGS existent : un écrit en OCAML et un écrit en C++. Il existe de légères différences entre les deux versions. Par exemple, la version OCAML est plus complète pour ce qui concerne la partie fonctionnelle du langage. Ces interprètes sont librements accessibles à partir de la page web http://www.lami.univ-evry.fr/mgs.

Références

[BB89] G. Berry and G. Boudol. The chemical abstract machine. Technical Report RR-1133, Inria, Institut National de Recherche en Informatique et en Automatique, 1989.

[BH00] R. Brown and A. Heyworth. Using rewriting systems to compute left kan extensions and induced actions of categories. *Journal of Symbolic Computation*, 29(1):5–31, January 2000.

[Ble93] G. Blelloch. NESL: A nested data-parallel language (version 2.6). Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, April 1993.

[BM86] J. P. Banâtre and D. Le Métayer. A new computational model and its discipline of programming. Technical Report RR-0566, Inria, 1986.

[BM91] J. P. Banâtre and D. Le Métayer. Introduction to Gamma. In J. P. Banâtre and D. Le Métayer, editors, *Research Directions in High-Level Parallel Programming Languages*, pages 197–202. Springer Lecture Notes in Computer Science 574, June 1991.

[BNTW95] P. Buneman, S. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, 18 September 1995.

[FMP00] M. Fisher, G. Malcolm, and R. Paton. Spatio-logical processes in intracellular signalling. *BioSystems*, 55:83–92, 2000.

[Gia00] J.-L. Giavitto. A framework for the recursive definition of data structures. In *Proceedings of the 2nd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP-00)*, pages 45–55. ACM Press, September 20–23 2000.

[GM01a] J.-L. Giavitto and O. Michel. Declarative definition of group indexed data structures and approximation of their domains. In *Proceedings of the 3nd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP-01)*. ACM Press, September 2001.

[GM01b] J.-L. Giavitto and O. Michel. MGS: a programming language for the transformations of topological collections. Technical Report 61-2001, LaMI – Université d'Évry Val d'Essonne, May 2001.

[GMS96] J.-L. Giavitto, O. Michel, and J. Sansonnet. Group-based fields. In *Parallel Symbolic Languages and Systems (International Workshop PSLS'95)*, volume 1068, pages 209–215, 1996.

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

[Man01] V. Manca. Logical string rewriting. *Theoretical Computer Science*, 264:25–51, 2001.

[Pau98] G. Paun. Computing with membranes. Technical Report TUCS-TR-208, TUCS - Turku Centre for Computer Science, November 11 1998.

[Rém92] D. Rémy. Syntactic theories and the algebra of record terms. Technical Report 1869, INRIA-Rocquencourt, BP 105, F-78 153 Le Chesnay Cedex, 1992.

[RS92] G. Rozenberg and A. Salomaa. *Lindenmayer Systems*. Springer, Berlin, 1992.

[LJ92] A. Lindenmayer and H. Jürgensen. Grammars of development: discrete-state models for growth, differentiation, and gene expression in modular organisms. In G. Ronzenberg and A. Salomaa, editors, *Lindenmayer Systems, Impacts on Theoretical Computer Science, Computer Graphics and Developmental Biology*, pages 3–21. Springer Verlag, February 1992.

[MW72] G. J. Mitchison and M. Wilcox. Rules governing cell division in anabaena. *Nature*, 239:110–11, 1972.

[KL87] C. G. Koster and A. Lindenmayer. Discrete and continuous models for heterocyst diffrentiation in growing filaments of blue-green bacteria. *Acta Biotheoretica*, 36:249–273, 1987.

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

[BKKMR] P. Borovanský and C. Kirchner and H. Kirchner and P.-E. Moreau and C. Ringeissen, An Overview of ELAN, In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of the International Workshop on Rewriting Logic and its Applications (Electronic Notes in Theoretical Computer Science* volume 15) , Elsevier Science, September 1998