N° d'identification : 2006EVRY0028

UNIVERSITÉ D'ÉVRY-VAL-D'ESSONNE
U.F.R. SCIENCES FONDAMENTALES ET APPLIQUÉES

THÈSE

présentée pour obtenir

le GRADE de DOCTEUR EN SCIENCES
DE L'UNIVERSITÉ d'ÉVRY

Spécialité : INFORMATIQUE

par

Antoine Spicher

Sujet : **Transformation de collections topologiques de dimension arbitraire. Application à la modélisation de systèmes dynamiques**

Soutenue le 1er Décembre 2006 devant le jury composé de :

MM.	Gaétan Hains	*Président*
	Jacques Malenfant	*Rapporteur*
	Przemyslaw Prusinkiewicz	*Rapporteur*
	Roberto Di Cosmo	*Examinateur*
	Pascal Lienhardt	*Examinateur*
	Olivier Michel	*Encadrant*
	Jean-Louis Giavitto	*Directeur de la Thèse*

Table des matières

Table des figures

Table des grammaires

Table des règles

Chapitre 1

Motivations et contributions

1 Introduction

Les travaux de recherche effectués dans cette thèse s'inscrivent dans le cadre du projet MGS. Ce projet poursuit deux objectifs complémentaires :

1. étudier et développer l'apport de *notions* et d'outils de nature *topologique* dans les langages de programmation ;
2. appliquer ces notions et ces outils à la conception et au développement de nouvelles structures de données et de contrôle à la fois expressives et efficaces pour la modélisation et la simulation de *systèmes dynamiques à structure dynamique* ($(\mathrm{SD})^2$).

Ces études se concrétisent par le développement d'un langage de programmation expérimental et par son application à la modélisation et à la simulation de systèmes dynamiques, en particulier dans le domaine de la biologie et de la morphogenèse. Ce langage est également nommé MGS.

Dans ce cadre, notre travail a consisté à étudier et développer les notions de *collection topologique de dimension arbitraire* et de *transformation*.

Une *collection topologique* est un ensemble de valeurs muni d'une relation de voisinage ; une *transformation* est une fonction définie par cas sous la forme de règles de réécriture s'appuyant sur la notion de voisinage. Les notions de collection topologique et de transformation permettent la spécification de modèles exécutables de systèmes dynamiques de manière extrêmement concise et dans un cadre déclaratif.

L'objectif de ces travaux de recherche est de répondre à plusieurs questions qui se sont posées au cours du développement du projet MGS. Parmi les plus importantes :

- Est-il possible de manipuler de manière purement déclarative des représentations spatiales complexes ?
- La prise en compte de relations topologiques dans un langage de programmation permet-elle de simplifier la spécification et la simulation des systèmes dynamiques (SD) et plus particulièrement des SD à structure dynamique ?
- Est-il possible de donner un « contenu calculatoire » à une forme différentielle, afin de se rapprocher des formalismes mathématiques utilisés dans les applications ?
- Est-il possible d'étendre les mécanismes d'évaluation du langage afin de permettre, avec les mêmes structures de contrôle, le développement de simulations déterministes et de simulations stochastiques ?
- Est-il possible de formaliser de manière synthétique les importants développements logiciels qui ont été réalisés dans le projet ?

En tentant de répondre à ces questions, nos travaux se sont organisés suivant trois directions :

1. Le premier axe de recherche concerne le développement de la notion de collection topologique. Ce développement a deux objectifs : généraliser cette notion afin de prendre en compte les besoins en modélisation, en particulier la représentation d'objets spatiaux complexes de dimension arbitraire, et fonder théoriquement cette notion à partir de la notion de complexe cellulaire abstrait développée en topologie algébrique combinatoire.
2. La seconde direction de recherche concerne la notion de transformation et la sémantique formelle des programmes MGS. Ce travail a permis de développer la manipulation déclarative de collections topologiques de dimension arbitraire, d'étendre le langage de motifs de chemins des règles d'une transformation, de concevoir un nouveau langage de motifs plus puissant afin de filtrer des sous-collections de forme arbitraire et de développer des stratégies stochastiques d'applications des règles d'une transformation.
3. Enfin, nous avons validé par de nombreux exemples significatifs les nouvelles constructions que nous avons conçues et implantées dans l'interprète MGS. Ces applications sont pour la plupart des exemples non triviaux de systèmes dynamiques à structure dynamique. La variété des domaines abordés, le volume et l'importance des exemples traités permettent de juger de la pertinence des constructions proposées, mettent en évidence le gain en expressivité apporté par l'approche topologique proposée, et valident les hypothèses du projet MGS.

Plan du chapitre. Dans la suite de ce chapitre nous présentons succinctement plusieurs langages ayant inspiré originairement MGS, les notions de collection topologique et leur transformation, puis nous détaillons les trois parties de notre travail. Pour clore ce chapitre nous listons nos contributions.

2 Collection topologique, transformation et langages de programmation bio-inspirés

Une des motivations originelles de MGS est la volonté d'unifier plusieurs modèles de calcul[1] initialement inspirés par des processus biologiques. Du point de vue topologique adopté dans MGS, ces langages se fondent tous sur un mécanisme de substitution dans une structure de données correspondant à un espace particulier. Ces différents modèles de calcul sont décrits dans le paragraphe suivant, avant d'introduire succinctement les notions de collection topologique et de transformation.

2.1 Des langages de programmation bio-inspirés

Gamma. Gamma propose un formalisme où toute notion de séquentialité est absente. Utilisant la réécriture de multi-ensemble comme unique structure de contrôle, un programme peut se décrire par une métaphore chimique où les réactions chimiques correspondent aux règles de réécriture et où la solution chimique est formalisée par la notion de multi-ensemble. La métaphore chimique autorise une description simple et directe des processus parallèles ainsi que du non-déterminisme. Des extensions ont ensuite été développées pour apporter aux données une structuration absente du formalisme initial ; une forme de relation de voisinage entre les données est alors introduite. La structure de multi-ensemble reste néanmoins prépondérante (il s'agit de multi-ensembles structurés), le voisinage étant encodé à l'aide d'un mécanisme d'adressage et de définition de type [BL86, BFL01].

P systèmes. Les P systèmes se fondent sur une métaphore du fonctionnement de la cellule biologique pour organiser un ensemble de calculs chimiques. Les P systèmes correspondent à des multi-ensembles imbriqués, ou *compartiments*, contenant différents objets élémentaires. Chaque compartiment est caractérisé par un jeu de règles de réécriture localisées permettant aux objets de réagir entre eux. Divers opérateurs additionnels permettent le transport d'objets entre compartiments ou la création et la destruction de compartiments. Les systèmes ainsi construits sont parallèles et non-déterministes. Les P systèmes offrent des mécanismes puissants permettant de définir de nouvelles classes de langages formels. L'étude des P systèmes s'est initialement concentrée sur des problèmes de calculabilité. Leur utilisation s'étend à présent à la modélisation et à la simulation [Pău00, Pău01].

L systèmes. Les systèmes de Lindenmayer ont été initialement proposés en 1968 par le biologiste Aristid Lindenmayer afin de donner une description formelle du développement d'organismes filamentaires. Le formalisme mis en place est puissant : le système est représenté par une chaîne de symboles, chaque symbole représentant une partie de l'organisme et la contiguïté dans la chaîne représentant le voisinage entre les parties. L'évolution du système est alors capturé par des règles de réécriture parallèle sélectionnant une sous-chaîne pour la remplacer par une nouvelle chaîne. Ce cadre a été développé suivant deux directions différentes. Tout d'abord, les L systèmes fournissent un moyen pour engendrer des mots et ils ont été utilisés dans la définition de langages conduisant à une hiérarchie propre aux L systèmes (à la façon de la hiérarchie de Chomsky). Leur effectivité a également rencontré un succès considérable dans modélisation et la simulation de processus de croissance et de développement, en particulier en botanique [RS92, PLH+90].

[1] Le point de vue qui nous intéresse ici est celui des langages de programmation : à chacun de ces modèles de calcul correspond une classe de langages de programmation et nous ne nous intéressons pas à l'étude de la complexité des algorithmes exprimés dans ces modèles de calcul mais à l'expressivité permise par les constructions propres à ce modèle dans un langage associé.

Les automates cellulaires. Les automates cellulaires ont été introduits par John von Neumann afin de comparer la notion d'organisme vivant et de machine, en particulier la notion d'auto-reproduction. Un automate cellulaire peut se décrire par un réseau prédéfini de sites (par exemple un voisinage NEWS), chaque site possédant un état pris dans un ensemble fini. À un instant $t+1$, l'état de chaque site est mis à jour suivant une règle d'évolution prédéfinie qui combine son état et ceux des sites voisins à l'instant t. Le fonctionnement de l'automate correspond à la mise à jour, dans une suite d'étapes discrètes, de l'état des sites [vN66].

2.2 Un point de vue unificateur

Une constatation. Les quatre exemples de modèles de calcul que nous venons d'évoquer brièvement ont été inspirés par des processus chimiques ou biologiques. Ils ont donné lieu tous les quatre à des développements formels très riches mais ont aussi été utilisés pour fonder des *langages de programmation non conventionnels* qui ont été abondamment utilisés dans la modélisation et la simulation de processus naturels et artificiels.

Malgré leurs différences (ils ont été créés à différents moments sur une période de cinquante ans et avec des motivations très diverses), on peut noter que ces quatre paradigmes se présentent tous comme des langages de règles et qu'on peut les décrire comme des formes particulières de *réécriture*. Par réécriture nous entendons ici le mécanisme qui consiste à substituer une partie par une autre dans un objet.

Les multi-ensembles et les chaînes sont des structures de données dites *monoïdales* [Man01, GM01b] et les techniques de réécriture associées sont bien maîtrisées : il s'agit de considérer des constructeurs *modulo* associativité (pour les chaîne) et *modulo* associativité-commutativité (pour les multi-ensembles).

En revanche, il n'est pas usuel de considérer les automates cellulaires comme de la réécriture de tableau, principalement parce qu'il n'en existe pas de définition inductive libre (un tableau n'est pas un terme). De plus, la réécriture ne concerne qu'un seul site (même si la partie droite d'une règle d'évolution dépend des voisins de ce site). Néanmoins, certaines variations de ce modèle, comme les gaz sur réseaux [TN87], spécifient bien la réécriture d'un ensemble de sites voisins.

L'approche topologique. Il est donc tentant d'englober ces modèles de calculs dans un cadre permettant de les définir comme des cas particuliers d'un mécanisme général de substitution des parties de la structure de données. Pour cela, il faut disposer d'une notion générale d'objet et de partie, capable de généraliser les multi-ensembles, les mots, les tableaux...

Le projet MGS propose de s'appuyer sur des notions issues de la topologie algébrique pour définir ce cadre. L'idée est de voir une solution chimique, une imbrication de membranes, une chaîne ou encore un réseau comme un espace où prend place le calcul. Nous appellerons cet espace une *collection topologique*. Les mécanismes de substitution et les règles qui apparaissent dans Gamma, les P systèmes, les L systèmes ou les automates cellulaires seront vus comme des *transformations* de cet espace.

Il est encore trop tôt pour affirmer que le cadre théorique que nous développons dans cette thèse pour formaliser les notions de collection topologique et de transformation est adapté à l'étude formelle de ces différents mécanismes de calcul. Cependant, les exemples développés dans cette thèse montrent que ces notions offrent, au moins au niveau syntaxique et pour ce qui est de la programmation, un cadre uniforme et expressif qui permet d'intégrer simplement ces différents paradigmes dans un même langage.

2.3 La notion de collection topologique

Nous appelons *collection* tout ensemble d'éléments « organisés ». Les structures de données manipulées par les langages de programmation correspondent à différentes sortes de collections : tableaux, arbres, ensembles, multi-ensembles (ou *bags*), séquences (ou listes), termes, etc.

Nous appelons *collection topologique* une collection dont les éléments sont organisés par une *relation de voisinage*. Nous noterons « , » la relation de voisinage : x , y signifie que l'élément y est un voisin de l'élément x (dans une certaine collection). La relation de voisinage permet de spécifier une notion de sous-collection : une sous-collection est un sous-ensemble *connexe* de la collection. Cette relation permet aussi de spécifier la notion de *chemin* : un chemin est une suite d'éléments tous distincts $[x_0, x_1, \ldots, x_n]$ tels que x_i , x_{i+1}.

De nombreuses structures de données classiques peuvent être considérées comme des collections topologiques. Par exemple, une *séquence* est une collection topologique telle que :

- chaque élément possède au plus un voisin ;
- chaque élément ne peut être le voisin que d'un élément au plus ;
- il n'existe pas de cycle dans la relation de voisinage.

Un ensemble, un tableau, un graphe, un arbre, ..., peuvent aussi être vus comme une collection topologique avec une relation de voisinage adéquate justifiée par les opérations habituelles sur ces structures de données.

Ce point de vue topologique sur une structure de données peut s'intégrer dans un langage fonctionnel classique (comme ML) où il ouvre d'intéressantes perspectives qui justifient à elles seules son étude :

- cette notion offre un *cadre uniforme* pour spécifier et manipuler des structures de données qui étendent la notion de type algébrique ;
- elle permet d'étendre la *définition par cas*, un mécanisme puissant de spécification de fonction, à tous les types de données (y compris les structures de données qui ne sont pas des types algébriques comme par exemple les tableaux) ;
- elle offre un cadre alternatif à la notion de *polytypisme* qui n'est pas restreint aux types de données algébriques ;
- elle apporte une *nouvelle notion de réécriture* qui ne se réduit pas aux approches existantes.

Cependant, les notions de collection topologique et de transformation ont été initialement motivées en MGS par la volonté d'unifier plusieurs modèles de programmation et par les problèmes particuliers posés par la simulation de systèmes dynamiques à structure dynamique en biologie. Cette dernière motivation, présentée dans le chapitre 7, est à l'origine du point de vue adopté dans cette thèse.

Dans les travaux antérieurs, en particulier [Coh04], les collections topologiques étaient représentées par un graphe de voisinage : les sommets du graphe correspondent aux éléments de la collection et les arcs à la relation de voisinage. Cependant, le cadre théorique adéquat pour la formalisation de la relation de voisinage est la notion de *complexe cellulaire* développée en topologie algébrique (les graphes sont un cas particulier de complexes cellulaires de dimension 1). Ce cadre théorique abstrait est développé au chapitre 3, implanté (voir l'annexe A) et validé à travers de nombreux exemples dans le domaine de la simulation (voir la partie III).

2.4 La notion de transformation

Une transformation est un mécanisme de calcul qui permet d'associer une collection topologique à une autre. Elle est constituée d'un ensemble de *transformations élémentaires*. Étant

donnée une collection C, une transformation élémentaire spécifie :

1. une sous-collection A de C
2. le calcul d'une nouvelle collection B à partir de A
3. la substitution de B en lieu et place de A dans C

La spécification d'une transformation élémentaire demande de préciser comment sélectionner A, comment calculer B et les contraintes de la substitution de A par B. Les premier et troisième points dépendent de la relation de voisinage définie sur C.

2.5 Plongement dans un langage de programmation fonctionnel

Il est possible d'introduire les notions de collection topologique et de transformation dans un langage fonctionnel comme ML :

- une collection topologique (caractérisée par sa relation de voisinage) correspond à un type de données ;
- une transformation correspond à une fonction définie par un ensemble de *règles*.

Une règle spécifie une transformation élémentaire et prend la forme suivante :

$$a \Rightarrow h(a)$$

où a spécifie une sous-collection A à sélectionner dans la collection C et où le calcul de B correspond à la fonction h. L'expression a peut s'interpréter comme un motif de filtrage.

Insistons sur le fait qu'une transformation est une fonction ordinaire appliquée et utilisée exactement comme les fonctions définies par abstraction. Dans un langage fonctionnel permettant l'ordre supérieur, comme MGS, une transformation peut être placée en argument ou bien retournée comme résultat d'une application.

Cette approche est mise en œuvre dans le langage expérimental MGS. Ce langage dispose d'un interprète qui correspond à une version dynamiquement typée du langage (le typage et la compilation ont été étudiés dans [Coh04]). Les travaux de cette thèse visent à étendre les notions de collection topologique et de transformation et à les fonder à partir de constructions théoriques développées dans le domaine de la topologie algébrique.

3 Organisation de cette thèse

Ce document se structure en quatre parties.

3.1 Première partie : Une introduction à MGS

La première partie décrit le langage MGS. Les notions de collection topologique, de transformation ainsi que leur intégration dans un langage fonctionnel sont présentées de manière informelle à travers des exemples dans le chapitre 2. L'objectif de ce chapitre est double : il doit aider à la compréhension des concepts décrits et des problématiques abordées, et il doit convaincre le lecteur que les notions de collection topologique et de transformation sont intéressantes et utiles pour la programmation en général.

Différentes sortes de collections topologiques sont présentées dans ce chapitre, en particulier les collections topologiques fondées sur la notion de G-carte et celles fondées sur la notion de complexe cellulaire abstrait. Un des enjeux de cette thèse est de montrer que ces structures de données peuvent être manipulées de manière purement fonctionnelle. C'est un enjeu important mais peu de travaux dans ce sens ont produit des résultats probants.

3.2 Deuxième partie : Formalisation

La seconde partie de ce document introduit et étudie les notions nécessaires à la formalisation des structures de données et de contrôle présentées précédemment. L'enjeu est de donner une description rigoureuse des constructions que nous avons conçues et réalisées dans l'interprète. Le travail d'implantation est très important : l'interprète représente environ 50 000 lignes de codes OCaml, sans compter l'interfaçage vers de nombreuses bibliothèques. Donner une sémantique permet alors de présenter cet effort sous une forme synthétique. Cela permet également de préciser clairement plusieurs points difficiles à clarifier en langage ordinaire (les stratégies d'application des règles en sont un bon exemple).

Trois problèmes se posent pour la formalisation du langage MGS : la formalisation des structures de données, la formalisation du filtrage et la formalisation des stratégies d'applications (en particulier les stratégies non-déterministes). Ces trois problèmes sont abordés dans les trois premiers chapitres de cette partie. Le dernier chapitre poursuit la formalisation des transformations, à partir d'un point de vue algébrique : une transformation est le pendant informatique de la notion de forme différentielle (discrète) en physique.

Chapitre 3 : Collections topologiques. Une structure de données dédiée à la simulation de $(\mathrm{SD})^2$ doit offrir à la fois une grande *expressivité* afin de représenter aisément des structures arbitrairement complexes, une certaine *souplesse* pour autoriser la modification à la volée de la structure (capturant le caractère dynamique de cette dernière) et être bien *adaptée* à l'expression des lois d'évolution spécifiant la dynamique du système.

Des structures de données possédant ces différentes propriétés ont déjà été étudiées et utilisées en modélisation géométrique pour la représentation d'objets physiques. Partant de la notion d'*interaction* et des concepts élaborés en modélisation du solide, nous proposons d'utiliser la notion de *complexe cellulaire abstrait* issue de la topologie algébrique. Un complexe cellulaire permet la représentation discrète d'un espace à travers un ensemble de *cellules topologiques*, abstractions d'un espace élémentaire, recollées suivant une relation de voisinage appelée *relation d'incidence*. Le complexe cellulaire fournissant la description de la structure du système, nous décorons cet espace par des valeurs, ce qui aboutit à la notion de *chaîne topologique*. Cette notion est également issue de la topologie algébrique et permet d'associer un élément d'un groupe mathématique aux cellules d'un complexe.

Afin de démontrer que les structures de données usuelles peuvent être prises en compte par le point de vue topologique que nous proposons, nous illustrons cette formalisation topologique de la notion de collection en exprimant un grand nombre de structures de données standards. Nous terminons enfin le chapitre en définissant un type universel de collections topologiques, type qui sera utilisé dans la définition de la sémantique du langage dans les chapitres suivants.

Chapitre 4 : Une sémantique naturelle pour MGS. Les différents concepts élaborés lors du développement du projet MGS, notamment les transformations, sont présentés en général de manière informelle. Nous proposons dans ce chapitre de définir une sémantique du langage MGS.

Nous nous focalisons sur un langage noyau, mini-MGS, fondé sur un λ-calcul non-typé et étendu par la spécification de deux types de transformations différentes : les *transformations de chemins* et les *patches*. La sémantique est présentée dans le style de la *sémantique naturelle* en établissant plusieurs relations d'évaluation. L'évaluation de l'application des transformations demande le développement de trois étapes essentielles :

1. L'application d'une règle de transformation : elle formalise le filtrage d'une sous-collection ainsi que l'évaluation de la partie droite des règles conditionnée par la sous-collection filtrée

à l'aide des variables de filtre.

2. La stratégie d'application des règles : une sous-partie de la sémantique du langage doit spécifier la politique imposée pour l'application des règles. Cette sous-partie est modulaire suivant la stratégie que l'on souhaite utiliser. Nous proposons la programmation de quatre stratégies différentes, toutes présentes dans l'interprète MGS. Les stratégies stochastiques sont étudiées à part dans le chapitre 5.
3. La reconstruction : il s'agit de l'opération permettant l'assemblage des sous-collections évaluées en partie droite des règles. C'est à cet endroit que les modifications topologiques des structures apparaissent. Ces modifications ne posent aucune difficulté au vu des concepts élaborés chapitre 3.

La définition de la sémantique est suivie de trois exemples d'utilisation : le premier concerne la spécification d'une modification topologique, le deuxième étudie l'application d'une transformation selon la stratégie utilisée, et le troisième présente un exemple de preuve construite sur la sémantique.

Chapitre 5 : Stratégies d'application probabilistes. La sémantique développée au chapitre 4 ne décrit pas de manière satisfaisante les stratégies stochastiques disponibles dans l'interprète MGS. En effet, si ces stratégies sont intéressantes du point de vue de la simulation, c'est qu'il est possible de contrôler les probabilités d'applications des règles d'une transformation. Une « vraie » sémantique des opérations stochastiques doit donc spécifier l'ensemble des résultats possibles, pondérés par leurs probabilités d'évaluation. Par comparaison, la sémantique que nous avons proposée au chapitre précédent établit simplement une relation entre une expression et les résultats retournés possiblement par son évaluation.

L'introduction de probabilités dans les sémantiques des langages n'est pas une extension simple. En effet, un grand nombre de précautions doivent être prises pour que les opérations sur les lois de probabilité utilisées soient correctes. On pense notamment à la vérification de l'indépendance de deux événements afin de multiplier leur probabilité.

Après avoir défini la notion de densité de probabilité qui permet d'associer une probabilité à tout résultat possible de l'évaluation d'une expression, et cela en posant les hypothèses nécessaires à leur bonne utilisation, nous modifions entièrement la sémantique définie dans le chapitre 4 pour prendre en compte des stratégies stochastiques.

Nous développons en particulier deux stratégies probabilistes : une stratégie stochastique standard inspirée des travaux concernant la réduction stochastique d'un système de réécriture de termes et une stratégie plus sophistiquée qui permet de faire dépendre les probabilités d'application du nombre d'occurrences de ces applications. Cette stratégie permet notamment la définition en MGS de simulations à événements discrets probabilistes comme par exemple la simulation exacte de réactions chimiques suivant l'approche proposée par D.T. Gillespie. La méthode dite de *Gillespie* est présentée en détail dans le chapitre 10 qui lui est complètement dédié.

Chapitre 6 : Analogie avec les formes différentielles. Pour clore cette partie, nous proposons une nouvelle perspective sur la notion de transformation. Les chapitres précédents fournissent une formalisation des transformations via un mécanisme calculatoire permettant d'évaluer le résultat de l'application d'une transformation sur une collection. Néanmoins, ils ne fournissent pas de sens profond à cette opération.

Intuitivement, les transformations permettent d'itérer une fonction localement sur un espace défini par une collection topologique et de « sommer » les résultats locaux pour construire le

résultat global de l'application. Cette description n'est pas sans rappeler l'intégration de formes différentielles. En effet, une forme différentielle évalue localement des résultats qui seront ensuite sommés sur toute l'étendue d'un domaine. Il est donc extrêmement intéressant d'étudier et d'analyser le lien qui peut exister entre les transformations MGS et le calcul différentiel pour deux raisons : cette analogie peut se poursuivre et des notions développées dans le domaine du calcul différentiel peuvent peut-être enrichir la notion de transformation (on pense notamment à des mécanismes comme la dérivation ou l'intégration par partie). La seconde raison concerne notre domaine d'application : en effet, les SD sont souvent spécifiés à l'aide d'équations différentielles continues, ensuite discrétisées et intégrées numériquement. La formalisation d'un lien entre transformation et forme différentielle permettrait de simplifier ce processus et de se rapprocher encore des formalismes employés naturellement par les modélisateurs.

Dans le chapitre 3, les collections topologiques sont présentées suivant le point de vue des chaînes topologiques de la topologie algébrique. Cette branche des mathématiques permet de définir un équivalent discret à la notion de forme différentielle. L'équivalent discret d'une forme différentielle s'appelle une *cochaîne* ; les cochaînes sont des homomorphismes de chaînes. Il est possible de définir simplement une cochaîne par une transformation mais nous allons plus loin en définissant en MGS, les équivalents des notions d'opérateur différentiel et de laplacien.

La chapitre s'organise comme suit. Après avoir rappelé de façon non exhaustive la théorie des formes différentielles et sa discrétisation, nous formalisons le concept de cochaîne poursuivant ainsi les définitions présentées dans le chapitre 3. Nous mettons en avant un certain nombre d'opérateurs notamment utilisés en physique discrète, permettant de manipuler les valeurs décorant les cellules topologiques d'une chaîne topologique. Nous terminons le chapitre en construisant l'analogie entre calcul différentiel et les concepts développés dans le projet MGS, et en illustrant cette analogie par la programmation, presque immédiate au vu des définitions précédentes, des opérateurs discrets en MGS.

Le chapitre s'achève en évoquant le lien entre les opérations étudiées et les algèbres de programmes, en détaillant un exemple de programme algorithmique pouvant être spécifié par une simple équation discrète.

3.3 Troisième partie : Exemples et applications

La troisième partie de ce document est consacrée à l'illustration du langage à travers des exemples importants. Ces chapitres permettent de répondre affirmativement à deux des questions posées précédemment : *oui*, il est possible de manipuler de manière purement déclarative des représentations spatiales complexes, et *oui*, la prise en compte de relations topologiques dans un langage de programmation permet de simplifier la spécification et la simulation des $(SD)^2$.

Chapitre 7 : Introduction aux exemples d'applications. Ce chapitre introductif décrit succinctement le domaine d'application privilégié du projet MGS et présente *l'analyse a priori* qui nous a amenés à concevoir, développer et formaliser les notions de collection topologique et de transformation pour la simulation des $(SD)^2$. Cette analyse se concentre sur la description de l'évolution du système en termes d'*interactions locales* entre des entités plus élémentaires qui composent ce système. Ces interactions induisent naturellement une structure topologique sur l'état du système, structure qui peut être utilisée pour définir la structure de données qui représentera l'état du système dynamique et pour spécifier la fonction d'évolution.

Le chapitre se termine par une analyse des types de motifs que nous avons utilisés : trois classes de motifs se dégagent, correspondant aux applications « classiques » en physique qui exhibent un caractère très local, aux applications « algorithmiques » qui nécessitent des motifs complexes et aux $(SD)^2$ qui sont un hybride des deux.

Chapitre 8 : Modifications topologiques. L'un des enjeux du développement des transformations de collections topologiques est la spécification de modifications structurelles de l'état des systèmes afin de simuler l'évolution de (SD)². Le langage de motifs de chemin initialement développé en MGS s'est révélé trop limité pour traiter ces exemples, ce qui a motivé le développement du langage de patches. Ce chapitre met en avant la construction de structures topologiques complexes à l'aide des patches. Nous illustrons la puissance des motifs de patches à travers deux exemples.

Le premier exemple concerne l'implantation en MGS de différents algorithmes de subdivision de maillage. Cette opération non-triviale est très importante et trouve ses applications en CAO, en géométrie algorithmique et en informatique graphique. Nous avons implanté un grand nombre d'algorithmes de subdivision afin d'illustrer la facilité de programmation et l'abstraction des programmes MGS vis-à-vis des structures de données sous-jacentes encodant la structure des objets. Le même programme MGS permet aussi bien de subdiviser des G-cartes que des complexes cellulaires abstraits alors que l'opération de subdivision est très souvent programmée de façon *ad hoc* pour chaque structure de données. Ce résultat est possible en MGS car le programme est la traduction directe des descriptions mathématiques des algorithmes. De ce point de vue, cet exemple illustre le bilan positif du projet MGS par rapport à cette problématique.

Le second exemple illustre l'utilisation des transformations pour modéliser des systèmes auto-assemblés (*sef-assembly*). Ces systèmes sont présents dans beaucoup de domaines et posent le problème de l'émergence de propriétés globales par la spécification uniquement locale des interactions. Nous nous concentrons sur le développement d'un espace fractal : le triangle de Sierpinski. Deux approches sont proposées : l'une permet la construction d'un motif représentant la fractale sur un espace défini *a priori*, illustrant une forme d'auto-assemblage dit par *accrétion* ; l'autre décrit la construction de l'espace en terme de découpage de complexe cellulaire (auto-assemblage par sculpture ou *carving*).

Chapitre 9 : Quelques aspects de simulation numérique. En tant que langage dédié à la simulation de systèmes dynamiques, MGS doit fournir la possibilité de programmer la résolution de problème d'analyse numérique. À travers leur lien avec le calcul différentiel que nous avons détaillé dans le chapitre 6, les transformations doivent permettre l'expression de ces méthodes de résolutions numériques de façon expressive et proche de leur description mathématique. Ce chapitre illustre par deux exemples comment les transformations peuvent être utilisées dans le domaine de l'analyse numérique.

Le première exemple utilise le prétexte de la programmation en MGS d'un modèle d'oscillateurs couplés pour décrire la programmation de l'intégration numérique d'équations différentielles non-linéaires. Nous mettons en œuvre les deux méthodes les plus courantes d'intégration : la méthode d'Euler et les méthodes de Runge Kutta d'ordre 2 et 4. Nous observons très informellement à travers cet exemple les différentes performances de ces méthodes.

Le second exemple est plus consistant en terme de programmation. En effet, il consiste en la programmation en MGS d'un modèle de déplacement cellulaire. Le modèle originel, issu des travaux présentés dans [BMR+02], utilise la méthode des éléments finis dont nous programmons l'instanciation sur cet exemple en MGS. Cet exemple est paradigmatique des systèmes que nous souhaitons manipuler : il requiert la simulation d'un modèle mécanique et d'un modèle bio-chimique sur une structure hautement dynamique. Nous mettons en place un couplage de ces deux modèles dont les échelles de temps diffèrent de façon significative (une approximation quasi-statique est utilisée pour le modèle bio-chimique).

Chapitre 10 : La stratégie `'gillespie`. Ce chapitre d'exemples est particulier car il est dédié à l'une des stratégies d'application de règles de MGS : la stratégie `'gillespie`. D.T. Gillespie a proposé une méthode de simulation stochastique et exacte de réactions chimiques dans un milieu homogène. Nous proposons d'utiliser cette méthode par l'intermédiaire d'une stratégie d'application dédiée pour simuler avec MGS l'évolution de solutions chimiques. Nous profitons de la réécriture de multi-ensemble qu'offre l'interprète pour modéliser la solution et sa dynamique. Ce travail est le fruit d'une collaboration avec P. Prusinkiewicz de l'université de Calgary ; nous reprenons en grande partie un article (en cours de soumission) dans lequel les techniques stochastiques élaborées dans le cadre des L systèmes [Cie06] sont utilisés avec les P systèmes pour généraliser la méthode de simulation de D.T. Gillespie initialement développée pour des solutions homogènes, à des espaces compartimentés.

Nous commençons par rappeler les fondements mathématiques de la méthode de D.T. Gillespie. Il s'agit essentiellement de la définition de lois de probabilité sur le déclenchement des réactions chimiques pouvant se produite dans une solution. Nous terminons alors en étendant cet algorithme à une solution hétérogène où des molécules sont isolées par des compartiments.

Nous illustrons ces propos par deux exemples d'application. Le premier concerne la modélisation du switch génétique du phage λ, et permet de décrire l'utilisation de l'algorithme original de D.T. Gillespie. Le second est tiré d'un exemple plus complexe mettant en jeu des réactions isolées dans des compartiments. Il s'agit de l'infection du virus de la forêt de *Semliki* qui a déjà été modélisée en *brane calculus* [Car04] et qui nous permet d'illustrer notre extension de l'algorithme de D.T. Gillespie à des espaces hiérarchiquement organisés.

Chapitre 11 : Modification topologique et simulation biologique. Ce dernier chapitre de la partie concerne l'un des objectifs les plus motivants du projet MGS : la *morphogenèse*. Il s'agit d'étudier et de modéliser le développement et la construction de « formes complexes » dans la nature. La morphogenèse est l'exemple paradigmatique de $(\text{SD})^2$ demandant la représentation d'un espace arbitrairement complexe et l'expression des lois d'évolution du système.

Nous commençons ce chapitre par introduire l'intérêt du formalisme issu des collections topologiques et des transformations pour représenter le développement du vivant. Ce type de systèmes correspond à la notion de $(\text{SD})^2$ et illustre l'utilité d'introduire de nouveaux formalismes pour les modéliser.

Nous illustrons l'utilisation de MGS pour programmer un tel système : un modèle simpliste de la *neurulation*. La neurulation est un processus indispensable de l'embryogénèse chez les vertébrés à l'origine du système nerveux. Nous simplifions ce processus pour le décrire comme la déformation d'un feuillet épithélial passant alors d'une conformation planaire à une conformation cylindrique. Le système est représenté de façon discrète par un système masses/ressorts. Un modèle mécanique décrit *localement* la déformation des cellules épithéliales entraînant la déformation globale de toute la structure. La modification de la topologie est finalement programmée à l'aide d'un patch.

3.4 Quatrième partie : Conclusions et annexes

Dans cette partie, nous commençons par résumer nos travaux et discutons des nombreuses perspectives ouvertes par cette recherche.

Enfin, nous développons dans l'annexe A quelques aspects importants de l'implantation de l'interprète MGS et les deux développements annexes pour la visualisation de règles de réécriture et des simulations.

4 Contributions

Le travail réalisé au cours de cette thèse prend place dans le projet MGS[2]. On distingue trois types de contributions : la formalisation, le développement des exemples et l'implantation.

La formalisation. Mes contributions concernant la formalisation sont les suivantes :

- j'ai formalisé les collections topologiques à travers la notion de complexe de chaînes ;
- j'ai formalisé les transformations de chemins et de patches dans un cadre unique ;
- j'ai formalisé la sémantique du langage dans un style naturel ; cette formalisation comprend la formalisation du filtrage, des stratégies d'application (y compris les stratégies stochastiques) et de la reconstruction ;
- j'ai débuté la formalisation permettant de rapprocher la notion de transformation et le concept de forme différentielle.

Ces développements sont largement détaillés dans la partie II de ce document.

Les applications et exemples. J'ai développé un grand nombre d'exemples pour tester et valider les constructions du langage et les développements logiciels effectués. Les plus importants développements sont présentés dans ce document :

- algorithmes de subdivision de surface,
- auto-assemblage et construction de figures fractales,
- intégration numérique (co-encadrement du stage de 2^{e} année d'école d'ingénieur de Lionel Perret – ECP),
- méthode des éléments finis (co-encadrement du stage de DEA de Damien Boussié),
- modèle stochastique du switch génétique du phage λ,
- modèle d'infection virale,
- neurulation et morphogenèse.

L'implantation. Mes contributions au développement de l'interprète MGS sont les suivantes :

- j'ai implanté les expressions d'aiguillage et le mécanisme d'exceptions ;
- j'ai implanté les types mutuellement récursifs et les contraintes de types MGS ;
- j'ai participé au développement des collections topologiques ; j'ai en particulier conçu et implanté les collections de dimension arbitraire ;
- j'ai intégré à l'interprète la bibliothèque de manipulation de G-cartes du modeleur MOKA ; j'ai en particulier codé des fonctions de haut niveau (sur les cellules topologiques) pour la manipulation fonctionnelle des G-cartes, dans le cadre d'une collaboration avec le laboratoire SIC de l'université de Poitier ;
- j'ai participé à l'implantation de l'algorithme générique du filtrage de chemins ;

[2]Le projet MGS est mené dans l'équipe LIS *Langage, Interaction et Simulation* du laboratoire IBISC, Informatique, Biologie Intégrative et Systèmes Complexes. Ce laboratoire est issu de la fusion au 1er janvier 2006 du LaMI, Laboratoire des Méthodes Informatiques, et du LSC, Laboratoire des Systèmes Complexes. Le projet MGS a débuté dans l'équipe SPECIF du LaMI.

- j'ai notamment conçu un mécanisme d'optimisation d'une classe particulière de motifs pour le filtrage modulo associativité-commutativité, fondé sur la notion d'élément filtré contraint ;
- j'ai également développé l'opération de descente du filtrage de chemins (opérateur \/ des motifs de chemin) ;
- j'ai conçu et réalisé l'implantation des transformations de patches (langage de motifs, reconstruction des collections et optimisation du filtrage) ;
- j'ai participé au développement de nouvelles stratégies d'application de règles en collaboration avec O. Michel ; j'ai notamment aidé à mettre en place la stratégie fondée sur l'algorithme de D.T. Gillespie ;
- j'ai co-encadré avec O. Michel plusieurs stages :
  1. **Romain Derrasse** : stage de 2e année d'école d'ingénieur. Programmation d'une bibliothèque de graphes fondée sur OCamlGraph pour MGS.
  2. **Yann Julian** : stage de 2e année d'école d'ingénieur. Programmation d'un éditeur graphique de règles de réécriture des transformations de patches MGS. J'ai amené l'étudiant à suivre un schéma de développement « Modèle-Vue-Controleur » fixant l'architecture logiciel de ses contributions.
  3. **Fabien Thonnerieux** : stage de 2e année d'école d'ingénieur. Description et visualisation de scènes 3D pour le langage de programmation MGS. J'ai participé à l'encadrement de façon informelle.

Ces contributions sont les moins valorisées dans ce document mais ont pris une grande partie de mes activités de recherche durant mes trois années de thèse.

Les publications. Ces travaux ont donné lieu à plusieurs publications dans des livres [GS06a, GS06b], des journaux [SM05a, SMC+06], des conférences internationales [SMG04a, SM04a, GMCS05, SMG05, SM05b], des conférences nationales [SM04b, Spi05, SM06], et des rapports techniques [Spi03, SMG04b, GMCS04].

Première partie

Une introduction à MGS

Chapitre 2

Le langage MGS

L'objet de ce chapitre est d'introduire le lecteur aux concepts développés dans le projet MGS, ainsi qu'au langage de programmation éponyme. La particularité de ce langage est le point de vue topologique qu'il offre pour la manipulation de structures de données. Les *collections topologiques* proposent un support uniforme pour la représentation d'agrégats de données, et les *transformations* fournissent un cadre unifié pour la manipulation par réécriture de ces collections, à travers la définition par cas de fonctions.

Un interprète du langage MGS a été réalisé en parallèle aux développements théoriques issus du projet. Dans cette implantation, MGS est un langage applicatif (les calculs produisent des nouvelles valeurs et n'agissent pas par effet de bord) et est dynamiquement typé (les types sont

vérifiés à l'exécution du programme). Il est disponible librement à partir du site Internet du projet[1].

Dans ce chapitre, cette implantation de MGS est décrite. Section 1, nous commençons par présenter MGS comme un langage fonctionnel des plus classiques, fournissant des constantes élémentaires appelées *scalaires*, la définition de variables et de fonctions, des structures de contrôle et un mécanisme d'exceptions. La section 2 étend ce langage standard avec un nouveau type de valeurs, les *collections topologiques*, qui sont l'unique source de structuration de données. Nous terminons avec la présentation des transformations, section 3. Nous décrivons particulièrement les deux langages de motifs proposés par l'interprète offrant la possibilité d'utiliser d'une part les $\langle n, p \rangle$*-transformations* et d'autre part les *patches*.

1 MGS : un langage applicatif

Le cœur de l'interprète MGS implante un langage de programmation fonctionnel [Coh04, Coh05] proposant les quelques éléments de base suivants pour assurer les calculs : des valeurs scalaires avec leurs opérations, un environnement pour la définition et la manipulation de variables globales et locales, des définitions et appels de fonctions, des structures de contrôle dont un mécanisme d'exceptions, et des fonctions d'entrées/sorties.

1.1 Les constantes scalaires

Les valeurs *scalaires* sont les constantes atomiques (*i.e.* ne pouvant être décomposées) du langage. Le langage fournit :

- les valeurs usuelles (les entiers, les entiers longs provenant de la bibliothèque `bigint` d'OCaml [LDG+04, page 409], les flottants, les chaînes de caractères) ainsi que leurs opérations standards ; les opérations flottantes respectent la norme IEEE 754 [Gol91] ;
- les valeurs booléennes `true` et `false` ; en outre, toute valeur MGS est automatiquement convertie en booléen dans certains contextes particuliers comme les conditionnelles (par exemple l'entier `0`, le flottant `0.0` et la chaîne de caractères vide `""` valent `false`) ;
- les *symboles*, des valeurs symboliques `'Nom` où `Nom` est un identifiant quelconque ; leur fonctionnement est similaire à celui des constructeurs de *variants* polymorphes d'arité nulle en OCaml [LDG+04, page 14] ;
- la valeur indéfinie `<undef>` utilisée comme résultat d'un calcul non défini ;
- les *G-cartes* (section 2, page 38), les *cellules topologiques abstraites* (section 2, page 36) et les *directions* `posgbf` (section 2, page 33) utilisées pour la construction d'espaces topologiques que nous décrirons plus loin dans cette thèse.

1.2 Les variables

MGS distingue deux catégories de variables : les variables *globales* et *locales*.

Les variables globales MGS sont *mutables* de telle sorte que la valeur qui leur est associée peut être réaffectée à tout moment. L'accès à cette valeur est fait via le nom de la variable ; la définition et la réaffectation des variables sont dénotées par l'utilisation de l'opérateur `:=`. Soit `x` une variable globale à laquelle est associé l'entier `10` ; l'expression

```
x := x + 1 ;;
```

[1] Accessible à l'adresse `http://mgs.ibisc.univ-evry.fr/`

affecte l'entier `11` à cette variable et retourne la valeur entière `11`. Le caractère impératif des variables globales a été choisi pour faciliter leur utilisation et n'affecte en rien le caractère *fonctionnel* de MGS qui opère par calcul de nouvelles valeurs sans effet de bord.

Les variables locales sont définies à l'aide de structures

```
let x = ... and y = ... in ... ;;
```

proches de celles disponibles en OCaml. Par exemple, l'expression

```
let x = 2 in
  x * (let x = 10 in x)
;;
```

s'évalue en la valeur entière `20`.

1.3 Les fonctions

λ-expression. MGS est un langage d'*ordre supérieur* ; les fonctions peuvent être utilisées comme argument ou résultat d'autres fonctions. Les fonctions anonymes sont définies dans une syntaxe similaire à celle du λ-calcul. Par exemple, l'expression `\x.x` définit la fonction identité. L'application d'une λ-expression se fait à l'aide des parenthèses :

```
(\x.x+1)(10) ;;
```

renvoie l'entier `11`.

Les fonctions MGS sont toutes *curryfiées*. Par exemple, l'application partielle de la fonction anonyme `\x,y.x+y` à l'argument `1` retourne une fonction qui peut être utilisée pour calculer le successeur :

```
let prec = (\x.\y.x+y)(1) in prec(2) ;;
```

s'évalue en `3`. Pour alléger la syntaxe, l'écriture

```
\x.\y.x(y)
```

peut être réécrite en l'expression équivalente

```
\x,y.x(y)
```

Nommage des fonctions. Bien qu'il soit possible d'utiliser l'opérateur d'affectation des variables globales pour nommer de nouvelles fonctions, une construction particulière est dédiée à leur déclaration. L'expression

```
fun succ(x) = x + 1 ;;
```

associe la λ-expression `\x.x+1` à la nouvelle variable `succ`.

Cette syntaxe jouit du même allègement syntaxique que les λ-expressions : `f(1,2,3,4)` est équivalent à l'écriture plus lourde `(((f(1))(2))(3))(4)`.

L'identificateur de la fonction peut être utilisé dans sa propre définition :

```
fun fact(n) = if (n <= 0) then 1 else n*fact(n-1) fi ;;
```

offrant ainsi la possibilité de définir des fonctions récursives.

Arguments optionnels. MGS propose également un système d'arguments optionnels (similaire à l'esprit du travail présenté dans [FG95]) pouvant être liés à la déclaration d'une nouvelle fonction (en utilisant la construction `fun`). Le comportement obtenu correspond à celui des variables locales statiques disponibles en C/C++. La déclaration précédente peut alors être réécrite de la façon suivante :

```
fun f[x=1](z) = x := x + z ;;
```

L'*option* `x` est associée au nom de fonction `f`. Le comportement lors de l'appel à cette fonction est le suivant : deux appels consécutifs `f(10)` retournent respectivement `11` et `21`. Il est également possible de modifier la valeur de l'option `x` par

```
f[x=3](10) ;;
```

qui affecte la variable `x` à `3` puis retourne l'entier `13`. Finalement, un dernier appel `f(10)` retourne l'entier `23`.

Itération de fonction. L'itération d'une fonction est un processus essentiel en MGS. C'est pourquoi, un certain nombre de paramètres optionnels prédéfinis sont fournis offrant un contrôle sur l'application et l'itération de fonctions. Soit `f` une fonction MGS et `x` son argument :

`f[fixpoint](x) :` itération de la fonction `f` jusqu'à l'obtention d'un point fixe. On calcule la limite x de la suite

$$x_0 = \texttt{x}$$
$$x_{i+1} = \texttt{f}(x_i)$$

telle que x`==f(`x`)`, si $(x_i)_{i\in\mathbb{N}}$ est convergente. Dans le cas contraire, le comportement de l'itération n'est pas défini.

La fonction de comparaison `==` utilisée pour vérifier l'égalité peut être paramétrée. Soit `cmp` une fonction de comparaison fournie par le programmeur, l'expression

```
f[fixpoint=cmp](x0)
```

retourne la valeur x' vérifiant `cmp(`x'`,f(`x'`))`.

`f[iter=`n`](x) :` n itérations de `f` sur `x` :

$$(\overbrace{\texttt{f} \circ \cdots \circ \texttt{f}}^{n \text{ fois}})\texttt{(x)}$$

Le paramètre n peut être remplacé par un prédicat à un argument (le résultat de l'itération en cours) qui stoppe l'application une fois vérifiée.

`f[prelude=`g`](x) :` fonction appliquée avant les itérations.

`f[interlude=`h`](x) :` fonction appliquée entre chaque itération.

`f[postlude=`i`](x) :` fonction appliquée après la dernière itération.

Les quatre dernières options peuvent être combinées pour obtenir le comportement suivant :

```
f[iter=3,prelude=g,interlude=h,postlude=i](x)
```

équivaut à l'expression :

```
i(f(h(f(h(f(g(x)))))))
```

Par exemple le programme suivant illustre l'utilisation de ces options :

```
fun pp(mess, v) =
  stdout << mess << " iteration=" << iteration << ", value=" << v ;
  v
;;
fun succ(x) = x+1 ;;
succ[ prelude   = pp("pre:  "),
      postlude  = pp("post:  "),
      interlude = pp("inter:  "),
      iter      = 3 ]
    (11)
;;
```

La fonction `pp` écrit un message sur la *sortie standard* (l'utilisation de l'opérateur `<<` est similaire à celle de ce même opérateur en C++ ; voir les fonctions système page 22) ; la variable `iteration` est locale aux fonctions associées aux paramètres optionnels prédéfinis et correspond à un compteur d'itération. L'appel à la fonction `succ` génère alors la sortie suivante :

```
pre:  iteration=0, value=11
inter:  iteration=1, value=12
inter:  iteration=2, value=13
post:  iteration=3, value=14
```

1.4 Les structures de contrôle

MGS propose trois structures de contrôle : la conditionnelle, l'aiguillage et la boucle.

Les conditionnelles. L'évaluation conditionnée d'une expression se fait à travers l'instruction standard

`if` exp_b `then` exp_t `else` exp_f `fi`

La valeur de l'expression ci-dessus est celle de exp_t si l'expression exp_b est évaluée en la valeur booléenne `true` (ou une valeur équivalente), ou celle de exp_f sinon.

Les expressions d'aiguillage. Les expressions d'aiguillage correspondent à des conditionnelles imbriquées. Elles sont similaires au `switch` qu'on peut trouver en C et en C++ :

`switch` exp
  `case` scl_1 : exp_1
  ...
  `case` scl_n : exp_n
  `[default :` exp_d`]`
`endswitch`

Si l'expression exp s'évalue en une valeur scalaire égale à scl_i, l'expression exp_i est évaluée. Dans le cas où le scalaire n'apparaît pas dans la liste, l'expression par défaut exp_d est évaluée si celle-ci est spécifiée. Dans le cas contraire, la valeur `<undef>` est retournée.

Les boucles. L'expression `while` exp_b `do` exp évalue l'expression exp tant que l'expression exp_b s'évalue en `true`. La valeur retournée par cette construction est la valeur de exp. Si la condition exp_b n'est pas vérifiée dès la première itération, la valeur retournée est `<undef>`. Par exemple :

```
while (true) do
  stdout << "OK\n"
;;
```

écrit une infinité de fois le mot « OK » sur la sortie standard.

1.5 Les exceptions

Les structures `raise ...` et `try ... with ...` permettent l'utilisation des exceptions en MGS. Le mécanisme est similaire à celui fourni dans OCaml [LDG+04], au `catch/throw` de C++ [Str86], ou encore au `setjump/longjump` de C [KR78].

Pour *lever* une exception, le mot clé `raise` est utilisé de la façon suivante :

raise *symb*

Cette exception est identifiée par un symbole MGS dénoté ici par *symb*. Une séquence d'arguments peut alors lui être associée :

raise *symb* `(`arg_1`, `arg_2`, ...)`

Ces exceptions peuvent être *rattrapées* à l'aide de la construction suivante :

try *exp*
with
  `|` $symb_1$ `(`arg_{11}`, ...)` `->` exp_1
  `|` $symb_2$ `(`arg_{21}`, ...)` `->` exp_2
`...`

Si l'expression *exp* lève une exception, le symbole l'identifiant est comparé aux $symb_i$ de la liste. Si le symbole $symb_i$ correspond, l'expression exp_i est alors évaluée où les variables libres arg_{ij} sont substituées par les arguments de l'exception. Ainsi, l'expression

```
try
  raise 'S (1, 2, 3)
with
  | 'T (x, y)    -> x * y
  | 'S (x, y, z) -> x + y + z
;;
```

retourne l'entier 6.

1.6 Les fonctions système

MGS propose un certain nombre de fonctions système. Les principales servent à gérer les entrées/sorties pour stocker le résultat de calculs, ou encore à exécuter des logiciels compagnons permettant par exemple une visualisation graphique de ces données.

L'opérateur principal des sorties système est `<<`. À l'image de son homologue en C++, il permet d'envoyer dans un canal sortant des valeurs MGS : `"file" << v` écrit la valeur `v` dans le fichier spécifié par `"file"`. La valeur retournée lors de l'évaluation de cette expression est `"file"`, de telle sorte que l'utilisation chaînée de `<<` est possible :

```
"file" << v1 << v2 << v3 << "\n" ;;
```

Le premier argument peut être :

- la chaîne de caractères `"stdout"` correspondant à la sortie standard,
- la chaîne de caractères `"stderr"` correspondant à la sortie standard des erreurs,
- le chemin relatif ou absolu vers un fichier exécutable (celui-ci étant alors exécuté avec en entrée standard les valeurs passées en partie droite de `<<`),
- le chemin relatif ou absolu vers un fichier non exécutable (celui-ci est créé s'il n'existe pas ou écrasé dans le cas contraire, et reçoit les valeurs fournies en partie de droite de `<<`).

L'expression `close("file")` permet alors de fermer le fichier.

Voici un exemple d'utilisation de l'opérateur `<<` pour produire une sortie GNUPLOT [WK04] :

```
"/usr/bin/gnuplot" << "plot cos(x) ;\n" ;;
```

ouvre un canal (via l'appel système `popen`) vers un nouveau processus GNUPLOT et lui envoie la commande dessinant le graphe de la fonction `cos`. L'expression suivante

```
"/usr/bin/gnuplot" << "plot x+sin(x) ;\n" ;;
```

remplace le dessin de la fonction `cos` par celui de la fonction `x+sin(x)` dans la fenêtre GNUPLOT précédemment ouverte. Finalement

```
close("/usr/bin/gnuplot") ;;
```

ferme le canal (appel système `pclose`).

2 Collections topologiques : les structures de données MGS

2.1 Point de vue local des structures de données

Les valeurs calculables à partir du langage tel qu'il vient d'être présenté sont atomiques et ne peuvent pour l'instant pas être agrégées pour former des organisations de données plus complexes.

L'unique source d'organisation des données qu'autorise MGS est le concept de *collection topologique*, offrant au programmeur un point de vue topologique pour traiter ces constructions. On entend par *collection*, un agrégat de données, c'est-à-dire un ensemble de valeurs scalaires ou de (sous-)collections muni d'une organisation précisant comment ces données sont agencées les unes par rapport aux autres. Une collection est donc une *structure de données* au sens algorithmique du terme.

Une *collection topologique* est une collection où la structure est capturée par une *relation de voisinage* entre les données, c'est-à-dire en fournissant à partir d'un des éléments de la collection, l'ensemble des autres données qui lui sont directement liées.

L'origine de ce concept vient du souhait d'unifier les structures de données standards (telles que les séquences, les (multi-)ensembles, les graphes, les tableaux, les matrices, etc) dans un même cadre. La généricité offerte par les collections topologiques provient des « déplacements usuels » suivis dans la structure de données, c'est-à-dire de l'ordre de parcours de ses éléments. Cet ordre est capturé par une notion de voisinage rapprochant deux éléments contigus dans la structure. Cette relation de voisinage est fondée sur le fait que le choix d'un programmeur pour une structure de données est motivé par l'utilisation algorithmique de l'objet représenté. Pour

illustrer cette affirmation, nous proposons d'observer la construction des voisinages pour deux structures de données différentes : les listes et les matrices.

- Les listes, telles qu'elles sont définies en OCaml, invitent par leur construction inductive à parcourir leurs éléments de gauche à droite, par exemple lors de descentes récursives. Pour que ce parcours soit possible, chaque élément de la liste doit avoir un voisin vers la droite (excepté pour le dernier élément). Ce simple point de vue local sur l'organisation des éléments est suffisant pour définir une collection topologique « émulant » les listes.
- Pour les matrices, il est d'usage[2] de les représenter par un vecteur de lignes ou un vecteur de colonnes. Le choix de l'une ou l'autre de ces deux représentations est influencé par son utilisation : souhaite-t-on parcourir les éléments par lignes ou par colonnes ? La représentation par colonnes facilite évidemment les parcours par colonnes, mais accentue fortement les difficultés de programmation des algorithmes par lignes, et réciproquement. Le point de vue local renverse la situation en capturant ces deux parcours dans une unique notion de voisinage : chaque élément e_{ij} (exceptés les éléments aux bords) possède 4 voisins, $e_{i+1,j}$, $e_{i,j+1}$, $e_{i-1,j}$ et $e_{i,j-1}$. Ce voisinage a été historiquement utilisé par von Neumann. Pour les parcours en diagonal, il suffit de considérer 4 voisins supplémentaires $e_{i+1,j+1}$, $e_{i-1,j-1}$, $e_{i-1,j+1}$ et $e_{i+1,j-1}$, utilisant ainsi le voisinage de Moore.

Toute structure de données peut ainsi être caractérisée par une relation locale entre ses éléments [GM02a].

2.2 Notions de voisinage et d'incidence

Les collections topologiques capturent cette organisation locale des structures de données à l'aide d'une relation de voisinage qui associe à chaque élément de la collection les éléments qui lui sont à proximité. Il est commode de représenter cette relation à l'aide d'un *graphe de voisinage* où les nœuds représentent les éléments de la collection et sont donc décorés par leur valeur et les arcs, orientés, représentent la relation de voisinage. Du point de vue des collections topologiques, la topologie des listes que nous venons de voir, peut être décrite par un graphe linéaire où deux nœuds sont liés par un arc si les éléments correspondants sont contigus dans la liste. L'organisation des matrices peut être décrite par un graphe régulier où chaque sommet présente un nombre constant de voisins (4 ou 8 pour notre exemple).

Cette approche par graphe de voisinage permet de représenter l'ensemble des structures de données standards qu'on peut attendre d'un langage de programmation. Cependant, on peut remarquer que seules des entités de dimension 0 (les sommets) et 1 (les arcs) peuvent être manipulées, et que seuls les sommets sont décorés alors que les arcs restent vierges d'information. Le point de départ de cette thèse est l'extension du graphe de voisinage à une structure plus générale permettant d'une part de décorer les arcs, et d'autre part d'augmenter l'expressivité du graphe en y ajoutant des éléments de dimension supérieure à 1 ; on peut ainsi représenter facilement des structures de données plus spécialisées :

- les objets manipulés dans le domaine de la modélisation géométrique (maillage, triangulation, etc) [Sha01, Lie94],
- la modélisation d'objets physiques et biologiques pour la simulation [PS93, Pal94, Ton01, ES04],
- la modélisation et la représentation de connaissance [VG98]

[2] D'autres représentations sont encore possibles ; on pense en particulier à l'utilisation en analyse numérique de représentations compressées lorsque la matrice présente un nombre conséquent d'éléments nuls.

- etc.

Cette généralisation du graphe de voisinage à une structure spatiale de dimension supérieure s'appuie sur la notion de *complexe cellulaire* ; il s'agit d'une structure mathématique provenant de la *topologie algébrique* qui permet la représentation discrète d'espaces topologiques. L'utilisation qui en est faite par MGS est la création d'espaces topologiques arbitrairement complexes (c'est-à-dire avec des voisinages plus ou moins hétérogènes) servant de structure et de relation de voisinage pour les collections topologiques. De plus, le cadre offert par les complexes cellulaires permet une manipulation dynamique de la structure qu'il est possible de modifier pour construire des espaces plus complexes par transformation de relation de voisinage (raffinement, découpage, etc. de l'espace sous-jacent). En autorisant les modifications topologiques, les collections topologiques sont un outil élégant pour supporter les changements de structures dans les modélisations de systèmes dynamiques à structure dynamique [Gia03].

Un complexe cellulaire est un ensemble d'objets élémentaires de différentes dimensions appelés *cellules topologiques*, ou plus simplement *k-cellules* pour les cellules de dimension k. Une k-cellule est la représentation abstraite d'un espace « simple[3] » 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 dimensions différentes, et chaque partition va être représentée par une k-cellule où k est la dimension de la partition en question. Une relation entre les cellules, appelée *relation d'incidence*, est alors définie pour relater l'organisation de la partition, c'est-à-dire l'agencement entre les cellules topologiques. Cette relation repose sur la notion de bord : intuitivement, un triangle apparaissant dans une triangulation est une 2-cellule bordée par trois arcs eux-mêmes liés par trois sommets. De façon informelle, les cellules topologiques *incidentes* à une k-cellule d'un complexe cellulaire sont les cellules apparaissant dans son bord (ayant par conséquent une dimension inférieure à k) et les cellules la contenant dans leur bord (et donc de dimension supérieure à k). En particulier, on appellera *faces* (resp. *cofaces*) les cellules incidentes de dimension $k-1$ (resp. de dimension $k+1$). Nous définissons également le $\langle n, p\rangle$-voisinage de deux n-cellules : deux n-cellules sont $\langle n, p\rangle$-voisines si elles sont incidentes à une même p-cellule. Par exemple, deux sommets sont $\langle 0, 1\rangle$-voisins si un arc les relie. La relation d'incidence est une relation d'ordre partiel sur les cellules topologiques ; il est commode de la représenter par un *graphe d'incidence*, reposant sur le diagramme de Hasse de l'ensemble des cellules topologiques partiellement ordonné, où les nœuds sont les cellules topologiques du complexe, et où les arcs dénotent la relation de *face* entre les cellules. La figure 1 de gauche donne un exemple de complexe cellulaire.

Un complexe cellulaire définit uniquement la structure d'un espace à la façon d'une matrice dans laquelle aucun élément ne serait défini. À partir de cette structure, on définit une *collection topologique* comme l'association de données aux cellules topologiques du complexe. Cette façon de procéder se rapproche de la notion de *champ de données* où les données sont indexées par les éléments de $\mathbb{Z}$ [Lis93, LC94]. Les champs de données utilisés en parallélisme (data-field) correspondent à une généralisation des régions arbitraires de $\mathbb{Z}^n$ et généralisent la notion de tableau. Les collections topologiques permettent d'aller plus loin en considérant des espaces plus structurés et/ou plus complexes que $\mathbb{Z}^n$, représentés par des complexes cellulaires. Elles décrivent donc des *fonctions partielles* de complexes cellulaires, également appelés dans ce contexte espaces

[3] On entend par « simple » un espace homéomorphe à la boule ouverte unité de $\mathbb{R}^k$. De façon plus précise, les cellules topologiques de dimension inférieure à 2 sont obligatoirement homéomorphes à ces boules. Au delà, la caractérisation combinatoire des complexes cellulaires peut engendrer des cellules non-homéomorphes. Il apparaît d'ailleurs que savoir si une opération sur un complexe cellulaire n'engendre que des cellules homéomorphes à des boules est indécidable.

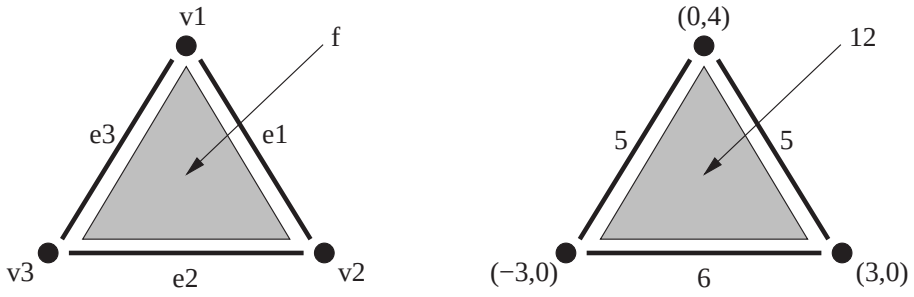


FIG. 1 – Sur la gauche, un exemple de complexe cellulaire : il est composé de trois 0-cellules (v_1, v_2, v_3), de trois 1-cellules (e_1, e_2, e_3), et d'une 2-cellule f. Le bord de f est constitué de ses cellules incidentes v_1, v_2, v_3, e_1, e_2 et e_3. Plus particulièrement, les trois arcs sont les faces de f, et par conséquent, f est une coface de e_1, e_2 et e_3. Sur la droite, des données sont associées aux cellules topologiques : des positions pour les sommets, des longueurs pour les arcs et une aire pour f.

de *positions* ou d'*indices*, vers un ensemble de valeurs (typiquement les valeurs du langage MGS). Cette généralisation des champs de données à des espaces arbitrairement complexes trouve déjà une utilisation en physique discrète où elle prend le nom de *chaîne topologique* [PS93, Pal94, Ton01, ES04]. Les chaînes topologiques sont utilisées pour représenter des champs vectoriels ou scalaires sur des espaces discrets. La figure 1 de droite présente un exemple de collection topologique définie sur le complexe de gauche. Une définition plus formelle de ces notions sera donnée dans le chapitre 3, page 63.

2.3 Quelques voisinages

Bien que leur définition soit générique, MGS propose de classer les collections topologiques pour retrouver et manipuler plus facilement des structures de données déjà connues. Ces différentes classes de collections topologiques sont appelées *types*. L'interprète n'implante pas de système statique d'inférence de types ; néanmoins, les types sont vérifiés dynamiquement à l'exécution. Un système d'inférence de type pour les collections topologiques et les transformations a fait l'objet d'une thèse [Coh04] et ne sera donc pas plus détaillé ici. Les différents types de collections sont distingués suivant les propriétés vérifiées par la relation d'incidence : en agissant sur celle-ci, on peut introduire une forme de régularité ou d'uniformité dans la structure. Les quelques sections suivantes présentent des collections topologiques spécifiques proposées dans l'interprète MGS ainsi que la forme la plus générale de collection, les *chaînes abstraites*. Nous commençons par détailler les concepts liés à la notion de type de collections topologiques.

2.3.1 Généralités sur les types des collections

Chaque type de collection est caractérisé par un nom. Ce nom t est notamment utilisé pour

- définir un prédicat éponyme vérifiant que son argument est une collection du type t (à la façon de la définition de type en Haskell [HF92]) ;
- dénoter une collection topologique vide, `t:()` ;
- spécifier une fonction de conversion `tify` d'une valeur quelconque en une valeur de type t.

Par exemple, le type des séquences MGS est noté `seq` ; les expressions

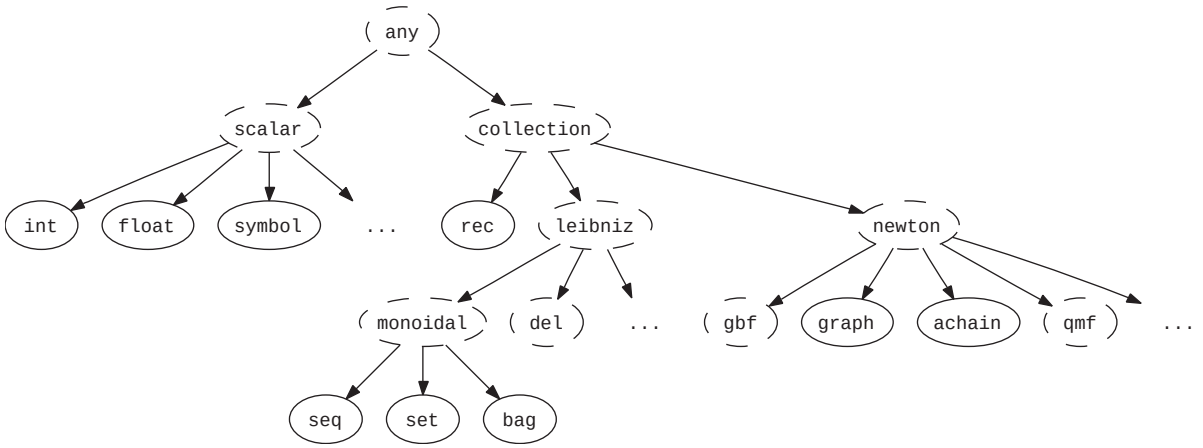


FIG. 2 – Hiérarchie des types MGS.

```
seq((1,2,3)) ;;
seq(1) ;;
```

retournent respectivement les valeurs booléennes `true` et `false` (les séquences MGS sont notées entre parenthèses et les éléments sont séparés par des virgules). La séquence vide est notée `seq:()`, et la fonction `seqify` permet de convertir toute valeur MGS (scalaire comme collection topologique) en séquence.

Hiérarchie de type et sous-typage. Les types de collection sont organisés hiérarchiquement et cette hiérarchie est étendue à tous les types de valeurs MGS. La figure 2 présente cette organisation. Hormis les types des valeurs scalaires qui ont déjà été présentés plus haut et le type `any` représentant la classe générale de toutes les valeurs MGS, les nœuds de cet arbre sont l'objet des paragraphes suivants où chaque type de collections est décrit.

Type abstrait et type concret. Plusieurs types de collections sont paramétrés par des valeurs MGS afin de calculer l'information nécessaire pour construire la relation d'incidence. Ces types génériques sont dits *abstraits*, la construction t`:()` ne pouvant être évaluée. À l'inverse, les autres types sont qualifiés de *concrets*, et pour eux, l'expression t`:()` est valide. Les exemples suivants présentent des types de collections des deux sortes. Cette différence apparaît sur la figure 2 : les types entourés en pointillés sont abstraits, et les types entourés par des cercles pleins sont concrets.

Collection leibnizienne et collection newtonienne. Les mathématiciens Leibniz et Newton avaient chacun une conception de l'espace différente. Pour le premier, l'espace est défini à partir des éléments qu'il contient et de leurs « positions » relatives les uns par rapport aux autres : *l'espace est construit de façon relative aux éléments qu'il contient.* À l'inverse, Newton possède une vision de l'espace comme d'un objet ayant une existence propre sur lequel d'autres objets sont placés et se déplacent.

Cette distinction apparaît dans les collections MGS. Certaines sont construites sur une relation entre les éléments ; leur structure résulte d'un calcul sur les données qui la compose. Elles sont dites *leibniziennes*. Dans ces collections, il n'est pas possible d'associer la valeur spéciale `<undef>` à une position, cette dernière n'existant que si une valeur la décore. La collection vide

ne contient aucune position ; celles-ci sont créées au fur et à mesure de l'ajout des éléments. Nous voyons parmi ces collections :

- les collections *monoïdales*, construites inductivement comme les éléments d'un monoïde dont l'opération est l'ajout d'un élément dans la collection ; il s'agit des séquences, des ensembles et des multi-ensembles ; et
- les collections fondées sur la triangulation de Delaunay.

La concaténation de deux collections leibniziennes rassemble les éléments des collections et recalcule le voisinage relatif entre les données.

Les collections *newtoniennes* s'opposent aux leibniziennes par la définition *a priori* de leur structure. L'espace engendré est alors décoré par des données. Dans ces collections, les positions existent indépendamment de leur décoration. On associe la valeur spéciale `<undef>` aux positions non décorées. La collection vide contient donc toutes les positions de l'espace décorées par la valeur `<undef>`. Nous présenterons parmi ces collections :

- les *enregistrements*, correspondant aux tables d'association ou dictionnaires,
- les *GBF*, extension de la notion de tableau,
- les *graphes*,
- les *chaînes abstraites*,
- les collections fondées sur la notion de *G-carte* .

Pour ces collections, la concaténation n'est possible que pour deux topologies partageant le même espace de positions ; les deux collections sont alors superposées, une fonction de collision gère alors les positions décorées dans les deux collections (la concaténation peut par exemple être asymétrique, donnant la priorité sur les éléments d'un des opérandes).

2.3.2 Les enregistrements

Il s'agit d'un dictionnaire associant une valeur MGS à une clé également appelée *champ* ; bien que les clés puissent être des valeurs MGS quelconques, l'utilisation des symboles est privilégiée. Dans les éléments de syntaxe qui suivent, nous utilisons des identifiants pour nommer les champs (ils sont automatiquement convertis en symboles par l'interprète). Pour associer une valeur de n'importe quel type il est nécessaire de passer par une *définition par extension*, une syntaxe commune à tous les types de collections présentée plus loin dans ce chapitre (voir page 44).

La structure d'enregistrement est celle d'une table d'association lorsqu'elle est utilisée avec des clés de valeur quelconque. Une utilisation restreinte aux clés de type `symbol` rapproche les enregistrements MGS des enregistrements Pascal, des `struct` en C ou encore des `record` OCaml. Voici la syntaxe réservée à l'utilisation de clés de type `symbol` :

- *Construction* : un enregistrement se construit à l'aide des accolades :

  ```
  { a = 1, b = "rouge" }
  ```

  est un expression qui construit un enregistrement avec deux champs identifiés par les symboles `'a` et `'b` dont les valeurs associées respectives sont l'entier `1` et la chaîne de caractères `"rouge"`.

- *Accès* : il se fait grâce à la notation pointée :

  ```
  { a = (1+2), b = "rouge" }.a
  ```

  retourne la valeur du champ `'a`, c'est-à-dire `3`. L'accès à un champ inexistant renvoie la valeur `<undef>`.

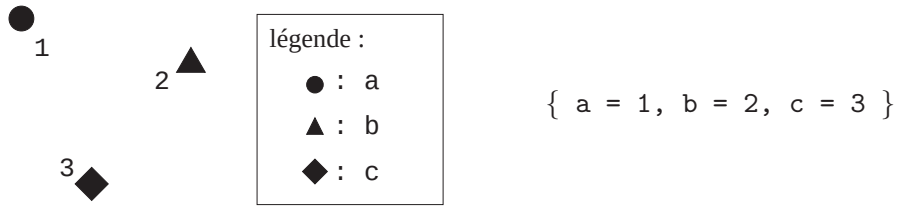


FIG. 3 – Exemple d'enregistrement : la topologie d'un enregistrement est un graphe sans arc où chaque sommet représente une clé et les décorations, les valeurs associées.

- *Concaténation* : une surcharge de l'opérateur `+` permet de fusionner deux enregistrements pour en calculer un nouveau. Cette concaténation est *asymétrique* avec priorité à droite :

  ```
  { a = 1, b = "rouge" } + { a = 3, c = true } ;;
  ```

  retourne l'enregistrement `{ a = 3, b = "rouge", c = true }`

Relation d'incidence. Les enregistrements sont une sorte de collection topologique très particulière où aucun élément n'a de voisin. Il s'agit en effet d'une structure sans organisation : la topologie correspondante est un complexe cellulaire de dimension 0, c'est-à-dire un graphe de voisinage *sans arc*. Chaque sommet représente une clé et la valeur associée à chaque clé décore le sommet en question. La figure 3 schématise une telle topologie. L'espace des positions est toujours défini comme l'ensemble des clés possibles : il s'agit d'une collection newtonienne.

Typage. Le type des enregistrements est `rec` (voir figure 2). Il s'agit d'un type concret : `rec:()` dénote la table d'association vide (`<undef>` est associé à chaque clé).

Le type `rec` peut être spécialisé afin de spécifier de nouvelles propriétés sur les champs et la valeur qui leur est associée. La déclaration d'un sous-type de `rec` (venant se placer directement sous le nœud `rec` de la hiérarchie 2) est possible à l'aide du mot clé `record` ; les exemples suivants présentent les différentes spécifications disponibles :

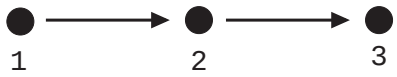
```
record R = { a } ;;
record S = { b, ~c } + R ;;
record T = S + { a = 1, d:string } ;;
```

La première expression spécifie un nouveau type d'enregistrement de nom `R`. Les enregistrements de type `R` doivent contenir *au moins* un champ identifié par le symbole `'a`. La seconde déclaration définit le type d'enregistrement `S` dont les valeurs contiennent au moins les deux champs `'a`, `'b` mais ne doivent pas associer de valeur au champ `'c`. La dernière ligne spécifie le type `T` des enregistrements également de type `S` dont la valeur associée au champ `'a` est l'entier `1`, et qui contient un champ supplémentaire `'d` dont la valeur associée est une chaîne de caractères.

2.3.3 Les séquences

Il s'agit d'une collection monoïdale. À la manière des types de données algébriques, à partir de la séquence vide `seq:()` et de l'opérateur d'ajout `::`, les séquences sont construites par ajouts successifs des données. Cet opérateur d'ajout est *associatif à droite* ; les parenthèses sont donc inutiles :

```
1 :: (2 :: (3 :: seq:())) ;;
```




```
(1, 2, 3)
```

FIG. 4 – Exemple de séquence : la topologie d'une séquence est un graphe linéaire où chaque sommet représente un élément et chaque arc la contiguïté.

construit une séquence de 3 éléments. On peut réécrire cette expression de la façon suivante :

```
1::2::3::seq:() ;;
1, 2, 3, seq:() ;;
1, 2, 3 ;;
```

La virgule MGS construit par défaut une séquence.

Relation d'incidence. Dans une séquence, chaque élément possède un voisin à droite : il s'agit de la tête de la queue sur laquelle il a été ajouté. La topologie induite par ce voisinage est celle d'un complexe cellulaire de dimension 1 linéaire où les sommets sont décorés par les éléments de la séquence et où un arc, orienté d'un sommet à un autre exprime la contiguïté des éléments dans la séquence. La figure 4 schématise une telle topologie. L'espace des positions est construit au fur et à mesure de l'ajout des éléments : il s'agit d'une collection leibnizienne.

Typage. Le type des séquences est `seq` (voir figure 2). Il s'agit d'un type concret, `seq:()` dénote la séquence vide. Dans la suite du manuscrit, les séquences sont également appelées des *listes*.

2.3.4 Les multi-ensembles

Les *multi-ensembles* sont des ensembles dans lesquels plusieurs occurrences d'un même élément peuvent coexister. Comme les séquences, il s'agit d'une collection monoïdale. L'opérateur d'ajout est *associatif* et *commutatif* ; l'ordre dans lequel les éléments sont ajoutés n'a pas d'importance. Les expressions

```
1, 1, 2, 3, 3, 3, bag:() ;;
2, 3, 3, 1, 3, 2, bag:() ;;
```

calculent le même multi-ensemble contenant deux occurrences de l'entier `1`, une de `2` et trois de `3`. La virgule est surchargée pour spécifier l'ajout d'un élément dans un multi-ensemble.

Relation d'incidence. La topologie associée à cette construction est un graphe complet où chaque élément est voisin de tous les autres. Il s'agit d'un complexe cellulaire de dimension 1. La figure 5 schématise une telle topologie. L'espace des positions est construit au fur et à mesure de l'ajout des éléments : il s'agit d'une collection leibnizienne.

Typage. Le type des multi-ensembles est `bag` (voir figure 2). Il s'agit d'un type concret, `bag:()` dénote le multi-ensemble vide.

2.3.5 Les ensembles

Contrairement aux multi-ensembles, il ne peut y avoir plus d'une occurrence d'un même élément. L'opérateur d'ajout est donc *associatif*, *commutatif* et *idempotent*. L'expression

```
1, 1, 2, 3, 3, 3, set:() ;;
```

évalue un ensemble contenant 3 éléments : `1`, `2` et `3`. La virgule est surchargée pour spécifier l'ajout d'un élément dans un ensemble.

Relation d'incidence. La topologie associée à cette construction est un graphe complet où chaque élément est voisin de tous les autres et n'apparaît qu'une seule fois. Il s'agit d'un complexe cellulaire de dimension 1. La figure 6 schématise une telle topologie. L'espace des positions est construit au fur et à mesure de l'ajout des éléments : il s'agit d'une collection leibnizienne.

Typage. Le type des ensembles est `set` (voir figure 2). Il s'agit d'un type concret, `set:()` dénote l'ensemble vide.

2.3.6 Les collections Delaunay

La triangulation de Delaunay et le diagramme de Voronoï [Aur91, OBSC00, BK03], que nous allons décrire ci-dessous, définissent une technique qui permet de construire le maillage d'un sous-ensemble d'un espace métrique de dimension finie à partir d'un nombre suffisant de points de ce sous-ensemble.

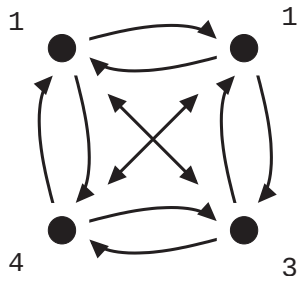
On désigne par P un ensemble composé de N points p_i de l'espace $\mathbb{R}^n$ (appelés aussi *sites* ou *germes*) : $P = \{p_i \in \mathbb{R}^n, i = 1, \ldots, N\}$.

On définit la *région de Voronoï* associée à un site p_i comme l'ensemble de points plus proches de p_i que de n'importe quel autre élément de P. Soit $Vor(p_i)$ la région de Voronoï associée à $p_i \in P$; on écrit :

$$Vor_p(p_i) = \{p \in \mathbb{R}^n, \forall p_j \in P,\ p_j \neq p_i,\ d(p, p_i) \leq d(p, p_j)\}$$

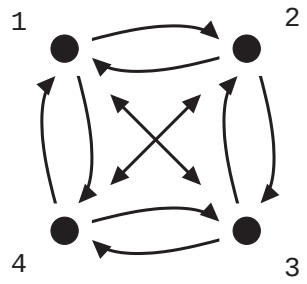
où $d(a, b)$ dénote une distance entre a et b. En travaillant dans $\mathbb{R}^2$ et en utilisant la distance euclidienne, ces régions sont des polygones dits de Voronoï (voir figure 7). On décrit le diagramme de Voronoï comme l'union des régions de Voronoï de tous les points de P.

$$Vor(P) = \bigcup_{p \in P} Vor_p(P)$$




```
(1, 1, 3, 4, bag:())
```

FIG. 5 – Exemple de multi-ensemble : la topologie d'un multi-ensemble est un graphe complet où chaque sommet représente un élément.




```
(1, 2, 3, 4, set:())
```

FIG. 6 – Exemple d'ensemble : la topologie d'un ensemble est un graphe complet où chaque sommet représente un élément.

À partir du diagramme de Voronoï, on construit son dual (voir figure 8) : c'est-à-dire un nouveau diagramme, dit de Delaunay, où on relie par un segment toutes les paires de sites dont les régions de Voronoï correspondantes sont adjacentes (séparées par une arête du diagramme de Voronoï).

Relation d'incidence. Ce diagramme fournit un voisinage entre les éléments de P à l'origine de la définition d'une nouvelle collection topologique MGS. La topologie est donnée par un complexe cellulaire de dimension 1 où les sommets sont les éléments de P. Les arcs correspondent aux segments du diagramme de Delaunay. La figure 9 schématise une telle topologie. L'espace des positions est construit au fur et à mesure de l'ajout des éléments : il s'agit d'une collection leibnizienne.

Typage. Le type des collections Delaunay est `del` en référence à Delaunay. Contrairement à ceux décrits précédemment, ce type de collection est *abstrait*. En effet, le calcul du voisinage requiert trois paramètres :

1. la dimension de l'espace euclidien dans lequel on se place,
2. un booléen précisant s'il faut calculer les caractéristiques du diagramme de Voronoï (par exemple en dimension 2, l'aire des polygones et la longueur de leurs arcs),
3. une « méthode » permettant d'extraire à partir des éléments leur position dans cet espace.

Le type des collections Delaunay possède donc un constructeur de type concret dont le mot clé est `delaunay` :

```
delaunay(2, true) E2 = \e.(e.px, e.py) ;;
```

Cette construction définit un nouveau type de collection Delaunay, nommé `E2`, dont la dimension est 2 (les points sont donc des éléments de $\mathbb{R}^2$) et pour lequel le calcul des caractéristiques du

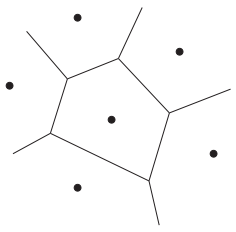


FIG. 7 – Polygone de Voronoï d'un point p_i.

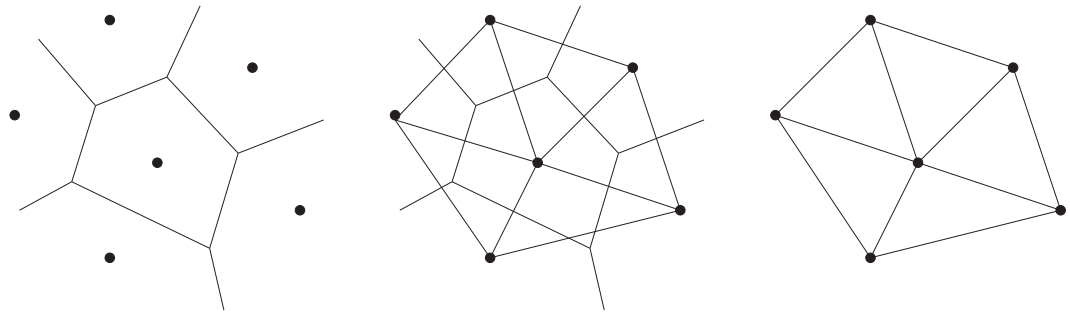


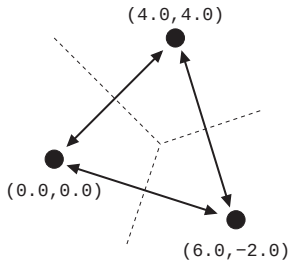
FIG. 8 – Construction du diagramme de Delaunay (à droite), dual du diagramme de Voronoï (à gauche).

diagramme de Voronoï est demandé. La fonction permettant d'extraire la position des points est donnée par la λ-abstraction `\e.(e.px, e.py)` qui retourne la liste des valeurs des champs `px` et `py` de son argument (celui-ci doit donc être un enregistrement contenant au moins ces deux champs). La collection vide correspondante est notée `E2:()` et une fonction `E2ify` est automatiquement générée. L'expression suivante génère un exemple de collection Delaunay de type `E2` :

```
   { px = 1.0 , py = 2.0 }
:: { px = 2.0 , py = 2.0 }
:: { px = 4.0 , py =-5.0 }
:: { px = 0.0 , py = 0.0 }
:: E2:() ;;
```

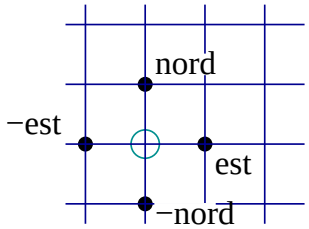
2.3.7 Les collections GBF

L'acronyme GBF signifie *Group Based Field*, ce qui peut être traduit par *champ de données fondé sur une structure de groupe* (voir [Mic96, GMS96]). Comme son nom l'indique, une collection GBF exhibe une structure construite sur les éléments d'un groupe. On rencontre ce type d'organisation lorsqu'on souhaite manipuler des voisinages réguliers. Par exemple, pour discrétiser un plan par un maillage carré, on utilise le voisinage suivant :




```
delaunay(2, true) D = id ;;
(0.0,0.0)::(4.0,4.0)::(6.0,-2.0)::D:()
```

FIG. 9 – Exemple de collection Delaunay : la topologie d'une collection Delaunay est un graphe où chaque sommet représente un élément, et chaque arc est un segment du diagramme de Delaunay.



Chaque élément possède un voisin suivant chaque direction *nord*, *sud*, *est* et *ouest*. Pour générer ce maillage, on construit une structure de groupe dont les générateurs sont deux des quatre directions (les deux autres correspondant à leurs inverses).

De façon plus formelle, soit $\mathcal{D} = \{a, b, c, \ldots\}$ l'ensemble des directions permettant de se déplacer d'un élément du maillage vers ses voisins ; on note $\mathcal{V}oisin(d, p)$ le voisin suivant la direction d d'un point p. Cette direction d peut être identifiée à l'opération de déplacement élémentaire donnée par la fonction $p \mapsto \mathcal{V}oisin(d, p)$ où $d \in \mathcal{D}$. Ces déplacements peuvent être composés par une opération + induisant une structure de groupe :

- elle est associative,
- pour chaque direction $d \in \mathcal{D}$, il existe un déplacement inverse noté $-d \in \mathcal{D}$,
- et il existe un déplacement nul qui consiste à « ne pas se déplacer ».

L'application des déplacements en un point est l'action du groupe sur l'espace des points. Pour définir un espace homogène arbitraire, il faut donc spécifier deux choses :

1. le groupe des déplacements à partir des déplacements élémentaires $\mathcal{D}$;
2. l'ensemble des points sur lequel le groupe va agir.

Dans l'interprète MGS, le second point est traité en considérant que le groupe des déplacements agit sur lui-même : un élément du groupe correspond alors à un point de l'espace homogène et l'action du groupe correspond simplement à la loi du groupe : $\mathcal{V}oisin(d, p) = p + d$.

Pour spécifier un GBF, on utilise une *présentation* [Mic96] : il s'agit d'une liste de générateurs et d'une liste d'équations entre ces générateurs. La syntaxe d'une présentation est la suivante :

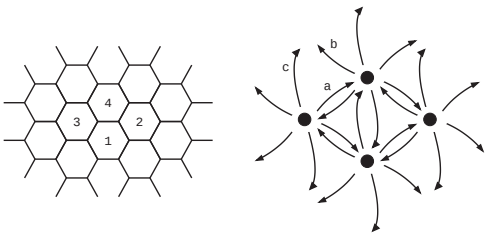
$$\langle g_1, g_2, \ldots, g_n ; e_1, e_2, \ldots, e_m \rangle$$

où $g_1, g_2, \ldots, g_n$ sont les générateurs et $e_1, e_2, \ldots, e_m$ les équations entre générateurs. Tout élément du groupe s'obtient alors comme une somme de générateurs. Ici, ils correspondent aux déplacements élémentaires qui définissent le voisinage homogène.

Dans notre exemple de grille, la présentation du GBF est :

$$\langle nord, est ; nord + est = est + nord \rangle$$

Relation d'incidence. À partir de la présentation d'un groupe, le graphe de Cayley fournit la topologie engendrée par le groupe : les nœuds sont les éléments du groupe et deux éléments sont liés par un arc si leurs positions ne diffèrent que de l'ajout ou du retrait d'un générateur. Il s'agit donc d'un complexe cellulaire de dimension 1. La figure 10 schématise une telle topologie. L'espace des positions est toujours déterminé par la présentation du groupe des déplacements indépendante des valeurs : il s'agit d'une collection newtonienne.




```
gbf hexa = < a, b, c ; a + b = c > ;;
(1,2)::(3,4)::seq:() following |a>, |b> ;;
```

FIG. 10 – Exemple de collection GBF : la topologie d'une collection GBF est donnée par le graphe de Cayley associé à la présentation du groupe des déplacements. Les deux graphes sont deux représentations duales du graphe de Cayley de la présentation `hexa`.

Typage. Le type résultant en MGS est un type abstrait appelé `gbf`. Le mot clé du même nom est alors utilisé pour spécifier des groupes particuliers et ainsi générer de nouveaux voisinages uniformes. Les groupes gérés par l'interprète MGS sont pour l'instant *abéliens*[4], les déplacements élémentaires sont commutatifs entre eux. La topologie de la grille carrée définie précédemment est construite en MGS de la façon suivante :

```
gbf grille = < nord, est > ;;
```

crée un nouveau type de GBF nommé `grille` à partir des deux générateurs `nord` et `est`. Ces derniers sont à l'origine de nouvelles valeurs MGS de type `posgbf`. Une *arithmétique* des déplacements est en effet mise en place pour manipuler les positions des GBF de type `grille` ; elle est fondée sur deux nouvelles valeurs `|nord>` et `|est>`, et les opérateurs `+`, `-` et `*`. Par exemple

```
|nord> + |est> == |est> - |nord> + 2*|nord> ;;
```

retourne la valeur booléenne `true`. Ces déplacements expriment également les coordonnées des points, partant d'une origine arbitrairement choisie et permettent d'identifier les positions des éléments du GBF (ce qui justifie le nom `posgbf` donné à ce type). La valeur `grille:()` correspond alors à la grille générée par le groupe associé aux générateurs `nord` et `est`, où toutes les positions ont une valeur indéfinie. La collection étant vide, la valeur `<undef>` est associée à chaque point du GBF. Afin de spécifier les décorations, MGS fournit l'opérateur `following` :

```
   (‘a, ‘a, ‘a)
:: (‘b, ‘b)
:: (‘c, ‘c, ‘c, ‘c)
:: seq:() following |nord>, |est> ;;
```

complète le GBF `grille:()` à partir de l'origine de l'espace engendré comme le montre la figure 11.

2.3.8 Les graphes quelconques

Les graphes quelconques correspondent au concept de graphe de voisinage. En effet, il s'agit de graphes orientés dont seuls les sommets sont étiquetés. Une implantation plus récente fondée sur la bibliothèque OCamlGraph a été développé dans [DR04] pour laquelle les arcs peuvent également être décorés. Cette mise à jour est à l'étude ; nous ne considérons donc dans ce document que l'ancienne version.

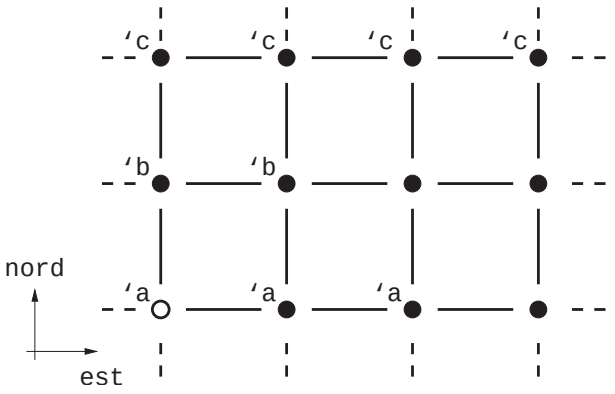
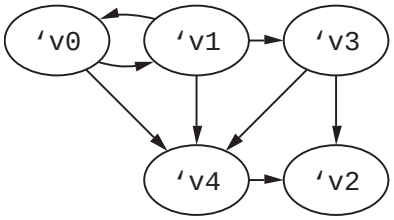


FIG. 11 – Décoration dans un GBF à l'aide de l'opérateur `following` (voir le programme décrit dans le texte).




```
g := [ 0 := ‘v0 :> 1, 4;
       1 := ‘v1 :> 0, 3, 4;
       2 := ‘v2 ;
       3 := ‘v3 :> 2, 4;
       4 := ‘v4 :> 2
     ] ;;
```

FIG. 12 – Exemple de graphe.

Relation d'incidence. Le voisinage des graphes repose directement sur leur topologie : un graphe *est* un complexe cellulaire de dimension 1. La figure 12 schématise une telle topologie. L'espace des positions est déterminé à la définition du graphe et ne peut être modifié : il s'agit d'une collection newtonienne.

Typage. Le type MGS pour ces graphes est `graph`. Il s'agit d'un type concret dont la structure est spécifiée par une construction syntaxique dédiée, comme le montre l'exemple de la figure 12.

2.3.9 Les chaînes abstraites

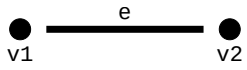
Cette collection est la plus générale en terme de voisinage arbitraire que l'interprète MGS fournit. Elle correspond à une implantation des complexes cellulaires qui ont été décrits précédemment.

Relation d'incidence. Contrairement aux collections présentées jusqu'ici, les chaînes abstraites ne sont pas limitées à un graphe de voisinage où les sommets sont les éléments et les arcs la relation de voisinage. Elles correspondent à la construction de complexes cellulaires de dimension arbitraire. La figure 13 schématise une telle topologie. L'espace des positions est défini seul dans un premier temps. Il est ensuite décoré ce qui amène à considérer les chaînes abstraites comme un type de collections newtoniennes.

[4] Une extension vers les groupes automatiques a été envisagée dans [Del02].

Typage. Un type de données scalaire a été créé pour une nouvelle sorte de valeur appelée `acell` (pour *cellule abstraite*). Une `acell` est une cellule topologique ; c'est à travers ces objets que les complexes cellulaires vont être représentés : un complexe cellulaire n'est pas un objet élémentaire manipulable explicitement ; en revanche, il peut être parcouru à travers ses éléments constituants, les cellules topologiques représentées par les valeurs de type `acell`, en utilisant la relation d'incidence qui les relie. Des cellules topologiques fraîches peuvent être créées à l'aide de la fonction `new_acell`. Celle-ci demande en paramètre la dimension de la cellule créée ainsi que la liste de ses cofaces et de ses faces. Par exemple, pour créer un arc avec ses deux sommets, on écrit :

```
v1 := new_acell(0, seq:(), seq:()) ;;
v2 := new_acell(0, seq:(), seq:()) ;;
e  := new_acell(1, (v1, v2, seq:()), seq:()) ;;
```

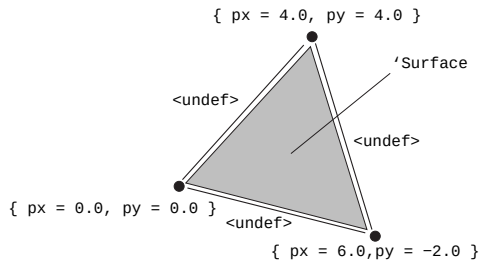


La cellule associée à la variable `e` est une cellule de dimension 1. Afin de conserver un caractère fonctionnel pur, la fonction `new_acell` fait une copie des faces et des cofaces données en paramètre. En effet, définir `v1` et `v2` comme faces de `e` signifie que `e` est une coface de `v1` et `v2`, ce qui va à l'encontre de la définition de ces deux mêmes cellules. De la même façon, si `v1` et `v2` étaient des éléments de complexes cellulaires différents, ceux-ci seraient copiés et unis en une nouvelle entité. Par souci de performance, une fonction équivalente à `new_acell` mais fonctionnant par effets de bord est également disponible ; il s'agit de `add_acell`. L'expression suivante

```
v1' := add_acell(0, seq:(), seq:()) ;;
v2' := add_acell(0, seq:(), seq:()) ;;
e'  := add_acell(1, (v1', v2', seq:()), seq:()) ;;
```

produit le même arc que précédemment, excepté que `v1'` et `v2'` sont dans ce cas effectivement les faces de `e'`. Les cellules topologiques associées à `v1'` et `v2'` ne sont pas copiées mais leurs définitions sont mises à jour. Sachant que la fonction `member` teste l'appartenance d'une valeur à une collection, les expressions

```
member(faces(e), v1) ;;
member(faces(e'), v1') ;;
```




```
v1 := add_acell(0, seq:(), seq:()) ;;
v2 := add_acell(0, seq:(), seq:()) ;;
v3 := add_acell(0, seq:(), seq:()) ;;
e1 := add_acell(1, (v1,v2), seq:()) ;;
e2 := add_acell(1, (v2,v3), seq:()) ;;
e3 := add_acell(1, (v3,v1), seq:()) ;;
f  := add_acell(2, (e1,e2,e3), seq:()) ;;
  { px = 0.0, py = 0.0 } * v1
+ { px = 4.0, py = 4.0 } * v2
+ { px = 6.0, py =-2.0 } * v3
+ 'Surface * f ;;
```

FIG. 13 – Exemple de chaîne abstraite : la topologie est définie de façon explicite par la création de nouvelles cellules ; les décorations sont ensuite placées sur les positions en utilisant les opérateurs de somme et de produit.

retournent respectivement `false` et `true`. Comme le montre l'exemple, des accesseurs sont fournis pour parcourir le complexe cellulaire :

- `faces(`c`)` retourne la liste des faces de c ;
- `cofaces(`c`)` retourne la liste des cofaces de c ;
- `icells(`c`)` retourne la liste des cellules incidentes à c ;
- `pcells(`c`,`p`)` retourne la liste des p-voisines de c.

Une fois créé, l'espace défini par les cellules topologiques abstraites peut être décoré par des valeurs du langage. On obtient alors le nouveau type de collection topologique qui nous intéresse, les `achain` pour *chaînes abstraites*. Les *chaînes topologiques* sont des objets issus de la topologie algébrique. Elles correspondent à des fonctions partielles associant aux cellules d'un complexe cellulaire une valeur. Elles sont souvent représentées par une somme formelle, style d'écriture utilisé également dans l'interprète. En reprenant l'exemple précédent, on écrit :

```
‘vertex * v1 + ‘edge * e ;;
```

pour construire la chaîne topologique associant la valeur symbolique `‘vertex` à la cellule associée à la variable `v1` et la valeur `‘edge` à `e`.

Afin de conserver l'intégrité des chaînes topologiques ainsi créées, il est nécessaire d'interdire la construction de toute nouvelle cellule topologique par la fonction `add_acell` ; l'utilisation de cette fonction est par conséquent prohibée sur les cellules utilisées dans la définition d'une chaîne topologique. Il reste néanmoins possible d'ajouter des cellules à un complexe grâce à la primitive `new_acell` qui crée une copie du support utilisé dans la définition de ce nouvel objet. Les opérations topologiques comme l'insertion, la suppression ou le raffinement des cellules topologiques, opérations classiques dans certains modeleurs topologiques, ne sont pas fournies par l'interprète MGS ; l'objectif de celui-ci n'est en effet que de montrer l'utilité des transformations que nous aborderons dans la section suivante, pour programmer ces opérations.

Nous verrons plus précisément l'utilisation des chaînes abstraites, d'une part théoriquement puisqu'elles sont à l'origine du formalisme mis en place dans cette thèse (voir le chapitre 3), mais également concrètement dans les différents exemples de modélisation que nous présentons chapitre 11.

2.3.10 Les G-cartes

La structure de chaîne abstraite vue dans la sous-section précédente est puissante en terme de représentation (voir le chapitre 3 pour une élaboration sur les complexes cellulaires) mais reste coûteuse pour les parcours de cellules. Les G-cartes constituent le second type de collection topologique permettant la représentation d'objets de dimension arbitraire ; elle est plus spécifique[5] que les chaînes abstraites et est dédiée aux modèles topologiques par *représentation de bord* (en anglais, *boundary representation*, voir [Lie91, Sha01]). L'implantation proposée dans l'interprète MGS repose sur une bibliothèque C++ développée au sein du projet MOKA[6], une plate-forme interactive de CAO fondée sur les G-cartes de dimension 3. Contrairement aux complexes abstraits, les *cartes généralisées*, ou G-cartes, sont une structure de données qui permet de représenter une classe particulière d'espaces topologiques : les *quasi-variétés* [Lie94]. La motivation de cette seconde implantation des complexes cellulaires tient au fait que cette structure de données présente

[5] De façon plus précise, les G-cartes ne permettent que de représenter des quasi-variétés. Cependant, à l'instar des modèles par graphes d'incidence, elles permettent de distinguer dans cette classe, certains espaces topologiques confondus dans une représentation par graphe d'incidence ; on pense en particulier à ceux faisant intervenir de la multi-incidence.

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

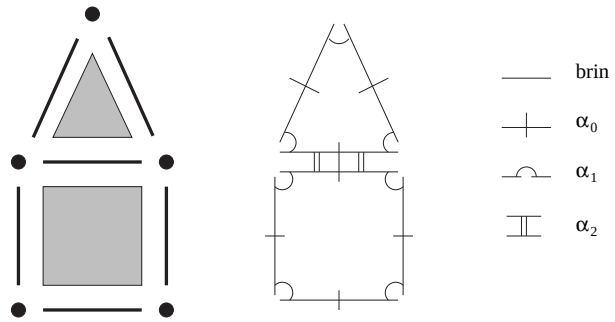


FIG. 14 – Description d'un complexe cellulaire en terme de brins et d'involutions.

un certain nombre de contraintes d'intégrité à respecter. De plus, le complexe cellulaire encodé par une G-carte n'est pas représenté de façon explicite mais doit être traité comme tel selon le point de vue que nous adoptons. L'intégration de cette structure dans MGS illustre donc parfaitement la généricité des transformations dont la spécification est indépendante du type de collection topologique (chaînes abstraites et G-cartes peuvent être utilisées indépendamment dans les exemples présentés dans la partie III).

Voici une brève présentation de la structure de G-carte [Lév99, LM99]. Soit $\mathcal{K}$ un complexe cellulaire. Soit N, la plus grande dimension des cellules du complexe, N est appelé la dimension de $\mathcal{K}$. On considère le *graphe d'incidence* $\mathcal{G}$ de $\mathcal{K}$, c'est-à-dire le graphe induit par la relation d'incidence entre les cellules de $\mathcal{K}$. 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 (c_i est donc une face de c_{i+1}). 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 $\mathcal{K}$ 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 $\mathcal{K}$. 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 $b' = \alpha_i(b)$ si les deux tuples représentés par b et b' sont i-adjacents (voir figure 14). Pour que les objets représentés par des G-cartes respectent les propriétés topologiques des *quasi-variétés*, les α_i doivent être des involutions et respecter les propriétés suivantes :

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

Relation d'incidence. La topologie des G-cartes correspond au complexe cellulaire défini par l'ensemble de brins et les involutions. La figure 15 schématise une telle topologie. L'espace des positions est défini seul dans un premier temps. Il est ensuite décoré ce qui amène à considérer les G-cartes comme un type de collection newtonien.

Typage. Les collections topologiques reposant sur les G-cartes sont de type `qmf`. Il s'agit d'un type abstrait, ces collections étant paramétrées par une G-carte. Il nous faut donc dans un premier temps fournir les primitives pour la création des espaces de positions.

Nous avons souhaité cacher au programmeur MGS l'aspect « brins et involutions » des G-cartes qui ne s'intègre pas dans la notion de complexe cellulaire telle qu'elle a été présentée précédemment. Nous n'avons donc rendu accessibles dans le langage que des primitives manipulant les cellules topologiques masquant ainsi les mécanismes internes concernant les brins. Deux

nouveaux types scalaires ont été apportés à l'interprète. D'une part, les `gmaps` sont les objets dans lesquels les G-cartes sont définies. On crée une G-carte vide par

```
g := new_gmap() ;;
```

Les cellules topologiques propres aux G-cartes ont pour type `ncell`. La construction des cellules comme celles présentées ci-dessus pour les complexes cellulaires abstraits n'a pas été développée. En effet, nous avons trouvé plus intéressant d'utiliser directement l'outil MOKA pour construire nos objets, et de les charger dans l'interprète à l'aide d'une primitive MGS et du format d'import/export de données de MOKA. Si le fichier `my_object.moka` contient le complexe cellulaire intéressant, il suffit d'écrire :

```
load_moka("my_object.moka", g) ;;
```

pour remplir la G-carte vide que nous venons de créer. L'évaluateur retourne alors la séquence des cellules topologiques de type `ncell` de dimension 3 (la plus grande dimension d'un objet construit avec le noyau de MOKA) à partir de laquelle l'ensemble des cellules, toutes dimensions confondues, peuvent être retrouvées. L'expression `save_moka("my_object.moka", g)` qui effectue l'opération inverse de sauvegarde du contenu de `g`, est également fournie par l'interprète. Un grand nombre de primitives directement intégrées de la bibliothèque de MOKA permettent de créer des objets simples : `add_polygon`, `add_sphere`, `add_square`, `add_polyline`, etc.

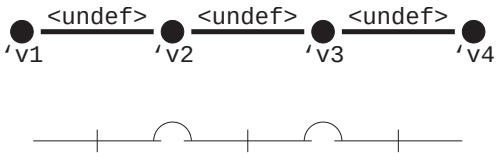
Une collection topologique reposant sur les G-cartes est donc paramétrée par un objet de type `gmap`. On peut alors créer un nouveau type de collection à partir d'une G-carte de la façon suivante :

```
qmf Q on g ;;
```

Le mot clé `qmf` correspond au type abstrait des collections topologiques fondées sur les G-cartes. Cette abréviation fait référence à *Quasi-ManiFold*, terme anglais pour quasi-variété. Ici, un nouveau type de `qmf`, nommé `Q`, est défini sur la G-carte `g`. Après cette définition, il n'est plus possible de modifier le contenu de `g`. Il n'existe pas de syntaxe propre aux quasi-variétés pour associer des valeurs aux `ncells` pour définir de nouvelle collection. On passe par une définition en extension, valide pour tout type de collections topologiques et qui sera décrite par la suite (voir page 44).

2.3.11 Imbrication de collections : les contraintes

Les types de données MGS fournissent des prédicats pour vérifier le type des valeurs manipulées. Les primitives et opérateurs MGS étant extrêmement surchargés et en l'absence de typage




```
g := new_gmap() ;;
l := add_polyline(g,4) ;;

qmf Q on g ;;
Q:{| 'v1@l.(0), 'v2@l.(1),
     'v3@l.(2), 'v4@l.(3) |} ;;
```

FIG. 15 – Exemple d'une collection `qmf` : les deux schémas décrivent d'une part l'organisation de la topologie en terme de brins et d'involution (la légende est la même que celle de la figure 14), et d'autre part le complexe cellulaire correspondant. Les valeurs sont associées aux cellules topologiques dans un second temps.

statique, ces prédicats s'avèrent être une bonne solution pour une vérification fine des types au cours de l'exécution d'un programme. Cependant, les prédicats définis automatiquement avec les collections topologiques ne considèrent que le type du *contenant* ; par exemple, le prédicat `seq` des séquences vérifie que l'argument est bien une séquence, mais ne permet pas de restreindre le type des éléments des séquences. Les *contraintes* MGS sont un moyen de définir des prédicats plus expressifs imposant le type du *contenu* et le nombre d'éléments d'une structure de données. Un exemple de contrainte de type a déjà été présenté : la spécification de types d'enregistrement permet de gérer l'absence ou la présence, ainsi que le type des champs que les enregistrements doivent contenir.

Grâce à un dispositif de déclaration de types mutuellement récursifs, les contraintes présentées ci-dessous permettent notamment de définir des structures de données imbriquées, c'est-à-dire des collections topologiques de collections. Ceci fournit une alternative aux types de données algébriques pour décrire des structures arborescentes. On peut notamment imaginer leur utilisation pour la spécification de fichiers XML [BPSM+06] à la façon du langage XMLSchema [FW04] ou des DTD (Document Type Definition).

Langage de déclaration de contraintes. Les contraintes sont définies en utilisant le mot clé `constraint` :

`constraint` *uid* `=` *cstr* `;;`

où *uid* dénote le nom de la contrainte à définir. La partie droite *cstr* de cette déclaration suit la grammaire suivante :

$$
\begin{array}{rcl}
cstr & ::= & symb \\
 & | & typeColl \\
 & | & \texttt{[}\ cstr\ \texttt{]}typeColl \\
 & | & typeColl\texttt{(}\ int\ \texttt{)} \\
 & | & \texttt{[}\ cstr\ \texttt{]}typeColl\texttt{(}\ int\ \texttt{)} \\
 & | & \texttt{\~{}}cstr \\
 & | & cstr\ \texttt{\&\&}\ cstr \\
 & | & cstr\ \texttt{||}\ cstr \\
 & | & \texttt{(}\ cstr\ \texttt{)}
\end{array}
$$

La syntaxe des contraintes est définie de façon inductive. Les cinq premières constructions correspondent respectivement à une constante symbolique, à un type de collection (*typeColl* peut s'instancier par exemple en `seq`, `set`, `bag`, etc), à un type de collection avec contraintes sur le type des éléments, à un type de collection avec contrainte sur le nombre d'éléments et à un type de collection avec à la fois contraintes sur le nombre et le type des éléments. Les constructions suivantes permettent de définir des négations, des conjonctions et des disjonctions de contraintes. Par exemple,

```
constraint intTriplet = [int]seq(3) ;;
```

définit une nouvelle contrainte `intTriplet` reconnaissant les séquences composées de 3 entiers. Dans ce second exemple,

```
    constraint option = `None | some
and record     some   = { Some:any } ;;
```

la contrainte `option` est définie comme soit la valeur du symbole `` `None ``, soit un enregistrement contenant un champ `Some` auquel est associé une valeur de n'importe quel type (voir le type `any` sur la hiérarchie de la figure 2).

Coloration et déclaration récursive de type. Les différentes constructions dédiées à la définition de type de collection s'insèrent dans un mécanisme d'héritage de type présenté figure 2. Par exemple, le type `grille` (défini page 35) hérite du type `gbf` et construit donc un nœud fils du nœud `gbf` dans cette hiérarchie. Cela signifie que si `g1` est de type `grille`, et `g2` est un GBF qui n'est pas de type `grille`, les expressions `gbf(g1)` et `gbf(g2)` s'évaluent en la valeur booléenne `true`, alors que `grille(g1)` et `grille(g2)` s'évaluent respectivement en `true` et `false`. Les constructions utilisant les mots clés `record`, `delaunay`, `qmf` et `constraint` construisent également des fils dans l'arbre de sous-typage. Un dernier mécanisme de création de nœud n'a pas encore été présenté ; il permet de « colorer » un type de collection, par exemple pour en spécialiser l'utilisation et bénéficier ainsi d'une forme primitive de sous-typage. Ainsi, l'expression

```
collection my_seq = seq ;;
```

crée une nouvelle *couleur* de séquence MGS. Les valeurs de type `my_seq` (construite à partir de la séquence vide `my_seq:()` elle-même créée à l'évaluation de l'expression précédente) sont également de type `seq`. À l'inverse, une séquence construite à partir de `seq:()` n'est pas de type `my_seq` :

```
seq((1, 2, 3, my_seq:())) ;;
my_seq((1, 2, 3, seq:())) ;;
```

retournent respectivement `true` et `false`. Partout où les séquences de type `seq` apparaissent, leur utilisation peut être restreinte à des séquences de type `my_seq`.

Un exemple. Afin d'illustrer les contraintes et la coloration de type, voici comment définir en MGS un prédicat pour la structure d'arbre binaire :

```
    collection node = seq
and constraint tree = 'Leaf | [ tree ]node(2) ;;
```

Un arbre est soit vide, dénoté par la constante `'Leaf`, soit une séquence de type `node` (une couleur du type `seq`), de longueur deux, contenant uniquement des arbres. Ceci est à comparer à la définition inductive suivante (syntaxe OCaml) :

```
type tree = Nil | Node of tree * tree ;;
```

La figure 16 schématise les représentations issues de ces deux différentes définitions.

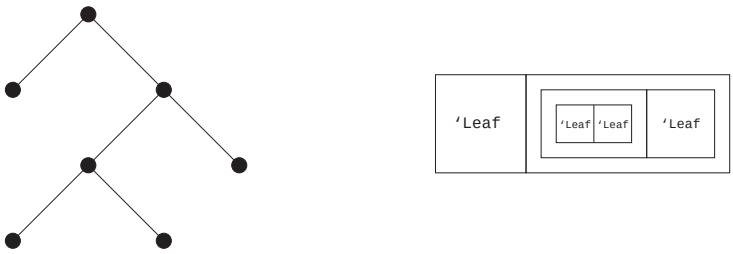


FIG. 16 – Représentation d'un arbre binaire : à gauche par un type de données algébrique, à droite par imbrication que collections topologiques.

2.4 Les fonctions sur les collections

Nous souhaitons utiliser le cadre unifié des collections topologiques afin de permettre une programmation indépendante du type de collection. Les transformations, présentées dans la section suivante, offrent ce point de vue générique. Néanmoins, un certain nombre de primitives *polytypiques* [Coh04] (c'est-à-dire s'appliquant sur n'importe quel type de collection) sont disponibles dans l'interprète. Les paragraphes suivants en donnent un bestiaire non-exhaustif. La documentation en ligne de l'interprète[7] détaille les fonctions additionnelles non décrites ici.

2.4.1 Les primitives MGS

Les fonctions `oneof` et `rest`. Les destructeurs `oneof` et `rest` [BNTW95] retournent respectivement un des éléments d'une collection et le reste de la collection, de telle sorte que si `c` est une collection, `oneof(c)` et `rest(c)` construisent une partition de `c`.

La fonction `take`. Il est possible de considérer les collections topologiques comme des fonctions partielles associant des valeurs du langage aux cellules topologiques d'un complexe cellulaire. Il est alors commode d'utiliser le terme de *position* pour désigner les cellules topologiques et d'*espace de positions* pour les complexes cellulaires. Cependant, les valeurs MGS correspondant à des collections ne sont pas implantées comme des fonctions, pour rappeler leur rôle de structure de données. L'accès à la décoration d'une position donnée est possible grâce à la fonction `take`. Ainsi,

```
take(c, p) ;;
```

retourne la valeur associée à la position `p` d'une collection topologique `c`. Une syntaxe infixe est également disponible :

```
c.(p) ;;
```

On peut noter que pour les enregistrements, l'expression `r.x` (pour obtenir la valeur associée à la clé `'x` dans `r`) s'écrit également `r.('x)` ; cette écriture est d'ailleurs plus générale puisqu'elle permet d'avoir accès aux champs dont la clé est une valeur autre qu'un symbole. Enfin, si la valeur de `p` ne correspond pas à une position définie dans `c`, la fonction `take` retourne la valeur `<undef>`.

Évidemment, le type des positions est propre à chaque type de collection. Pour les séquences, seuls les sommets du complexe cellulaire sont décorés ; ce complexe étant un graphe linéaire (par conséquent homomorphe à $\mathbb{N}$), les positions des éléments sont donc naturellement représentées par des entiers, leur index dans la séquence :

```
take((1,2,3,4), 2) ;;
```

retourne l'entier `3` (l'indexation commençant à 0 comme en C/C++). Pour les ensembles, le complexe cellulaire est un graphe complet ; la position d'un élément est donc simplement sa propre valeur. Pour les multi-ensembles, afin de distinguer les différentes occurrences d'un même élément, celles-ci sont numérotées par des entiers. La position dans un multi-ensemble est donc donnée par un couple (valeur, occurrence). Le type de position sera `posgbf` pour une collection GBF, `acell` pour les chaînes abstraites et `ncell` pour les collections de type `qmf`.

[7] Accessible à l'adresse `http://mgs.ibisc.univ-evry.fr/Online_Manual/`

La définition par extension. La définition par extension des collections est un moyen explicite de spécifier une collection par une liste de couples (position, valeur) :

`extension("`*typeColl*`", (`p_1`,`v_1`) ::... ::(`p_n`,`v_n`) :: seq:()) ;;`

Le premier argument est une chaîne de caractères donnant le type de la collection à définir, et le second est une séquence de couples (position, valeur) indiquant comment décorer l'espace associé au type de collection précisé. Par exemple, pour construire la séquence `(1, 2, 3)`, on écrit

```
extension("seq", (0,11) ::(1,22) ::(2,33) :: seq:()) ;;
```

L'interprète fournit également du sucre syntaxique pour une notation infixe plus commode :

```
seq:{| 11@0, 22@1, 33@2 |} ;;
```

La fonction `map`. Cet opérateur applique une fonction sur tous les éléments d'une collection et retourne la collection des résultats.

```
map(\x.(x+1), (3,4,5)) ;;
```

retourne la séquence `(4,5,6)`. La fonction passée en argument s'applique sur chaque élément de la séquence. Une variante de l'opérateur existe pour appliquer la fonction sur les éléments mais également sur leurs positions. Il s'agit de la fonction `map_indexed`. Dans ce cas, la fonction argument prend deux arguments, le premier représentant la position et le second la valeur de l'élément courant :

```
map_indexed(\p,x.(10 * p + x), (3,4,5)) ;;
```

s'évalue en la séquence `(3,14,25)`.

La fonction `fold`. Ce deuxième itérateur permet de parcourir l'ensemble des éléments décorant une collection topologique pour calculer une seule valeur. Soit c_1, ..., c_n les éléments d'une collection topologique c. L'expression

`fold(`f`, `*zero*`, `c`) ;;`

calcule $f(c_1, (\ldots, f(c_n, zero)\ldots))$. Si la collection ne contient pas d'élément, la valeur *zero* est directement retournée. L'ordre d'énumération des éléments n'est pas spécifié et peut varier pour des collections identiques en type et en taille. La fonction d'accumulation f prend deux arguments, l'élément courant et le résultat courant.

Par exemple, il est particulièrement simple d'exprimer la somme des éléments d'une collection :

```
fold(\e,acc.(e + acc), 0, (1,2,3)) ;;
```

retourne l'entier `6`. La variante `fold_indexed` est également disponible.

La fonction `iter`. Cet opérateur applique une fonction ayant un effet de bord sur chaque élément d'une collection. Le résultat évalué est alors `<undef>`.

```
a := 0 ;;
iter(\x.(a:=a+x), (1,2,3)) ;;
```

retourne `<undef>` mais la valeur associée à la variable `a` est mise à jour pour l'entier `6`. La variante `iter_indexed` est également disponible.

La fonction `forall`. Il s'agit du prédicat universel sur les éléments d'une collection. L'expression

`forall(`p`,`c`) ;;`

évalue le calcul $p(c_1) \wedge \cdots \wedge p(c_n)$ où les c_i sont les éléments de la collection c, et p est un prédicat.

La fonction `exists`. Il s'agit du prédicat existentiel sur les éléments d'une collection. L'expression

`exists(`p`,`c`) ;;`

évalue le calcul $p(c_1) \vee \cdots \vee p(c_n)$ où les c_i sont les éléments de la collection c, et p est un prédicat.

2.4.2 Le polytypisme proposé dans MGS

Dans le typage, le *polymorphisme* consiste à abstraire par une variable de type, le type d'une variable. Grâce à cette abstraction, la fonction `List.map` de OCaml peut s'appliquer aussi bien sur des listes d'entiers que des listes de flottants par exemple. On remarque cependant qu'il existe une fonction `map` pour chaque type de structure ; par exemple, pour appliquer une même fonction sur les éléments d'un tableau, OCaml fournit une seconde fonction `Array.map`, également polymorphe sur le type des éléments du tableau. Transposé à MGS, le polymorphisme permet l'abstraction sur le type du contenu d'une collection topologique.

Le *polytypisme* consiste à étendre le polymorphisme au *contenant*, c'est-à-dire au type de la collection topologique lui-même.

Dans [CW85], L. Cardelli et P. Wegner proposent une classification des différentes classes de polymorphisme. Cette classification peut être étendue pour le polytypisme. Il en existe deux grandes classes, elles-mêmes divisées en deux :

1. polymorphisme/polytypisme *universel* :
   - *paramétrique* : définition de fonctions pouvant s'appliquer sur plusieurs types différents. Par exemple, le `List.map` de OCaml correspond à du polymorphisme paramétrique (les types doivent partager une structure commune), et le `map` MGS à du polytypisme paramétrique (la structure peut être différente) ;
   - *par inclusion* : concept d'héritage ou de sous-typage. Par exemple, l'héritage des objets en C++ est du polymorphisme par inclusion, la notion de coloration MGS (avec le mot clé `collection`) correspond à du polytypisme par inclusion.
2. polymorphisme/polytypisme *ad-hoc* :
   - *par surcharge* : la surcharge permet l'utilisation d'un même opérateur sur des arguments de types différents. Par exemple, les opérateurs de comparaison `<` et `>` sont surchargés pour comparer n'importe quelles valeurs ; la primitive `oneof` est un exemple de polytypisme par surcharge ;
   - *par coercion* : cette dernière classe correspond aux opérateurs qui modifient le type de ses arguments afin d'effectuer un calcul. L'addition en C/C++ transforme les entiers en flottants pour les additionner avec d'autres flottants. La concaténation de deux collections topologiques MGS fait de même avec ses arguments : soient `c1` et `c2` deux collections de types différents ; la concaténation retourne une collection dont le type est le sur-type commun le plus bas dans la hiérarchie des types (voir figure 2) si celui-ci n'est pas abstrait.

Pour plus de détails sur le polytypisme et les systèmes de type du langage MGS, le lecteur est invité à parcourir [Coh04].

3 Les transformations

Les collections topologiques permettent de représenter un grand nombre de structures de données. Pour les manipuler, MGS fournit une structure de contrôle, appelée *transformation*, permettant la définition par cas de fonctions sur les collections topologiques. La généricité est capturée par l'utilisation de la relation de voisinage locale fournie avec chaque collection. Cette relation est le point de départ pour l'élaboration de motifs filtrant des sous-parties d'une collection. Chaque cas de la transformation est alors une règle de réécriture ; son application correspond à une modification locale de la collection.

3.1 Fonctions définies par cas

Dans les langages de type ML (comme Haskell ou OCaml), il est possible de définir des fonctions par cas sur des constantes. Ce système de définition se rapproche d'une structure d'aiguillage en C ou en MGS (avec le mot clé `switch`). La fonction *factorielle* s'écrit ainsi dans un pseudo-langage fonctionnel :

```
let rec fact = function
  | 0 -> 1
  | x -> x*f(x-1)
```

Cette fonction est spécifiée par deux cas suivant la valeur de son argument. Le premier cas filtre la constante `0` ; ainsi, si l'argument de la fonction est `0`, l'évaluation de l'expression `1` sera déclenchée. La seconde partie de cette définition absorbe un argument de n'importe quelle valeur. La variable `x` joue un rôle de *joker*, permettant la référence à la valeur de l'argument dans l'expression en partie droite.

Dans les langages Haskell et OCaml, la spécification des motifs ne se limite pas au filtrage de constantes (l'entier `0` dans notre exemple). La définition par cas des fonctions permet notamment l'expression de motifs filtrant des objets plus complexes que de simples valeurs scalaires, et construits à partir d'un *type de données algébrique*. Ces objets permettent la représentation de structures définies inductivement. C'est le cas des arbres binaires :

```
type Ab = Leaf of int
        | Node of Ab * Ab ;;
```

Les arbres binaires sont soit des feuilles (`Leaf`), soit des nœuds (`Node`). `Leaf` et `Node` sont les *constructeurs* du type `Ab`. Ces constructeurs sont paramétrés par des attributs permettant leur décoration par des valeurs du langage. Ici, on associe un entier à chaque feuille et un couple d'arbres binaires à chaque nœud (représentant les fils droit et gauche). On utilise alors les constructeurs pour définir de nouveaux motifs expressifs spécifiant la structure des éléments à filtrer et des variables de motif pour référencer les attributs qui leur sont associés. Par exemple, on définit par cas la somme des valeurs associées aux feuilles d'un arbre de type `Ab` de la façon suivante :

```
let rec sum = function
  | Leaf n -> n
  | Node(l, r) -> sum(l) + sum(r)
```

Les définitions par cas présentent de nombreux avantages : elles facilitent le raisonnement équationnel sur les fonctions et fournissent un mécanisme concis et expressif pour la définition de fonctions. Néanmoins, étendre ce mécanisme à des types de données non algébriques reste une question ouverte.

3.2 Réécriture locale de sous-collection

Par leur définition inductive, les valeurs des types de données algébriques correspondent à une organisation hiérarchique ; on les appelle également *arbre formel* rappelant que les valeurs sont construites par une racine colorée par un constructeur et pouvant exhiber un certain nombre de fils suivant la définition du type. Lors du filtrage tel qu'il vient d'être décrit, les différents cas pour la définition des fonctions correspondent aux différents constructeurs sur lesquels les racines sont définies[8]. Ainsi, pour la fonction `sum`, deux cas se présentent correspondant à une feuille ou à un nœud de l'arbre. Le filtrage consomme entièrement l'arbre, les sous-arbres étant nommés par des variables du filtre puis traités par des appels récursifs le cas échéant.

Contrairement aux types de données algébriques, il n'est pas possible de représenter les collections topologiques par une racine unique permettant de filtrer toute la structure de données. Cependant, en partant d'un des éléments de la collection puis en suivant la relation de voisinage, il est possible de sélectionner une partie de la collection topologique. Là où le filtrage des structures de données algébriques est *global* dans le sens où toute la structure est filtrée, le filtrage des collections topologiques est *local* : une sous-partie est sélectionnée.

Les *transformations* étendent en ce sens la définition par cas de fonctions aux collections topologiques. Elles correspondent à une nouvelle structure de contrôle fournie par MGS, définie par cas, où chaque cas est décrit par une règle de réécriture $\alpha \rightarrow \beta$:

- α est un *motif* qui sélectionne par filtrage une sous-partie de la collection, appelée *sous-collection*,
- β est une expression qui calcule une nouvelle collection à substituer en lieu et place de la sous-collection filtrée par α.

Une transformation est définie par un ensemble de règles de réécriture et son application sur une collection topologique correspond aux trois opérations interdépendantes suivantes :

1. l'*application d'une règle* avec :
   - le *filtrage* d'une sous-collection, et
   - l'*évaluation de la partie droite*.
2. la *stratégie d'application des règles*, pour choisir les règles à appliquer et pour gérer les collisions si plusieurs applications de règles sont possibles, et enfin
3. la *reconstruction*.

La primitive `map` décrite plus haut peut être programmée de cette façon. Soit l'expression `map(f,c)` appliquant la fonction `f` sur tous les éléments de `c`. Détaillons les différents points précisés ci-dessus dans le cadre de cette fonction :

- La règle de réécriture : la transformation locale de la fonction `map` est capturée par la règle de réécriture $x \rightarrow \texttt{f}(x)$. Le motif de la règle est constitué d'une seule variable filtrant un élément x. La partie droite évalue l'application de la fonction `f` sur l'élément filtré.
- La stratégie d'application : elle doit définir dans le contexte de la fonction `map` l'application de la règle partout où cela est possible, c'est-à-dire sur chaque élément de la collection `c`. La stratégie MGS par défaut, appelée *maximale parallèle* et explicitée plus loin dans ce chapitre, correspond à cette attente.
- La reconstruction : elle consiste ici à recréer une structure identique à celle de la collection `c` et à la décorer avec les valeurs calculées par la partie droite de la règle pour chaque instance de filtre.

[8] Il est également possible de distinguer par deux cas deux racines de même constructeur dont les valeurs des attributs diffèrent.

Cette transformation est programmée et utilisée en MGS par :

```
trans map[fct=\x.x] = { x => fct(x) } ;;

map[fct=f,strategy=default](c) ;;
```

La transformation `map` est définie par une seule règle de réécriture. Un argument optionnel est également défini pour paramétrer la fonction à appliquer localement. Finalement, l'application de cette transformation sur une collection `c` est faite en précisant en paramètre la fonction `f` à appliquer et la stratégie à utiliser.

Application d'une règle. Ce premier point consiste à définir l'application *locale* d'une règle de la transformation sur la collection. Le motif de la règle spécifie la sélection d'une sous-partie de la collection. Cette étape est appelée *filtrage* et la sous-partie filtrée, *sous-collection*. La partie droite de la règle est une expression MGS qui permet le calcul d'une nouvelle sous-collection. L'application de la règle sur une sous-collection filtrée consiste alors à évaluer la partie droite. Ainsi, *l'application d'une règle sur une collection retourne un couple (sous-collection filtrée, sous-collection calculée).*

L'enjeu déterminant de l'application d'une règle consiste à spécifier de façon élégante une sous-collection, c'est-à-dire de développer un langage de spécification de motifs concis et expressif. Deux langages sont développés dans l'interprète MGS et correspondent à deux utilisations différentes des transformations :

1. Le premier concerne la mise à jour des valeurs associées aux cellules topologiques sans modifier la structure. L'expérience, avec notamment la modélisation et la simulation en physique discrète, nous ont conduits à utiliser le $\langle n, p\rangle$-voisinage comme relation de voisinage pour déplacer de l'information entre les n-cellules, information véhiculée par les p-cellules. Il en découle un premier type de transformation, les $\langle n, p\rangle$*-transformations*, appelées également *transformations de chemins*.
2. Le second type de manipulation des collections consiste à en modifier la topologie. Nous avons donc développé un second langage de motifs dédiés à cette opération appelé *transformation de patches*, ou plus simplement *patch*. La construction des motifs ne se restreint plus à l'utilisation du $\langle n, p\rangle$-voisinage, mais s'étend à n'importe quelle organisation reposant sur la relation d'incidence qui lie les cellules topologiques. Les motifs expriment alors une sous-partie du graphe d'incidence.

Stratégie d'application des règles. La transformation est composée de plusieurs règles de réécriture. La *stratégie d'application des règles* consiste à choisir quelle règle doit être appliquée. Il s'agit donc de l'algorithme utilisé lors de l'application des règles pour choisir quelle règle doit s'appliquer lorsque plusieurs motifs sont susceptibles de filtrer la même sous-collection.

On appelle *occurrence de filtre* ou *instance de filtre* une sous-collection susceptible d'être sélectionnée lors de l'application d'une règle. En considérant l'ensemble de toutes les occurrences de filtre de toutes les règles, il est possible de décrire une stratégie d'application suivant les deux critères orthogonaux suivants :

1. *synchrone/asynchrone* : l'application asynchrone d'un ensemble de règles consiste à considérer parmi toutes les occurrences de filtre possibles l'une d'entre-elles ainsi que la règle qui a permis de la filtrer ; la règle est alors appliquée une et une seule fois sur l'instance désignée. En revanche, une stratégie synchrone consiste à sélectionner plusieurs instances disjointes de filtre, et les règles de réécriture correspondantes, et de les appliquer en parallèle sur les sous-collections désignées.

2. *priorité* : elle caractérise le choix de l'instance parmi l'ensemble de toutes les occurrences de filtre de toutes les règles. Ce choix peut être déterminé de façon prédéfinie (désignant comme prioritaires les instances concernant les règles les plus prioritaires) ou de façon stochastique (les instances sont tirées au sort suivant des probabilités associées à chaque règle).

L'application de plusieurs règles en parallèle étant possible, nous considérons qu'après l'étape de gestion de la stratégie, les règles sont appliquées fournissant une *liste de couples (collection filtrée, collection calculée)*.

Reconstruction. Il s'agit de la dernière étape de l'application de la stratégie. En fonction de la liste des couples (sous-collection filtrée, sous-collection calculée) et du type de la collection, les sous-collections calculées sont réarrangées les unes par rapport aux autres pour construire la collection résultat de l'application.

Les sections suivantes décrivent en détail ces trois étapes. Nous commençons par la description du filtrage de chemins, puis nous continuons par le filtrage de patches (page 52). Nous terminons enfin par les stratégies d'application de règles (page 55) disponibles dans l'interprète, et le fonctionnement de la reconstruction (page 58).

3.3 Les transformations de chemins

Les chemins permettent de spécifier les sous-collections comme des séquences d'éléments *voisins* deux à deux (nous dirons aussi *contigus* ou *adjacents*). Les éléments constituant les chemins sont de même dimension. Soit n la dimension des cellules d'un chemin ; la relation de voisinage entre deux n-cellules est donnée par le $\langle n, p\rangle$-voisinage défini précédemment. Ainsi, deux n-cellules voisines consécutives dans un $\langle n, p\rangle$-chemin ont une p-cellule incidente en commun.

Par exemple, comme il a été dit précédemment, le voisinage entre les éléments des structures de données standards peut être représenté par un graphe de voisinage. Le $\langle 0, 1\rangle$-voisinage permet, sur ce type de structure, de parcourir les sommets en suivant les arcs, construisant ainsi des chemins de sommets dans le graphe de voisinage.

Une transformation de chemins est une expression MGS de la forme :

```
trans <n,p> id = {
  Motif => exp;
  ...
} ;;
```

où *exp* est une expression MGS. La transformation s'évalue en une fonction qui attend une collection topologique comme argument. Elle est accessible dans l'environnement par l'identificateur *id*. L'application de cette transformation sur une collection topologique entraîne l'application des règles définies entre accolades. Les paramètres n et p sont des entiers fixant les paramètres de la relation de voisinage pour la construction des chemins.

Les motifs. La syntaxe des motifs, inspirée de celle des expressions régulières, suit la grammaire suivante :

Motif

m ::= `x` | c | `_` | `<undef>`
| m , m | $m\delta m$ | m`+` | m`*` | $m\delta$`+` | $m\delta$`*`
| m`:`P | m `/` exp | m `as x` | m `|` m | m `\/` m

où `x` est une variable de motif, P est un prédicat, *exp* est une expression s'évaluant en un booléen et δ est une valeur de type `posgbf` (dénotant un sous-ensemble particulier de la relation de voisinage pour les GBF). Les explications qui suivent donnent une sémantique informelle de ces motifs :

variable : Une variable de motif `x` filtre une n-cellule σ de la collection décorée par une valeur v (dans le cadre des collections newtoniennes, cette valeur doit être définie, *i.e.* différente de `<undef>`). La variable permet alors de référer à l'élément filtré dans une garde ou en partie droite de la règle :

- la variable `x` prend la valeur v,
- la variable `^x` prend la valeur σ.

Les noms de variables ne sont utilisables qu'une fois par motif : les motifs sont *linéaires*. Ainsi, le motif `x, x` est interdit.

Si la variable n'est pas utilisée, on préférera le motif anonyme « _ » afin de ne pas nommer inutilement un élément filtré.

constante : Un motif c, où c est une constante, filtre une n-cellule dont la valeur associée v est égale à la constante c.

vide : Le motif `<undef>` filtre une n-cellule qui n'est pas décorée par une valeur (uniquement sur des collections newtoniennes). Certaines collections ayant un espace de positions infini, le motif `<undef>` ne peut apparaître seul, mais uniquement comme voisin d'un élément défini (comme par exemple dans `x, <undef>`).

voisinage : Le motif m`,` m' filtre un chemin de n-cellules $e_0, \ldots, e_k, e_{k+1}, \ldots, e_l$ où $e_0, \ldots, e_k$ (resp. $e_{k+1}, \ldots, e_l$) est filtré par m (resp. m') et tel que e_k et e_{k+1} soient $\langle n, p\rangle$-voisins. De même pour le motif $m\delta m'$ mais avec la contrainte supplémentaire que la p-cellule incidente à la fois à e_k et e_{k+1} doit correspondre à un déplacement δ dans une collection GBF. Dans ce dernier cas, le voisinage utilisé est le $\langle 0, 1\rangle$-voisinage.

nommage : La construction m `as x` sert à lier la variable `x` au $\langle n, p\rangle$-chemin filtré par m. La valeur associée à `x` est une séquence MGS de valeurs, et la valeur associée à `^x` est la séquence des positions décorées par ces valeurs.

garde : Le motif m `/` e filtre un $\langle n, p\rangle$-chemin filtré par le motif m et vérifiant l'expression e. Par exemple `(x, y) / x > y` filtre deux n-cellules σ_0 et σ_1 telles que la valeur en σ_0 est plus grande que la valeur en σ_1. Le motif m`:`P est équivalent au motif m `as x /` P`(x)` (où `x` est une variable fraîche).

répétition : Le motif m`*` peut filtrer un chemin vide, un $\langle n, p\rangle$-chemin filtré par m ou une suite de $\langle n, p\rangle$-chemins filtrés par m et dont la jonction est constituée de deux cellules $\langle n, p\rangle$-voisines. Le symbole « * » (étoile) est utilisé en référence aux expressions régulières où il exprime la répétition éventuellement nulle d'une expression régulière. Le motif m`+` filtre les mêmes chemins que m`*` excepté le chemin vide.

Dans le cadre des collections GBF, le motif $m\delta$`*` (resp. $m\delta$`+`) est similaire à m`*` (resp. m`+`) mais les chemins filtrés par m doivent être joints par des positions voisines selon la direction δ.

alternative : Le motif m_1 `|` m_2 filtre un $\langle n, p\rangle$-chemin filtré par m_1 **ou** un chemin filtré par m_2.

descente : Le motif m_1 `\/` m_2 permet de filtrer en *profondeur* dans des imbrications de collections. Par exemple, soit la séquence suivante :

```
(0, 1, (2, 3, 4, 5), 6, (7, 8, 9)) ;;
```

et le motif

```
w, (x \/ (4, y)) as z
```

L'opérateur `\/`, prononcé « dans lequel », permet de prolonger le filtrage à l'intérieur de la collection filtrée par `x`. Dans notre exemple, `x` sélectionne un élément (qui doit être une collection topologique) *dans lequel* un chemin peut être filtré par le motif `4, y`. Seule la sous-séquence `(2, 3, 4, 5)` convient pour l'élément `x` car elle est la seule à contenir l'entier `4`; on note alors que la variable `y` filtre l'entier `5`. Au niveau initial du motif, `w` filtre l'élément qui précède la sous-séquence `(2, 3, 4, 5)`, c'est-à-dire l'entier `1`. Lors de la descente dans l'élément filtré par `x`, le sous-chemin `(4, 5)` étant filtré, il est consommé et détruit par le filtrage : la variable `x` réfère donc à la séquence `(2, 3)`. La variable `z` nomme le sous-motif ayant filtré la sous-séquence, on lui associe donc la valeur `(2, 3, 4, 5)`.

Ces constructions fournissent un langage puissant pour spécifier les $\langle n, p\rangle$-chemins. Par exemple, le motif

```
(x:int / x<3)+ as S / (size(S) < 5) && (fold(\x,y.(x+y),0,S) > 10)
```

sélectionne une sous-collection `S` d'entiers inférieurs à 3, telle que le cardinal de `S` soit inférieur à 5 et telle que la somme des éléments de la collection soit supérieur à 10. Si ce motif est appliqué sur une séquence (resp. un ensemble, un multi-ensemble), `S` dénote une sous-séquence (resp. un sous-ensemble, un sous-multi-ensemble). Appliqué sur une chaîne abstraite, il dénote une sous-chaîne composée de n-cellules $\langle n, p\rangle$-voisines.

Partie droite d'une règle. La partie droite d'une règle est une expression MGS. Elle spécifie la collection à substituer en lieu et place de la sous-collection filtrée en partie gauche. Dans le cadre des transformations de chemins, il existe un point de vue alternatif : 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^{e} élément du chemin filtré est remplacé par le i^{e} élément de la partie droite. L'écriture des règles s'en trouve alors plus concise.

La situation n'est pas la même selon que la collection est newtonienne ou leibnizienne. Pour les collections newtoniennes, il n'est pas possible de substituer un chemin par une séquence de longueur différente : par exemple, on ne peut pas substituer une sous-partie d'une grille par une autre sous-partie ne respectant pas le même patron sans détruire la topologie de la gille. En revanche les collections leibniziennes autorisent de telles modifications.

Afin d'exprimer l'accès au voisinage de la sous-collection filtrée en partie droite, des primitives supplémentaires sont disponibles. Elles permettent d'itérer sur les cellules incidentes d'un élément filtré. Voici un exemple illustrant leur utilisation :

```
trans <n,p> average = {
  x => (
    let total = neighborfold(\a,b.(a+b), 0, x)
    and size  = neighborsize(x)
      total / size )
} ;;
```

Cette $\langle n, p\rangle$-transformation agit tel un `map` filtrant toutes les `n`-cellules d'une collection pour la décorer par la moyenne des valeurs décorant ses $\langle$`n,p`$\rangle$-voisines. La primitive `neighborfold` correspond à du sucre syntaxique pour :

`fold(`*f*`,` *zero*`, neighbors(self,n,p,^x))`

La variable `self` qui ne peut être utilisée que dans la portée d'une transformation, réfère à la collection argument de cette transformation. La primitive `neighbors(`*coll*`,`*n*`,`*p*`,`*pos*`)` retourne la liste des valeurs décorant les $\langle n, p\rangle$-voisines de la cellule *pos* dans la collection *coll*. La primitive `neighborsize` fonctionne de la même façon. Pour les chaînes abstraites, l'interprète dispose également des fonctions `facesfold`, `cofacesfold`, etc.

Exemple. Afin d'illustrer l'utilisation des transformations de chemins, nous proposons un exemple de transformation d'ensemble permettant de calculer les nombres premiers. La transformation fonctionne de la façon suivante : si deux éléments de l'ensemble peuvent être sélectionnés et que l'un est divisé par l'autre, alors ce dernier est supprimé de l'ensemble. La transformation s'écrit :

```
trans <0,1> prime = {
  x:int, y:int / (x % y == 0) => y
} ;;
```

Cette transformation utilise le $\langle 0, 1\rangle$-voisinage pour être appliquée sur un ensemble où seuls les sommets sont décorés et où les arcs correspondent à la relation de voisinage. La règle filtre deux éléments dont les valeurs associées sont des entiers et telle que `y` divise `x`. Dans ce cas, `x` n'est pas premier et est supprimé. Ce calcul est appliqué en parallèle sur plusieurs couples d'éléments disjoints de la collection en utilisant la stratégie par défaut de MGS. La règle ne spécifiant pas que l'élément restant n'est pas divisible par un autre élément de l'ensemble, il est donc nécessaire d'appliquer la transformation jusqu'à l'obtention d'un point fixe. À terme, il ne reste effectivement plus dans l'ensemble que des entiers premiers entre eux. Pour générer l'ensemble des nombres premiers inférieurs à `100` par exemple, il suffit de générer un ensemble contenant tous les entiers de `2` à `100` et d'itérer la transformation `prime`. En supposant qu'un tel ensemble est lié à la variable `c`, l'application est donnée par :

```
prime[fixpoint](c) ;;
```

3.4 Les patches

Les transformations de chemins permettent la mise à jour des valeurs associées aux cellules topologiques. Les *patches* sont à l'inverse destinés à en modifier la structure. Leur utilisation est pour le moment réservée aux collections topologiques de type `achain` et `qmf`.

Leur définition suit la même syntaxe que celle des transformations de chemins :

```
patch id = {
  PMotif => exp;
  ...
} ;;
```

Un patch est également un ensemble ordonné de règles de réécriture. Un nouveau langage a été introduit pour spécifier les motifs. Comme dans les transformations de chemins, la partie droite reste une expression. Afin d'illustrer la nouvelle syntaxe des règles de réécriture, nous

FIG. 17 – Insertion d'un sommet sur un arc.

considérons la modification topologique décrite figure 17 : un arc nommé `e`, dont les faces sont appelées `v1` et `v2`, est filtré et remplacé par deux nouveaux arcs `e1` et `e2` ainsi qu'un nouveau sommet `v` ; l'arc `e1` est bordé par les sommets `v1` et `v`, et l'arc `e2` par les sommets `v2` et `v`. Cette opération permet d'insérer un nouveau sommet sur un arc.

Les motifs. Les motifs décrivent la liste des cellules topologiques à filtrer. Elles sont caractérisées par une dimension, ainsi que la liste partielle de leurs faces et de leurs cofaces. On définit ainsi un ensemble de contraintes sur les cellules topologiques ainsi que sur la relation d'incidence qui les relie les unes aux autres. Voici la grammaire des motifs de patch :

PMotif
m ::= c | $c\,m$

PClause
c ::= `x:[dim=`exp_d`, faces=`exp_f`, cofaces=`exp_{cf}`,` exp_b`]`

Un motif est une liste finie de *clauses* et une clause (*PClause*) correspond à un élément à filtrer. Ces clauses sont caractérisées par plusieurs informations qui vont contraindre la recherche d'une sous-collection :

- `x` est une variable de motif qui permet de faire référence à la cellule filtrée par la clause n'importe où dans la règle. Contrairement aux motifs de chemin, les motifs de patch ne sont pas linéaires. Il est possible d'utiliser une variable qui n'est définie que plus loin dans le motif. Néanmoins, un nom correspond à une et une seule cellule filtrée. Si deux clauses partagent le même identificateur, elles filtrent la même cellule ; les prédicats des deux clauses doivent être vérifiées.
- L'expression exp_d associée au champ `dim` s'évalue en un entier indiquant la dimension de la cellule filtrée par la clause.
- Les expressions exp_f et exp_{cf} associées respectivement aux champs `faces` et `cofaces` sont des expressions MGS qui s'évaluent en des séquences de cellules topologiques. On peut utiliser ici les variables de motif ou faire directement référence à des cellules particulières de la structure. Ces séquences contraignent la relation d'incidence que doivent respecter les cellules filtrées. Elles ne sont pas exhaustives ; une cellule peut posséder des (co)faces en plus de celles imposées par le motif.
- Une dernière expression optionnelle peut être ajoutée à cette liste. Il s'agit de l'expression exp_b dont l'évaluation doit retourner `true`. Cela permet par exemple d'imposer des propriétés à la valeur associée à la cellule filtrée par la clause.

Le motif correspondant à la partie gauche de la règle de la figure 17 s'écrit alors de la manière suivante :

```
e :[ dim = 1, faces = (v1,v2), cofaces = seq:() ]
v1:[ dim = 0, faces = seq:(),  cofaces = (e)    ]
v2:[ dim = 0, faces = seq:(),  cofaces = (e)    ]
```

Ce motif présente un grand nombre de redondances. Par exemple, les faces de l'arc `e` étant `v1` et `v2`, il n'est pas nécessaire de préciser que `e` appartient aux cofaces de `v1` et `v2`. Cette information permet également d'inférer les dimensions de `v1` et `v2` à partir de celle de `e`. On réduit les informations redondantes au moyen de *sucre syntaxique*. Deux opérateurs, `<` et `>`, sont introduits entre les clauses des motifs :

PMotif

$m \quad ::= \quad c \quad | \quad c\ o\ m$

Op

$o \quad ::= \quad \varepsilon \quad | \quad < \quad | \quad >$

où ε dénote le mot vide. L'opérateur infixe binaire `<` (resp. `>`) contraint l'élément filtré par son opérande gauche à être une face (resp. une coface) de la cellule filtrée par l'opérande droite. Par exemple, le motif `v:[...] < e:[...]` signifie que la cellule filtrée par `v` est une face de celle filtrée par `e`. Ce motif est équivalent à `e:[...] > v:[...]`. L'absence d'opérateur entre deux clauses dénote l'absence de contrainte sur la relation d'incidence qui les lie. Le motif précédent peut alors être réduit à :

```
v1 < e:[ dim = 1 ] > v2
```

Les stratégies d'application de règles synchrones permettent d'appliquer une règle sur plusieurs sous-collections en même temps. Ces instances de motif sont *disjointes* et ne peuvent partager une sous-partie. Certaines informations du motif sont uniquement utilisées pour nommer le voisinage des éléments filtrés. C'est par exemple le cas pour la règle de la figure 17 où les sommets `v1` et `v2` ne servent qu'à la reconstruction.

Pour autoriser certains éléments à être filtrés plusieurs fois, nous introduisons dans le motif un opérateur unaire « ~ ». Celui-ci signifie que la clause correspond à une cellule répondant toujours aux contraintes structurelles imposées par le motif, mais qui ne sera pas considérée comme filtrée après la recherche de l'instance. Autrement dit, l'élément sera filtré mais *non-consommé* par l'étape de filtrage ; il pourra alors être à nouveau sélectionné lors de la recherche des occurrences de filtre suivantes. La règle de la consommation est la suivante :

- toute cellule peut être filtrée sans être consommée (clause dont l'identificateur est précédé par l'opérateur ~) ;
- seules des cellules non-consommées peuvent être consommées (clause dont l'identificateur **n**'est **pas** précédé par l'opérateur ~).

Ainsi, dans notre exemple, pour que l'insertion de sommet s'applique sur tous les arcs du complexe, l'opérateur de non-consommation est utilisé de la façon suivante :

```
~v1 < e:[ dim = 1 ] > ~v2
```

Partie droite d'une règle. La partie droite d'une règle est une expression MGS comme dans les transformations de chemins. Pour ces dernières, le type de valeur attendu est une séquence, structure correspondant à la notion de chemin. Dans le cas des patches, la partie droite doit spécifier une sous-partie d'un graphe d'incidence. Autorisant la modification de la topologie, de nouvelles cellules apparaissent alors que d'autres, filtrées, ne sont pas réécrites. La partie droite décrit la nouvelle relation d'incidence entre les nouvelles et les anciennes cellules respectant la modification topologique.

Les nouvelles sous-collections sont décrites à l'aide d'une syntaxe spéciale. Celle-ci évalue des séquences d'un type particulier destinées à la spécification de nouvelles sous-collections. Cette syntaxe est proche de celle des motifs sans sucre syntaxique :

PRhs
r ::= e | $e\ r$

PRhsElt
e ::= `x:[val=`exp_v`]`
| $symb$`:[dim=`exp_d`, faces=`exp_f`, cofaces=`exp_{cf}`, val=`exp_v`]`

Cette spécification (*PRhs* pour *Patch Right-hand-side*) de la sous-collection à réécrire correspond à une liste d'éléments (*PRhsElt*) de deux types :

1. Les éléments filtrés et conservés en partie droite sont référencés via les identificateurs `x` qui ont permis de les nommer en partie gauche de la règle ; cette première construction permet de mettre à jour la valeur qui leur est associée.
2. De nouvelles cellules topologiques peuvent être introduites dans le nouveau complexe cellulaire. Ces éléments n'existent pas avant l'étape de reconstruction. Pour les identifier, nous utilisons donc des symboles (*symb*), c'est-à-dire des constantes MGS. Ce symbole peut apparaître ailleurs dans la partie droite : il fait alors référence à la cellule qui sera créée à l'application de la règle. Chaque nouvel élément est caractérisé par sa dimension exp_d, la liste de ses faces exp_f (pouvant faire apparaître à la fois d'anciens éléments à travers l'utilisation des variables de filtre, et des nouveaux éléments en utilisant les symboles), la liste de ses cofaces exp_{cf} et la valeur qui lui est associée exp_v.

En suivant cette syntaxe, la règle schématisée figure 17 s'écrit de la façon suivante en MGS :

```
~v1 < e:[ dim = 1 ] > ~v2 =>
   ‘e1:[ dim = 1, faces = (^v1,‘v), cofaces = cofaces(^e), val = ... ]
   ‘v :[ dim = 0, faces = seq:(),   cofaces = (‘e1,‘e2),   val = ... ]
   ‘e2:[ dim = 1, faces = (^v2,‘v), cofaces = cofaces(^e), val = ... ]
```

Les éléments filtrés par `v1` et `v2` ne sont pas réécrits en partie droite de la règle. En effet, ils ne sont pas consommés et ne peuvent donc pas être réécrits. Les cofaces de `‘e1` et `‘e2` sont celles de `e` de telle sorte que si l'arc `e` borde une cellule de dimension 2, les cellules spécifiées par `‘e1` et `‘e2` bordent également cette cellule après l'application.

Des exemples de modifications de la topologie des collections sont donnés dans le chapitre 8.

3.5 Stratégies d'application de règles

La stratégie d'application des règles est une politique guidant l'application d'une transformation sur une collection topologique. Elle est précisée lors de l'évaluation d'une telle application au moyen du paramètre optionnel `strategy`. L'application d'une transformation `T` avec une stratégie `s` sur une collection topologique `c` s'écrit :

```
T[ strategy = s ](c) ;;
```

La valeur de `s` est un symbole MGS ; les différentes stratégies fournies par l'interprète sont :

`‘default :` stratégie maximale parallèle (synchrone) avec priorité des premières règles sur les dernières,

`‘asynchronous :` stratégie asynchrone avec priorité des premières règles sur les dernières,

‘`singleStochastic` : stratégie maximale parallèle (synchrone) avec une priorité choisie aléatoirement entre les règles,

‘`multiStochastic` : stratégie maximale parallèle (synchrone) sans priorité entre les règles,

‘`stochastic` : stratégie stochastique (asynchrone) avec probabilité explicite sur chaque règle,

‘`gillespie` : stratégie asynchrone inspirée de l'algorithme de D.T. Gillespie.

Lorsqu'aucune stratégie n'est spécifiée, l'interprète effectue par défaut une application maximale parallèle des règles (‘`default`).

3.5.1 Stratégie ‘`default` : maximale parallèle

Les sous-collections sont filtrées par les motifs des règles de façon disjointe. Après l'étape de filtrage, on assure qu'il n'existe pas de sous-collection non filtrée qui puisse être filtrée par l'un des motifs des règles de la transformation ; c'est en cela que cette stratégie est dite maximale. Lorsque deux motifs filtrent une même partie de la collection, une priorité est donnée à l'une des règles. En pratique, l'ordre choisi est l'ordre dans lequel les règles sont spécifiées. Après la sélection des sous-collections à faire évoluer, l'application des règles se fait en parallèle. Cette stratégie trouve une forte motivation dans le cadre de la simulation puisqu'elle supporte l'idée que les sous-parties d'un système évoluent en parallèle et de façon indépendante. Elle a notamment été utilisée dans les systèmes de Lindenmayer, des systèmes de réécriture maximale parallèle de chaînes [PLH+90, PH92, LJ92, RS92].

Par exemple, soient la transformation

```
trans T = { x => x + 1 ; x => x * 10 } ;;
```

et la collection topologique

```
c := (1,2,3,4,5,6) ;;
```

L'application maximale parallèle de `T` sur `c` produit la séquence `(2,3,4,5,6,7)`. En effet, la première règle, dotée d'une priorité supérieure à la seconde, est appliquée jusqu'à ce qu'il ne reste plus d'élément à filtrer. Par conséquent, la seconde règle n'est jamais appliquée, tous les éléments étant consommés par la première.

3.5.2 Stratégie ‘`asynchronous`

Avec cette stratégie, une seule règle est appliquée une seule fois et la priorité est donnée aux premières règles spécifiées. En d'autres termes, une seule sous-collection est réécrite si cela est possible.

Avec la transformation `T` et la collection `c` précédentes, le résultat est similaire à `c` à ceci près qu'un élément est incrémenté de 1. La position de cet élément n'est pas prévisible car la recherche des instances de filtre est non-déterministe.

3.5.3 Stratégie ‘`singleStochastic`

Cette stratégie est identique à la stratégie maximale parallèle à ceci près que la priorité entre les règles est choisie arbitrairement avant chaque application. Par exemple, en reprenant la transformation `T` et la collection topologique `c`, cette stratégie peut produire deux résultats (son application est non-déterministe) : `(2,3,4,5,6,7)` et `(10,20,30,40,50,60)` avec la même probabilité.

3.5.4 Stratégie `‘multiStochastic`

Pour cette stratégie également maximale parallèle, il n'y a pas de priorité entre les règles. Une règle est choisie au hasard parmi celles spécifiées dans la transformation, puis appliquée une seule fois. Le processus est itéré jusqu'à ce qu'aucune règle ne puisse plus être appliquée. Pour l'exemple précédent, 64 résultats différents peuvent être produits :

```
(2,3,4,5,6,7)
(10,3,4,5,6,7)
(10,20,4,5,6,7)
...
(10,20,30,40,50,60)
```

3.5.5 Stratégie `‘stochastic`

Cette stratégie est asynchrone (une seule règle est appliquée une seule fois) et stochastique (chaque règle spécifie avec quelle probabilité elle peut être choisie). Sur chaque règle, une probabilité est fournie sous la forme du paramètre `P` défini dans la flèche de la règle. Ce paramètre peut être un entier, un flottant ou une λ-abstraction recevant la collection argument lorsqu'elle est évaluée et qui doit retourner un entier ou un flottant :

```
trans T = {
  x ={ P = \x.(0.25) }=> x + 1;
  x ={ P = 0.25      }=> x + 10;
  x ={ P = 0.5       }=> x + 100
} ;;
```

3.5.6 Stratégie `‘gillespie`

La stratégie d'application `‘gillespie` permet de paramétrer l'application des règles de manière à émuler l'algorithme original de D.T. Gillespie développé dans le cadre de la simulation exacte et stochastique de réactions chimiques [Gil77, Gil01]. L'algorithme de D.T. Gillespie permet de déterminer dans une solution chimique la prochaine réaction ayant lieu en tenant compte des cinétiques chimiques. Chaque règle de la transformation correspond à une réaction et il lui est associée une constante, dite *constante stochastique*, correspondant à la probabilité moyenne qu'une combinaison des réactants réagisse dans le prochain intervalle de temps infinitésimal. En fonction de cette constante et du nombre de fois où la règle peut s'appliquer, l'algorithme de D.T. Gillespie détermine de façon probabiliste la date de la prochaine réaction et l'évolution des réactants.

Les équations auto-catalytiques de *Lotka-Volterra* sont un exemple standard de réactions pouvant être simulées par l'algorithme de D.T. Gillespie [Gil77]. Considérant une espèce proie Y_1, une espèce prédateur Y_2 et une ressource infinie X, l'évolution des effectifs de chaque espèce suit les réactions suivantes :

$$\begin{array}{rcll} X + Y_1 & \rightarrow & X + 2\,Y_1 & \text{avec} \quad C = 0.001 \\ Y_1 + Y_2 & \rightarrow & 2\,Y_2 & \text{avec} \quad C = 0.01 \\ Y_2 & \rightarrow & Z & \text{avec} \quad C = 0.1 \end{array}$$

La première règle décrit la reproduction des individus de l'espèce proie Y_1 contre une certaine quantité de nourriture X ; la seconde règle exprime la reproduction de l'espèce prédatrice Y_2

se nourrissant des individus de l'espèce Y_1 ; enfin, la dernière règle spécifie la disparition des individus de Y_2 de mort naturelle. La traduction en terme de transformation MGS est la suivante :

```
trans lotka_volterra = {
  ‘X, ‘Y1  ={ C = 0.001 }=> #2 ‘Y1, ‘X;
  ‘Y1, ‘Y2 ={ C = 0.01  }=> #2 ‘Y2;
  ‘Y2      ={ C = 0.1   }=> ‘Z
} ;;
```

L'expression suivante itère l'application de la transformation `lotka_volterra` sur un multi-ensemble contenant 100 individus de chaque espèce jusqu'à ce que le compteur de temps `tau`, cumulant les temps d'attente entre chaque application de règle, dépasse la valeur 5.0 :

```
lotka_volterra[fixpoint = (\x1,x2.(tau >= 5.0)),
               strategy = ‘gillespie
              ](bagify((#100 ‘X, #100 ‘Y1, #100 ‘Y2))) ;;
```

L'expression `#100 ‘X` construit une séquence de longueur 100 contenant des constantes symboliques `‘X`.

La stratégie `‘gillespie`, dans son utilisation usuelle, est réservée à des multi-ensembles d'espèces chimiques. Notons cependant que l'algorithme peut se généraliser au cas plus général d'une simulation par événements discrets où chaque évolution possible est affectée d'une probabilité qui dépend de l'état du système. L'algorithme de D.T. Gillespie est donc un algorithme important qui trouve son utilité bien au-delà de la simulation de réactions chimiques. Nous présentons en détail cette stratégie ainsi que son utilisation sur des espaces hétérogènes (des multi-ensembles imbriqués) dans le chapitre 10.

3.6 La reconstruction de collection

Comme cela a été dit précédemment, l'application des règles de la transformation retourne une liste de couples (sous-collection filtrée, sous-collection calculée). Cette liste est appelée un *batch*. La reconstruction consiste à réduire ce batch en une nouvelle valeur. Dans l'interprète actuel, le type de collection est préservé par une transformation ; en d'autres termes, l'application d'une transformation sur une séquence (resp. un ensemble, un multi-ensemble, un GBF, une chaîne abstraite, une G-carte, etc.) retourne une séquence (resp. un ensemble, un multi-ensemble, un GBF, une chaîne abstraite, une G-carte, etc.). Nous détaillons la reconstruction des $\langle n, p\rangle$-transformations et des patches séparément.

3.6.1 Reconstruction des $\langle n, p\rangle$-transformations

Pour les transformations de chemins, les sous-collections filtrées sont spécifiées par des listes de positions, et les sous-collections calculées par des listes de valeurs. Dénotons le batch d'une transformation appliquée sur une collection `c` de la façon suivante :

$$\texttt{[(}l_0\texttt{,}l'_0\texttt{); ...; (}l_N\texttt{,}l'_N\texttt{)]}$$

Nous cherchons à spécifier le résultat c' de la transformation en fonction de ce batch. On distingue deux types de reconstruction selon que la collection topologique est newtonienne ou leibnizienne :

- Pour les collections leibniziennes, partant d'une collection vide (étant leibnizienne, elle ne contient aucune position), les éléments des séquences l'_i sont ajoutés à l'aide de l'opérateur `::`. Une permutation des couples du batch peut être nécessaire pour réinsérer les éléments dans l'ordre ; on pense en particulier aux séquences où l'ordre d'ajout importe.
- Pour les collections newtoniennes, les longueurs de l_i et de l'_i doivent être égales (il ne peut y avoir de modification topologique d'une collection newtonienne) ; il suffit alors de placer à chaque position l_i`.(`j`)` d'une collection vide la valeur l'_i`.(`j`)` en utilisant une définition en extension par exemple.

Les deux paragraphes suivants présentent l'application de la transformation

```
trans <0,1> bubble_sort = {
  x, y / (x > y) => y, x ;
} ;;
```

calculant une étape d'un tri par bulle, sur une séquence et un GBF.

Tri à bulle d'une séquence. Soit la séquence MGS suivante

```
c := (30,20,10,50,40,60) ;;
```

Supposons que l'application de la transformation `bubble_sort` avec la stratégie par défaut construit le batch suivant :

```
[ ((3,4),(40,50)) ;((1,2),(10,20)) ;((0),(30)) ;((5),(60)) ]
```

Dans un premier temps, le batch est trié de telle sorte que les sous-séquences filtrées s'enchaînent suivant l'ordre des positions dans la séquence initiale :

```
[ ((0),(30)) ;((1,2),(10,20)) ;((3,4),(40,50)) ;((5),(60)) ]
```

Les éléments des sous-séquences calculées sont alors ajoutés dans l'ordre dans la séquence vide

```
30 :: 10 :: 20 :: 40 :: 50 :: 60 :: seq:()
```

L'application de la transformation retourne finalement la séquence

```
(30,10,20,40,50,60)
```

Tri à bulle d'un GBF. Soit le GBF suivant

```
gbf array = < right > ;;
c := (30,20,10,50,40,60) following |right> ;;
```

Il s'agit d'un GBF à une dimension, homomorphe à $\mathbb{Z}$. Supposons que l'application de la transformation précédente retourne le batch suivant :

```
[ ((4*|right>, 3*|right>), (50, 40)) ;
  ((0*|right>, 1*|right>), (20, 30)) ;
  ((2*|right>), (10)) ;
  ((5*|right>), (60)) ]
```

Ce batch diffère du précédent pour illustrer le fait que la stratégie par défaut de MGS n'est pas déterministe. À partir de ce batch, le résultat est calculé à l'aide d'une définition par extension

```
array:{| 50 @ 4*|right>,
         40 @ 3*|right>,
         20 @ 0*|right>,
         30 @ 1*|right>,
         10 @ 2*|right>,
         60 @ 5*|right>,
|} ;;
```

construisant le résultat `c'`

```
20 |right> 30 |right> 10 |right> 40 |right> 50 |right> 60
```

3.6.2 Reconstruction des patches

Quel que soit le type de la collection (`achain` ou `qmf`) sur laquelle le patch est appliqué, la reconstruction génère un ensemble de cellules topologiques fraîches ainsi que la relation d'incidence qui les relie. Chacune des collections calculées du batch correspond à une séquence d'éléments à réécrire. La liste complète de ces éléments est alors réduite ; partant d'un ensemble de cellules topologiques S vide, les éléments présents dans les parties droites sont ajoutés à S successivement :

- Soit σ une cellule filtrée par le motif `x:[...]` qu'il faut réécrire en `x:[val=`v`]`. La cellule σ est dans un premier temps dupliquée en σ', et σ' est ajoutée à S. Pour toute cellule de τ' de S correspondant à une copie d'une cellule τ appartenant à la structure de la collection initiale et telle que τ et σ sont incidentes, la cellule σ' est définie comme incidente à τ'. De cette façon, l'ensemble de la relation d'incidence de la structure initiale est conservée pour tout couple de cellules réécrites en partie droite. Dans la structure finale, la valeur v est associée à la cellule σ'.
- Soit un nouvel élément à créer `'e:[dim=`d`, faces=`f`, cofaces=`cf`, val=`v`]`. Une d-cellule fraîche σ' est créée et ajoutée à S. Soit e un élément de la liste évaluée par l'expression f (resp. cf) :
  - si e est une cellule τ de la structure initiale dont une copie τ' appartient à S, alors σ' est définie comme une coface (resp. une face) de τ' ;
  - si e est un symbole référant à une cellule τ' appartenant à S, σ' est définie comme une coface (resp. une face) de τ' ;

  Dans la structure finale, la valeur v est associée à la cellule σ'.

L'ensemble S muni de la relation d'incidence telle qu'elle vient d'être définie, est ensuite converti suivant le type de structure de données sur lequel la transformation est appliquée. Pour les collections `qmf`, les contraintes d'intégrité doivent être vérifiées pour que cette conversion n'échoue pas. Si la construction retournée ne les respecte pas, une exception est levée.

Deuxième partie

Formalisation

Chapitre 3

Collections topologiques

Dans ce chapitre, nous nous intéressons de manière plus technique à la notion de collection topologique introduite informellement dans le chapitre 2. Nous proposons une formalisation des collections topologiques sous forme de *chaînes topologiques*, objets issus de la *topologie*

algébrique[1]. Ce domaine étudie les espaces topologiques en leur associant un objet algébrique (groupe, espace vectoriel, etc.) dont les propriétés servent à déterminer un certain nombre d'invariants algébriques caractérisant la topologie de l'espace initial. Notre utilisation de la topologie combinatoire se limite à la définition des collections topologiques en terme de chaînes. Cette définition sera utilisée dans les chapitres suivants (voir les chapitres 4, 5 et 6). Notre propos ne porte pas sur les caractéristiques topologiques des espaces que nous souhaitons manipuler, mais sur le développement d'un formalisme unique pour représenter une large classe de structures de données.

Dans un premier temps (section 1), nous présentons les besoins issus des problématiques de la simulation de modèles de systèmes dynamiques à structure dynamique. Ces besoins nous ont poussés à élaborer une plate-forme de travail unique pour le représentation de structures, en nous inspirant notamment des réalisations faites dans le domaine de la représentation de solides pour la modélisation géométrique et la physique discrète. À partir de ces considérations, nous décrivons les notions de *complexe cellulaire* (section 2) et de *complexe de chaînes* (section 3) pour finalement élaborer la formalisation des collections topologiques (section 4). Dans la même section, nous montrons à travers un grand nombre d'exemples comment des structures de données standards peuvent être considérées du point de vue des collections topologiques. Nous terminons ce chapitre (section 5) par la définition d'un type de collections topologiques « universel » dans lequel toute structure peut être représentée, qui servira dans la définition de la sémantique du langage (chapitre 4).

1 Une structure de données pour la simulation

En informatique, les *structures de données* sont définies comme un moyen d'organiser des données dans la mémoire d'un ordinateur de telle sorte qu'elles puissent être accédées efficacement. Le choix d'une structure de données nécessite souvent la représentation d'une organisation abstraite de ces données et dépend fortement de son utilisation ultérieure. Le développement d'un langage dédié à la modélisation et la simulation de systèmes dynamiques tels que MGS amène naturellement à offrir des structures de données adaptées à la représentation de l'état du système. Ces structures doivent être :

- *expressives*, c'est-à-dire permettre une représentation directe et efficace de l'état du système, et
- *adaptées* à l'expression des lois d'évolution du système, afin de simplifier le calcul de l'état suivant.

1.1 Les besoins de la simulation

Afin de répondre à ces deux objectifs, notre point de départ pour représenter l'état d'un système est la notion d'*interaction* [Rau03, GMCS05]. L'évolution du système est spécifié à travers l'interaction de ses composants. Le fait que deux entités (atomiques ou non) soient en interaction ou puissent potentiellement interagir, les placent dans un certain espace l'une voisine de l'autre. Cet espace peut être l'espace physique : souvent, il n'y a pas d'interactions à distance,

[1] Inventée au début du XXe siècle pour résoudre des problèmes géométriques, la topologie algébrique connut un grand développement grâce à l'introduction de constructions algébriques de plus en plus abstraites [Mor04]. Elle permet d'étudier certaines situations purement topologiques à l'aide de concepts algébriques tels que les modules, les groupes, etc. En particulier, les problèmes de l'homéomorphisme entre deux espaces topologiques, ou de l'équivalence de deux topologies sur un même ensemble sous-jacent peuvent être abordés de façon constructive en introduisant des groupes dits d'homotopie, resp. d'homologie [RS97].

ce qui signifie que seules des entités physiquement proches interagissent. Mais il peut aussi être abstrait : par exemple, dans les réseaux sociaux, les individus qui interagissent sont ceux qui partagent des affinités indépendamment de leur localisation physique.

Quoi qu'il en soit, l'évolution d'un système, exprimée sous la forme d'interactions « locales » entre les entités qui composent ce système, induit une organisation de ces entités qui s'exprime sous la forme de relations spatiales (par des relations de voisinage).

Inversement, supposons que l'on ait à exprimer l'évolution d'un système sous la forme d'un ensemble d'interactions. Il sera plus simple d'exprimer ces interactions s'il est possible de structurer l'état du système de telle manière que les interactions n'impliquent que des éléments « voisins » dans cette structure. En effet, dans ce cas, il sera possible d'expliciter l'évolution du système par une fonction ou une relation préservant la structure sur laquelle elle s'applique (un morphisme).

Une structure de données adaptée à cette notion générale d'interaction doit donc présenter les caractéristiques suivantes :

- La structure de données doit permettre la représentation explicite de relations de voisinage entre des entités arbitraires.
- Ces relations de voisinage doivent être elles-mêmes arbitraires. En effet, les interactions sont de toutes sortes. Bien que dans certains systèmes les entités qui les composent soient organisées de façon régulière, dans le cas général, cette organisation spatiale est arbitraire.
- L'organisation spatiale explicitée par la structure de données doit être labile. Autrement dit, la même structure de données doit être capable de représenter plusieurs organisations différentes. En effet, au cours de l'évolution d'un système, son organisation peut changer (sa *structure* est *dynamique*). Par exemple, deux parties en interaction d'un système peuvent s'éloigner physiquement rendant les interactions ultérieures impossibles. Il faut que la structure de données soit suffisamment souple pour permettre cette évolution.
- La structure de données doit permettre la représentation de relations spatiales discrètes. Ce point est plus discutable, beaucoup de systèmes se décrivant à travers une organisation continue (par exemple, la température en tout point d'un volume). Cependant, deux arguments nous ont amenés à nous concentrer sur des organisations spatiales « discrètes ». Le premier argument est que le traitement informatique d'entités continues passe souvent par une discrétisation qui ramène le problème continu à un modèle dont la structure est discrète. On pense par exemple aux méthodes des différences finies ou des éléments finis [ZT00], aux méthodes cellulaires ou mimétiques développées par E. Tonti [Ton01] et d'autres, ou encore à des approches comme celle du calcul différentiel discret [Hir03, DKT06]. Le second argument est que nous nous intéressons principalement à la simulation de processus de morphogenèse. Dans ce type d'application, les événements morphogenétiques correspondent à des changements discrets affectant la structuration d'un ensemble fini d'entités composant le système (comme par exemple la pousse d'un entre-nœud dans le cas de la croissance des plantes, la migration cellulaire dans le cas de la gastrulation, ...) [GS06a, GGMP02b].
- Les lois d'évolution expriment souvent comment une interaction entre quelques éléments du système modifie *localement* l'état de ces éléments. La structure doit donc offrir les moyens nécessaires pour en modifier localement l'état laissant inchangé le reste des données.
- La *dimension* des composants du système joue également un rôle pour les communications d'informations. Par exemple, deux cellules biologiques communiquant *via* des canaux biochimiques entre leurs membranes peuvent être représentées par deux objets abstraits de

dimension 3 communiquant par l'intermédiaire d'un médium de dimension 2 correspondant ici aux membranes cellulaires en contact.

En résumé, MGS doit fournir *des structures de données adaptées à la représentation d'organisations spatiales discrètes et arbitrairement complexes, suffisamment souples pour autoriser les modifications locales de ces organisations, et offrant la notion de dimension.*

1.2 Représentation de l'espace et modélisation des solides

La modélisation géométrique est un domaine de l'informatique dont les techniques de représentation d'objets sont susceptibles de répondre aux besoins exprimés ci-dessus. Ces structures de données permettent de représenter des espaces complexes dans le cadre de la CAO, en infographie ou encore en modélisation discrète de systèmes physiques. V. Shapiro donne dans [Sha01] une bonne introduction à la modélisation mathématique et informatique des solides tridimensionnels. Il décrit dans les sections 2 et 3 les modèles mathématiques pour la représentation de l'espace ainsi que leurs représentations informatiques. Dans cette section, nous nous pencherons principalement sur les modèles mathématiques en vue de la définition d'une sémantique dans le chapitre suivant. V. Shapiro décrit deux grands types de modèles pour la représentation de l'espace :

- Le premier est un modèle fondé sur la manipulation de sous-ensembles continus de $\mathbb{R}^3$. De façon générale, tous les sous-ensembles de $\mathbb{R}^3$ ne représentent pas adéquatement un solide ; on impose pour cela des propriétés qu'ils doivent vérifier pour les considérer comme tels. On demande par exemple qu'ils soient égaux à la fermeture de leur intérieur. Nous ne rentrerons pas plus dans les détails, le lecteur intéressé se référera à [Sha01]. Ce type de modèle induit une représentation *implicite* et *constructive* définie à travers un ensemble de tests déterminants si un point de l'espace appartient ou non au solide. Ainsi, un solide X est représenté par l'ensemble :

  $$X = \{p \in \mathbb{R}^3 \mid A(p) = \text{true}\}$$

  où A est le prédicat d'appartenance au solide. Les prédicats les plus simples correspondent à une liste d'égalités et d'inégalités. De tels espaces peuvent être construits itérativement partant d'objets élémentaires composés et raffinés à l'aide d'opérations ensemblistes (union, intersection et différence). L'une des représentations de ce type les plus connues est appelée *CSG* pour *Constructive Solid Geometry.*

- Le second modèle est fondé sur la caractérisation combinatoire d'un solide par un ensemble de solides simples appelés *cellules.* Dans ce modèle la notion de *bord* a une importance capitale. En effet, pour représenter les solides, on définit un *complexe cellulaire* comme un assemblage de cellules ; cet assemblage est effectué de telle sorte qu'il vérifie la *condition aux frontières* qui demande que le bord relatif de chaque cellule soit l'union d'autres cellules du complexe. Dans de bonnes conditions, les notions de complexes cellulaires et de bord permettent la définition d'opérations purement algébriques avec lesquelles il est possible de construire le bord d'un solide ou encore de définir une algèbre de formes différentielles discrètes sur ces espaces. Ce type de modèle donne naissance à des représentations *énumératives* et *combinatoires* permettant la génération des points appartenant aux solides. Chaque cellule permet d'énumérer les points qui la composent. Par exemple, pour un arc, une fonction de $]0,1[\rightarrow \mathbb{R}^2$ est associée à la cellule permettant de parcourir les points d'un bout (*i.e.* un sommet) à l'autre. Des propriétés de voisinage (intuitivement les équations de deux arcs adjacents à un même sommet doivent se superposer de façon continue et/ou dérivable en ce sommet) sont alors à vérifier pour assurer

des propriétés telles que la continuité ou la différentiabilité des espaces décrits. Dans le cadre de la modélisation des solides, les complexes cellulaires sont de dimension 3, mais les concepts qu'ils apportent sont indépendants de cette dimension et peuvent être généralisés à toute dimension.

Le second modèle possède les caractéristiques énumérées précédemment : l'assemblage des cellules permet la description discrète et combinatoire d'espaces arbitrairement complexes de dimension quelconque. De plus, V. Shapiro précise ([Sha01] page 13) que cette approche conduit à la définition de structures de données pouvant être contrôlées localement et de façon incrémentale, facilitant de ce fait la génération de points et les modifications locales.

Dans le domaine de l'informatique graphique, en modélisation géométrique et en CAO, des représentations cellulaires spécialisées sont couramment utilisées. Citons les arêtes ailées (*winged edges*) [Bau74, Sha01], les demi-arêtes (*half-edges*) [Wei85], les tuples de cellules (*cell tuples*) [Bri89] ou encore les G-cartes [Lie91].

Dans les domaines d'application cités, les assemblages de cellules sont arbitraires. Dans d'autres domaines, ils peuvent êtres réguliers. Par exemple, en géométrie algorithmique [BK03] les assemblages sont souvent des maillages réguliers (par exemple des triangulations de surfaces). De même, en imagerie ou en géométrie discrète, on utilise souvent des assemblages réguliers (pixels ou voxels).

Dans la suite de ce chapitre, nous voulons préciser cette notion d'assemblage de cellules et son utilisation pour unifier les différents types de structures de données utilisées dans MGS pour représenter l'état d'un système dynamique et faciliter l'expression de son évolution. Nous nous plaçons donc au niveau d'un formalisme et non pas au niveau de l'implantation.

1.3 Complexe cellulaire géométrique

Les définitions suivantes sont tirées de la thèse de G. Berti [Ber01]. Les complexes cellulaires sont des objets issus de la topologie algébrique. Une formalisation très générale des complexes cellulaires est celle offerte par la notion de *CW-complexe* reposant sur la notion de *k-cellule* :

Définition 1 (*k*-cellule) *Soit X un espace de Hausdorff*[2]*. L'ensemble $c \subset X$ est une* k-cellule topologique ouverte *si c est homéomorphe à l'intérieur de la k-boule ouverte $\mathcal{B}^k = \{x \in \mathbb{R}^k \mid \|x\| < 1\}$. L'entier k est appelé la* dimension *de c.*

Un ensemble de cellules « collées » les unes aux autres de façon appropriée forme une structure appelée un *CW-complexe.*

Définition 2 (CW-complexe) *Un* CW-complexe $\mathcal{C}$ *est un système de cellules topologiques c ouvertes disjointes paire à paire, avec une topologie de Hausdorff sur $\|\mathcal{C}\| = \bigcup_{c\in\mathcal{C}} c$. Pour chaque k-cellule c, il existe une fonction caractéristique continue*

$$\Phi : \overline{\mathcal{B}^k} \longmapsto \overline{c}$$

qui est un homéomorphisme de $\mathcal{B}^k$ dans c, et qui fait correspondre $\mathbb{S}^{k-1} = \{x \in \mathbb{R}^k \mid \|x\| = 1\}$ à l'union d'un sous-ensemble fini[3] *de cellules de $\mathcal{C}$ de dimension inférieure, appelées le* bord *de c ; ce bord est donc homéomorphe à une décomposition de $\mathbb{S}^{k-1}$.*

De plus $F \in \|\mathcal{C}\|$ est fermé dans $\|\mathcal{C}\|$ ssi l'intersection de F avec chaque cellule fermée $\overline{c}$ est fermée dans c[4]*.*

[2] Un espace est *séparé* ou de *Hausdorff* lorsque deux points distincts quelconques admettent des voisinages disjoints.

[3] Cette condition correspond à la lettre C de CW-complexe, pour « *closure finite* ».

[4] Cette seconde condition correspond à la lettre W de CW-complexe, pour « *weak topology* ».

On peut trouver en appendice de [Hat02] ou dans [Mun84] une description précise des CW-complexes ainsi que du type de topologie définie à partir de ces objets. Ils sont introduits par Whitehead [Whi49] pour la mise en place d'une théorie de l'homotopie[5] ; leur utilisation est restée standard depuis. En effet, malgré une construction très générale, on peut définir des opérations sur les CW-complexes et étudier leurs propriétés. Les CW-complexes sont néanmoins peu utilisés en informatique (voir cependant [Har99]) car les espaces représentés sont trop généraux du point de vue de la modélisation géométrique ou de la visualisation graphique. Cette généralité nous est utile, car nous voulons aller au-delà de la représentation d'objets volumiques issus du monde réel et intégrer dans un même cadre toutes les structures de données présentées dans le chapitre précédent. Il faut par exemple pouvoir représenter des graphes infinis correspondant aux graphes de Cayley des GBFs. Si les CW-complexes sont suffisamment généraux pour notre objectif, ils sont cependant « trop riches ».

1.4 La géométrie en trop

Les complexes cellulaires paraissent adaptés à l'élaboration de la structure de données de MGS. Néanmoins, les aspects géométriques qu'ils mettent en avant ne correspondent pas à la structure de données que l'on souhaite créer aussi arbitraire que possible. Par exemple, elle doit être capable de représenter une solution chimique où les éléments sont des molécules d'espèces chimiques différentes pouvant réagir les unes avec les autres. En représentant les molécules par des 0-cellules et les possibilités de réaction par des 1-cellules reliant les molécules, on crée une organisation adaptée à la représentation de la solution. En revanche, qu'en est-il de la fonction caractéristique à associer à chaque cellule ? Au-delà du fait de savoir si cette fonction a un sens ou non dans notre exemple, elle n'apparaît pas comme indispensable. Le simple fait d'organiser les cellules comme mentionné ci-dessus, est suffisant pour représenter la solution. Il nous faut donc relâcher les contraintes géométriques imposées dans la définition des CW-complexes pour ne conserver que l'organisation des cellules : on définit ainsi les *complexes cellulaires abstraits*.

Dans les sections suivantes, les notions de complexes cellulaires abstraits et de chaînes topologiques sont définies pour finalement aboutir à la définition formelle des collections topologiques.

2 Complexes cellulaires abstraits

2.1 Définition

Un point important dans la définition des complexes cellulaires géométriques concerne la séparation entre l'organisation topologique donnée par l'agencement des cellules topologiques, et le plongement géométrique correspondant à la paramétrisation de chaque cellule par une fonction caractéristique. Dans sa thèse [Ber01], G. Berti propose également une notion de *complexe cellulaire abstrait* où seule importe l'organisation des cellules les unes par rapport aux autres. Elles deviennent donc des objets abstraits représentant des espaces simples dont seule la dimension est précisée. La *réalisation géométrique* fait alors le lien entre complexe abstrait et complexe géométrique : la réalisation géométrique d'un complexe cellulaire abstrait est un complexe cellulaire géométrique possédant la même organisation cellulaire.

Une définition des complexes cellulaires abstraits existe depuis 1908 ([Ste08], référence provenant de [Kov01]). Cette définition est donnée à la fois dans [Kov01] et dans [Kle00] où un

[5] Deux fonctions continues d'un espace topologique sont dites *homotopiques* si l'une peut être « déformée continûment » en l'autre. Une telle déformation est appelée *homotopie*. Intuitivement, l'étude des homotopies d'un espace complexe permet de le caractériser topologiquement : deux espaces ayant des classes d'homotopies différentes sont topologiquement différents.

historique sur la genèse de la notion de complexe cellulaire telle qu'elle est connue maintenant, est présenté.

Définition 3 (Complexe cellulaire abstrait) *Soit S un ensemble arbitraire. Soit une relation binaire $\mathcal{I} \subset S \times S$ entre les éléments de S notée de façon infixe par $\sigma \prec \tau$ où $\sigma, \tau \in S$. Soit une fonction dim : $S \to \mathbb{N}$ assignant un entier positif ou nul noté $dim(\sigma)$ à chaque élément $\sigma \in S$. Les deux axiomes suivants sont satisfaits :*

- *AC_1 :* Si $\sigma_1 \prec \sigma_2$ et $\sigma_2 \prec \sigma_3$ pour $\sigma_i \in S$, alors $\sigma_1 \prec \sigma_3$ *(transitivité)*
- *AC_2 :* Si $\sigma \prec \tau$ alors $dim(\sigma) < dim(\tau)$ *(monotonie)*

Le triplet $\mathcal{K} = (S, \mathcal{I}, dim)$ (noté également $(S, \prec, dim)$) est un complexe cellulaire abstrait. *Les éléments de S sont appelés les* cellules *du complexe $\mathcal{K}$; en particulier, si $dim(\sigma) = n$ pour $\sigma \in S$ alors n est la* dimension *de σ et σ est appelé n-*cellule *; il est également d'usage de préciser la dimension de la cellule en exposant : σ^n signifie que σ est une n-cellule. On pourra écrire $\sigma \in \mathcal{K}$ et $A \subset \mathcal{K}$ pour $\sigma \in S$ et $A \subset S$. La relation $\prec$ est appelée* relation d'incidence.

On note $\mathcal{K}_p = \{\sigma \in \mathcal{K} \mid dim(\sigma) = p\}$ où $p \in \mathbb{N}$, l'ensemble des p-cellules de $\mathcal{K}$. On dira qu'un complexe cellulaire abstrait est de dimension finie *s'il existe un entier N tel que $\forall p > N$, $\mathcal{K}_p = \emptyset$. Le plus petit entier N vérifiant cette propriété est alors appelé la* dimension *du complexe $\mathcal{K}$.*

La figure 1 de gauche présente un exemple de complexe cellulaire de dimension 2. Souvent nous utilisons le terme de sommet (ou de nœud) pour désigner des 0-cellules, et d'arc pour désigner des 1-cellules. Pour alléger les écritures nous utiliserons les deux conventions suivantes :

1. Soit $\mathcal{K}$ un complexe cellulaire abstrait, on note $S^{\mathcal{K}}$, $\prec^{\mathcal{K}}$ et $dim^{\mathcal{K}}$ les composants de $\mathcal{K}$ tels que $\mathcal{K} = (S^{\mathcal{K}}, \prec^{\mathcal{K}}, dim^{\mathcal{K}})$.
2. Une notation par extension des complexes cellulaires abstraits est également donnée. Par exemple, un complexe défini par un arc avec ses deux sommets est dénoté par extension de la manière suivante :

$$\mathcal{K} = (\{\sigma_0^0, \sigma_1^0, \sigma_2^1\}, \{\sigma_0^0 \prec \sigma_2^1, \sigma_1^0 \prec \sigma_2^1\}, \{\sigma_0^0 \mapsto 0, \sigma_1^0 \mapsto 0, \sigma_2^1 \mapsto 1\})$$

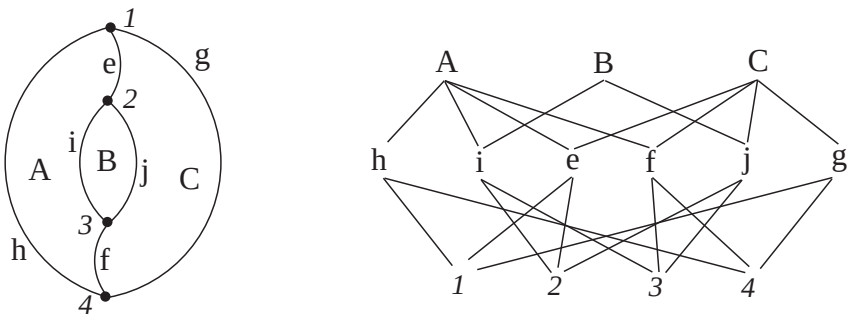


FIG. 1 – Un complexe cellulaire abstrait : sur le schéma de gauche, un complexe cellulaire abstrait composé de trois 2-cellules A, B et C, de six 1-cellules e, f, g, h, i et j, et de quatre 0-cellules, *1*, *2*, *3* et *4*. À droite, le graphe d'incidence associé au complexe cellulaire construit sur le diagramme de Hasse de l'ensemble des cellules topologiques ordonnées par la relation d'incidence. Les 1-cellules i et j sont les *faces* B ; B appartient donc aux *cofaces* de i et j. Les 2-cellules A et C sont 1-*voisines* par e et f, et 0-*voisines* par *1*, *2*, *3* et *4*.

On remarque aisément que la relation d'incidence est une relation d'ordre partiel sur les cellules topologiques. Un complexe cellulaire abstrait apparaît donc comme un ensemble partiellement ordonné de cellules topologiques muni, en plus de la relation d'ordre, d'une fonction décorant chaque cellule de sa dimension. Nous utilisons plus loin cette propriété de la relation d'incidence (voir section 2.5 page 74).

2.2 Retour à la géométrie *sans* géométrie

Cette définition des complexes cellulaires abstraits peut paraître extrêmement générale. En effet, au vu de la notion de dimension, le fait de pouvoir associer à un arc plus de deux sommets semble incohérent : on s'attend à construire un arc comme une cellule portée par deux sommets. Bien qu'on conserve par la suite cette définition sans aucune restriction, il faut tout de même signaler le concept d'*homéomorphisme combinatoire* présenté dans [Kov01] qui apporte à ce problème une solution purement combinatoire, sans refaire référence à un plongement géométrique.

Pour cette restriction, V. Kovalevsky définit les notions de *m-boule* et de *m-sphère combinatoires*. Il caractérise comme *bien définies* (*proper* en anglais) les m-cellules dont le bord est une $(m-1)$-sphère. Ces définitions étant mutuellement récursives, il utilise comme élément de base la 1-cellule bien définie : une 1-cellule est bien définie si elle est bordée par exactement deux 0-cellules. Finalement, les complexes cellulaires abstraits sont alors bien définis si toutes les cellules qui les composent le sont : bien que purement combinatoire, cette restriction réintroduit une forme de géométrie dans les complexes cellulaires abstraits permettant la représentation de variétés avec les notions de m-boule et de m-sphère combinatoires. Le lecteur intéressé pourra se référer à [Kov01] (section 2.2 page 41) pour une définition plus formelle.

Nous n'avons pas souhaité prendre en compte cette restriction de la définition des complexes cellulaires abstraits pour laisser la possibilité de créer des objets dont une réalisation géométrique aurait peu de sens. En effet, nous voulons pouvoir considérer dans le même cadre des organisations relatant des espaces topologiques et/ou géométriques réels mais également certaines organisations impliquées par des structures de données algorithmiques, n'ayant pas obligatoirement de réalité géométrique.

À partir de maintenant, nous emploierons l'expression « complexe cellulaire », voire plus simplement encore « complexe », pour complexe cellulaire abstrait.

2.3 Sous-complexes cellulaires et opérations

Afin de modifier localement l'état d'un système, on doit être capable d'en sélectionner une sous-partie. La notion de *sous-complexe cellulaire* permet cela :

Définition 4 (Sous-complexe cellulaire) *Le* sous-complexe cellulaire $\mathcal{K}' = (S', \prec', dim')$ *d'un complexe cellulaire abstrait* $\mathcal{K} = (S, \prec, dim)$ *est un complexe cellulaire abstrait contruit sur l'ensemble de cellules* $S' \subseteq S$*, dont la relation d'incidence* $\prec'$ *hérite de* $\prec$ *sur les éléments de* S'*, et dont la fonction de dimension* dim' *est la restriction de* dim *à* S'*.*

Pour assurer les modifications topologiques, nous définissons deux opérations sur les complexes cellulaires : l'*union* de deux complexes cellulaires et la *restriction* d'un complexe cellulaire par un ensemble de cellules.

L'union. L'union permet la fusion de deux complexes cellulaires. Cette opération n'est pas possible pour tout couple de complexes. En effet, l'union demande aux deux opérandes de partager les mêmes caractéristiques : par exemple, si une cellule est commune aux deux complexes,

elle doit y avoir la même dimension. De la même façon, les relations d'incidence ne doivent pas se contredire.

Définition 5 (Union de complexes) *Soient* $\mathcal{K}_1 = (S_1, \prec_1, dim_1)$ *et* $\mathcal{K}_2 = (S_2, \prec_2, dim_2)$ *deux complexes cellulaires abstraits vérifiant les conditions suivantes :*

- UC_1 *: les dimensions coïncident,* i.e. $\forall \sigma \in S_1 \cap S_2,\ dim_1(\sigma) = dim_2(\sigma)$,
- UC_2 *: les ordres sont cohérents,* i.e. $\forall \sigma, \sigma' \in S_1 \cap S_2,\ \sigma \prec_i \sigma' \Rightarrow \sigma' \not\prec_j \sigma$ *où* $i \neq j \in \{1, 2\}$.

*On définit comme l'*union $\mathcal{K}_1 \cup \mathcal{K}_2$ *de* $\mathcal{K}_1$ *et de* $\mathcal{K}_2$*, le complexe* $\mathcal{K} = (S, \prec^*, dim)$ *tel que :*

- $S = S_1 \cup S_2$,
- $\prec^*$ *est la fermeture transitive sur tout* S *de la relation* $\prec = \prec_1 \cup \prec_2$,
- $\forall \sigma \in S_1,\ dim(\sigma) = dim_1(\sigma)$ *et* $\forall \sigma \in S_2,\ dim(\sigma) = dim_2(\sigma)$.

La condition UC_2 découle naturellement de la condition UC_1, considérant AC_2 pour les complexes $\mathcal{K}_1$ et $\mathcal{K}_2$. En effet, soient deux cellules $\sigma, \tau \in S_1 \cap S_2$ telles que $\sigma \prec_1 \tau$ et $\tau \prec_2 \sigma$. Alors, AC_1 pour $\mathcal{K}_1$ implique que $dim_1(\sigma) < dim_1(\tau)$. Ceci entraîne par UC_1 que $dim_2(\sigma) < dim_2(\tau)$. Or $\tau \prec_2 \sigma$; AC_2 n'est donc pas vérifiée pour tous les éléments de $\mathcal{K}_2$ ce qui contredit le fait que $\mathcal{K}_2$ soit un complexe cellulaire abstrait. Si σ et τ sont liées par la relation d'incidence de $\mathcal{K}_2$, on a forcément $\sigma \prec_2 \tau$. Il suffit donc de vérifier UC_1 pour que l'union de deux complexes cellulaires abstraits soit définie. La figure 2 présente un exemple d'union de complexes. Les cellules topo-

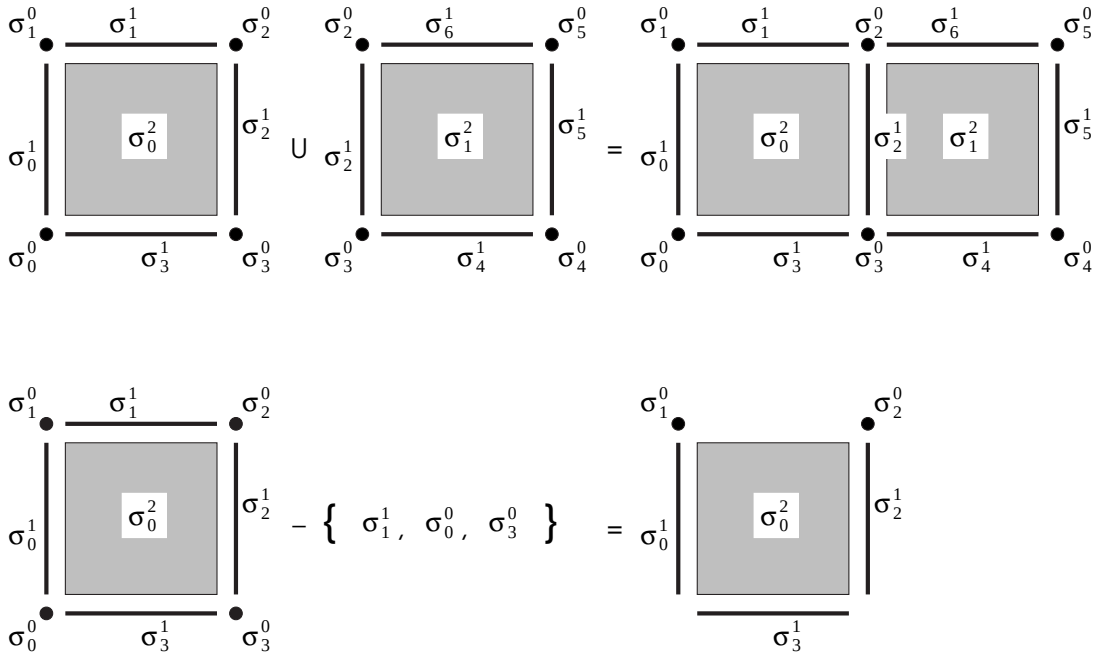


FIG. 2 – Exemple d'union et de restriction de complexe : en haut une union de deux complexes de dimension 2 partageant un arc et deux sommets, en bas une restriction d'un complexe par un ensemble $T = \{\sigma_1^1, \sigma_0^0, \sigma_3^0\}$. On constate que les cellules topologiques ont une existence propre : elles ne sont pas liées à un unique complexe cellulaire, et ne dépendent pas de l'existence de cellule incidente. Ainsi, l'arc σ_2^1 est partagé par deux complexes et l'arc σ_3^1 existe toujours après la restriction, même en l'absence des sommets σ_0^0 et σ_3^0.

logiques existent d'elles-mêmes ; elles peuvent être partagées par plusieurs complexes cellulaires abstraits.

Le théorème suivant établit que l'union de deux complexes cellulaires abstraits est toujours un complexe cellulaire abstrait.

Théorème 1 *Soient $\mathcal{K}_1 = (S_1, \prec_1, dim_1)$ et $\mathcal{K}_2 = (S_2, \prec_2, dim_2)$ deux complexes cellulaires abstraits vérifiant les conditions UC_1 et UC_2. Le complexe $\mathcal{K}_1 \cup \mathcal{K}_2$ est également un complexe cellulaire abstrait, c'est-à-dire qu'il vérifie les conditions AC_1 et AC_2.*

Preuve :

- AC_1 : trivial par la fermeture transitive $\prec^*$
- AC_2 : La condition de monotonie se vérifie facilement pour deux éléments de S_1 en utilisant la condition AC_2 de $\mathcal{K}_1$, et pour deux éléments de S_2 en utilisant AC_2 de K_2. Le dernier cas concerne la propriété de monotonie pour deux éléments, l'un de S_1 et l'autre S_2, n'appartenant pas à $S_1 \cap S_2$.
  Soient $\sigma_1 \in S_1$ et $\sigma_2 \in S_2$ tels que $\sigma_1 \prec^* \sigma_2$. Il existe alors σ_i et σ_j de $S_1 \cap S_2$ tels que :

  $$\sigma_1 \prec_1 \sigma_i \prec \sigma_j \prec_2 \sigma_2$$

  Ainsi, on a

  $$dim(\sigma_1) = dim_1(\sigma_1) < dim_1(\sigma_i) < dim_1(\sigma_j) = dim_2(\sigma_j) < dim_2(\sigma_2) = dim(\sigma_2)$$

  Le cas où $\sigma_1 \in S_2$ et $\sigma_2 \in S_1$ est symétrique. On obtient donc $\forall \sigma_1, \sigma_2 \in S, \sigma_1 \prec^* \sigma_2 \Rightarrow dim(\sigma_1) < dim(\sigma_2)$. AC_2 est donc vérifiée.

■

Restriction. La restriction est une opération qui permet de supprimer des cellules du complexe, sans toucher à la relation d'incidence entre les autres cellules. Cette opération ne nécessite aucune condition.

Définition 6 (Restriction d'un complexe) *Soient $\mathcal{K} = (S, \prec, dim)$ et $T \subset S$ respectivement un complexe cellulaire abstrait et un ensemble de cellules topologiques. On définit comme la* restriction *$\mathcal{K} - T$, le complexe $\mathcal{K}' = (S', \prec', dim')$ tel que :*

- *$S' = S - T$*
- *$\prec'$ est la restriction de $\prec$ à S'*
- *dim' coïncide avec dim sur les éléments de S'*

La figure 2 présente un exemple de restriction de complexe. Cette opération construit un nouveau complexe cellulaire abstrait. Les cellules topologiques ont une existence propre ; un arc peut exister sans les sommets qui le bordent. Cette situation n'est pas standard en topologie algébrique mais elle nous convient ici.

Théorème 2 *Soient $\mathcal{K} = (S, \prec, dim)$ et $T \subset S$ respectivement un complexe cellulaire abstrait et un ensemble de cellules topologiques, le complexe $\mathcal{K} - T$ est également un complexe cellulaire abstrait.*

Preuve : Ce théorème découle directement de la définition 6. ■

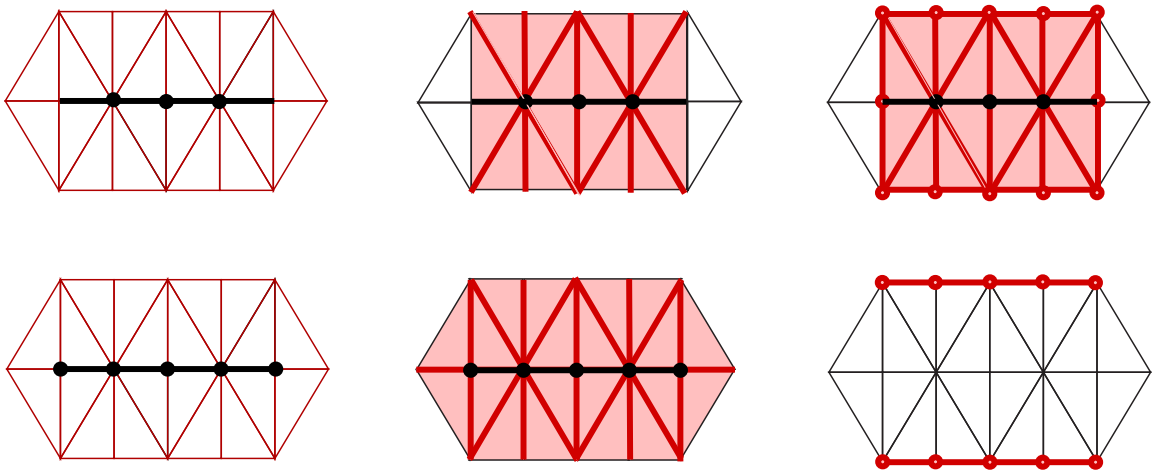


FIG. 3 – Exemples de *fermeture*, d'*étoile* et de *liaison*. Le sous-ensemble S est composé de quatre 1-cellules et de trois 0-cellules. En haut, S est représenté en gras sur le complexe cellulaire abstrait $\mathcal{K}$, puis St S et $\overline{\text{St}}$ S. En bas, $\overline{S}$, St $\overline{S}$ et finalement Lk $S = \overline{\text{St}}\ S - \text{St}\ \overline{S}$ sont représentés.

2.4 Voisinages

En combinant de façon appropriée les deux opérations précédentes, on peut modifier localement la topologie d'un complexe cellulaire. Afin de spécifier ces modifications topologiques, nous proposons d'utiliser des opérateurs de voisinages. Ils permettent de déterminer dans un complexe cellulaire comment les cellules topologiques sont liées entre elles. Le premier jeu d'opérateurs donné ci-dessous permet de déterminer à partir d'un sous-ensemble de cellules topologiques, les autres cellules topologiques du complexe qui les « entourent » au sens de la relation d'incidence. Ils permettent de déterminer les cellules touchées par une modification de la topologie par exemple.

Définition 7 (Fermeture et étoile [Axe98]) *Soit $\mathcal{K}$ un complexe cellulaire abstrait et $S \subseteq K$.*

On appelle la fermeture *de S, l'ensemble $\overline{S} = \{\sigma \in \mathcal{K} \mid \exists \tau \in S,\ \tau \preceq \sigma\}$.*

*On appelle l'*étoile *de S, l'ensemble* St $S = \{\sigma \in \mathcal{K} \mid \exists \tau \in S,\ \tau \succeq \sigma\}$. *On note la fermeture de l'étoile* $\overline{\text{St}}$ $S = \overline{\text{St}}\ \overline{S}$.

Ces opérateurs peuvent être composés pour désigner des sous-ensembles particuliers de cellules topologiques. La figure 3 (reprise de [Axe98]) décrit un exemple d'une telle composition, la *liaison*, définie par Lk $S = \overline{\text{St}}\ S - \text{St}\ \overline{S}$ (voir la thèse d'U. Axen [Axe98] pour plus de détails sur l'utilisation de la liaison dans la définition d'une notion de propagation combinatoire d'ondes).

Le second jeu d'opérateurs permet la description de sous-parties du complexe. Ils serviront à spécifier les sous-complexes à transformer. Nous définissons en particulier la notion de $\langle n, p \rangle$-*chemin*, point de départ des transformations de chemins.

Définition 8 (Relation de voisinage) *Soit $\mathcal{K}$ un complexe cellulaire, $\sigma, \tau \in \mathcal{K}$. On dit que σ et τ sont* incidentes, *notée $\sigma \gtrless \tau$, si*

$$(\sigma = \tau) \vee (\sigma \prec \tau) \vee (\tau \prec \sigma)$$

En particulier, on appelle τ une face *de σ si elle vérifie :*

$$dim(\tau) = dim(\sigma) - 1$$

on dit alors que σ est une coface *de τ et on écrit $\tau < \sigma$.*

Deux cellules σ_1 et σ_2 de $\mathcal{K}$ sont connectées*, et on écrit*

$$\sigma_1, \sigma_2$$

*s'il existe une cellule τ de $\mathcal{K}$ telle que σ_1 et σ_2 appartiennent à $\overline{\mathrm{St}}\ \{\tau\}$. On définit alors naturellement le $\langle n, p\rangle$-*voisinage*, une restriction de la connectivité sur les dimensions des cellules : on dira que deux n-cellules σ_1 et σ_2 sont $\langle n, p\rangle$-*voisines*, relation notée $\sigma_1 {}^{n}_{p}\sigma_2$, si*

$$\exists \tau \in \mathcal{K},\ (dim(\tau) = p) \wedge \begin{cases} (\sigma_1 \prec \tau) \wedge (\sigma_2 \prec \tau) & si\ n < p \\ (\tau \prec \sigma_1) \wedge (\tau \prec \sigma_2) & sinon \end{cases}$$

*On appelle $\langle n, p\rangle$-*chemin*, une séquence de n-cellules $\langle n, p\rangle$-voisines. La valeur de n étant donnée par les dimensions des cellules, on pourra écrire « $,_p$ » pour « $,^n_p$ ».*

On définit quatre opérations sur les cellules :

1. *l'ensemble des faces :* `faces` $\sigma = \{\tau \in \mathcal{K} \mid \tau < \sigma\}$,
2. *l'ensemble des cofaces :* `cofaces` $\sigma = \{\tau \in \mathcal{K} \mid \sigma < \tau\}$,
3. *l'ensemble des cellules incidentes :* `icells` $\sigma = \{\tau \in \mathcal{K} \mid \sigma \geqslant \tau\}$,
4. *l'ensemble des p-voisines :* $\texttt{pcells}_p$ $\sigma = \{\tau \in \mathcal{K} \mid \sigma,_p \tau\}$.

Par exemple, sur la figure 1, on a :

- `faces` B $= \{\mathrm{i}, \mathrm{j}\}$,
- `cofaces` B $= \emptyset$,
- `icells` B $= \{\mathrm{A}, \mathrm{C}, \mathrm{i}, \mathrm{j}, 2, 3\}$,
- $\texttt{pcells}_0$ B $= \{\mathrm{A}, \mathrm{C}\}$ par *2* et *3*,
- $\texttt{pcells}_1$ B $= \{\mathrm{A}, \mathrm{C}\}$ par i et j.

2.5 *Poset* et graphe d'incidence

Pour terminer cette section consacrée aux complexes cellulaires abstraits, nous en définissons une représentation graphique commode : le *graphe d'incidence*. Cette représentation est fondée sur le diagramme de Hasse, une représentation des ensembles partiellement ordonnés.

Définition 9 (Poset) *Un* ensemble partiellement ordonné*, ou* poset *(pour* partially ordered set*), est un ensemble muni d'une relation d'ordre partiel (*i.e. *réflexive, transitive et antisymétrique).*

On peut montrer que la relation d'incidence définit un ordre partiel sur l'ensemble des cellules.

Théorème 3 *Soient $\mathcal{K} = (S, \prec, dim)$ un complexe cellulaire abstrait, $\sigma, \tau \in \mathcal{K}$. La relation $\preceq$ définie par*

$$\sigma \preceq \tau \Leftrightarrow (\sigma \prec \tau) \vee (\sigma = \tau)$$

est une relation d'ordre partiel sur les éléments de $\mathcal{K}$. $(S, \preceq)$ possède donc une structure de poset*.*

Preuve :

- réflexivité : par définition de $\preceq$,

- transitivité : héritée de AC_1,
- antisymétrie : soient $\sigma, \tau \in \mathcal{K}$ avec $\sigma \preceq \tau$ et $\tau \preceq \sigma$. Par AC_2, on sait que $dim(\sigma) \leq dim(\tau)$ et $dim(\tau) \leq dim(\sigma)$, donc $dim(\sigma) = dim(\tau)$. Or si $\sigma \prec \tau$ (resp. $\sigma \succ \tau$), alors $dim(\sigma) < dim(\tau)$ (resp. $dim(\sigma) > dim(\tau)$), ce qui n'est pas possible. On déduit alors que $\sigma = \tau$.

■

On utilise un diagramme de Hasse pour représenter graphiquement un complexe cellulaire.

Définition 10 (Diagramme de Hasse, graphe d'incidence) *Un* diagramme de Hasse *est une représentation graphique d'un* poset. *Un point est dessiné pour chaque élément du* poset *et un arc joint deux sommets en respectant les lois de construction suivantes :*

1. *Si $x \prec y$ appartiennent au* poset, *alors le point correspondant à x apparaît plus bas que le point correspondant à y.*
2. *Un arc est dessiné entre deux points correspondant aux éléments x et y si $x \prec y$ et s'il n'existe pas d'élément z tel que $x \prec z \prec y$ ou $y \prec z \prec x$.*

Soit $\mathcal{K} = (S, \prec, dim)$ un complexe cellulaire abstrait. Le graphe d'incidence *de $\mathcal{K}$ est le diagramme de Hasse du* poset $(S, \preceq)$.

À droite de la figure 1 est donné le graphe d'incidence du complexe cellulaire abstrait de gauche.

3 Complexe de chaîne

3.1 Chaîne topologique

La figure 4 montre que la structure décrite par un complexe cellulaire abstrait n'est pas suffisamment expressive pour représenter toutes les façons dont les cellules peuvent être connectées : deux complexes topologiquement différents peuvent partager le même graphe d'incidence [Ber01]. En fait, la topologie d'une cellule n'est pas complètement décrite par la liste de ses cellules incidentes : on doit également préciser comment celles-ci sont organisées. La notion d'orientation

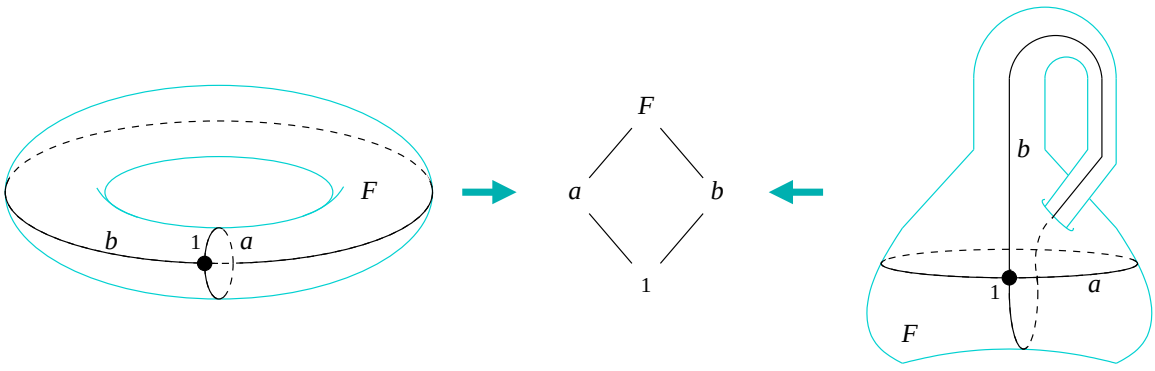


FIG. 4 – Tore et bouteille de Klein : à gauche, un complexe cellulaire représentant un tore, à droite une bouteille de Klein ; les traits clairs correspondent aux 2-cellules. Au centre, les deux complexes partagent le même graphe d'incidence mais correspondent à deux objets topologiquement différents.

peut par exemple pallier ce manque[6] : sur la figure 4, le tore est orientable alors que la bouteille de Klein ne l'est pas. La notion de *chaîne topologique* permet la description de l'organisation des cellules incidentes : informellement, une chaîne est un ensemble de cellules muni d'une structure de groupe abélien et d'un opérateur de bord. La structure de groupe décrit comment les cellules sont assemblées à l'aide de sa loi (notée additivement), et l'opérateur de bord fournit une chaîne décrivant le bord d'une cellule (c'est-à-dire les cellules incidentes de dimension inférieure) et, par extension, le bord d'une chaîne.

Soient σ et σ' deux cellules ; assembler σ et σ' signifie définir la chaîne $c = \sigma + \sigma'$ (ou $c = \sigma' + \sigma$, l'ordre n'ayant pas d'importance dans un groupe abélien). Le neutre 0 du groupe correspond à l'absence de cellule. L'opération d'ajout d'une cellule σ'' à une chaîne c étant décrite par l'opérateur d'addition $c+\sigma''$, on utilise l'inverse de σ'' pour en dénoter la suppression $c - \sigma''$. Ceci justifie la structure de groupe. Enfin, l'un des objectifs théoriques des chaînes est également de calculer le bord d'un espace (représenté aussi par une chaîne) en fonction du calcul du bord isolé de chacune des cellules qui le composent. Il est alors naturel de définir l'opérateur ∂ comme un homomorphisme de groupe : $\partial(\sigma + \sigma') = \partial(\sigma) + \partial(\sigma')$. Ces considérations motivent les définitions suivantes :

Définition 11 (Chaîne topologique, p-chaîne) *Soient $\mathcal{K}$ un complexe cellulaire abstrait, p un entier et G un groupe abélien arbitraire noté additivement ($+_G$). Le neutre de G est noté 0_G. L'ensemble $C_p(\mathcal{K}, G)$ dénote l'ensemble des p-*chaînes *de dimension p du complexe $\mathcal{K}$ à valeur dans G comme l'ensemble des fonctions totales c_p de $\mathcal{K}_p$ dans G nulles presque partout, c'est-à-dire $c_p(\sigma) \neq 0_G$ pour un nombre fini de $\sigma \in \mathcal{K}_p$.*

On définit l'addition $+_{C_p(\mathcal{K},G)}$ entre deux chaînes c_1 et c_2 de $C_p(\mathcal{K}, G)$, l'opération qui construit une nouvelle chaîne de $C_p(\mathcal{K}, G)$ telle que, quelque soit $\sigma \in \mathcal{K}_p$:

$$(c_1 +_{C_p(\mathcal{K},G)} c_2)(\sigma) = c_1(\sigma) +_G c_2(\sigma)$$

Intuitivement, l'ensemble $C_p(\mathcal{K}, G)$ représente l'ensemble de toutes les façons possibles d'assembler les p-cellules de $\mathcal{K}$ selon les degrés de liberté offerts par le groupe G. Une p-chaîne peut être représentée par une somme formelle finie ; soit $c_p \in C_p(\mathcal{K}, G)$:

$$c_p = \sum_{\sigma \in \mathcal{K}_p} c_p(\sigma).\sigma$$

Par convention, l'écriture $c_p = \alpha_1\sigma_1 + \cdots + \alpha_n\sigma_n$ définit une p-chaîne de $C_p(\mathcal{K}, G)$ (avec $\alpha_i \in G - \{0_G\}$ et $i \neq j \Rightarrow \sigma_i \neq \sigma_j$).

Théorème 4 *L'ensemble $C_p(\mathcal{K}, G)$ muni de la loi $+_{C_p(\mathcal{K},G)}$ est un groupe abélien.*

Preuve :

1. $+_{C_p(\mathcal{K},G)}$ est associatif, par associativité de $+_G$,
2. $+_{C_p(\mathcal{K},G)}$ est commutatif, par commutativité de $+_G$,
3. soit $0_{C_p(\mathcal{K},G)} \in C_p(\mathcal{K}, G)$ telle que

$$\forall \sigma \in \mathcal{K}_p, 0_{C_p(\mathcal{K},G)}(\sigma) = 0_G$$

$0_{C_p(\mathcal{K},G)}$ est neutre pour $+_{C_p(\mathcal{K},G)}$.

[6] bien que ça ne soit pas toujours le cas. Nous avons choisi cette représentation parce qu'elle est proche de notre formalisme de calcul et que ses limitations n'interviennent pas dans nos travaux.

4. Chaque p-chaîne c_p admet un inverse (appelé ici opposé, la loi étant additive) noté $-_{C_p(\mathcal{K},G)}c_p$, défini par :
$$\forall \sigma \in \mathcal{K}_p, (-_{C_p(\mathcal{K},G)}c_p)(\sigma) = -_G c_p(\sigma)$$

■

3.2 Groupe de chaînes à coefficients dans un groupe abélien arbitraire G

Définition 12 (Groupe des chaînes, complexe de chaînes) *Soient $\mathcal{K}$ un complexe cellulaire abstrait de dimension N, et G un groupe abélien arbitraire. Le* groupe des chaînes *sur $\mathcal{K}$ à coefficients dans G est défini par :*

$$C(\mathcal{K}, G) = \bigoplus_{i\in\mathbb{N}} C_i(\mathcal{K}, G) = C_0(\mathcal{K}, G) \oplus C_1(\mathcal{K}, G) \oplus \cdots \oplus C_N(\mathcal{K}, G) \oplus \ldots$$

où $\oplus$ est la somme directe de groupes abéliens.

Un complexe de chaînes *$C(\mathcal{K}, G, \partial)$ est une suite $(C_p(\mathcal{K}, G), \partial_p)_{p\in\mathbb{N}}$ de groupes abéliens et d'homomorphismes $\partial_p : C_p(\mathcal{K}, G) \to C_{p-1}(\mathcal{K}, G)$*

Les groupes $C_i(\mathcal{K}, G)$ où $\mathcal{K}_i = \emptyset$ (parce que $i > N$, la dimension de $\mathcal{K}$ par exemple) sont des *groupes triviaux*, c'est-à-dire des groupes réduits à leur élément neutre.

On dira qu'un complexe de chaîne est de dimension finie si les $C_p(\mathcal{K}, G)$ sont triviaux excepté pour un nombre fini de p.

La définition habituelle de la notion de complexe de chaîne requiert que $\partial_{p-1} \circ \partial p = 0$, signifiant intuitivement qu'un bord n'a pas de bord (le bord d'un volume est une surface fermée par exemple). Nous n'imposons pas cette condition ici pour les mêmes raisons qui nous ont poussés à étendre les complexes abstraits vers une définition indépendante de la géométrie.

L'usage est d'utiliser p en indice pour indiquer qu'une chaîne c est une p-chaîne : c_p. En revanche, pour simplifier les notations, l'indice p des homomorphismes de bord ∂_p sera souvent omis, laissant le contexte préciser les dimensions mises en jeu. Les ensembles $C_p(\mathcal{K}, G)$ et $C(\mathcal{K}, G, \partial)$ sont abrégés respectivement en C_p et C lorsque le contexte le permet. En ce qui concerne les notations des groupes, l'indice porté par les opérateurs et les neutres précisant le groupe auquel ils font référence, sera omis si le contexte suffit à déterminer la nature du groupe.

Exemple du complexe de chaînes $C(\mathcal{K}, \mathbb{Z}/2\mathbb{Z}, \partial)$. $\mathbb{Z}/2\mathbb{Z}$ dénote l'ensemble des entiers modulo 2. L'utilisation de $\mathbb{Z}/2\mathbb{Z}$ pour les coefficients de chaîne autorise la représentation de la présence, $c_p(\sigma) = 1$, ou de l'absence, $c_p(\sigma) = 0$, d'une p-cellule σ dans une chaîne c_p. Une

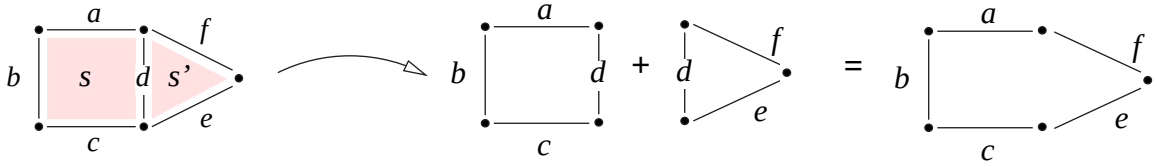


FIG. 5 – Complexe de chaînes dans $C(\mathcal{K}, \mathbb{Z}/2\mathbb{Z}, \partial)$: à gauche le complexe cellulaire $\mathcal{K}$ est composé de deux 2-cellules, l'une carrée s et l'autre triangulaire s', et de leurs bords. À droite le calcul de la chaîne $\partial_2(s + s')$ de $C(\mathcal{K}, \mathbb{Z}/2\mathbb{Z})$. La 1-cellule d apparaissant dans les deux sous-ensembles $\partial_2(s)$ et $\partial_2(s')$ disparaît.

chaîne de $C(\mathcal{K}, \mathbb{Z}/2\mathbb{Z})$ est alors simplement la fonction caractéristique d'un sous-ensemble de $\mathcal{K}$. Un exemple est donné figure 5. Une chaîne $c = e + f$ correspond à la fonction c définie par $c(e) = c(f) = 1$ et $c(\sigma) = 0$ pour $\sigma \neq e$ et $\sigma \neq f$. Cette chaîne peut aussi être écrite $c = 1.e + 1.f + 0.g + 0.h + \ldots$ mais il est commode de ne pas écrire les p-cellules à coefficient nul en accord avec la notation additive. On obtient alors $c = 1.e + 1.f$, ou de manière plus ambiguë $c = e + f$.

Pour transformer le groupe de chaînes $C(\mathcal{K}, \mathbb{Z}/2\mathbb{Z})$ en un complexe de chaînes $C(\mathcal{K}, \mathbb{Z}/2\mathbb{Z}, \partial)$, on définit dans un premier temps l'opérateur de bord sur chaque p-cellule σ^p de $\mathcal{K}$:

$$\partial_p(1.\sigma^p) = \sum_{\sigma_j^{p-1} < \sigma^p} 1.\sigma_j^{p-1}$$

que nous étendons par linéarité à toute p-chaîne $c \in C_p(\mathcal{K}, \mathbb{Z}/2\mathbb{Z})$:

$$\partial_p(c) = \sum_{\sigma_i^p} \sum_{\sigma_i^p < \sigma_j^{p-1}} c(\sigma_i^p).\sigma_j^{p-1}$$

Suivant l'exemple de la figure 5, supposons que $c = s + s'$. La 1-cellule d n'est pas dans le bord de c puisque s et s' sont collées suivant d ; d est une cellule intérieure. En revanche d appartient à la fois au bord de s et à celui de s'. Soit $\partial_p s = d + \sum \sigma'_j$ et $\partial_p s' = d + \sum \sigma''_k$. Alors nous devons avoir $d + \sum \sigma'_j + d + \sum \sigma''_k = \sum \sigma'_j + \sum \sigma''_k$, ce qui est *automatiquement* obtenu car $d + d = 2.d = 0$ dans $C_p(\mathcal{K}, \mathbb{Z}/2\mathbb{Z})$.

Exemple du complexe de chaînes $C(\mathcal{K}, \mathbb{Z}, \partial)$. Le groupe de chaînes entières $C(\mathcal{K}, \mathbb{Z})$ fournit une algèbre permettant de compter les cellules topologiques, sans la restriction « pair/impair » des coefficients de $\mathbb{Z}/2\mathbb{Z}$. Soient les complexes de la figure 6 : la 1-chaîne $2a + 1b - 3e$ de $C(\mathcal{K}, \mathbb{Z})$ comptent la 1-cellule a deux fois, la 1-cellule b une fois et la 1-cellule e moins trois fois.

Les coefficients négatifs permettent la prise en compte de l'orientation des cellules. Ainsi, l'orientation relative des cellules topologiques de la figure 6 amène à définir l'opérateur de bord ∂ de telle sorte que $\partial d = 4 - 2$ et $\partial s = a + b - c$. La linéarité de l'opérateur profite une fois de plus des propriétés du groupe $\mathbb{Z}$ pour supprimer les cellules comptées positivement et négativement un même nombre de fois, ce qui se révèle utile pour manipuler les bords. Ainsi, $\partial(s + s') = (a + b - c) + (d - e - b) = a - c + d - e$: la cellule b est supprimée automatiquement par le calcul. Sur l'exemple de droite en revanche, les cellules sont positivement orientées du bas vers le haut (relativement à la représentation). Cela permet de compter le nombre de fois où une cellule est utilisée dans la construction du bord : $\partial(v + w) = (j - f - e + d) + (g - h - i + d) = j - f - e + g - h - i + 2d$.

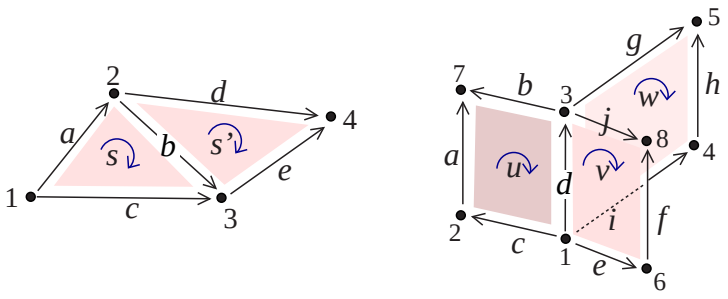


FIG. 6 – Complexe de chaînes dans $C(\mathcal{K}, \mathbb{Z}, \partial)$: l'utilisation de coefficients permet de compter les cellules topologiques. Les coefficients négatifs expriment alors l'orientation des cellules.

Valeurs arbitraires et transport de valeurs. Que ce soit le groupe $\mathbb{Z}$, le groupe $\mathbb{Z}/2\mathbb{Z}$ comme nous venons de le voir, ou encore un groupe abélien quelconque G, les chaînes permettent d'associer un élément de ce groupe à des cellules (en nombre fini). Nous cherchons maintenant à associer les éléments d'un ensemble arbitraire de valeurs Val aux cellules topologiques. L'étiquetage des cellules topologiques d'un complexe cellulaire abstrait peut être représenté par une fonction partielle $l : \mathcal{K} \to \text{Val}$. Cette fonction peut être étendue en une fonction totale en associant un élément $\perp$, tel que $\perp \notin \text{Val}$, à toutes les cellules n'ayant pas d'image par l. Cependant la fonction l ne possède pas la structure de groupe nécessaire à la définition d'un groupe de chaînes.

L'ensemble initial Val n'est pas naturellement muni d'une structure de groupe. Pour pallier ce défaut de structure, nous ne considérons plus l'ensemble Val mais l'ensemble Abel(Val). Soit un ensemble arbitraire E auquel on souhaite apporter une structure de groupe abélien. On définit Abel(E) par la présentation de groupe [Mic96] suivante :

$$\text{Abel}(E) = \langle E \ ; \ \{a +_{\text{Abel}(E)} b = b +_{\text{Abel}(E)} a, \forall (a,b) \in E^2\} \rangle$$

Abel(E) est le groupe abélien libre finiment engendré par les éléments de E. Le groupe Abel(Val) a une structure plus riche que celle de Val et autorise l'association à une cellule topologique d'un *multi-ensemble généralisé* d'éléments de Val. Dans un multi-ensemble généralisé, un élément peut avoir une multiplicité négative. On peut alors voir Abel(V) comme les fonctions totales de V dans $\mathbb{Z}$ nulles presque partout. Là encore, les éléments de Abel(Val) peuvent être représentés par des sommes formelles ; soit $f \in \text{Abel(Val)}$:

$$f = \sum_{v \in \text{Val}} f(v).v$$

Pour alléger les écritures, les fonctions $1.v \in \text{Abel(Val)}$ seront simplement dénotées v. Nous utilisons ici l'injection canonique Inj_{Val} qui envoie les éléments de Val sur les éléments de Abel(Val) :

$$\begin{array}{rccc} \text{Inj}_{\text{Val}} : & \text{Val} & \to & \text{Abel(Val)} \\ & \sigma & \mapsto & 1.\sigma \end{array}$$

L'élément $\perp$ mis en avant précédemment correspond alors au neutre du groupe Abel(Val), noté $0_{\text{Abel(Val)}}$. Il est intéressant de noter que si Val possède déjà une structure de groupe, la loi Abel(Val) $+_{\text{Abel(Val)}}$ ne coïncide pas avec la loi du groupe Val. Par exemple, en prenant $\text{Val} = \mathbb{Z}$, on a $x +_{\text{Abel}(\mathbb{Z})} (-x) \neq 0_{\text{Abel}(\mathbb{Z})}$. En effet, les deux éléments x et $-x$ sont des générateurs de Abel($\mathbb{Z}$) et sont par conséquent indépendants (ce qui n'est pas le cas dans $\mathbb{Z}$ où ils sont opposés).

En étiquetant les cellules d'un complexe cellulaire abstrait de cette façon, l'opération de bord ∂ peut être interprétée comme effectuant une opération de *transport* (voir [Ton74, Ton76, PS93]). L'opérateur de bord sur une cellule σ est défini par :

$$\partial\sigma = \sum_{\tau < \sigma} \tau \qquad \text{et on étend linéairement } \partial \text{ par : } \partial\left(\sum_{\sigma} \alpha_\sigma \sigma\right) = \sum_{\sigma} \alpha_\sigma \partial\sigma$$

En considérant que σ est une cellule ayant plusieurs faces, l'effet de l'opérateur ∂ ainsi défini est un transport d'informations envoyant le coefficient de σ à chacune de ses faces τ. Le résultat de ce calcul est conservé dans une somme formelle grâce aux propriétés d'Abel(Val), et aucun coefficient n'est perdu. Cette somme formelle peut être interprétée comme « la collision de valeurs transportées en une cellule τ ». Un homomorphisme d'Abel(Val) dans lui-même permet alors de résoudre ces conflits, le résultat de l'application étant finalement attribué comme coefficient de

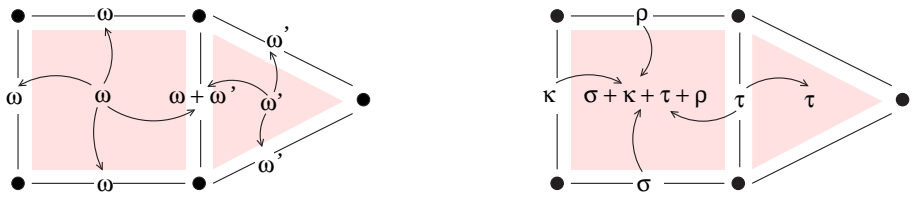


FIG. 7 – Transport de valeurs à l'aide du groupe Abel(Val) : le complexe cellulaire sous-jacent est celui de la figure 5. À droite, l'opérateur de bord est utilisé pour transporter les valeurs des 2-cellules vers leurs faces. À gauche, un opérateur dual est utilisé pour le transport vers les cofaces. Cet opérateur est présenté dans le chapitre 6 et dans [Mun84].

τ. Cette opération est similaire aux opérations de communication considérées dans les langages à parallélisme de données [Hil85]. La figure 7 de gauche propose un exemple concret d'utilisation des transports de valeurs. Supposons que les cellules du complexe sont décorées par des réels (on considère par conséquent les chaînes de $C(\mathcal{K}, \text{Abel}(\mathbb{R}), \partial)$). Par exemple, prenons $\omega = 1.6$ et $\omega' = 3.1$ dans la chaîne $\omega.s + \omega'.s'$:

$$\partial(1.6s + 3.1s') = 1.6a + 1.6b + 1.6c + (1.6 +_{\text{Abel}(\mathbb{R})} 3.1)d + 3.1e + 3.1f$$

Les valeurs 1.6 de s et 3.1 de s' se sont rencontrées en d. On souhaite maintenant combiner ces deux valeurs pour obtenir de nouveau un réel. On utilise alors l'homomorphisme $h : \text{Abel}(\mathbb{R}) \rightarrow (\mathbb{R}, +_{\mathbb{R}})$ qui interprète la somme $+_{\text{Abel}(\mathbb{R})}$ comme la somme réelle $+_{\mathbb{R}}$. On étend facilement l'homomorphisme de groupe h à un homomorphisme de chaîne en définissant $h(\alpha.\sigma) = \alpha \times h(\sigma)$ pour toutes cellules σ et en utilisant la linéarité. La somme $(1.6 +_{\text{Abel}(\mathbb{R})} 3.1)$ est alors combinée en la valeur $h(1.6 +_{\text{Abel}(\mathbb{R})} 3.1) = 1.6 +_{\mathbb{R}} 3.1 = 4.7$. Un autre choix de résolution des collisions aurait pu être pris : au lieu d'utiliser l'homomorphisme h, il est également possible de travailler directement dans $C(\mathcal{K}, (\mathbb{R}, +_{\mathbb{R}}), \partial)$. La fonction de collision résolvant les rencontres de coefficients devient directement la loi du groupe des coefficients de la chaîne. Néanmoins, l'utilisation d'Abel($\mathbb{R}$) puis d'un homomorphisme h défini *a posteriori* est plus générale. Par exemple, il n'est pas facile d'exprimer une multiplication comme fonction de collision en travaillant avec des coefficients dans $(\mathbb{R}, +_{\mathbb{R}})$. Alors qu'en utilisant Abel($\mathbb{R}$) en premier lieu, il suffit de changer l'homomorphisme h pour obtenir une multiplication des valeurs. La fonction de collision ne doit pas dépendre de l'ordre des recombinaisons, ce qui motive l'utilisation de groupes *abéliens*.

4 Collection topologique

Intuitivement, une collection topologique est une structure de données dont l'organisation est fondée sur un complexe cellulaire abstrait décoré par des valeurs. L'ensemble des valeurs est celui donné par le langage. La sémantique du chapitre suivant en donne une définition formelle. Pour cette section, nous considérons un ensemble arbitraire Val de valeurs à associer à chaque cellule d'un complexe cellulaire. Dans un premier temps, nous voyons comment construire cette association en terme de chaîne topologique à valeurs arbitraires, pour finalement définir les collections topologiques. La seconde partie de cette section est consacrée à la description des structures de données standards en termes de collections topologiques.

4.1 Définition formelle

4.1.1 Valeurs arbitraires et chaînes à coefficients dans un groupe G

Le calcul du bord d'un complexe et le calcul du transport de valeurs entre les cellules demandent l'utilisation de deux complexes de chaînes différents : l'un pour associer les valeurs d'un groupe abélien G pour le calcul topologique du bord, et l'autre pour associer les éléments du groupe abélien Abel(Val) pour associer des valeurs arbitraires. Il est possible de fusionner ces deux décorations en une seule, à travers l'utilisation du produit cartésien $\text{Abel(Val)} \times G$; ce produit présente également une structure de groupe abélien. Il est néanmoins mal adapté pour être utilisé tel quel. En effet, soit v et g deux éléments respectifs de Abel(Val) et de G, les éléments $(v, 0_G)$ et $(0_{\text{Abel(Val)}}, g)$ appartiennent à ce produit cartésien. Ces deux couples sont construits sur les éléments neutres des deux groupes ; dans notre contexte, cela signifie que lors de l'application de l'opérateur ∂, la cellule portant de telles valeurs doit être considérée comme absente, c'est-à-dire décorée par le neutre du produit cartésien $(0_{\text{Abel(Val)}}, 0_G)$. En d'autre terme, on voudrait satisfaire l'équation suivante :

$$(v, 0_G) = (0_{\text{Abel(Val)}}, g) = (0_{\text{Abel(Val)}}, 0_G)$$

J. Munkres [Mun84] définit pour cela le produit tensoriel de deux groupes :

Définition 13 (Produit tensoriel) *Soit A et B deux groupes abéliens. Soit $F(A, B)$ le groupe abélien engendré par l'ensemble $A \times B$. Soit $R(A, B)$, le sous-groupe de $F(A, B)$ de tous les éléments de la forme :*

$$(a + a', b) - (a, b) - (a', b)$$
$$(a, b + b') - (a, b) - (a, b')$$

pour $a, a' \in A$ et $b, b' \in B$. On définit

$$A \otimes B = F(A, B)/R(A, B)$$

appelé le produit tensoriel *de A et de B. La classe du couple $(a, b) \in A \times B$ est dénotée $a \otimes b$.*

Nous utilisons donc le groupe abélien $\text{Abel(Val)} \otimes G$ pour construire les collections topologiques.

Il n'est pas évident de « comprendre » le sens des éléments de $\text{Abel(Val)} \otimes G$. Voici une représentation alternative : nous les identifions aux éléments de $\text{Abel}_G(\text{Val})$, l'ensemble des fonctions totales de V dans G nulles presque partout[7]. Les éléments de $\text{Abel}_G(\text{Val})$ peuvent également être représentés par des sommes finies formelles ; soit $f \in \text{Abel}_G(\text{Val})$,

$$f = \sum_{v \in V} f(v).v$$

Nous définissons l'addition des éléments de $\text{Abel}_G(\text{Val})$; soit $f, f' \in \text{Abel}_G(\text{Val})$,

$$f +_{\text{Abel}_G(\text{Val})} f' = \sum_{v \in V} (f(v) + f'(v)).v$$

Théorème 5 $\text{Abel}_G(\text{Val})$ *muni de son addition $+_{\text{Abel}_G(\text{Val})}$, possède une structure de groupe commutatif.*

Preuve :

- $+_{\text{Abel}_G(\text{Val})}$ est associatif, par associativité de $+_G$,

[7] Notons en particulier que $\text{Abel(Val)} = \text{Abel}_{\mathbb{Z}}(\text{Val})$.

- $+_{\mathrm{Abel}_G(\mathrm{Val})}$ est commutatif, par commutativité de $+_G$,
- soit $0_{\mathrm{Abel}_G(\mathrm{Val})} \in \mathrm{Abel}_G(\mathrm{Val})$ telle que

$$\forall v \in V, \quad 0_{\mathrm{Abel}_G(\mathrm{Val})}(v) = 0_G$$

  $0_{\mathrm{Abel}_G(\mathrm{Val})}$ est neutre pour $+_{\mathrm{Abel}_G(\mathrm{Val})}$.
- Chaque fonction f admet un inverse noté $-_{\mathrm{Abel}_G(\mathrm{Val})}f$, défini par :

$$\forall v \in V, \quad (-_{\mathrm{Abel}_G(\mathrm{Val})}f)(v) = -_G f(v)$$

■

On peut alors montrer le théorème suivant, identifiant les éléments de $\mathrm{Abel}_G(\mathrm{Val})$ à ceux de $\mathrm{Abel}(\mathrm{Val}) \otimes G$.

Théorème 6 *Il existe un isomorphisme*

$$\mathrm{Abel}(\mathrm{Val}) \otimes G \simeq \mathrm{Abel}_G(\mathrm{Val})$$

qui envoie $(\sum n.v) \otimes g$ *sur* $\sum (ng).v$.

Preuve : Cette preuve est inspirée de celle du théorème 50.2 de J. Munkres [Mun84].

Le point de départ de cette preuve est que pour toute forme bilinéaire de $A \times B$ dans C (A, B et C étant des groupes abéliens), il existe un homomorphisme de $A \otimes B$ dans C. Ainsi, la fonction de $\mathrm{Abel}(\mathrm{Val}) \times G$ dans $\mathrm{Abel}_G(\mathrm{Val})$ envoyant $((\sum n.v), g)$ sur $\sum (ng).v$, étant une forme bilinéaire, induit un homomorphisme ϕ tel que :

$$\begin{array}{rccc} \phi : & \mathrm{Abel}(\mathrm{Val}) \otimes G & \to & \mathrm{Abel}_G(\mathrm{Val}) \\ & (\sum n.v) \otimes g & \mapsto & \sum (ng).v \end{array}$$

Soit ψ :

$$\begin{array}{rccc} \psi : & \mathrm{Abel}_G(\mathrm{Val}) & \to & \mathrm{Abel}(\mathrm{Val}) \otimes G \\ & \sum g.v & \mapsto & \sum 1.v \otimes g \end{array}$$

On montre trivialement que ψ est un homomorphisme (linéarité et envoi du neutre de $\mathrm{Abel}_G(\mathrm{Val})$ sur celui de $\mathrm{Abel}(\mathrm{Val}) \otimes G$).

Soit $\sum g.v \in \mathrm{Abel}_G(\mathrm{Val})$, on a :

$$\begin{array}{rcl} \phi\psi(\sum g.v) & = & \phi\left(\sum 1.v \otimes g\right) \\ & = & \sum \phi\left(1.v \otimes g\right) \\ & = & \sum \left(1g\right).v \\ & = & \sum g.v \end{array}$$

De la même façon, soit $(\sum n.v) \otimes g \in \mathrm{Abel}(\mathrm{Val}) \otimes G$, on a :

$$\begin{array}{rcl} \psi\phi((\sum n.v) \otimes g) & = & \psi\left(\sum (ng).v\right) \\ & = & \sum \left(1.v \otimes (ng)\right) \\ & = & \sum \left(n.v \otimes g\right) \\ & = & (\sum n.v) \otimes g \end{array}$$

Les deux homorphismes ϕ et ψ sont donc inverses. ϕ est donc un isomorphisme. ■

4.1.2 Collections topologiques et chaînes topologiques

Nous pouvons désormais définir formellement les collections topologiques :

Définition 14 (Collection topologique) *Un* type de collection topologique *est un quintuplet*

$$\mathcal{T}(\mathcal{K}, G, \partial, \mathrm{Val}, pred)$$

tel que :

- $\mathcal{K}$ *est un complexe cellulaire abstrait de dimension finie,*
- $C(\mathcal{K}, G, \partial)$ *est un complexe de chaînes, appelé la* forme *des collections de type* $\mathcal{T}$*,*
- Val *est un ensemble arbitraire de valeurs,*
- *pred est un prédicat désignant un sous-ensemble de* $C(\mathcal{K}, \mathrm{Abel}_G(\mathrm{Val}))$*.*

Une collection topologique *est un couple* $(\mathcal{T}, c)$ *où* $\mathcal{T}$ *est un type de collection topologique* $\mathcal{T}(\mathcal{K}, G, \partial, \mathrm{Val}, pred)$ *et* c *est une chaîne de* $C(\mathcal{K}, \mathrm{Abel}_G(\mathrm{Val}))$ *telle que* $pred(c)$ *est vrai (*c *est dans le sous-ensemble de* $C(\mathcal{K}, \mathrm{Abel}_G(\mathrm{Val}))$ *caractérisé par pred).*

Lorsque le contexte est suffisamment clair, le type de collection n'est pas précisé ; on dit alors qu'une chaîne c est une collection topologique (ou plus simplement une collection). On écrit $c \in \mathcal{T}$ si $\mathcal{T}$ est le type de c.

4.2 Exemples de collections topologiques

Dans cette sous-section, nous voulons montrer que les structures de données usuelles utilisées en algorithmique classique peuvent être facilement représentées par des collections topologiques. Tous ces exemples décrivent de façon formelle la présentation des collections topologiques donnée dans le chapitre 2 (section 2 page 26).

4.2.1 Les séquences

Les séquences sont des collections topologiques de dimension 1 : elles sont composées de sommets portant des valeurs et correspondant aux éléments de la séquence. Les arcs entre les sommets représentent la contiguïté de ces éléments : deux sommets sont reliés par un arc si les éléments qu'ils représentent sont contigus dans la séquence.

Le complexe $\mathcal{K}^{Seq}$. Les sommets sont représentés par des entiers i et les arcs par les couples de $\{(i, i+1), i \in \mathbb{N}\}$. Le complexe $\mathcal{K}^{Seq}$ suivant est utilisé pour définir la structure linéaire des séquences :

$$\begin{aligned} \mathcal{K}_0^{Seq} &= \mathbb{N} \\ \mathcal{K}_1^{Seq} &= \{(i, i+1), i \in \mathbb{N}\} \end{aligned}$$

où les faces de chaque arc $(i, i+1)$ sont les sommets i et $i+1$ (ce qui définit entièrement la relation d'incidence $\prec^{\mathcal{K}^{Seq}}$ sur les éléments de $\mathcal{K}^{Seq}$).

La forme $C(\mathcal{K}^{Seq}, \mathbb{Z}, \partial^{Seq})$. En pratique, chaque élément de la séquence a un voisin à droite. Une direction, de gauche à droite, est donc privilégiée. On oriente ainsi le complexe $\mathcal{K}^{Seq}$ en ce sens pour définir la forme de ce type de collection topologique ; le groupe utilisé pour calculer les bords est $\mathbb{Z}$ auquel on associe l'opérateur ∂ défini par :

$$\begin{aligned} \partial_0^{Seq}(i) &= 0 \\ \partial_1^{Seq}(i, i+1) &= (i+1) -_{C_0(\mathcal{K}^{Seq}, \mathbb{Z})} i \end{aligned}$$

La forme des séquences est $C(\mathcal{K}^{Seq}, \mathbb{Z}, \partial^{Seq})$.

Le prédicat $pred^{Seq}$**.** Pour définir le type des séquences, il reste à restreindre l'ensemble $C(\mathcal{K}^{Seq}, \mathrm{Abel}_{\mathbb{Z}}(\mathrm{Val}))$ de telle sorte que seuls les sommets soient décorés. De plus, à partir d'un certain rang, les sommets ne sont plus décorés (les séquences sont de longueur finie), et aucun élément non décoré ne doit apparaître avant ce rang. Soit $l \in C(\mathcal{K}^{Seq}, \mathrm{Abel}_{\mathbb{Z}}(\mathrm{Val}),)$:

$$pred^{Seq}(l) \Leftrightarrow \exists n \in \mathbb{N} \text{ tel que } \begin{cases} \forall i < n \in \mathbb{N},\ \exists v \in \mathrm{Val},\ l(i) = 1.v \\ \forall i \geq n \in \mathbb{N},\ l(i) = 0_{\mathrm{Abel}_{\mathbb{Z}}(\mathrm{Val})} \\ \forall i \in \mathbb{N},\ l((i, i+1)) = 0_{\mathrm{Abel}_{\mathbb{Z}}(\mathrm{Val})} \end{cases}$$

n est appelée la *longueur* de l. On note $|l|$ la longueur de la séquence l.

Finalement, on définit $\textsc{Seq}(\mathrm{Val}) = \mathcal{T}(\mathcal{K}^{Seq}, \mathbb{Z}, \partial^{Seq}, \mathrm{Val}, pred^{Seq})$, l'ensemble des séquences.

Notation. Pour simplifier les écritures, nous utiliserons la notation des listes OCaml :

$$l = [x_0\,;\,\ldots\,;\,x_{n-1}] \Leftrightarrow l(i) = \begin{cases} x_i & \text{si } i < n \\ 0_{\mathrm{Abel}_{\mathbb{Z}}(\mathrm{Val})} & \text{sinon} \end{cases}$$

Nous notons [] la liste vide et nous utilisons également les opérations standards hd , last , :: et @. Soient l, l_1 et l_2 des listes, n et n_1 les longueurs respectives de l et l_1, et x un élément de Val, on a :

- $\mathrm{hd}\ l = x \Leftrightarrow l(0) = x$
- $\mathrm{last}\ l = x \Leftrightarrow l(n-1) = x$
- $l' = x::l$ est la liste telle que

$$l'(i) = \begin{cases} x & \text{si } i = 0 \\ l(i-1) & \text{sinon} \end{cases}$$

- $l' = l_1 @ l_2$ est la liste telle que

$$l'(i) = \begin{cases} l_1(i) & \text{si } i < n_1 \\ l_2(i-n_1) & \text{sinon} \end{cases}$$

Afin d'alléger les écritures, nous utilisons les notations suivantes :

- Soient $l = [\sigma_1, \ldots, \sigma_n] \in \textsc{Seq}(\mathcal{K})$ et $c \in \mathcal{T}(\mathcal{K}, G, \partial, \mathrm{Val}, pred)$

$$l' = c(l) = [c(\sigma_1)\,;\,\ldots\,;\,c(\sigma_n)]$$

  est une liste de $\textsc{Seq}(\mathrm{Val})$.

- Soient $l_\sigma = [\sigma_1\,;\,\ldots\,;\,\sigma_n] \in \textsc{Seq}(\mathcal{K})$ et $l_v = [v_1\,;\,\ldots\,;\,v_n] \in \textsc{Seq}(\mathrm{Val})$

$$c = l_\sigma.l_v = v_1.\sigma_1 +_{C(\mathcal{K},\mathrm{Abel}_G(\mathrm{Val}))} \cdots +_{C(\mathcal{K},\mathrm{Abel}_G(\mathrm{Val}))} v_N.\sigma_N$$

  est une chaîne de $\mathcal{T}(\mathcal{K}, G, \partial, \mathrm{Val}, pred)$ (G, $pred$, ∂ et h sont précisés dans le contexte).

- Soient V un sous-ensemble de Val et l une séquence de $\textsc{Seq}(\mathrm{Val})$ de longueur n

$$V \sqcup l = V \cup \{l(i) \,|\, 0 \leq i < n\}$$

  est un sous-ensemble de Val.

4.2.2 Les ensembles

La topologie des ensembles est un graphe complet où les sommets correspondent aux éléments de la structure de données, et les arcs représentent la relation de voisinage. En l'absence d'ordre entre les éléments, on considère que chaque élément est atteignable à partir de n'importe quel autre.

Le complexe $\mathcal{K}^{Set}$. Un ensemble est un sous-ensemble de Val. Les sommets du graphe complet sont donc représentés par les éléments de Val. La valeur associée à chaque sommet va dénoter sa présence ou son absence de l'ensemble. Le complexe cellulaire $\mathcal{K}^{Set}$ est défini par :

$$\begin{array}{rcl} \mathcal{K}_0^{Set} & = & \text{Val} \\ \mathcal{K}_1^{Set} & = & \{\{v_1, v_2\} \subset \text{Val} \mid v_1 \neq v_2\} \end{array}$$

Les faces d'un arc $\{v_1, v_2\}$ sont naturellement les sommets v_1 et v_2.

La forme $C(\mathcal{K}^{Set}, \mathbb{Z}/2\mathbb{Z}, \partial^{Set})$. Contrairement aux séquences, aucune direction n'est privilégiée pour orienter les arcs. On utilise donc le groupe $\mathbb{Z}/2\mathbb{Z}$ pour définir la forme du type de collection des ensembles ; ce groupe permet d'associer 1 aux sommets appartenant à l'ensemble qu'on souhaite construire, et 0 aux sommets absents. Le calcul du bord est donné par :

$$\begin{array}{rcl} \partial_0^{Set}(v) & = & 0 \\ \partial_1^{Set}(\{v_1, v_2\}) & = & v_1 +_{C_0(\mathcal{K}^{Set}, \mathbb{Z}/2\mathbb{Z})} v_2 \end{array}$$

La forme des ensembles est $C(\mathcal{K}^{Seq}, \mathbb{Z}/2\mathbb{Z}, \partial^{Set})$.

Le prédicat $pred^{Set}$. De façon analogue aux séquences, il faut restreindre l'ensemble des chaînes de $C(\mathcal{K}^{Set}, \text{Abel}_{\mathbb{Z}/2\mathbb{Z}}(\text{Val}))$ aux chaînes dont seuls les sommets sont décorés. Soit un ensemble $s \in C(\mathcal{K}^{Set}, \text{Abel}_{\mathbb{Z}/2\mathbb{Z}}(\text{Val}))$,

$$pred^{Set}(s) \Leftrightarrow \left\{ \begin{array}{l} \forall v \in \text{Val}, \left\{ \begin{array}{ll} s(v) = 1.v & \text{si } v \in s \\ s(v) = 0.v = 0_{\text{Abel}_{\mathbb{Z}/2\mathbb{Z}}(\text{Val})} & \text{sinon} \end{array} \right. \\ \\ \forall e \in \mathcal{K}_1^{Set},\ s(e) = 0_{\text{Abel}_{\mathbb{Z}/2\mathbb{Z}}(\text{Val})} \end{array} \right.$$

Finalement, on définit $\textsc{Set}(\text{Val}) = \mathcal{T}(\mathcal{K}^{Set}, \mathbb{Z}/2\mathbb{Z}, \partial^{Set}, \text{Val}, pred^{Set})$, le type des ensembles.

Notation. On peut alors définir les opérations d'union et d'intersection d'ensembles :

- L'union : soient $s_1, s_2 \in \textsc{Set}(\text{Val})$, $\forall v \in \text{Val}$

$$(s_1 \cup s_2)(v) = \left\{ \begin{array}{ll} 1.v & \text{si } s_1(v)(v) = 1 \vee s_2(v)(v) = 1 \\ 0.v = 0_{\text{Abel}_{\mathbb{Z}/2\mathbb{Z}}(\text{Val})} & \text{sinon} \end{array} \right.$$

- L'intersection : soient $s_1, s_2 \in \textsc{Set}(\text{Val})$, $\forall v \in \text{Val}$

$$(s_1 \cap s_2)(v) = (s_1(v)(v).{}_{\mathbb{Z}/2\mathbb{Z}} s_2(v)(v)).v$$

  où « $.{}_{\mathbb{Z}/2\mathbb{Z}}$ » dénote la multiplication standard dans $\mathbb{Z}/2\mathbb{Z}$.

4.2.3 Les multi-ensembles.

Les multi-ensembles peuvent être construits comme les ensembles mais il faut néanmoins prendre en compte du nombre d'occurrences d'une valeur.

Le complexe $\mathcal{K}^{Bag}$. Soit M un multi-ensemble d'éléments de Val. M peut être représenté par un ensemble $M \subseteq \mathbb{N} \times \text{Val}$. Si $v \in M$ avec une multiplicité n, alors, $(0, v), \ldots, (n-1, v) \in \mathbb{N} \times \text{Val}$. Nous utilisons cette représentation pour construire le complexe cellulaire $\mathcal{K}^{Bag}$

$$\begin{array}{rcl} \mathcal{K}_0^{Bag} & = & \mathbb{N} \times \text{Val} \\ \mathcal{K}_1^{Bag} & = & \{\{v_1, v_2\} \subset (\mathbb{N} \times \text{Val}) \mid v_1 \neq v_2\} \end{array}$$

Encore une fois, les faces d'un arc $\{v_1, v_2\}$ sont naturellement les sommets v_1 et v_2 de $\mathcal{K}_0^{Bag}$.

La forme $C(\mathcal{K}^{Bag}, \mathbb{Z}/2\mathbb{Z}, \partial^{Bag})$. La forme des multi-ensembles est exactement la même que celle des ensembles :

$$\begin{array}{rcl} \partial_0^{Bag}(v) & = & 0 \\ \partial_1^{Bag}(\{v_1, v_2\}) & = & v_1 +_{C_0(\mathcal{K}^{Bag}, \mathbb{Z}/2\mathbb{Z})} v_2 \end{array}$$

La forme des ensembles est $C(\mathcal{K}^{Seq}, \mathbb{Z}/2\mathbb{Z}, \partial^{Set})$.

Le prédicat $pred^{Bag}$. On construit le type Bag(Val) des multi-ensembles d'éléments de Val de la même façon que Set(Val) tout en considérant la multiplicité. Soit un multi-ensemble $m \in C(\mathcal{K}^{Bag}, \text{Abel}_{\mathbb{Z}/2\mathbb{Z}}(\text{Val}))$ $m \in \text{Bag}(\text{Val})$, on a

$$pred^{Bag}(m) \Leftrightarrow \left\{ \begin{array}{l} \forall v \in \text{Val},\ \exists N_v \in \mathbb{N},\ \left\{ \begin{array}{ll} \forall i < N_v & m(i, v) = 1.v \\ \forall j \geq N_v & m(j, v) = 0_{\text{Abel}_{\mathbb{Z}/2\mathbb{Z}}(\text{Val})} \end{array} \right. \\ \\ \forall e \in \mathcal{K}_1^{Bag},\ m(e) = 0_{\text{Abel}_{\mathbb{Z}/2\mathbb{Z}}(\text{Val})} \end{array} \right.$$

La première condition fixe un entier N_v en dessous duquel on associe à la position (i, v) la valeur $1.v$ pour désigner la présence de la i^{e} occurrence de v dans m, et au-delà duquel il n'y a plus d'occurrence : N_v définit la multiplicité de v dans m.

Finalement, on définit $\text{Bag}(\text{Val}) = \mathcal{T}(\mathcal{K}^{Bag}, \mathbb{Z}/2\mathbb{Z}, \partial^{Bag}, \text{Val}, pred^{Bag})$, le type des multi-ensembles.

Notation. On définit l'union et l'intersection :

- L'union : soient $m_1, m_2 \in \text{Bag}(\text{Val})$, $\forall v \in \text{Val}$ de multiplicités p_1 et p_2 respectivement pour m_1 et m_2, et $\forall i \in \mathbb{N}$

$$(m_1 \cup m_2)((i, v)) = \left\{ \begin{array}{ll} 1.v & \text{si } i < (p_1 + p_2) \\ 0.v & \text{sinon} \end{array} \right.$$

- L'intersection : soient $m_1, m_2 \in \text{Bag}(\text{Val})$, $\forall v \in \text{Val}$ de multiplicités p_1 et p_2 respectivement pour m_1 et m_2, et $\forall i \in \mathbb{N}$

$$(m_1 \cap m_2)((i, v)) = \left\{ \begin{array}{ll} 1.v & \text{si } i < min(p_1, p_2) \\ 0.v & \text{sinon} \end{array} \right.$$

4.2.4 Les graphes de Delaunay

Les fondements théoriques de cette collection ont été présentés chapitre 2, page 31. On rappelle que cette collection résulte d'une triangulation d'un ensemble fini $P = \{p \in \mathbb{R}^n\}$ de points de $\mathbb{R}^n$. Soit $p \in \mathbb{R}^n$, on appelle

$$\text{Vor}(p_i) = \{q \in \mathbb{R}^n \mid d(p_i, q) < d(p_j, q),\ p_i \neq p_j \in P\}$$

la région de Voronoï de $p_i \in P$. Ces régions sont disjointes et correspondent à des boules ouvertes de $\mathbb{R}^n$; on les représente par des n-cellules. Il découle de cette partition de l'espace un complexe cellulaire géométrique de dimension n. En particulier, les régions de Voronoï de deux éléments de P, p_i et p_j, sont $(n-1)$-voisines si $\overline{\text{VOR}(p_1)} \cap \overline{\text{VOR}(p_2)} \neq \emptyset$, où

$$\overline{\text{VOR}(p_i)} = \{q \in \mathbb{R}^n \mid d(p_i, n) \leq d(p_j, q),\ p_i \neq p_j \in P\}$$

décrit un sous-espace de $\mathbb{R}^n$ de dimension $n-1$.

Le complexe $\mathcal{K}^{Del}$. Pour construire le graphe de Delaunay, on utilise le *dual* du complexe cellulaire issu de la découpe de $\mathbb{R}^n$ en diagramme de Voronoï. De façon informelle (voir le chapitre 6 pour une définition formelle), le dual d'un complexe correspond au renversement de son graphe d'incidence : si $\sigma_1 \prec \sigma_2$ dans le primal, la relation s'inverse dans le dual, $\sigma_1 \succ \sigma_2$. La dimension est également modifiée : les p-cellules d'un complexe primal de dimension n deviennent les $(n-p)$-cellules du dual. Dans le cas des diagrammes de Voronoï, les régions de Voronoï deviennent simplement des sommets, correspondant directement aux éléments de P. Ces sommets sont reliés par des arcs, duaux des connexions de dimension $n-1$ entre les régions de Voronoï. Nous construisons alors le complexe cellulaire abstrait $\mathcal{K}^{Del}$. Soit un ensemble fini $P \subset \mathbb{R}^n$:

$$\begin{array}{rcl} \mathcal{K}_0^{Del} & = & P \\ \mathcal{K}_1^{Del} & = & \{\{p_1, p_2\} \subset P, p_1 \neq p_2, \text{VOR}(p_1) \text{ et } \text{VOR}(p_2) \text{ sont } (n-1)\text{-voisines}\} \end{array}$$

La forme $C(\mathcal{K}^{Del}, \mathbb{Z}/2\mathbb{Z}, \partial^{Del})$. Muni de l'opérateur de bord

$$\begin{array}{rcl} \partial_0^{Del}(p) & = & 0 \\ \partial_1^{Del}(\{p_1, p_2\}) & = & p_1 +_{C_0(\mathcal{K}^{Del}, \mathbb{Z}/2\mathbb{Z})} p_2 \end{array}$$

on définit la forme $C(\mathcal{K}^{Del}, \mathbb{Z}/2\mathbb{Z}, \partial^{Del})$ des collections topologiques de Delaunay sur l'ensemble P.

Le prédicat $pred^{Del}$. Il en découle le type des collections de Delaunay $\text{DEL}_P(\text{Val})$ à valeurs dans Val et dont le support est donné par les points P. Soit $d \in C(\mathcal{K}^{Del}, \text{Abel}_{\mathbb{Z}/2\mathbb{Z}}(\text{Val}))$:

$$pred^{Del}(d) \Leftrightarrow \left\{ \begin{array}{ll} \forall p \in P & d(p) = 1.v \text{ avec } v \in \text{Val} \\ \forall e \in \mathcal{K}_1^{Del} & d(p) = 0_{\text{Abel}_{\mathbb{Z}/2\mathbb{Z}}(\text{Val})} \end{array} \right.$$

Finalement, on définit $\text{DEL}_P(\text{Val}) = \mathcal{T}(\mathcal{K}^{Del}, \mathbb{Z}/2\mathbb{Z}, \partial^{Del}, \text{Val}, pred^{Del})$, le type des collections de Delaunay sur P.

Notation. En pratique, une collection de Delaunay est générée à partir d'un ensemble ou d'une séquence d'éléments de Val qui décorent les sommets, et d'une fonction qui calcule l'ensemble P en fonction de ces valeurs. Soit une telle fonction $f : \text{Val} \to \mathbb{R}^n$, et une séquence $l \in \text{SEQ}(\text{Val})$ de longueur N. Soit $P = \{f(l(i)) \in \mathbb{R}^n \mid 0 \leq i < N\}$, l'ensemble des positions engendrées par l et f. On note $\text{DELAUNAY}_f(l)$, la collection de $\text{DEL}_P(\text{Val})$ engendrée par l et f :

$$\text{DELAUNAY}_f(l) = \sum_{i=0}^{N-1} (1.f(l(i))).l(i)$$

4.2.5 Les collections GBF

Ces collections newtoniennes proposent un voisinage régulier entre les éléments qui la composent. Ce voisinage est généré par l'utilisation d'un groupe G finiment engendré par un ensemble de déplacements élémentaires $D = \{d_1, \ldots, d_n\}$. Partant d'une position d'origine choisie arbitrairement, toutes les autres positions sont atteignables suivant une suite de déplacements élémentaires D.

Le complexe $\mathcal{K}^{Gbf}$. Les positions sont donc représentables par les éléments du groupe G lui-même. Chaque position est directement voisine de $2n$ autres positions, suivant les directions élémentaires de D, ainsi que leurs opposées. On peut alors construire un graphe de voisinage $\mathcal{K}^{Gbf}$ fondé sur ces considérations :

$$\begin{array}{rcl} \mathcal{K}_0^{Gbf} & = & G \\ \mathcal{K}_1^{Gbf} & = & G \times D \end{array}$$

Chaque arc $(g, d) \in G \times d$ a donc deux faces : g et $g +_G d$.

La forme $C(\mathcal{K}^{Gbf}, \mathbb{Z}, \partial^{Gbf})$. Un arc (g, d) de $\mathcal{K}_1^{Gbf}$ est naturellement orienté de la position g à la position $g +_G d$. On définit donc l'opérateur de bord :

$$\begin{array}{rcl} \partial_0^{Gbf}(g) & = & 0 \\ \partial_1^{Gbf}((g,d)) & = & (g +_G d) -_{C_0(\mathcal{K}^{Gbf},\mathbb{Z})} g \end{array}$$

pour définir la forme $C(\mathcal{K}^{Gbf}, \mathbb{Z}, \partial)$ des collections GBF sur G.

Le prédicat $pred^{Gbf}$. Soit $c \in C(\mathcal{K}^{Gbf}, \mathrm{Abel}_{\mathbb{Z}}(\mathrm{Val}))$, un GBF sur le groupe de déplacements G

$$pred^{Gbf}(c) \Leftrightarrow \left\{ \begin{array}{ll} \forall p \in G & \exists v \in \mathrm{Val},\ c(p) = 1.v \ \vee\ c(p) = 0_{\mathrm{Abel}_{\mathbb{Z}}(\mathrm{Val})} \\ \forall p \in \mathcal{K}_1^{Gbf} & c(p) = 0_{\mathrm{Abel}_{\mathbb{Z}}(\mathrm{Val})} \end{array} \right.$$

Finalement, on définit $\textsc{Gbf}_G(\mathrm{Val}) = \mathcal{T}(\mathcal{K}^{Gbf}, \mathbb{Z}, \partial^{Gbf}, \mathrm{Val}, pred^{Gbf})$, le type des collections GBF sur G.

5 La collection topologique universelle

Avec les exemples précédents, on distingue différents types de collections topologiques ; ces types ont été mis en place dans l'interprète du langage. Cependant, pour alléger la définition de la sémantique du mini-langage MGS que nous proposons dans le chapitre 4, nous ne considérons qu'une forme générale de collections topologiques présentée dans cette section.

Ce type de collection topologique est universel dans le sens où les exemples présentés précédemment en font partie. La description ci-dessous a pour but la définition des opérations de construction des collections topologiques ainsi que l'introduction de quelques notations pour la sémantique.

5.1 Ensemble des positions et complexe cellulaire

Le point de départ de la définition des collections topologiques est la construction des complexes cellulaires qui les supportent. Le chapitre d'introduction au langage MGS (chapitre 2) décrit les collections comme des fonctions décorant un espace de positions représenté par un

complexe cellulaire abstrait. La terminologie utilisée dans cette introduction parle de *positions* plutôt que de cellules topologiques.

Les collections topologiques universelles sont construites sur un *ensemble de positions universelles* :

Définition 15 (Ensemble des positions $\mathcal{S}_0$) *Soit $\mathcal{S}_0$, un ensemble arbitraire de symboles. $\mathcal{S}_0$ est l'*ensemble des positions universelles*.*

Cet ensemble est exempt de toute organisation topologique : les positions de $\mathcal{S}_0$ n'ont pas de dimension intrinsèque et ne sont liées par aucune relation d'incidence. En effet, ces notions sont définies par les complexes cellulaires eux-mêmes. Soit $S \subset \mathcal{S}_0$, on définit un complexe cellulaire sur S comme un triplet $\mathcal{K} = (S, \prec, dim)$ vérifiant les conditions de transitivité et de monotonie (AC_1 et AC_2) de la définition 3 page 69. Nous dirons d'un tel complexe qu'il est *construit sur* $\mathcal{S}_0$. Nous notons $\mathbb{K}$ l'ensemble des complexes cellulaires abstraits construits sur $\mathcal{S}_0$.

Notations. Les éléments de $\mathcal{S}_0$ n'ont un sens réel que dans le contexte d'un complexe cellulaire $\mathcal{K} \in \mathbb{K}$. Aussi, une position n'est plus considérée comme un simple élément de $\mathcal{S}_0$, mais comme un couple $(\sigma, \mathcal{K})$ où $\sigma \in \mathcal{K}$. Pour simplifier cette écriture, nous employons la notation $\sigma^{\mathcal{K}}$ rappelant le complexe cellulaire $\mathcal{K}$ auquel appartient σ. En notant $\mathcal{S} = \mathcal{S}_0 \times \mathbb{K}$, nous écrirons toujours $\sigma^{\mathcal{K}} \in \mathcal{S}$, l'exposant étant une annotation explicitant le contexte dans lequel la position σ est utilisée.

Ce point du vue, définissant de façon totalement explicite les cellules topologiques sans référence à aucune dimension et sans incidence prédéfinie, n'est pas usuel. Classiquement, une cellule topologique n'existe que dans un et un seul complexe cellulaire. Ici, on peut manipuler deux complexes cellulaires $\mathcal{K}$ et $\mathcal{K}'$ partageant une cellule $\sigma \in \mathcal{S}$, dans lesquels les cellules incidentes à σ sont différentes. L'avantage d'une telle écriture est de faciliter l'expression des modifications topologiques dans les complexes cellulaires. En effet, il suffit de remplacer une sous-partie d'un complexe par une autre pour en modifier la topologie localement, sans toucher au reste du complexe ; l'incidence liant les cellules qui ne sont pas mises en jeu dans la modification topologique reste inchangée. En revanche, il est nécessaire à tout moment de préciser dans quel complexe $\mathcal{K}$ une cellule est considérée, d'où la notation $\sigma^{\mathcal{K}}$.

5.2 Génération des positions

Afin de construire incrémentalement les collections topologiques, on utilise un générateur de positions retournant une position fraîche de $\mathcal{S}$ (c'est-à-dire qui n'a pas encore été utilisée pour construire une autre collection topologique).

Définition 16 (La fonction posGen**)** *La fonction* posGen *construit une nouvelle cellule topologique $\sigma^{\mathcal{K}}$ à partir d'un entier n, la dimension de σ dans $\mathcal{K}$, et la liste de ses prédécesseurs et de ses successeurs dans $\mathcal{K}$:*

$$\begin{array}{rrcl} \text{posGen}: & \mathbb{Z} \times \textsc{Seq}(\mathcal{S}) \times \textsc{Seq}(\mathcal{S}) & \to & \mathcal{S} \\ & (n, [\ldots; \sigma_i^{\mathcal{K}_i}; \ldots], [\ldots; {\sigma'}_j^{\mathcal{K}'_j}; \ldots]) & \mapsto & \sigma^{\mathcal{K}} \end{array}$$

où

$$\mathcal{K} = \mathcal{K}_0 \cup \cdots \cup \mathcal{K}_i \cup \cdots \cup \mathcal{K}'_j \cup \ldots$$

avec $\mathcal{K}_0 = (S, \prec^*, dim)$ où

- $S = \{\sigma\} \cup \{\ldots, \sigma_i, \ldots\} \cup \{\ldots, \sigma'_j, \ldots\}$

- $\prec^*$ *est la fermeture transitive de la relation suivante :*

$$\{\ldots,(\sigma_i \prec \sigma),\ldots\} \cup \{\ldots,(\sigma \prec \sigma'_j),\ldots\}$$

- *la fonction dim est définie par* $\forall \tau \in S$ *:*

$$dim(\tau) = \begin{cases} n & si \quad \tau = \sigma \\ dim_i(\tau) & si \quad \exists i, \tau = \sigma_i \\ dim'_j(\tau) & si \quad \exists j, \tau = \sigma'_j \end{cases}$$

Cette définition demande quelques précisions. Nous nous plaçons dans le contexte où l'on cherche à générer une nouvelle cellule topologique. On donne à cette cellule une dimension n, une séquence de cellules de dimensions inférieures qui lui seront incidentes $[\ldots;\sigma_i^{\mathcal{K}_i};\ldots]$, et une séquence de cellules de dimensions supérieures qui lui seront également incidentes $[\ldots;\sigma'^{\mathcal{K}'_j}_j;\ldots]$. Les futures cellules incidentes sont prises chacune dans le contexte d'un complexe cellulaire. Soit $\sigma \in \mathcal{S}_0$, une cellule fraîche qui n'a pas encore été utilisée pour construire un complexe cellulaire abstrait. D'après les arguments de la fonction, la cellule σ est utilisée dans le cadre d'un complexe cellulaire où sa dimension est n et est incidente aux cellules σ_i et aux cellules σ'_j. Un complexe cellulaire $\mathcal{K}_0 \in \mathbb{K}$ est défini pour spécifier ces informations. Finalement, les complexes $\mathcal{K}_i$ et $\mathcal{K}'_j$ contenant les cellules σ_i et les cellules σ'_j, ainsi que le complexe $\mathcal{K}_0$ sont unis pour ne former qu'un seul complexe $\mathcal{K}$. C'est dans ce complexe que σ sera utilisée, d'où le retour $\sigma^{\mathcal{K}} \in \mathcal{S}$ de la fonction posGen.

Cette fonction n'est évidemment définie que si $\mathcal{K}$ est défini, c'est-à-dire que les éléments de l'union vérifient deux à deux la condition UC_1. On définit donc un prédicat vérifiant l'applicabilité de posGen.

Définition 17 (La fonction predGen**)** *Le prédicat* predGen *vérifie que le calcul*

$$\text{posGen}(n,[\ldots;\sigma_i^{\mathcal{K}_i};\ldots],[\ldots;\sigma'^{\mathcal{K}'_j}_j;\ldots])$$

est bien défini si :

$$\begin{aligned} \text{predGen}(n,[\ldots;\sigma_i^{\mathcal{K}_i};\ldots],[\ldots;\sigma'^{\mathcal{K}'_j}_j;\ldots]) = & \\ & \forall i_1,i_2 \quad \tau \in \mathcal{K}_{i_1} \cap \mathcal{K}_{i_2} \Rightarrow dim^{\mathcal{K}_{i_1}}(\tau) = dim^{\mathcal{K}_{i_2}}(\tau) \\ \wedge \quad & \forall j_1,j_2 \quad \tau \in \mathcal{K}'_{j_1} \cap \mathcal{K}'_{j_2} \Rightarrow dim^{\mathcal{K}'_{j_1}}(\tau) = dim^{\mathcal{K}'_{j_2}}(\tau) \\ \wedge \quad & \forall i,j \quad \tau \in \mathcal{K}_i \cap \mathcal{K}'_j \Rightarrow dim^{\mathcal{K}_i}(\tau) = dim^{\mathcal{K}'_j}(\tau) \\ \wedge \quad & \forall i \quad n > dim^{\mathcal{K}_i}(\sigma_i) \\ \wedge \quad & \forall j \quad n < dim^{\mathcal{K}'_j}(\sigma'_j) \end{aligned}$$

Si une cellule τ apparaît dans deux complexes cellulaires, elle doit y avoir la même dimension. De plus, les cellules σ_i (resp. des σ'_j) étant spécifiées comme inférieures (resp. supérieures) au sens de la relation d'incidence dans le complexe cellulaire généré, la condition de monotonie des complexes cellulaires impose que la cellule σ créée soit de dimension supérieure à σ_i (resp. de dimension inférieure à σ'_j).

5.3 Ensemble des collections topologiques

Finalement, partant de l'ensemble de positions abstraites $\mathcal{S}_0$ avec l'ensemble des complexes cellulaires de $\mathbb{K}$, et en notant Val l'ensemble des valeurs qu'on souhaite manipuler, les collections topologiques sur Val sont des éléments de $\mathcal{T}(\mathcal{K}, \mathbb{Z}, \partial, \text{Val}, \mathit{pred})$ tels que :

- $\mathcal{K} \in \mathbb{K}$ est un complexe construit sur $\mathcal{S}_0$,
- on utilise le groupe $\mathbb{Z}$ et l'opérateur de bord ∂ définis dans le cadre du transport de valeurs entre les cellules de $\mathcal{K}$,
- le prédicat *pred* accepte tous les éléments de $C(\mathcal{K}, \mathbb{Z})$.

On définit alors l'ensemble des collections topologiques universelles à valeur dans un ensemble arbitraire Val par :

$$\mathsf{CollType}(\text{Val}) = \bigcup_{\mathcal{K} \in \mathbb{K}} \mathcal{T}(\mathcal{K}, \mathbb{Z}, \partial, \text{Val}, \mathit{pred})$$

Notation. De façon générale, les collections sont dénotées par la variable C. Comme pour les cellules topologiques, la notation $C^{\mathcal{K}}$ signifie que la collection C est construite sur le complexe cellulaire $\mathcal{K}$; il s'agit d'une information supplémentaire donnée sous la forme d'une annotation, qui permet de préciser le complexe cellulaire sur lequel est construit C, et non un opérateur particulier.

Pour dénoter une modification du domaine d'une collection, on utilise l'opérateur $\downarrow$.

Définition 18 (L'opérateur $\downarrow$) *Soient $C^{\mathcal{K}} \in \mathsf{CollType}(\text{Val})$ une collection topologique à valeur dans un ensemble arbitraire* Val*, et $\mathcal{K}' \in \mathbb{K}$ un complexe cellulaire abstrait. L'écriture $C^{\mathcal{K}}{\downarrow}\mathcal{K}'$ définit une nouvelle collection de* $\mathsf{CollType}(\text{Val})$ *telle que $\forall \sigma \in \mathcal{K}'$:*

$$C'^{\mathcal{K}'}(\sigma) = (C^{\mathcal{K}}{\downarrow}\mathcal{K}')(\sigma) = \begin{cases} C^{\mathcal{K}}(\sigma) & \mathit{si}\ \sigma \in \mathcal{K} \cap \mathcal{K}' \\ 0_{\text{Abel(Val)}} & \mathit{sinon} \end{cases}$$

L'opérateur $\downarrow$ modifie le domaine d'une collection topologique pour un nouveau complexe cellulaire abstrait de $\mathbb{K}$.

La concaténation de deux collections topologiques est donnée par l'opérateur d'addition :

Définition 19 (Concaténation de collections topologiques) *Soient $C_1^{\mathcal{K}_1}$ et $C_2^{\mathcal{K}_2}$ deux collections topologiques de* $\mathsf{CollType}(\text{Val})$*. La* concaténation *de $C_1^{\mathcal{K}_1}$ et $C_2^{\mathcal{K}_2}$, notée $C_1^{\mathcal{K}_1} + C_2^{\mathcal{K}_2}$, définit une nouvelle collection topologique telle que :*

$$C_1^{\mathcal{K}_1} + C_2^{\mathcal{K}_2} = C_1^{\mathcal{K}_1}{\downarrow}(\mathcal{K}_1 \cup \mathcal{K}_2) +_{C(\mathcal{K}_1 \cup \mathcal{K}_2, \text{Abel(Val)})} C_2^{\mathcal{K}_2}{\downarrow}(\mathcal{K}_1 \cup \mathcal{K}_2)$$

Les changements de topologie sont alors décrits de façon concise. Soit $C^{\mathcal{K}}$ une collection topologique dans laquelle on filtre la sous-collection restreinte au sous-complexe $\mathcal{K}' \subset \mathcal{K}$, pour la remplacer par la collection $C'^{\mathcal{K}''}$. La modification de topologie est donnée par :

$$C^{\mathcal{K}}{\downarrow}(\mathcal{K} - S^{\mathcal{K}'}) + C'^{\mathcal{K}''}$$

6 Substitution dans les collections topologiques et réécriture de graphe

Un des objectifs de la formalisation que nous venons de présenter est de simplifier la définition de la reconstruction des collections. Dans le cadre de la réécriture de terme ou de graphe, cette

reconstruction correspond à un mécanisme de substitution. Ce mécanisme est réputé difficile à implanter en toute généralité dans le cadre des graphes. Les graphes étant des complexes cellulaires particulier de dimension 1, il est intéressant de comparer notre approche avec celle de la réécriture de graphe.

6.1 Réécriture de graphe

Un graphe est défini à partir d'un ensemble de sommets, d'un ensemble d'arcs et d'une fonction attribuant à chaque arc ses sommets[8] :

Définition 20 (Graphe) *Un* graphe *est un triplet* $\langle \mathrm{V}, \mathrm{E}, \mathrm{att} \rangle$ *composé d'un ensemble* V *de sommets, d'un ensemble* E *d'arcs et d'une fonction* $\mathrm{att} : \mathrm{E} \to \mathrm{V}^*$ *donnant pour chaque arc* $e \in \mathrm{E}$ *l'ensemble des sommets* $\mathrm{att}(e)$ *incidents à cet arc.*

La réécriture de graphe est fondée sur les notions de *morphisme* de graphes et de *pushout* (approche du double pushout [EPS73]).

Morphisme de graphes et pushout. Intuitivement, un morphisme de graphes est une fonction d'un graphe dans un autre :

Définition 21 (Morphisme de graphes) *Soient deux graphes* $G = \langle \mathrm{V}_G, \mathrm{E}_G, \mathrm{att}_G \rangle$ *et* $H = \langle \mathrm{V}_H, \mathrm{E}_H, \mathrm{att}_H \rangle$. *Un* morphisme *de graphes* $g : G \mapsto H$ *consiste en deux fonctions* $g_\mathrm{V} : \mathrm{V}_G \to \mathrm{V}_H$ *et* $g_\mathrm{E} : \mathrm{E}_G \to \mathrm{E}_H$ *tels que :*

$$\begin{array}{c} \mathrm{V}_G \\ \downarrow{\scriptstyle g_\mathrm{V}} \\ \mathrm{V}_H \end{array} \qquad\qquad \begin{array}{ccc} \mathrm{V}_G^* & \xleftarrow{\mathrm{att}_G} & \mathrm{E}_G \\ {\scriptstyle g_\mathrm{V}^*}\downarrow & & \downarrow{\scriptstyle g_\mathrm{E}} \\ \mathrm{V}_H^* & \xleftarrow[\mathrm{att}_H]{} & \mathrm{E}_H \end{array}$$

Un morphisme de graphe g est un *isomorphisme* si les fonctions g_V et g_E sont des bijections.

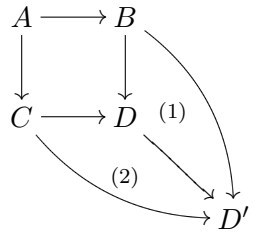
Définition 22 (Pushout) *Soit le diagramme qui commute suivant :*

$$\begin{array}{ccc} A & \longrightarrow & B \\ \downarrow & & \downarrow \\ C & \longrightarrow & D \end{array}$$

où les lettres A, B, C et D sont des graphes et les flèches représentent des morphismes entre ces graphes. Ce diagramme est un pushout *si pour tous morphismes* $B \to D'$ *et* $C \to D'$, *tel que*

$$A \to B \to D' = A \to C \to D'$$

il existe un unique morphisme $D \to D'$ *tels que les diagrammes (1) et (2) suivants commutent :*



[8] On note que, dans cette définition, le nombre de sommets associés à un arc n'est pas limité à deux. On appelle alors *hyper-graphes* les graphes dont certains arcs, appelés *hyper-arcs*, possèdent un nombre de sommets différents de 2.

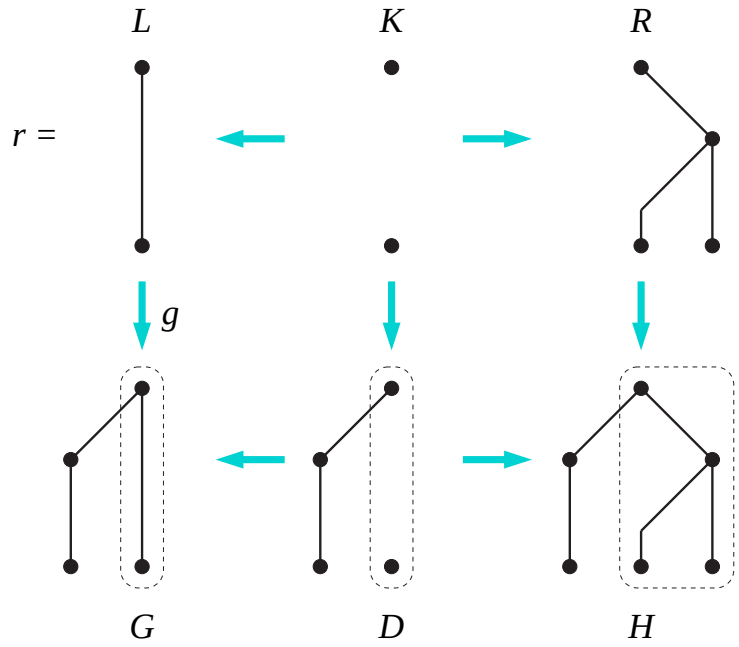


FIG. 8 – Exemple de réécriture de graphe : L est la partie gauche de la règle spécifiant la forme du sous-graphe transformé, R est la partie droite de la règle décrivant ce par quoi il est remplacé, et K correspond aux éléments communs des graphes L et R. G est le graphe à transformer. Le sous-graphe entouré en pointillés est isomorphe (par g) à L. D et H correspondent à la transformation du sous-arc sélectionné par L, en K et R.

Réécriture de graphes. Les règles de réécriture de graphe se présentent sous la forme de triplets $\langle L \leftarrow K \rightarrow R \rangle$ signifiant intuitivement que l'on souhaite supprimer le sous-graphe $L - K$ pour le remplacer par $R - K$. Le sous-graphe L représente la partie filtrée du graphe, le sous-graphe K la partie de L qui est conservée par la règle, et R les objets (arcs et sommets) créés par la règle et liés aux éléments de K.

Un pas de transformation $G \Rightarrow H$ correspond à l'application d'une règle. Soient $r = \langle L \leftarrow K \rightarrow R \rangle$ une règle et $g : L \rightarrow G$ un morphisme injectif. Alors G est transformé en H par l'application de la règle r filtrant une sous-partie de G par g, si et seulement s'il existe deux pushouts tels que :

$$\begin{array}{ccccc} L & \longleftarrow & K & \longrightarrow & R \\ \downarrow & & \downarrow & & \downarrow \\ G & \longleftarrow & D & \longrightarrow & H \end{array}$$

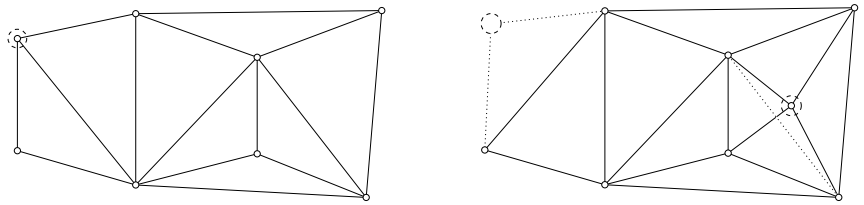
La figure 8 décrit une transformation de graphe.

6.2 Transformation de graphe *versus* réécriture de graphe

La notion de transformation se rapproche de la notion de réécriture [DJ90, Der93], ou plus exactement de *système de réduction*. Un système de réduction abstrait [Ibă04, chapitre 4] est une paire $(\mathcal{T}, \rightarrow)$ formé d'une ensemble $\mathcal{T}$, appelé son domaine, et d'une relation binaire $\rightarrow$ sur $\mathcal{T}$. Une transformation T spécifie un système de réduction : $\mathcal{T}$ correspond à l'ensemble des collections topologiques d'un type donnée (i.e. où la relation de voisinage est définie de manière uniforme) et la relation $\rightarrow$ est définie par $C \rightarrow T(C)$.

Un système de réduction abstrait permet d'abstraire plusieurs notions de réécriture. Nous avons vu que les graphes correspondaient à un cas particulier de collection topologique. Cependant, une transformation ne correspond à aucune notion classique de réécriture de graphe. En effet, l'application d'une règle peut changer le voisinage d'un élément x qui n'est pas filtré par cette règle.

Par exemple la substitution topologique des graphes de Delaunay renvoie un graphe dont la relation de voisinage n'est pas fonction de la relation de voisinage sur la collection en argument. Les schémas ci-dessous illustrent bien ceci. Le schéma de gauche représente la collection en argument. Le chemin filtré est entouré de pointillés (ici, une seule position).



Le « déplacement » de l'élément filtré fait que deux éléments de la collection initiale y étant voisins mais n'étant pas voisins de l'élément filtré ne sont plus voisins dans la collection renvoyée. Sur le schéma de droite, on peut voir qu'une arête n'étant pas liée à l'élément filtré a disparu. Cet exemple montre la difficulté d'utiliser les substitutions topologiques dans le cadre habituel de la réécriture de graphe [Cou93].

Cependant, les *patches*, une forme particulière de transformation que nous décrivons dans ce document, correspondent à une certaine forme de réécriture de graphe quand ils sont appliqués à des collections de dimension 1.

Les recherches menées dans le domaine de la réécriture se focalisent surtout sur les liens entre réécriture et théorie équationnelle avec des problématiques comme la terminaison, la confluence, la recherche de paire critique ou d'ordre bien fondé, la complétion, etc.

Dans notre contexte, nous nous intéressons à la manipulation de ces transformations pour simuler le caractère dynamique de la structure des (SD)^2. Si les transformations permettent de définir de nouveaux mécanismes de réécriture, ces mécanismes sont pour nous utiles pour la programmation et trouvent de nombreuses applications, comme le montrent les exemples de la partie III. L'étude de ces mécanismes du point de vue de la réécriture est néanmoins un travail à effectuer, une perspective intéressante qui dépasse cependant le cadre de cette thèse.

7 Bilan

Dans ce chapitre, nous nous sommes concentrés sur la définition d'un formalisme permettant la représentation de n'importe quel type de structure de données. Nous nous sommes penchés dans un premier temps sur les besoins requis par la simulation de systèmes dynamiques à structure dynamique. Nous nous sommes ensuite inspirés des outils et formalismes élaborés dans le cadre de la CAO, de l'infographie et de la modélisation de solides qui répondent en partie à nos besoin, pour développer la notion de complexe de chaîne, à la base de la formalisation des collections topologiques. Nous avons illustré ce formalisme par des exemples de structures de données standards considérées du point de vue formel des collections topologiques. Finalement, nous avons élaboré une forme universelle de collection que nous utiliserons dans les chapitres suivants.

En nous intéressant aux chaînes topologiques, sans exprimer de contrainte particulière, nous avons créer une structure de données « universelle » qui permet de représenter n'importe quelle

chaîne, à un renommage près des cellules. Ce chapitre a mis en avant un important effort théorique pour aboutir à ce résultat. Il est important d'en distinguer les apports pour comprendre l'intérêt de notre démarche :

- En utilisant la notion de chaîne topologique, nous avons formalisé les collections topologiques comme des fonctions associant les éléments d'un groupe particulier, $\mathrm{Abel}_G(\mathrm{Val})$, à un espace discret composé de briques atomiques, les cellules topologiques. L'intérêt de l'utilisation des chaînes et non d'une simple fonction de l'ensemble des cellules topologiques dans un ensemble arbitraire de valeur Val est double :

  1. L'ensemble Abel(Val) permet de manipuler des « multi-ensembles » généralisés de valeurs, c'est-à-dire où les éléments peuvent apparaître avec des occurrences négatives. Ceux-ci correspondent à la forme de représentation la plus lâche d'une fonction de collision. En effet, par la structure de groupe que nous avons associée à Abel(Val), les collisions de valeurs pendant les transports (formalisés par l'opérateur de bord ∂) se résument à l'insertion dans ce multi-ensemble de nouveaux éléments. Il suffit donc d'exprimer la fonction de collision comme un morphisme de Abel(Val) vers l'ensemble qu'il semble intéressant d'utiliser : Abel(Val) contient toutes les informations à la gestion des collisions. Enfin, la définition du produit tensoriel $\mathrm{Abel}(\mathrm{Val}) \otimes G$ permet de gérer les collisions dans un cadre où le transport est contraint par les éléments du groupe G, et cela en conservant uniquement les informations nécessaires pour ne pas introduire de redondance.

     Du point de vue de l'implantation, nous avons évité de manipuler les éléments de Abel(Val), en gérant les collisions au cas par cas (notamment pour l'étape de reconstruction des transformations) afin d'être plus efficaces. Cela nous semble à présent inutile. En effet, un élément de Abel(Val) peut être représenté par une table d'associations applicative ou une table de hachage associant à un élément $v \in \mathrm{Val}$ un entier relatif. Nous avons également montré qu'un produit tensoriel $\mathrm{Abel}(\mathrm{Val}) \otimes G$ pouvait être identifié aux éléments de $\mathrm{Abel}_G(\mathrm{Val})$. Là encore, l'implantation est satisfaite par une table de hachage associant aux éléments de Val un élément de G.

  2. La topologie algébrique permet d'une part l'étude de la topologie des espaces par une représentation algébrique, mais permet également de définir un calcul différentiel discret sur les objets qu'elle manipule. La définition des chaînes en est la première étape. Il semble intéressant de prolonger ce que nous avons commencé dans ce chapitre en étudiant le calcul différentiel discret et comment celui-ci peut s'interpréter pour la simulation de systèmes dynamiques. Nous ne poursuivons pas plus ces propos, ceux-ci étant l'objet du chapitre 6.

- Le langage MGS étant dédié à la simulation des systèmes dynamiques à structure dynamique, nous avons défini l'ensemble des concepts présentés dans ce chapitre pour simplifier la formalisation des modifications topologiques. Les collections topologiques universelles ont été développées dans ce sens. Quatre opérations élémentaires simples permettent d'une part de définir les collections et d'autre part d'en modifier la topologie :

  1. La fonction posGen permet de créer une nouvelle cellule topologique fraîche, construisant automatiquement le complexe cellulaire dans lequel cette cellule doit être considérée.

  2. Les sommes formelles représentant les collections permettent d'associer une valeur v de Val à une cellule σ générée par posGen : $\mathrm{Inj}_{\mathrm{Val}}(v).\sigma$, ou plus simplement $v.\sigma$.

3. L'opérateur $\downarrow$ autorise la modification du domaine de définition, et donc de la structure, d'une collection topologique.
4. La concaténation permet de fusionner deux collections, construisant alors l'union des complexes cellulaires qui les supportent.

Ces quatre opérations sont suffisantes pour spécifier des modifications topologiques et définir ainsi une sémantique des transformations comme nous le voyons chapitres 4 et 5.

Chapitre 4

Une sémantique naturelle pour MGS

Dans ce chapitre, nous donnons une sémantique naturelle d'un sous-ensemble du langage MGS tel qu'il a été présenté chapitre 2. Cette sémantique décrit formellement les manipulations qu'autorisent les transformations, une forme de réécriture locale, sur les collections topologiques.

Le sous-ensemble du langage MGS que nous proposons d'étudier s'appelle mini-MGS. Il s'agit d'un langage strict (l'évaluation d'une application entraîne l'évaluation de tous ses arguments), non-typé (le typage du langage MGS a été étudié dans [Coh04]) et purement fonctionnel (le caractère impératif des variables globales MGS n'est pas pris en compte). Il permet d'étudier les mécanismes de filtrage et de reconstruction des collections topologiques, ainsi que les stratégies d'application de règles des transformations MGS. Ce chapitre ne démontre en aucun cas l'expressivité du langage MGS. En particulier, les primitives telles que `neighborfold` ne sont pas introduites dans mini-MGS : leur implantation ne poserait aucune difficulté mais surchargerait

inutilement la présentation des mécanismes sous-jacents au concept de transformation. En revanche, elles seront étudiées dans le chapitre 6.

Nous commençons par définir la grammaire du langage mini-MGS (section 1). Nous donnons ensuite les domaines dans lesquels les expressions de ce langage sont évaluées (section 2) pour ensuite décrire la sémantique naturelle du langage à travers des relations d'évaluation (associant une valeur à une expression mini-MGS) définies par induction sur les constructions des programmes mini-MGS (section 3). Afin d'illustrer nos propos, nous donnons d'une part quelques exemples qui jalonnent ces différentes sections et d'autre part trois exemples plus élaborés (section 4) montrant les mécanismes mis en œuvre lors de modifications topologiques, l'intérêt des stratégies d'application des règles et l'utilisation de la sémantique pour l'élaboration de preuve sur les programmes mini-MGS.

1 La syntaxe

1.1 La grammaire

La syntaxe du langage mini-MGS est donnée par la grammaire 1 où $n \in \mathbb{Z}$, $f \in \mathbb{F}$ sont les flottants standards (norme IEEE 754 [Gol91]), $s \in \mathbb{S}$ un ensemble de symboles, et $\mathtt{x} \in \mathcal{X}$ l'ensemble des identificateurs du langage. On utilise également un ensemble étendu d'identificateurs, noté $\overline{\mathcal{X}}$ et défini par :

$$\overline{\mathcal{X}} = \mathcal{X} \cup \{\mathtt{\hat{}x} \mid \mathtt{x} \in \mathcal{X}\}$$

Ainsi, `^x` n'est pas une construction syntaxique, mais un élément de $\overline{\mathcal{X}}$.

La grammaire 1 définit sept ensembles :

1. $\mathcal{C}$: l'ensemble des constantes dénotées[1] par la variable *cte* ;
2. $\mathcal{F}$: l'ensemble des symboles de fonctions prédéfinies dénotés par la variable *fct* ;
3. Σ : l'ensemble des expressions dénotées par la variable e ;
4. Γ : l'ensemble des règles de transformations de chemins dénotées par γ ;
5. M : l'ensemble des motifs de chemins dénotés par μ ;
6. Ψ : l'ensemble des règles de patches dénotées par ψ ;
7. Π : l'ensemble des motifs de patches dénotés par π.

Les constructions syntaxiques de cette grammaire correspondent aux descriptions informelles suivantes :

- Les constantes *cte* du langage sont les entiers, les flottants, les booléens (on note $\mathcal{B} = \{\text{true}, \text{false}\}$ l'ensemble des booléens), les symboles (construits en pratique sur l'expression régulière `'(a-zA-Z)(a-zA-Z0-9)*`), la valeur particulière `<undef>` et la collection topologique vide notée {||}. En plus de ces constantes, quelques fonctions prédéfinies sont disponibles :
  - `PosGen` : une fonction à trois variables pour générer de nouvelles positions fraîches pour construire les collections topologiques (voir la fonction posGen du chapitre 3 page 89) ;
  - `Extension` : une fonction à deux variables associant une valeur à une position. La notation infixe suivante sera utilisée :

$$\{\mid e_1 \mathbin{@} e_2 \mid\} \quad \equiv \quad \texttt{Extension}(e_1)(e_2)$$

[1] Si la variable x dénote une certaine sorte de données, il en va de même pour les variables $x_0, x_1, \ldots$ et $x', x'', \ldots$

$\mathcal{C}$

cte ::= n | f | `true` | `false` | s | `<undef>` | {||} | fct

$\mathcal{F}$

fct ::= `PosGen` | `Extension` | `Concat` | `Cons` | ...

Σ

e ::= cte
| `x`
| `^x`
| e(`)`e`)`
| e`<`e`,`e`>(`e`)`
| λ`x·`e
| `if` e `then` e `else` e
| `trans<x,x>{` ... `;` γ `;` ... `}`
| `patch{` ... `;` ψ `;` ... `}`

Γ

γ ::= $\mu \Rightarrow e$

M

μ ::= `_` | cte
| μ`,` μ | μ `as x` | μ `/` e | μ `|` μ | μ `\/` μ | μ`*`

Ψ

ψ ::= $\pi \Rightarrow e$

Π

π ::= `x:[ dim =` e`,` e `in faces,` e `in cofaces,` e `]`
| `~x:[ dim =` e`,` e `in faces,` e `in cofaces,` e `]`
| π π

GRAM. 1: Grammaire du langage mini-MGS

Cette expression construit une collection topologique où la cellule issue de l'évaluation de e_2 est décorée par la valeur de e_1 ;

– `Concat` : une fonction à deux variables concaténant deux collections topologiques. La notation infixe suivante sera utilisée :

$$e_1\,\texttt{,}\,e_2 \quad \equiv \quad \texttt{Concat(}e_1\texttt{)(}e_2\texttt{)}$$

Les expressions e_1 et e_2 doivent évaluer deux collections topologiques. Ainsi pour construire un arc reposant sur deux sommets, le premier étant décoré par l'entier 1, le second par l'entier 2 et l'arc lui-même par le symbole `'edge`, on écrit :

```
let v1 = PosGen 0 [] [] in
let v2 = PosGen 0 [] [] in
let e  = PosGen 1 [v1;v2] [] in
  {| 1 @ v1 |}, {| 2 @ v2 |}, {| 'edge @ e |}
```

– `Cons` : une fonction à deux variables insérant un élément en tête d'une séquence. On utilisera également la notation `::` standard :

$$e_1\,\texttt{::}\,e_2 \quad \equiv \quad \texttt{Cons(}e_1\texttt{)(}e_2\texttt{)}$$

Les listes sont construites à partir de la collection vide {||} (que l'on écrit également `[]` dans le cadre des séquences) et du constructeur `::` usuel, comme en OCaml. On utilisera

dans les exemples le sucre syntaxique `[`e_1`,...,`e_n`]` pour

$$e_1 \texttt{ :: (... :: (}e_n \texttt{ :: [])...)}$$

Pour distinguer les éléments de la syntaxe et les valeurs évaluées, on les écrit en utilisant la `police courier`. Par exemple, la constante `1.0` s'évalue en la valeur 1.0.

- Les variables du langage sont de la forme `x` ou `^x`.
- Deux formes d'application existent en mini-MGS. La première est l'application standard pour les λ-expressions et les patches. La seconde est réservée à l'application des transformations de chemins ; deux arguments supplémentaires sont utilisés pour spécifier les paramètres n et p du $\langle n, p\rangle$-voisinage. Soit `c` une variable dénotant une collection topologique, `P(c)` applique à la collection `c` le patch associé à la variable `P`, et `T<0,1>(c)` applique à `c` la transformation associée à `T` en utilisant le $\langle 0, 1\rangle$-voisinage.
- Les fonctions sont définies en utilisant les λ-expressions. La déclaration de variable locale n'est pas donnée dans la grammaire. En effet, une construction telle que le `let ... in ...` correspond à l'application d'une λ-expression. Cependant, une telle construction est utilisée dans nos exemples, simplifiant grandement la syntaxe. Ce sucre syntaxique est défini par l'équivalence suivante :

$$\texttt{let x = }e_1\texttt{ in }e_2 \quad \equiv \quad \texttt{(}\lambda\texttt{x}\cdot e_2\texttt{)(}e_1\texttt{)}$$

- Aucune construction syntaxique n'est fournie dans notre grammaire pour définir des fonctions récursives. En effet, il est possible de les déclarer à l'aide d'un opérateur de point fixe défini directement par l'utilisateur. En considérant que la sémantique qui sera donnée dans la suite de ce chapitre implante un évaluateur *strict* et *non-typé*, il est par exemple possible de définir la fonction *factorielle* de la façon suivante :

```
let U      = \f·(f(f)) in
let factfn = \f·(\n·(if (n<=1)
                       then 1
                       else n * (U(f))(n-1)
                 ))
in
let fact   = U(factfn) in
  fact(3) ;;
```

L'opérateur de point fixe `U` est adapté à une évaluation stricte[2] et non-typée.

- La conditionnelle est assurée par la construction standard `if...then...else...`.
- Une transformation de chemins est définie à l'aide du mot clé `trans`. Les variables `n` et `p` réfèrent aux paramètres du voisinage utilisé. Les motifs de chemin sont construits inductivement avec le *joker* et les constantes pour filtrer des éléments isolés, la virgule pour concaténer deux chemins, le choix entre deux chemins, ... (voir le chapitre 2 page 49 pour une description informelle des motifs de chemin).
- Les patches sont définis à l'aide du mot clé `patch`. Les motifs permettent de filtrer une liste non-vide d'éléments, certains étant consommés et les autres non par le filtrage (voir le chapitre 2 page 52 pour une description informelle des motifs de patch).

Ces explications sont formellement définies par la sémantique construite dans la suite de ce chapitre.

[2] L'opérateur non-typé plus standard `Y=`λ`f·(`λ`x·f(x(x)))(`λ`x·f(x(x)))` ne convient pas à un cadre strict.

1.2 Validité des expressions de Σ

Toutes les expressions de l'ensemble Σ ne correspondent pas à des programmes mini-MGS valides. Par exemple, pour que l'évaluation puisse aboutir, on demande que toutes les variables utilisées dans un programme soient liées. De plus, concernant les motifs, la définition de variables de filtre suit des règles strictes. Les fonctions et prédicats définis dans la suite de cette section permettent de déterminer si une expression est valide en ce qui concerne les variables ; ils ne permettent pas en revanche de déterminer si une expression est bien typée. Le lecteur intéressé par le typage des programmes MGS se référera à [Coh04].

1.2.1 Les variables libres

Nous commençons par définir une fonction qui calcule l'ensemble des variables libres d'une expression de Σ. L'ensemble des variables de filtre d'un motif étant nécessaire à ce calcul, une seconde fonction est donnée pour construire l'ensemble des variables liées d'un motif.

Définition 23 (Les fonctions `Libre` et `Liee`) *Les fonctions*[3]

$$\begin{array}{rcl} \texttt{Libre}_\Sigma & : & \Sigma \to \mathcal{P}(\overline{\mathcal{X}}) \\ \texttt{Libre}_\Gamma & : & \Gamma \to \mathcal{P}(\overline{\mathcal{X}}) \\ \texttt{Libre}_\mathrm{M} & : & \mathrm{M} \to \mathcal{P}(\overline{\mathcal{X}}) \\ \texttt{Libre}_\Psi & : & \Psi \to \mathcal{P}(\overline{\mathcal{X}}) \\ \texttt{Libre}_\Pi & : & \Pi \to \mathcal{P}(\overline{\mathcal{X}}) \end{array}$$

calculent respectivement l'ensemble des variables libres d'une expression, d'une règle de $\langle n, p\rangle$-transformation, d'un motif de chemin, d'une règle de patch et d'un motif de patch.

Les fonctions

$$\begin{array}{rcl} \texttt{Liee}_\mathrm{M} & : & \mathrm{M} \to \mathcal{P}(\overline{\mathcal{X}}) \\ \texttt{Liee}_\Pi & : & \Pi \to \mathcal{P}(\overline{\mathcal{X}}) \end{array}$$

calculent respectivement les variables de filtre des motifs de chemin et de motifs de patch.

Ces fonctions mutuellement récursives sont définies inductivement sur la grammaire des expressions :

$$\begin{array}{lcl} \texttt{Libre}_\Sigma(cte) & = & \emptyset \\ \texttt{Libre}_\Sigma(\texttt{x}) & = & \{\texttt{x}\} \\ \texttt{Libre}_\Sigma(\texttt{\^{}x}) & = & \{\texttt{\^{}x}\} \\ \texttt{Libre}_\Sigma(e_1\,(e_2)) & = & \texttt{Libre}_\Sigma(e_1) \cup \texttt{Libre}_\Sigma(e_2) \\ \texttt{Libre}_\Sigma(e_1\texttt{<}e_2, e_3\texttt{>}(e_4)) & = & \bigcup_i \texttt{Libre}_\Sigma(e_i) \\ \texttt{Libre}_\Sigma(\lambda\texttt{x}\cdot e) & = & \texttt{Libre}_\Sigma(e) - \{\texttt{x}\} \\ \texttt{Libre}_\Sigma(\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3) & = & \texttt{Libre}_\Sigma(e_1) \cup \texttt{Libre}_\Sigma(e_2) \cup \texttt{Libre}_\Sigma(e_3) \\ \texttt{Libre}_\Sigma(\texttt{trans<n,p>\{ } \ldots \texttt{ ; } \gamma_i \texttt{ ; } \ldots \texttt{ \}}) & = & \bigcup_i \texttt{Libre}_\Gamma(\gamma_i) - \{\texttt{n}, \texttt{p}, \texttt{self}\} \\ \texttt{Libre}_\Sigma(\texttt{patch\{ } \ldots \texttt{ ; } \psi_i \texttt{ ; } \ldots \texttt{ \}}) & = & \bigcup_i \texttt{Libre}_\Psi(\psi_i) - \{\texttt{self}\} \end{array}$$

Les transformations de chemins et les patches lient la variable `self` à la collection sur laquelle ils s'appliquent ; en effet, cette variable se réfère à l'argument de la transformation lorsque celle-ci est appliquée. Pour les $\langle n, p\rangle$-transformations, les deux variables `n` et `p` faisant référence aux dimensions utilisées pour le voisinage sont également liées. L'appel à $\texttt{Libre}_\Gamma$ et à $\texttt{Libre}_\Psi$ permet de déterminer les variables libres des règles γ_i et ψ_i des transformations de chemin et des patches.

$$\texttt{Libre}_\Gamma(\mu \Rightarrow e) \quad = \quad \texttt{Libre}_\mathrm{M}(\mu) \cup (\texttt{Libre}_\Sigma(e) - \texttt{Liee}_\mathrm{M}(\mu))$$

[3] Soit E un ensemble, $\mathcal{P}(E)$ dénote les parties de E.

$$\texttt{Libre}_\Psi(\pi \Rightarrow e) \quad = \quad \texttt{Libre}_\Pi(\pi) \cup (\texttt{Libre}_\Sigma(e) - \texttt{Liee}_\Pi(\pi))$$

Pour chaque règle, les variables libres sont les variables libres des parties gauches, auxquelles s'ajoutent les variables libres de l'expression en partie droite privées des variables de filtre définies dans le motif. Les fonctions $\texttt{Libre}_\text{M}$ et $\texttt{Liee}_\text{M}$ (resp. $\texttt{Libre}_\Pi$ et $\texttt{Liee}_\Pi$) calculent les variables libres et les variables de filtre des motifs de chemin (resp. des motifs de patch).

$$\begin{array}{lcl}
\texttt{Libre}_\text{M}(_) & = & \emptyset \\
\texttt{Libre}_\text{M}(cte) & = & \emptyset \\
\texttt{Libre}_\text{M}(\mu_1\texttt{ , }\mu_2) & = & \texttt{Libre}_\text{M}(\mu_1) \cup (\texttt{Libre}_\text{M}(\mu_2) - \texttt{Liee}_\text{M}(\mu_1)) \\
\texttt{Libre}_\text{M}(\mu\texttt{ as x}) & = & \texttt{Libre}_\text{M}(\mu) \\
\texttt{Libre}_\text{M}(\mu\texttt{ / }e) & = & \texttt{Libre}_\text{M}(\mu) \cup (\texttt{Libre}_\Sigma(e) - \texttt{Liee}_\text{M}(\mu)) \\
\texttt{Libre}_\text{M}(\mu_1\texttt{ | }\mu_2) & = & \texttt{Libre}_\text{M}(\mu_1) \cup \texttt{Libre}_\text{M}(\mu_2) \\
\texttt{Libre}_\text{M}(\mu_1\texttt{ \textbackslash/ }\mu_2) & = & \texttt{Libre}_\text{M}(\mu_1) \cup (\texttt{Libre}_\text{M}(\mu_2) - \texttt{Liee}_\text{M}(\mu_1)) \\
\texttt{Libre}_\text{M}(\mu\texttt{*}) & = & \texttt{Libre}_\text{M}(\mu) \\
\\
\texttt{Liee}_\text{M}(_) & = & \emptyset \\
\texttt{Liee}_\text{M}(cte) & = & \emptyset \\
\texttt{Liee}_\text{M}(\mu_1\texttt{ , }\mu_2) & = & \texttt{Liee}_\text{M}(\mu_1) \cup \texttt{Liee}_\text{M}(\mu_2) \\
\texttt{Liee}_\text{M}(\mu\texttt{ as x}) & = & \texttt{Liee}_\text{M}(\mu) \cup \{\texttt{x}, \texttt{\^{}x}\} \\
\texttt{Liee}_\text{M}(\mu\texttt{ / }e) & = & \texttt{Liee}_\text{M}(\mu) \\
\texttt{Liee}_\text{M}(\mu_1\texttt{ | }\mu_2) & = & \emptyset \\
\texttt{Liee}_\text{M}(\mu_1\texttt{ \textbackslash/ }\mu_2) & = & \texttt{Liee}_\text{M}(\mu_1) \cup \texttt{Liee}_\text{M}(\mu_2) \\
\texttt{Liee}_\text{M}(\mu\texttt{*}) & = & \emptyset
\end{array}$$

On peut noter que lorsqu'un motif est nommé par une variable `x`, la variable `^x` est également liée. En effet, lorsqu'un élément d'une collection est filtré, il correspond à un couple (position, valeur). Nous avons arbitrairement choisi d'accéder à la valeur de l'élément filtré par la variable donnée dans le motif, ici `x`. Ainsi, si le motif `x` a filtré l'élément (σ, v) d'une collection, l'expression `x+3` s'évaluera en $v + 3$. L'accès à la position se fait à l'aide de la variable `^x`. L'expression `^x+3` s'évaluera en $\sigma + 3$. Un autre point important du calcul des variables liées concerne la disjonction de motifs. En effet, nous statuons ici sur le fait que l'ensemble des variables de filtre d'un motif μ_1 `|` μ_2 est vide car ces définitions doivent rester locales. Cette contrainte permet d'éviter des règles telles que :

```
a | b => a+b
```

Puisqu'une seule branche de la disjonction est utilisée lors du filtrage d'une sous-collection, l'une des deux variables `a` ou `b` n'est pas liée. Cette remarque est également vraie pour l'itération de motif μ`*`.

Nous procédons de la même façon pour déterminer les variables libres et liées des motifs de patch :

$$\begin{array}{lcl}
\texttt{Libre}_\Pi(\texttt{x:[dim = }e_1\texttt{ , }e_2\texttt{ in faces, }e_3\texttt{ in cofaces, }e_4\texttt{]}) & = & \bigcup_i \texttt{Libre}_\Sigma(e_i) - \{\texttt{x}, \texttt{\^{}x}\} \\
\texttt{Libre}_\Pi(\texttt{\~{}x:[dim = }e_1\texttt{ , }e_2\texttt{ in faces, }e_3\texttt{ in cofaces, }e_4\texttt{]}) & = & \bigcup_i \texttt{Libre}_\Sigma(e_i) - \{\texttt{x}, \texttt{\^{}x}\} \\
\texttt{Libre}_\Pi(\pi_1\ \pi_2) \quad = \quad \texttt{Libre}_\Pi(\pi_1) \cup (\texttt{Libre}_\Pi(\pi_2) - \texttt{Liee}_\Pi(\pi_1)) & & \\
\\
\texttt{Liee}_\Pi(\texttt{x:[dim = }e_1\texttt{ , }e_2\texttt{ in faces, }e_3\texttt{ in cofaces, }e_4\texttt{]}) & = & \{\texttt{x}, \texttt{\^{}x}\} \\
\texttt{Liee}_\Pi(\texttt{\~{}x:[dim = }e_1\texttt{ , }e_2\texttt{ in faces, }e_3\texttt{ in cofaces, }e_4\texttt{]}) & = & \{\texttt{x}, \texttt{\^{}x}\} \\
\texttt{Liee}_\Pi(\pi_1\ \pi_2) & = & \texttt{Liee}_\Pi(\pi_1) \cup \texttt{Liee}_\Pi(\pi_2)
\end{array}$$

Ainsi, nous pouvons déterminer les variables libres de n'importe quelle expression mini-MGS. Soit l'expression e suivante

```
let z = 2 in
  trans<n,p>{ _ as x / (x==y), _ as y => x+z }
```

Les variables libres de cette expression sont réduites à l'ensemble :

$$\texttt{Libre}_{\Sigma}(e) = \{\texttt{y}\}$$

La variable `x` est liée en tant que variable de filtre, la variable `z` est liée à l'entier 2 par une définition locale, et les variables `n` et `p` correspondent à des arguments de la fonction résultant de la transformation (`n` et `p` ne sont pas libres au même titre que la variable `x` dans l'expression $\lambda\texttt{x}\cdot e$). Finalement, la variable `y` est utilisée avant sa propre définition. Les références en avant ne sont pas acceptées dans les motifs du mini-langage Σ. La variable `y` apparaît tout de même dans les variables de filtre du motif :

$$\texttt{Liee}_{\mathrm{M}}(\texttt{\_ as x / (x==y), \_ as y}) = \{\texttt{x}, \texttt{\^{}x}, \texttt{y}, \texttt{\^{}y}\}$$

Comme nous l'avons fait remarquer ci-dessus, l'ensemble des variables liées d'un motif itéré μ`*` est toujours vide. En effet, puisque le motif est une répétition, les variables définies ne peuvent servir que pour le filtrage de μ. Par exemple, pour le filtre

```
(_ as x, _ as y / x==y)*
```

les variables `x` et `y` n'ont de sens que dans le motif itéré. Le motif suivant n'a pas de sens :

```
(_ as x)* / x==1
```

Il est en effet impossible de savoir, parmi toutes les positions filtrées par `x`, laquelle doit avoir comme valeur 1. La variable `x` n'a pas de sens en dehors de l'itération.

1.2.2 Structure des motifs

Il est indispensable pour des questions de terminaison de l'algorithme de filtrage que les motifs filtrent *au moins* un élément. Les prédicats qui suivent permettent de vérifier que dans un motif (de chemin ou de patch) au moins un élément est consommé :

- Pour les transformations de chemin, l'itération de motif permet le filtrage d'un chemin de longueur nulle ; il faut donc s'assurer qu'un motif de chemin présente au moins un sous-motif en dehors d'une itération. Par exemple, `(_ as x)*` est interdit, alors que `(_ as x)*, _ as y` est autorisé, le sous-motif `_ as y` étant placé en dehors de toute itération consomme forcément un élément.
- Le problème apparaît sous une autre forme pour les motifs de patch. Les patches autorisent le filtrage sans consommation. Il s'agit d'un mécanisme qui permet d'utiliser la syntaxe des motifs pour sélectionner une sous-collection sans que celle-ci ne soit filtrée : les éléments filtrés mais non-consommés sont considérés comme non-filtrés après l'application de la règle (voir chapitre 2 page 52). Nous cherchons à assurer qu'au moins un élément est consommé par le motif.

On définit pour cela les fonctions `OkPath` et `OkPatch`.

Définition 24 (Le prédicat `OkPath`) *Le prédicat* $\texttt{OkPath}(\mu)$ *est vrai si dans le motif de chemin* μ *au moins un élément est consommé. Ce prédicat a pour signature :*

$$\texttt{OkPath} : \mathrm{M} \to \mathcal{B}$$

Le prédicat `OkPath` est défini inductivement sur la construction des motifs de chemin :

$$
\begin{array}{lcl}
\texttt{OkPath}(\_) & = & \text{true} \\
\texttt{OkPath}(cte) & = & \text{true} \\
\texttt{OkPath}(\mu_1\texttt{ , }\mu_2) & = & \texttt{OkPath}(\mu_1) \vee \texttt{OkPath}(\mu_2) \\
\texttt{OkPath}(\mu\texttt{ as x}) & = & \texttt{OkPath}(\mu) \\
\texttt{OkPath}(\mu\texttt{ / }e) & = & \texttt{OkPath}(\mu) \\
\texttt{OkPath}(\mu_1\texttt{ | }\mu_2) & = & \texttt{OkPath}(\mu_1) \wedge \texttt{OkPath}(\mu_2) \\
\texttt{OkPath}(\mu_1\texttt{ \textbackslash/ }\mu_2) & = & \texttt{OkPath}(\mu_1) \\
\texttt{OkPath}(\mu\texttt{*}) & = & \text{false}
\end{array}
$$

Définition 25 (Le prédicat `OkPatch`) *Le prédicat* $\texttt{OkPatch}(\pi)$ *est vrai si dans le motif de patch* π *au moins un élément est consommé. Ce prédicat a pour signature :*

$$\texttt{OkPatch} : \Pi \rightarrow \mathcal{B}$$

Le prédicat `OkPatch` est défini inductivement sur la construction des motifs de patch :

$$
\begin{array}{lcl}
\texttt{OkPatch(x:[dim =}\ e_1\texttt{, }e_2\texttt{ in faces, }e_3\texttt{ in cofaces, }e_4\texttt{])} & = & \text{true} \\
\texttt{OkPatch(\~{}x:[dim =}\ e_1\texttt{, }e_2\texttt{ in faces, }e_3\texttt{ in cofaces, }e_4\texttt{])} & = & \text{false} \\
\texttt{OkPatch}(\pi_1\ \pi_2) & = & \texttt{OkPatch}(\pi_1) \vee \texttt{OkPatch}(\pi_2)
\end{array}
$$

1.2.3 Validité des expressions

Voici les différentes contraintes imposées aux expressions construites sur la grammaire du langage mini-MGS :

- Dans une expression, des variables ne peuvent être utilisées que si elles sont définies (en d'autres termes, aucune variable libre ne doit apparaître).
- Dans un motif, il ne peut y avoir qu'une seule définition de chaque variable de filtre : les motifs sont *linéaires*.

Nous définissons un prédicat pour vérifier cela :

Définition 26 (Le prédicat `Correcte`) *Une expression e est* valide *si le prédicat* $\texttt{Correcte}_\Sigma(\emptyset, e)$ *est vrai. Ce prédicat est défini à l'aide des prédicats suivants :*

$$
\begin{array}{lcl}
\texttt{Correcte}_\Sigma & : & \mathcal{P}(\overline{\mathcal{X}}) \times \Sigma \rightarrow \mathcal{B} \\
\texttt{Correcte}_\Gamma & : & \mathcal{P}(\overline{\mathcal{X}}) \times \Gamma \rightarrow \mathcal{B} \\
\texttt{Correcte}_\mathrm{M} & : & \mathcal{P}(\overline{\mathcal{X}}) \times \mathrm{M} \rightarrow \mathcal{B} \\
\texttt{Correcte}_\Psi & : & \mathcal{P}(\overline{\mathcal{X}}) \times \Psi \rightarrow \mathcal{B} \\
\texttt{Correcte}_\Pi & : & \mathcal{P}(\overline{\mathcal{X}}) \times \mathrm{M} \rightarrow \mathcal{B}
\end{array}
$$

Le premier argument du prédicat est un ensemble de variables de $\overline{\mathcal{X}}$. Il s'agit des variables liées définissant le contexte dans lequel la validité de l'expression est considérée. Ces prédicats sont définis inductivement sur la construction des éléments de Σ.

$$
\begin{array}{lcl}
\texttt{Correcte}_\Sigma(l, cte) & = & \text{true} \\
\texttt{Correcte}_\Sigma(l, \texttt{x}) & = & \{\texttt{x}\} \in l \\
\texttt{Correcte}_\Sigma(l, \texttt{\^{}x}) & = & \{\texttt{\^{}x}\} \in l \\
\texttt{Correcte}_\Sigma(l, e_1\texttt{(}e_2\texttt{)}) & = & \texttt{Correcte}_\Sigma(l, e_1) \wedge \texttt{Correcte}_\Sigma(l, e_2) \\
\texttt{Correcte}_\Sigma(l, e_1\texttt{<}e_2\texttt{,}e_3\texttt{>(}e_4\texttt{)}) & = & \bigwedge_i \texttt{Correcte}_\Sigma(l, e_i) \\
\texttt{Correcte}_\Sigma(l, \lambda\texttt{x}\cdot e) & = & \texttt{Correcte}_\Sigma(l \cup \{\texttt{x}\}, e) \\
\texttt{Correcte}_\Sigma(l, \texttt{if }e_1\texttt{ then }e_2\texttt{ else }e_3) & = & \bigwedge_i \texttt{Correcte}_\Sigma(l, e_i) \\
\texttt{Correcte}_\Sigma(l, \texttt{trans<n,p>\{ ... ; }\gamma_i\texttt{ ; ... \}}) & = & \bigwedge_i \texttt{Correcte}_\Gamma(l \cup \{\texttt{n}, \texttt{p}, \texttt{self}\}, \gamma_i)
\end{array}
$$

$$\texttt{Correcte}_\Sigma(l, \texttt{patch\{ ... ; } \psi_i \texttt{ ; ... \}}) \quad = \quad \bigwedge\nolimits_i \texttt{Correcte}_\Psi(l \cup \{\texttt{self}\}, \psi_i)$$

Les cas de bases de cette définition inductive sont

- d'une part les constantes ne faisant référence à aucune variable, et qui sont par conséquent des expressions valides, et
- d'autre part les variables de $\overline{\mathcal{X}}$: si la variable n'appartient pas au contexte l, elle n'est pas définie, ce qui invalide l'expression.

Dans les cas d'induction, toutes les sous-expressions doivent être correctes. Le contexte est augmenté dans les cas de la λ-expression et des transformations.

Par exemple, les expressions $e_1 = \texttt{x::[]}$ et $e_2 = \texttt{let x = 2 in x::[]}$ sont respectivement non-valide et valide dans un contexte vide :

$$\begin{array}{rcl} \texttt{Correcte}_\Sigma(\emptyset, \texttt{x::[]}) & = & \texttt{Correcte}_\Sigma(\emptyset, \texttt{Cons}) \wedge \texttt{Correcte}_\Sigma(\emptyset, \texttt{x}) \wedge \texttt{Correcte}_\Sigma(\emptyset, \texttt{[]}) \\ & = & \text{true} \wedge \text{false} \wedge \text{true} \\ & = & \text{false} \end{array}$$

alors que

$$\begin{array}{rcl} \texttt{Correcte}_\Sigma(\emptyset, \texttt{let x = 2 in x::[]}) & = & \texttt{Correcte}_\Sigma(\emptyset, \texttt{(}\lambda\texttt{x}\cdot\texttt{x::[])(2)}) \\ & = & \texttt{Correcte}_\Sigma(\emptyset, 2) \wedge \texttt{Correcte}_\Sigma(\{\texttt{x}\}, \texttt{x::[]}) \\ & = & \text{true} \end{array}$$

Pour les transformations de chemins et les patches, les prédicats $\texttt{Correcte}_\Gamma$ et $\texttt{Correcte}_\Psi$ sont appelés récursivement sur chaque règle :

$$\begin{array}{rcl} \texttt{Correcte}_\Gamma(l, \mu \Rightarrow e) & = & \texttt{Correcte}_\text{M}(\mu) \wedge \texttt{OkPath}(\mu) \wedge \texttt{Correcte}_\Sigma(l \cup \texttt{Liee}_\text{M}(\mu), e) \\ \texttt{Correcte}_\Psi(l, \pi \Rightarrow e) & = & \texttt{Correcte}_\Pi(\pi) \wedge \texttt{OkPatch}(\pi) \wedge \texttt{Correcte}_\Sigma(l \cup \texttt{Liee}_\Pi(\pi), e) \end{array}$$

Pour chaque règle, on vérifie d'une part que la partie gauche est valide, puis on teste la validité de la partie droite en prenant en compte les variables de filtre définies dans le motif. À cela s'ajoute les tests de consommation, chaque motif devant filtrer au moins un élément (prédicats `OkPath` et `OkPatch`). La vérification des motifs de chemin se fait à l'aide du prédicat $\texttt{Correcte}_\text{M}$:

$$\begin{array}{rcl} \texttt{Correcte}_\text{M}(l, \_) & = & \text{true} \\ \texttt{Correcte}_\text{M}(l, cte) & = & \text{true} \\ \texttt{Correcte}_\text{M}(l, \mu_1\texttt{ , }\mu_2) & = & \texttt{Correcte}_\text{M}(l, \mu_1) \wedge \texttt{Correcte}_\text{M}(l \cup \texttt{Liee}_\text{M}(\mu_1), \mu_2) \\ & & \wedge\ (\texttt{Liee}_\text{M}(\mu_1) \cap \texttt{Liee}_\text{M}(\mu_2) = \emptyset) \\ \texttt{Correcte}_\text{M}(l, \mu\texttt{ as x}) & = & \texttt{Correcte}_\text{M}(l, \mu) \wedge \texttt{x} \notin \texttt{Liee}_\text{M}(\mu) \\ \texttt{Correcte}_\text{M}(l, \mu\texttt{ / }e) & = & \texttt{Correcte}_\text{M}(l, \mu) \wedge \texttt{Correcte}_\Sigma(l \cup \texttt{Liee}_\text{M}(\mu), e) \\ \texttt{Correcte}_\text{M}(l, \mu_1\texttt{ | }\mu_2) & = & \texttt{Correcte}_\text{M}(l, \mu_1) \wedge \texttt{Correcte}_\text{M}(l, \mu_2) \\ \texttt{Correcte}_\text{M}(l, \mu_1\texttt{ \textbackslash/ }\mu_2) & = & \texttt{Correcte}_\text{M}(l, \mu_1) \wedge \texttt{Correcte}_\text{M}(l \cup \texttt{Liee}_\text{M}(\mu_1), \mu_2) \\ & & \wedge\ (\texttt{Liee}_\text{M}(\mu_1) \cap \texttt{Liee}_\text{M}(\mu_2) = \emptyset) \\ \texttt{Correcte}_\text{M}(l, \mu\texttt{*}) & = & \texttt{Correcte}_\text{M}(l, \mu) \end{array}$$

Pour les constructions binaires μ_1`,` μ_2 et μ_1 `\/` μ_2, on vérifie que les deux motifs opérandes définissent des ensembles disjoints de variables de filtre ; on souhaite donc que $\texttt{Liee}_\text{M}(\mu_1) \cap \texttt{Liee}_\text{M}(\mu_2)$ soit vide. De la même façon, afin de lever toute ambiguïté la définition d'une variable de filtre μ `as x` n'est possible que si `x` n'est pas une variable de filtre du motif μ. La linéarité des motifs est assurée par ce prédicat. Ainsi, l'expression suivante n'est pas valide :

```
trans<n,p>{ _ as x, _ as x => x }
```

On lui préférera l'expression :

```
trans<n,p>{ _ as x, _ as y / (x == y) => x }
```

La validité des motifs de patch est donnée par le prédicat $\texttt{Correcte}_\Pi$:

$$\texttt{Correcte}_\Pi(l, \texttt{x:[dim =}\ e_1\texttt{,}\ e_2\ \texttt{in faces,}\ e_3\ \texttt{in cofaces,}\ e_4\ \texttt{]}) \quad = \quad \bigwedge_i \texttt{Correcte}_\Sigma(l \cup \{\texttt{x}, \texttt{\^{}x}\}, e_i)$$

$$\texttt{Correcte}_\Pi(l, \texttt{\~{}x:[dim =}\ e_1\texttt{,}\ e_2\ \texttt{in faces,}\ e_3\ \texttt{in cofaces,}\ e_4\ \texttt{]}) \quad = \quad \bigwedge_i \texttt{Correcte}_\Sigma(l \cup \{\texttt{x}, \texttt{\^{}x}\}, e_i)$$

$$\texttt{Correcte}_\Pi(l, \pi_1\ \pi_2) \quad = \quad \texttt{Correcte}_\Pi(l, \pi_1) \cup \texttt{Correcte}_\Pi(l \cup \texttt{Bound}_\Pi(\pi_1), \pi_2) \cup (\texttt{Bound}_\Pi(\pi_1) \cap \texttt{Bound}_\Pi(\pi_2) = \emptyset)$$

La linéarité des motifs de patch est assurée de la même façon que celle des motifs de chemin. Cependant, cette restriction n'apparaît pas pour les motifs de patch de l'interprète MGS présenté dans le chapitre 2. En fait, l'interprète dispose de sucre syntaxique pour la spécification des motifs de patch ; néanmoins tout motif accepté par l'interprète peut être réécrit en mini-MGS. Par exemple, considérons le motif suivant, filtrant un arc et ses sommets (les sommets ne sont pas consommés), tels que les valeurs associées à ces cellules vérifient le prédicat `Pred` :

```
~v1 < e:[ dim=1, Pred(e,v1,v2) ] > ~v2
```

Dans un premier temps, ce motif est étendu suivant la grammaire sans sucre syntaxique de l'interprète (voir chapitre 2 page 53) :

```
~v1:[ dim = <undef>, []        in faces, [^e] in cofaces, Pred(e,v1,v2) ]
  e:[ dim = 1,       [^v1,^v2] in faces, []   in cofaces, Pred(e,v1,v2) ]
~v2:[ dim = <undef>, []        in faces, [^e] in cofaces, Pred(e,v1,v2) ]
```

Les prédicats sont copiés dans toutes les clauses définissant une variable nécessaire à leur évaluation. Ici toutes les clauses sont mises en jeu dans l'évaluation de `Pred(e,v1,v2)`. Par conséquent cette garde est ajoutée à chaque clause. Finalement, les différentes clauses sont réordonnées (pour des raisons d'efficacité par exemple) et les conditions ne pouvant pas être évaluées sont supprimées :

```
~v1:[ dim = <undef>, []    in faces, []   in cofaces, true          ]
  e:[ dim = 1,       [^v1] in faces, []   in cofaces, true          ]
~v2:[ dim = <undef>, []    in faces, [^e] in cofaces, Pred(e,v1,v2) ]
```

2 Les domaines

Dans cette section, nous définissons le domaine des valeurs manipulées dans un programme ainsi que les domaines des règles de réduction pour la sémantique du langage mini-MGS. Pour chaque type de valeurs, les conventions d'écriture sont précisées.

2.1 Le domaine Val

Le langage mini-MGS permet la manipulation de deux types de valeurs : les valeurs atomiques (ou scalaires) et les valeurs structurées (ou collections topologiques).

2.1.1 Les scalaires

Les valeurs scalaires sont les suivantes :

- les valeurs entières dénotées par la variable n dont le domaine est $\mathbb{Z}$;

- les valeurs flottantes dénotées par la variable f dont le domaine est $\mathbb{F}$;
- les valeurs booléennes dénotées par la variable b dont le domaine est $\mathcal{B}$;
- les symboles dénotés par la variable s dont le domaine est $\mathbb{S}$;
- la valeur spéciale indéfinie `<undef>` ;
- les positions dénotées par la variable $\sigma^{\mathcal{K}}$ dont le domaine est $\mathcal{S}$ (défini chapitre 3 page 88) ;
- les *clôtures*, représentant les fonctions définies par le programmeur (mini-MGS étant un langage d'ordre supérieur). Les clôtures sont des couples $\langle e, \rho \rangle$ où $e \in \Sigma$ est une expression d'ordre supérieur et ρ est un environnement (voir la section 2.2). Il existe en mini-MGS trois expressions de ce type :
  1. λ`x`$\cdot e$: l'abstraction standard du λ-calcul ;
  2. `trans<n,p>{ ... ; `γ` ; ... }` : une transformation de chemins nécessitant deux entiers et une collection topologique pour être appliquée ;
  3. `patch{ ... ; `ψ` ; ... }` : un patch nécessitant également une collection topologique pour être appliquée.

  On note CIType l'ensemble des clôtures et la variable F en dénote les éléments.
- les *clôtures opaques*, représentant les fonctions prédéfinies de mini-MGS. Nous suivons en cela l'approche développée par G. Kahn dans [Kah87]. Les symboles de fonctions prédéfinies de $\mathcal{F}$ sont directement traduits en leur équivalent mathématique dans une version curryfiée, car mini-MGS ne permet l'application que d'un seul argument à la fois. Par exemple, une fonction f de signature

$$\mathrm{A} \times \mathrm{B} \rightarrow \mathrm{C}$$

est traduite en une clôture opaque fct_{f} de signature

$$\mathrm{A} \rightarrow \mathrm{B} \rightarrow \mathrm{C}$$

L'application d'un seul argument retourne alors une nouvelle clôture opaque. Par exemple, soit $a \in \mathrm{A}$, $fct_{\mathrm{f}}(a)$ est une clôture opaque de signature :

$$\mathrm{B} \rightarrow \mathrm{C}$$

Bien entendu, la clôture opaque fct_{f} est définie de telle sorte que pour $b \in \mathrm{B}$

$$\mathrm{f}(a, b) = fct_{\mathrm{f}}(a)(b)$$

Voici quelques exemples de fonctions prédéfinies :

– `PosGen` est traduit en fct_{posGen}. Il s'agit de la fonction posGen du chapitre 4, dans sa version curryfiée. La signature de posGen étant :

$$\mathbb{Z} \times \textsc{Seq}(\mathcal{S}) \times \textsc{Seq}(\mathcal{S}) \rightarrow \mathcal{S}$$

celle de fct_{posGen} est :

$$\mathbb{Z} \rightarrow \textsc{Seq}(\mathcal{S}) \rightarrow \textsc{Seq}(\mathcal{S}) \rightarrow \mathcal{S}$$

– `Extension` est traduit en fct_{ext} de signature Val $\rightarrow \mathcal{S} \rightarrow$ CollType(Val) (en considérant Val l'ensemble des valeurs du langage défini ci-après). Cette fonction crée de nouvelles collections topologiques dont une seule cellule $\sigma^{\mathcal{K}}$ est décorée par une valeur $v \in$ Val :

$$fct_{\mathrm{ext}}(v)(\sigma^{\mathcal{K}}) = (1.v).\sigma^{\mathcal{K}}$$

– `Concat` est traduit en fct_{concat} de signature

$$\mathsf{CollType}(\mathrm{Val}) \to \mathsf{CollType}(\mathrm{Val}) \to \mathsf{CollType}(\mathrm{Val})$$

et correspond à la somme de deux collections topologiques définie dans le chapitre 3 page 91 :

$$fct_{\text{concat}}(c_1)(c_2) = c_1 + c_2$$

– `Cons` est traduit en $fct_{::}$ de signature $\mathrm{Val} \to \textsc{Seq}(\mathrm{Val}) \to \textsc{Seq}(\mathrm{Val})$ de telle sorte que, pour $v \in \mathrm{Val}$ et $l \in \textsc{Seq}(\mathrm{Val})$

$$fct_{::}(v)(l) = v::l$$

On note $\mathsf{OpClType} = \{fct_{\text{posGen}}, fct_{\text{ext}}, fct_{\text{concat}}, fct_{::}, \ldots\}$ l'ensemble des clôtures opaques et la variable *fct* en dénote les éléments.

Finalement, le domaine des valeurs scalaires est défini par :

$$\mathsf{ScalType} = \mathbb{Z} \cup \mathcal{F} \cup \mathcal{B} \cup \mathbb{S} \cup \{\texttt{<undef>}\} \cup \mathcal{S} \cup \mathsf{ClType} \cup \mathsf{OpClType}$$

De façon générale, un scalaire est dénoté par la variable *sc*.

2.1.2 Les collections topologiques

L'ensemble des collections topologiques a été défini dans le chapitre 3, page 91. Ainsi, soit V un ensemble arbitraire de valeurs, on utilise l'ensemble des collections topologiques construites sur $\mathcal{S}$ à valeur dans V dénoté $\mathsf{CollType}(V)$. Nous considérons en particulier le sous-ensemble $\textsc{Seq}(V)$ de $\mathsf{CollType}(V)$ des séquences à valeur dans V. Deux constantes sont dédiées à leur construction :

1. la séquence vide notée `[]` (notons que les séquences sont des collections et que la séquence vide `[]` désigne également la collection vide $\{||\}$), et
2. la fonction prédéfinie `Cons` de $\mathcal{F}$.

Ces constantes ont pour objectif d'alléger l'écriture des programmes mini-MGS. En effet, les séquences sont énormément utilisées, d'une part pour la construction de nouvelles cellules topologiques (les deux derniers arguments de la fonction fct_{posGen} de $\mathsf{OpClType}$), et d'autre part pour la spécification la partie droite des règles des $\langle n, p\rangle$-transformations.

2.1.3 Le domaine Val

On définit alors le domaine Val, l'ensemble des valeurs manipulables en mini-MGS.

Définition 27 (Le domaine Val) *Le domaine* Val *des valeurs mini-MGS est le plus petit ensemble vérifiant l'équation :*

$$\mathrm{Val} = \mathsf{ScalType} \cup \mathsf{CollType}(\mathrm{Val})$$

Afin d'alléger l'écriture, l'ensemble des collections $\mathsf{CollType}(\mathrm{Val})$ est noté $\mathsf{CollType}$.

2.2 Les environnements

Une structure d'environnement est utilisée pour associer des valeurs aux variables de $\overline{\mathcal{X}}$.

Définition 28 (Environnement) *Un* environnement *est une fonction partielle de signature* $\mathcal{E} = \overline{\mathcal{X}} \to \text{Val}$ *qui associe une variable de* $\overline{\mathcal{X}}$ *à une valeur de* Val. *On note* $[\mathtt{x} \mapsto v]$ *(ou* $[\hat{}\mathtt{x} \mapsto v]$ *le cas échéant) la fonction qui associe la valeur* v *à la variable* `x` *et qui est indéfinie pour tous les autres éléments de* $\overline{\mathcal{X}}$.

De plus, si ρ *est un environnement, alors* $\rho' = \rho \uplus [\mathtt{x'} \mapsto v']$ *est l'environnement tel que :*

$$\rho'(\mathtt{x}) = \begin{cases} v' & \textit{si } \mathtt{x} = \mathtt{x'} \\ \rho(\mathtt{x}) & \textit{sinon} \end{cases}$$

L'écriture $[\mathtt{x} \mapsto v, \ldots, \mathtt{x'} \mapsto v']$ *est une abréviation de* $[\mathtt{x} \mapsto v] \uplus \cdots \uplus [\mathtt{x'} \mapsto v']$.

2.3 Les relations de réduction

Dans cette sous-section, nous définissons les signatures des relations de réduction utilisées pour définir la sémantique du langage mini-MGS. Il s'agit effectivement de relations, les transformations étant non-déterministes. Il est fréquent qu'une règle puisse s'appliquer sur plusieurs sous-collections distinctes. La relation associe alors tous les résultats possibles.

2.3.1 Evaluation des expressions

Cette première relation lie une expression à une valeur étant donné un environnement.

Définition 29 (Evaluation des expressions) *L'évaluation des expressions est donnée par une relation ternaire :*

$$\rho \vdash e \to v$$

de signature

$$\mathcal{E} \times \Sigma \times \text{Val}$$

se lisant : « dans l'environnement ρ*, l'expression* e *s'évalue en la valeur* v *»*

Pour alléger les écritures, on utilise également l'évaluation *typée* des expressions, où l'on précise le sous-ensemble de Val dans lequel l'expression doit s'évaluer :

$$\rho \vdash e \to_T v$$

où $T \subset \text{Val}$. Cela correspond à la règle d'inférence suivante :

$$\frac{\rho \vdash e \to v}{\rho \vdash e \to_T v}\ v \in T$$

2.3.2 Application des transformations

Les transformations sont des ensembles de règles de réécriture. L'évaluation de leur application requiert trois étapes différentes (voir le chapitre 2, page 47) :

1. l'application d'une règle de réécriture retournant un couple (sous-collection filtrée, sous-collection calculée) ;
2. la stratégie d'évaluation des règles, appliquant les différentes règles de la transformation et construisant une liste de couples (sous-collection filtrée, sous-collection calculée) ;

3. la reconstruction de la collection par réduction de la liste de couple (sous-collection filtrée, sous-collection calculée).

L'application d'une règle nécessite la définition de deux relations : l'une concerne le filtrage, et l'autre l'évaluation de la partie droite de la règle en fonction de la sous-collection filtrée. En revanche, les deux dernières étapes sont factorisées en une unique relation dont la définition dépend de la stratégie d'application des règles. Nous fournirons sa définition pour différentes stratégies plus loin dans le chapitre.

Pour résumer, trois relations doivent être définies :

1. pour la stratégie d'application des règles et la reconstruction,
2. pour l'application d'une règle (évaluation de la partie droite en fonction de la sous-collection filtrée), et
3. pour le filtrage.

Ces trois relations de réduction sont dédoublées selon qu'elles concernent les $\langle n, p\rangle$-transformations ou les patches : 6 relations sont donc à définir.

Stratégie d'application et reconstruction. Nous définissons dans ce paragraphe une relation liant un ensemble de règles de réécriture mini-MGS R et une collection *argument* $C^{\mathcal{K}}$ construite sur le complexe cellulaire $\mathcal{K}$ à une collection topologique *résultat* $C'^{\mathcal{K}'}$, dans le contexte d'un environnement ρ et sachant qu'un sous-ensemble de cellules topologiques $S \subset \mathcal{K}$ ont déjà été filtrées par ailleurs. Cette relation est de la forme :

$$S, \rho \vdash \{M\}(C^{\mathcal{K}}) \rightsquigarrow C'^{\mathcal{K}'}$$

Il faut conserver à l'esprit que la définition de cette relation est récursive ; par conséquent, on regroupe dans l'ensemble S les cellules ayant déjà été filtrées au cours de la descente récursive et qui ne peuvent plus l'être par la suite. Cet ensemble a une importance capitale : intuitivement le nombre de cellules topologiques pouvant être filtrées diminuant à mesure que S grossit, l'algorithme de filtrage termine.

Nous distinguons ci-dessous sa définition concernant les $\langle n, p\rangle$-transformations et les patches.

Définition 30 (Stratégie et reconstruction) *L'évaluation de l'application d'un ensemble de règles d'une $\langle n, p\rangle$-transformation est donnée par la relation :*

$$S, \rho \vdash \{\ldots ; \gamma_i ; \ldots\}(C) \rightsquigarrow_{n,p} C'$$

de signature

$$\mathcal{P}(\mathcal{S}) \times \mathcal{E} \times \textsc{Seq}(\Gamma) \times \mathsf{CollType} \times \mathbb{Z} \times \mathbb{Z} \times \mathsf{CollType}$$

et se lisant : « dans l'environnement ρ et sachant que les éléments de S ne peuvent être filtrés, la $\langle n, p\rangle$-transformation composée de l'ensemble de règles $\{\ldots ; \gamma_i ; \ldots\}$ transforme C en C' ».

L'évaluation de l'application d'un ensemble de règles d'un patch est donnée par la relation :

$$S, \rho \vdash \{\ldots ; \psi_i ; \ldots\}(C) \rightsquigarrow C'$$

de signature

$$\mathcal{P}(\mathcal{S}) \times \mathcal{E} \times \textsc{Seq}(\Psi) \times \mathsf{CollType} \times \mathsf{CollType}$$

et se lisant : « dans l'environnement ρ et sachant que les éléments de S ne peuvent être filtrés, le patch composé de l'ensemble de règles $\{\ldots ; \psi_i ; \ldots\}$ transforme C en C' ».

Application d'une règle. Nous nous plaçons à nouveau dans le contexte d'un environnement ρ et d'un ensemble de cellules topologiques déjà filtrées S. Nous définissons alors l'application d'une règle $m\texttt{=>}e$ sur une collection $C^{\mathcal{K}}$ retournant le couple $\langle$sous-collection filtrée C_m, sous-collection calculée $C_e\rangle$ par la relation

$$S, \rho \vdash (m\texttt{=>}e)(C^{\mathcal{K}}) \rightharpoonup \langle C_m, C_e\rangle$$

Sachant que la sous-collection C_m désigne les cellules topologiques de $\mathcal{K}$ filtrées par le motif m, une nouvelle application de règle entraîne la mise à jour de S pour prendre en compte les cellules filtrées. C'est par ce moyen que le nombre de cellules filtrées augmente, jusqu'à ce qu'il ne reste plus de cellule à filtrer.

Nous distinguons ci-dessous la définition de cette relation concernant les $\langle n, p\rangle$-transformations et les patches.

Définition 31 (Application d'une règle) *L'évaluation de l'application d'une règle de $\langle n, p\rangle$-transformation est donnée par la relation :*

$$S, \rho \vdash \gamma(C) \rightharpoonup_{n,p} \langle l_\mu, l_e\rangle$$

de signature

$$\mathcal{P}(\mathcal{S}) \times \mathcal{E} \times \Gamma \times \mathsf{CollType} \times \mathbb{Z} \times \mathbb{Z} \times \textsc{Seq}(\mathcal{S}) \times \textsc{Seq}(\mathrm{Val})$$

se lisant : « dans l'environnement ρ sachant que les éléments de S ne peuvent être filtrés, la règle de $\langle n, p\rangle$-chemin γ filtre le $\langle n, p\rangle$-chemin l_μ dans la collection C pour le subsituer par la séquence l_e ».

L'évaluation de l'application d'une règle de patch est donnée par la relation :

$$S, \rho \vdash \psi(C) \rightharpoonup \langle S_\pi, C_e\rangle$$

de signature

$$\mathcal{P}(\mathcal{S}) \times \mathcal{E} \times \Psi \times \mathsf{CollType} \times \mathcal{P}(\mathcal{S}) \times \mathsf{CollType}$$

se lisant : « dans l'environnement ρ sachant que les éléments de S ne peuvent être filtrés, la règle de patch ψ filtre les positions S_π dans la collection C pour les subsituer par la collection C_e ».

On définit également les prédicat suivants

$$\begin{array}{lcl} S, \rho \vdash \gamma(C) \not\rightharpoonup_{n,p} & \equiv & \nexists\, \langle l_\mu, l_e\rangle \in \textsc{Seq}(\mathcal{S}) \times \textsc{Seq}(\mathrm{Val}) \text{ tel que } S, \rho \vdash \gamma(C) \rightharpoonup_{n,p} \langle l_\mu, l_e\rangle \\ S, \rho \vdash \psi(C) \not\rightharpoonup & \equiv & \nexists\, \langle S_\pi, C_e\rangle \in \mathcal{P}(\mathcal{S}) \times \mathsf{CollType} \text{ tel que } S, \rho \vdash \psi(C) \rightharpoonup \langle S_\pi, C_e\rangle \end{array}$$

pour exprimer le fait qu'aucune sous-collection ne peut être filtrée par la règle.

Il reste à définir les relations de filtrage des motifs de chemin et de patch. Cependant, ces relations étant spécifiques au type de transformation, nous préférons les présenter séparément. Il est important de noter qu'une traduction du langage de filtre de l'un vers l'autre est impossible :

- les motifs de patch ont une expressivité qui dépasse le simple cadre des chemins ;
- les motifs de chemin autorisent des constructions telles que la disjonction ou l'itération de sous-motifs, qui n'ont pas d'équivalent dans les motifs de patch.

Filtrage de chemins. Le filtrage est un processus complexe pendant lequel une partie de la collection est sélectionnée. Afin de ne pas filtrer plusieurs fois le même élément, on utilise l'ensemble S des cellules topologiques déjà filtrées. L'évaluation du filtrage d'un motif de chemin est donc conditionnée par cet ensemble de cellules qu'il est interdit de filtrer. Le retour de l'évaluation du filtrage est un $\langle n, p\rangle$-chemin, c'est-à-dire une séquence de cellules topologiques (qui n'apparaissent pas dans S). La définition des variables de filtre est également prise en compte : l'évaluation du filtrage retourne un nouvel environnement dans lequel les valeurs filtrées sont associées aux variables de filtre.

Définition 32 (Filtrage de chemins) *L'évaluation du filtrage pour des motifs de chemin est donnée par la relation :*

$$S, \rho \ \vdash \ \mu(C) \ \hookrightarrow_{n,p} \ \langle l, \rho' \rangle$$

de signature

$$\mathcal{P}(\mathcal{S}) \times \mathcal{E} \times \mathrm{M} \times \mathsf{CollType} \times \mathbb{Z} \times \mathbb{Z} \times \textsc{Seq}(\mathcal{S}) \times \mathcal{E}$$

se lisant : « dans l'environnement ρ et sachant que les cellules de S ne peuvent pas être filtrées, le motif de $\langle n, p\rangle$-chemin μ filtre dans la collection C le $\langle n, p\rangle$-chemin l et augmente l'environnement ρ en ρ' »

Filtrage de patches. Le filtrage de patch fournit un mécanisme supplémentaire permettant de filtrer *sans consommer* une cellule (voir chapitre 2 page 53). Cela permet d'utiliser l'expressivité du langage de filtre sans réellement filtrer certaines cellules topologiques. Celles-ci peuvent déjà avoir été filtrées et consommées par une application de règle précédente.

Cependant, il faut noter que durant le filtrage, chaque clause du motif doit sélectionner une cellule topologique différente de celle filtrée par les autres clauses. L'ensemble S des cellules déjà filtrées et consommées par des applications de règles précédentes n'est alors plus suffisant, il faut également préciser un second ensemble de cellules topologiques qui ont été sélectionnées (consommées ou non) par le filtrage en cours. Nous nommons ces deux ensembles de la façon suivante :

- S_{prec} : l'ensemble des cellules filtrées **et** consommées par l'application de règles précédentes ;
- S_{cur} : l'ensemble des cellules filtrées, **consommées ou non**, par le filtrage en cours.

De façon orthogonale, le filtrage sélectionne deux sortes de cellules, les cellules consommées et les cellules non-consommées. Là encore nous distinguons ces deux ensembles par deux noms différents :

- S_{cons} : l'ensemble des cellules **consommées** ;
- $S_{\overline{\text{cons}}}$: l'ensemble des cellules **non-consommées**.

Ces notations sont utilisées pour définir le filtrage des motifs de patch :

Définition 33 (Filtrage de patches) *L'évaluation du filtrage pour des motifs de patch est donnée par la relation :*

$$S_{\text{prec}}, S_{\text{cur}}, \rho \ \vdash \ \pi(C) \ \hookrightarrow \ \langle S_{\text{cons}}, S_{\overline{\text{cons}}}, \rho' \rangle$$

de signature

$$\mathcal{P}(\mathcal{S}) \times \mathcal{P}(\mathcal{S}) \times \mathcal{E} \times \Pi \times \mathsf{CollType} \times \mathcal{P}(\mathcal{S}) \times \mathcal{P}(\mathcal{S}) \times \mathcal{E}$$

se lisant : « dans l'environnement ρ, les cellules de S_{prec} ayant été filtrées par une application de règle précédente, les cellules de S_{cur} ayant été filtrées par le filtrage courant, le motif de patch π filtre et consomme dans la collection C l'ensemble de cellules topologiques S_{cons}, filtre mais ne consomme pas les cellules topologiques de $S_{\overline{\text{cons}}}$, et augmente l'environnement ρ en ρ' »

3 La sémantique

La sémantique du langage mini-MGS est donnée dans le style de la sémantique naturelle, décrit dans [Kah87]. Elle consiste en une axiomatisation permettant la vérification d'appartenance aux relations de réduction définies précédemment. Par exemple, étant donné un environnement ρ, une expression e et une valeur v, la sémantique donne un ensemble de règles pour vérifier :

$$\rho \vdash e \rightarrow v$$

L'appartenance aux relations d'évaluation est donnée par les règles d'inférence définies de façon inductive sur la grammaire 1 du langage mini-MGS.

La présentation de cette sémantique suit la structure suivante : nous commençons par donner les règles d'inférence définissant l'évaluation du noyau fonctionnel du langage puis nous illustrons cette partie de la sémantique par un exemple de construction de collection topologique. Finalement, la sémantique de l'application des transformations de chemins et des patches est spécifiée à travers les règles d'inférence pour le filtrage, l'application d'une règle et la gestion des stratégies et de la reconstruction.

3.1 Sémantique standard d'un mini-ML

Le système de règles d'inférence 1 fournit une sémantique pour l'évaluation d'un langage de type ML. Il définit le noyau fonctionnel de notre langage. L'écriture $\mathcal{D}(\rho)$ dénote le domaine de

$$\frac{}{\rho \vdash cte \rightarrow cte} \quad (1)$$

$$\frac{}{\rho \vdash \mathtt{x} \rightarrow \rho(\mathtt{x})}\ \mathtt{x} \in \mathcal{D}(\rho) \quad (2)$$

$$\frac{\rho \vdash e_1 \rightarrow \langle \lambda \mathtt{x} \cdot e, \rho' \rangle \qquad \rho \vdash e_2 \rightarrow v \qquad \rho' \uplus [\mathtt{x} \mapsto v] \vdash e \rightarrow v'}{\rho \vdash e_1(e_2) \rightarrow v'} \quad (3)$$

$$\frac{\rho \vdash e_1 \rightarrow_{\mathsf{OpClType}} fct \qquad \rho \vdash e_2 \rightarrow v \qquad fct(v) = v'}{\rho \vdash e_1(e_2) \rightarrow v'} \quad (4)$$

$$\frac{}{\rho \vdash \lambda \mathtt{x} \cdot e \rightarrow \langle \lambda \mathtt{x} \cdot e, \rho \rangle} \quad (5)$$

$$\frac{\rho \vdash e_1 \rightarrow \text{true} \qquad \rho \vdash e_2 \rightarrow v}{\rho \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \rightarrow v} \qquad \frac{\rho \vdash e_1 \rightarrow \text{false} \qquad \rho \vdash e_3 \rightarrow v}{\rho \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \rightarrow v} \quad (6)$$

RÈG. 1: La sémantique du noyau fonctionnel de mini-MGS.

définition de la fonction ρ.

Ces règles sont empruntées à la sémantique de mini-ML que G. Kahn donne dans [Kah87]. On distingue notamment l'application d'une fonction prédéfinie et d'une λ-expression. C'est par les clôtures opaques que nous construisant les collections topologiques dont voici un exemple d'utilisation.

Exemple de construction des collections topologiques. Dans cet exemple, nous cherchons à construire un arc dont les deux sommets sont déjà créés, puis définir une collection topologique associant le symbole `'edge` à l'arc et `'vertex` à l'un des deux sommets. Soient $\sigma_1^{\mathcal{K}_1}$ et $\sigma_2^{\mathcal{K}_2}$, deux sommets isolés ; pour $i \in \{1, 2\}$

$$\mathcal{K}_i = (\{\sigma_i\}, \emptyset, \{\sigma_i \mapsto 0\})$$

Les complexes sont composés uniquement des cellules σ_i dont la dimension est 0. La relation d'incidence est vide puisqu'il n'y a qu'une seule cellule par complexe. Soient `v1` et `v2` deux variables respectivement liées à σ_1 et σ_2 dans l'environnement $\rho = [\texttt{v1} \mapsto \sigma_1^{\mathcal{K}_1}, \texttt{v2} \mapsto \sigma_2^{\mathcal{K}_2}]$. L'expression mini-MGS suivante crée l'arc et décore la structure :

```
let e = PosGen(1)([v1,v2])([]) in
  {| 'edge@e |}, {| 'vertex@v1 |}
```

En supprimant le sucre syntaxique, nous obtenons l'expression e suivante :

```
(\e·(Concat(Extension('edge)(e))
           (Extension('vertex)(v1))
)) (PosGen(1)([v1,v2])([]))
```

à évaluer dans l'environnement ρ. Dans cette application, une fois évaluée, la λ-expression forme une clôture

$$\langle \texttt{\textbackslash e·(Concat(Extension('edge)(e))(Extension('vertex)(v1)))}, \rho \rangle$$

et l'argument construit un nouvel arc bordé par `v1` et `v2` :

$$\dfrac{\dfrac{\vdots}{\rho \vdash \texttt{PosGen(1)([v1,v2])} \rightarrow fct_{\text{posGen}}(1)([\sigma_1^{\mathcal{K}_1}, \sigma_2^{\mathcal{K}_2}])}\,4 \qquad \rho \vdash \texttt{[]} \rightarrow []}{\rho \vdash \texttt{PosGen(1)([v1,v2])([])} \rightarrow \sigma_3^{\mathcal{K}_3} = fct_{\text{posGen}}(1)([\sigma_1^{\mathcal{K}_1}, \sigma_2^{\mathcal{K}_2}])([])}\,4$$

Nous appliquons en effet la règle 4 d'application des clôtures opaques. Observons le calcul de $\sigma_3^{\mathcal{K}_3}$. Rappelons l'égalité suivante :

$$fct_{\text{posGen}}(1)([\sigma_1^{\mathcal{K}_1}, \sigma_2^{\mathcal{K}_2}])([]) = \text{posGen}(1, [\sigma_1^{\mathcal{K}_1}, \sigma_2^{\mathcal{K}_2}], [])$$

Le prédicat predGen (voir page 89) est vérifié :

$$\begin{array}{lcl} \text{predGen}(1, [\sigma_1^{\mathcal{K}_1}; \sigma_2^{\mathcal{K}_2}], []) & = & dim^{\mathcal{K}_1}(\sigma_1) < 1 \wedge dim^{\mathcal{K}_2}(\sigma_2) < 1 \\ & = & \text{true} \end{array}$$

Le complexe cellulaire abstrait $\mathcal{K}_3$ est alors défini par :

$$\begin{array}{lcl} \mathcal{K}_3 & = & (\{\sigma_1, \sigma_2, \sigma_3\}, \{\sigma_1 \prec \sigma_3, \sigma_2 \prec \sigma_3\}, \{\sigma_1 \mapsto 0, \sigma_2 \mapsto 0, \sigma_3 \mapsto 1\}) \cup \mathcal{K}_1 \cup \mathcal{K}_2 \\ & = & (\{\sigma_1, \sigma_2, \sigma_3\}, \{\sigma_1 \prec \sigma_3, \sigma_2 \prec \sigma_3\}, \{\sigma_1 \mapsto 0, \sigma_2 \mapsto 0, \sigma_3 \mapsto 1\}) \end{array}$$

dans lequel σ_3 apparaît comme l'arc incident aux deux sommets σ_1 et σ_2 comme nous souhaitions le construire.

Pour revenir à l'évaluation de l'expression e ci-dessus, on associe la cellule $\sigma_3^{\mathcal{K}_3}$ à la variable `e` dans l'environnement capturé par la clôture de la λ-expression, comme l'indique la règle 3. Un nouvel environnement ρ' est calculé :

$$\begin{array}{rcl} \rho' & = & \rho \uplus [\texttt{e} \mapsto \sigma_3^{\mathcal{K}_3}] \\ & = & [\texttt{e} \mapsto \sigma_3^{\mathcal{K}_3}, \texttt{v1} \mapsto \sigma_1^{\mathcal{K}_1}, \texttt{v2} \mapsto \sigma_2^{\mathcal{K}_2}] \end{array}$$

La sous-expression $e' =$ `Concat(Extension(‘edge)(e))(Extension(‘vertex)(v1))` est finalement évaluée dans l'environnement ρ'. Soient

- $e_1 =$ `Extension(‘edge)(e)`
- $e_2 =$ `Extension(‘vertex)(v1)`

La fonction `Extension` permet d'associer une valeur à une cellule topologique :

$$\frac{\vdots}{\rho' \vdash \texttt{Extension(‘edge)(e)} \rightarrow \texttt{‘edge}.\sigma_3^{\mathcal{K}_3}}\ 4$$

En effet, l'expression {| ‘edge @ e |} constuit la collection topologique suivante :

$$0_{\text{Abel(Val)}}.\sigma_1^{\mathcal{K}_3} + 0_{\text{Abel(Val)}}.\sigma_2^{\mathcal{K}_3} + (1.\texttt{‘edge}).\sigma_3^{\mathcal{K}_3} = (1.\texttt{‘edge}).\sigma_3^{\mathcal{K}_3}$$

Dans la suite, les positions décorées par $0_{\text{Abel(Val)}}$ ne seront plus écrites suivant les notations introduites dans le chapitre 3. De plus, nous abrégeons également la notation $(1.\texttt{‘edge}).\sigma_3^{\mathcal{K}_3}$ en $\texttt{‘edge}.\sigma_3^{\mathcal{K}_3}$; nous supposons que l'injection canonique Inj_{Val} est utilisée partout où c'est nécessaire. On évalue l'expression e_2 de la même façon :

$$\frac{\vdots}{\rho' \vdash \texttt{Extension(‘vertex)(v1)} \rightarrow \texttt{‘vertex}.\sigma_1^{\mathcal{K}_1}}\ 4$$

Finalement, la concaténation agit pour construire le résultat de notre calcul :

$$\frac{\dfrac{\vdots}{\rho' \vdash e_1 \rightarrow \texttt{‘edge}.\sigma_3^{\mathcal{K}_3}}\ 4 \quad \dfrac{\vdots}{\rho' \vdash e_2 \rightarrow \texttt{‘vertex}.\sigma_1^{\mathcal{K}_1}}\ 4 \quad C = \texttt{‘edge}.\sigma_3^{\mathcal{K}_3} + \texttt{‘vertex}.\sigma_1^{\mathcal{K}_1}}{\rho' \vdash e' \rightarrow C}\ 4$$

La collection résultat C s'évalue alors de la façon suivante :

$$\begin{array}{rcl} C & = & \texttt{‘vertex}.\sigma_1^{\mathcal{K}_1} + \texttt{‘edge}.\sigma_3^{\mathcal{K}_3} \\ & = & \texttt{‘vertex}.\sigma_1^{\mathcal{K}_1}\!\downarrow(\mathcal{K}_1 \cup \mathcal{K}_3) + \texttt{‘edge}.\sigma_3^{\mathcal{K}_3}\!\downarrow(\mathcal{K}_1 \cup \mathcal{K}_3) \\ & = & \texttt{‘vertex}.\sigma_1^{\mathcal{K}_3} + \texttt{‘edge}.\sigma_3^{\mathcal{K}_3} \end{array}$$

Il est en effet trivial de vérifier que $\mathcal{K}_3 \cup \mathcal{K}_1 = \mathcal{K}_3$.

3.2 Construction des clôtures et application des transformations

Dans cette section, nous définissons les règles d'inférence pour l'application des transformations. L'induction dont dépend la définition de la relation d'évaluation des expressions est alors complète. Le système de règles 2 fait le lien entre l'application d'un ensemble de règles (que nous verrons dans les sections suivantes) et l'évaluation des expressions.

Les règles 7 et 8 créent les clôtures pour l'application future des transformations, et les règles 9 et 10 évaluent cette application :

$$\frac{}{\rho \vdash \texttt{trans<n,p>\{\ldots\}} \rightarrow \langle \texttt{trans<n,p>\{\ldots\}}, \rho \rangle} \quad (7)$$

$$\frac{}{\rho \vdash \texttt{patch\{\ldots\}} \rightarrow \langle \texttt{patch\{\ldots\}}, \rho \rangle} \quad (8)$$

$$\frac{\begin{array}{l}\rho \vdash e_1 \rightarrow \langle \texttt{trans<n,p>\{\ldots\}}, \rho' \rangle \\ \rho \vdash e_2 \rightarrow_{\mathbb{Z}} n \\ \rho \vdash e_3 \rightarrow_{\mathbb{Z}} p \\ \rho \vdash e_4 \rightarrow_{\mathsf{CollType}} C\end{array} \qquad \emptyset, \rho' \uplus \left[\begin{array}{l}\texttt{self} \mapsto C, \\ \texttt{n} \mapsto n, \\ \texttt{p} \mapsto p\end{array}\right] \vdash \{\ldots\}(C) \rightsquigarrow_{n,p} C'}{\rho \vdash e_1\texttt{<}e_2, e_3\texttt{>(}e_4\texttt{)} \rightarrow C'} \quad (9)$$

$$\frac{\begin{array}{l}\rho \vdash e_1 \rightarrow \langle \texttt{patch\{\ldots\}}, \rho' \rangle \\ \rho \vdash e_2 \rightarrow_{\mathsf{CollType}} C\end{array} \qquad \emptyset, \rho' \uplus [\texttt{self} \mapsto C] \vdash \{\ldots\}(C) \rightsquigarrow C'}{\rho \vdash e_1\texttt{(}e_2\texttt{)} \rightarrow C'} \quad (10)$$

Règ. 2: Règles de sémantique pour l'application des transformations.

- La règle 9 : l'application d'une transformation de chemins passe par la syntaxe spéciale $e_1\texttt{<}e_2, e_3\texttt{>(}e_4\texttt{)}$. La première expression correspond à la clôture encodant la transformation. Les expressions e_2 et e_3 s'évaluent en deux entiers pour fixer les paramètres n et p de la transformation. Finalement e_4 est la collection topologique C passée en argument à la transformation. L'environnement est étendu en associant respectivement aux variables `n`, `p` et `self` les valeurs n, p et C. Finalement, l'ensemble des cellules déjà filtrées est initialement vide.
- La règle 10 : l'application d'un patch utilise la syntaxe standard de l'application. Exceptée la gestion des dimensions n et p pour les $\langle n, p \rangle$-transformations, cette règle suit la même structure que la règle 9.

Les sections suivantes sont organisées comme suit : nous commençons par la définition de la relation évaluant le filtrage d'un motif dans une collection. Nous continuons avec l'évaluation de l'application d'une règle sur une collection. Finalement, nous terminons avec la définition des stratégies et la gestion de la reconstruction avec la dernière relation d'évaluation de l'application d'un ensemble de règles de réécriture.

3.3 Filtrage

Dans le cas général, la donnée d'un motif et d'une collection topologique ne détermine pas une unique sous-collection filtrée. En effet, plusieurs sous-collections peuvent se présenter comme des instances de filtre. Le non-déterminisme du langage provient du filtrage : il n'existe pas une unique solution à l'évaluation du filtrage. Les règles d'inférence que nous proposons permettent de vérifier qu'une sous-collection de chemin (resp. de patch) est bien filtrée par un motif de chemin (resp. de patch).

Les grammaires de filtrage différant énormément entre $\langle n, p\rangle$-transformations et patches, nous les présentons séparément.

Filtrage de chemins. Le système de règles 3 donne la sémantique du filtrage de chemin. En voici une description règle par règle :

- Règle 11 : il s'agit du premier cas de base de l'induction sur la construction syntaxique des motifs. Le joker filtre un chemin de longueur 1 composé d'une seule cellule. Celle-ci doit être de dimension n dans le cadre d'une $\langle n, p\rangle$-transformation et ne doit pas avoir été filtrée par ailleurs (première condition : $\sigma \in \mathcal{K}_n - S$), et doit être décorée. Il n'existe donc qu'un nombre fini de cellules pouvant être filtrées. L'environnement n'est pas modifié.
- Règle 12 : filtre équivalent au joker, mais où l'on souhaite que la valeur associée à la cellule soit égale à cte[4].
- Règle 13 : ce motif assure la poursuite du filtrage. Les motifs μ_1 et μ_2 filtrent respectivement deux chemins l_1 et l_2. Ces deux chemins sont alors concaténés pour n'en former qu'un : $l_1@l_2$. Pour filtrer l_2, l'ensemble S des cellules déjà filtrées est complété par les cellules du chemin l_1, celles-ci ne devant pas être filtrées une seconde fois par le motif μ_2. Ainsi, les deux chemins l_1 et l_2 ne peuvent pas se recouvrir. L'environnement d'évaluation de μ_2 est ρ_1 pour prendre en compte les variables de filtre définies lors du filtrage de μ_1. L'environnement retourné pour le filtrage de $l_1@l_2$ est ρ_2, c'est-à-dire l'environnement retourné par le filtrage de l_2.
- Règle 14 : cette règle illustre la mise à jour d'un environnement par la définition d'une variable de filtre. L'environnement ρ est modifié pour tenir compte du nommage du chemin filtré par μ : $\rho' \uplus [\mathtt{x} \mapsto C(l), \mathtt{\hat{}x} \mapsto l]$. D'une part nous associons à la variable `^x` le chemin l filtré par μ (une séquence de positions) ; d'autre part, la variable `x` réfère à la séquence des valeurs successives associées aux cellules de l.
- Règle 15 : cette règle permet le filtrage de chemin gardé ; l'expression doit être évaluée en la valeur true.
- Règles 16 : deux règles définissent les deux branches possibles de la disjonction de motif.
- Règle 17 : cette règle permet de continuer le filtrage dans l'élément filtré. On suppose pour cela que μ_1 filtre un chemin de longueur 1, dont la cellule est décorée par une collection topologique $C(\sigma)$. Cela permet d'empêcher par exemple l'évaluation de motifs tels que `(w, x, y) \/ z` qui n'a pas de sens. La procédure de filtrage est alors exécutée sur la collection $C(\sigma)$ avec un ensemble de cellules déjà filtrées vide. En effet, le filtrage se poursuit sur une collection topologique différente. Il est important de remarquer que cette règle ne correspond pas à la sémantique implantée dans l'interprète MGS. Dans le chapitre 2, page 49, il est précisé que pour le motif

  ```
  w, (x \/ (4, y)) as z
  ```

  les variables `x` et `z` ne sont pas équivalentes : `x` correspond à la collection filtrée par la descente dans laquelle le chemin filtré par le sous-motif `(4, y)` est supprimé, alors que `z` correspond à la même collection où le sous-chemin est toujours présent. Dans le cadre de mini-MGS, les deux variables adressent la même valeur : la sous-collection filtrée dans laquelle le sous-chemin n'est pas supprimé. Nous avons souhaité rester simples dans la définition de

[4]Pour être rigoureux, on devrait écrire $C(\sigma) = 1.cte$; on suppose que l'injection canonique $\mathrm{Inj}_{\mathrm{Val}}$ est utilisée partout où cela est nécessaire.

$$\frac{}{S,\rho \vdash \_(C^{\mathcal{K}}) \hookrightarrow_{n,p} \langle [\sigma^{\mathcal{K}}], \rho\rangle} \quad \begin{cases}\sigma^{\mathcal{K}} & \notin \mathcal{K}_n - S \\ C^{\mathcal{K}}(\sigma^{\mathcal{K}}) & \neq 0_{\mathrm{Abel(Val)}}\end{cases} \tag{11}$$

$$\frac{}{S,\rho \vdash (cte)(C^{\mathcal{K}}) \hookrightarrow_{n,p} \langle [\sigma^{\mathcal{K}}], \rho\rangle} \quad \begin{cases}\sigma^{\mathcal{K}} & \notin \mathcal{K}_n - S \\ C^{\mathcal{K}}(\sigma^{\mathcal{K}}) & = cte\end{cases} \tag{12}$$

$$\frac{S,\rho \vdash \mu_1(C) \hookrightarrow_{n,p} \langle l_1, \rho_1\rangle \qquad S \sqcup l_1, \rho_1 \vdash \mu_2(C) \hookrightarrow_{n,p} \langle l_2, \rho_2\rangle \qquad (\mathrm{last}\ l_1),^n_p(\mathrm{hd}\ l_2)}{S,\rho \vdash (\mu_1\,,\,\mu_2)(C) \hookrightarrow_{n,p} \langle l_1 @ l_2, \rho_2\rangle} \tag{13}$$

$$\frac{S,\rho \vdash \mu(C) \hookrightarrow_{n,p} \langle l, \rho'\rangle}{S,\rho \vdash (\mu\ \mathtt{as}\ \mathtt{x})(C) \hookrightarrow_{n,p} \langle l, \rho' \uplus [\mathtt{x} \mapsto C(l), \hat{}\mathtt{x} \mapsto l]\rangle} \tag{14}$$

$$\frac{S,\rho \vdash \mu(C) \hookrightarrow_{n,p} \langle l, \rho'\rangle \qquad \rho' \vdash e \to \mathrm{true}}{S,\rho \vdash (\mu\ /\ e)(C) \hookrightarrow_{n,p} \langle l, \rho'\rangle} \tag{15}$$

$$\frac{S,\rho \vdash \mu_1(C) \hookrightarrow_{n,p} \langle l_1, \rho_1\rangle}{S,\rho \vdash (\mu_1 \mid \mu_2)(C) \hookrightarrow_{n,p} \langle l_1, \rho_1\rangle} \qquad \frac{S,\rho \vdash \mu_2(C) \hookrightarrow_{n,p} \langle l_2, \rho_2\rangle}{S,\rho \vdash (\mu_1 \mid \mu_2)(C) \hookrightarrow_{n,p} \langle l_2, \rho_2\rangle} \tag{16}$$

$$\frac{S,\rho \vdash \mu_1(C) \hookrightarrow_{n,p} \langle [\sigma], \rho_1\rangle \qquad \emptyset, \rho_1 \vdash \mu_2(C(\sigma)) \hookrightarrow_{n,p} \langle \_, \rho_2\rangle \qquad C(\sigma) \in \mathsf{CollType}}{S,\rho \vdash (\mu_1 \setminus\!/\ \mu_2)(C) \hookrightarrow_{n,p} \langle [\sigma], \rho_2\rangle} \tag{17}$$

$$\frac{}{S,\rho \vdash (\mu*)(C) \hookrightarrow_{n,p} \langle [], \rho\rangle} \qquad \frac{S,\rho \vdash (\mu\,,\,\mu*)(C) \hookrightarrow_{n,p} \langle l, \rho\rangle}{S,\rho \vdash (\mu*)(C) \hookrightarrow_{n,p} \langle l, \rho\rangle} \tag{18}$$

Règ. 3: Règles de sémantique pour le filtrage de chemin.

la sémantique ; la suppression du sous-chemin demande l'application d'une transformation construite à la volée associant la valeur spéciale `<undef>` aux cellules filtrées.

- Règles 18 : elles implantent l'itération de motif. La première est un axiome pour filtrer le chemin vide [] correspondant à un nombre de répétition nulle du motif ; la seconde réécrit le motif μ`*` en μ`, `μ`*`. Ces deux règles sont la traduction de l'égalité :

$$\mu\texttt{*} = \varepsilon \mid \mu\texttt{, }\mu\texttt{*}$$

où ε dénote le filtre de longueur 0. Nous rappelons que grâce au prédicat `OkPath`, il est assuré que le filtrage d'un motif de chemin consomme au moins un élément.

Filtrage de patches. Le système de règles 4 donnent la sémantique du filtrage de patch. En voici une description règle par règle :

- Règle 19 : Le motif filtre et consomme une cellule σ de dimension n donnée par l'expression e_1. Elle doit également respecter les incidences provenant de l'évaluation des expressions e_2 et e_3. Pour finir, la condition e_4 doit être vérifiée.
  La cellule σ ne doit pas appartenir à l'ensemble S_{cur} des cellules sélectionnées par le filtrage en cours. De plus, σ n'appartient pas à l'ensemble S_{prec} des cellules consommées par l'application d'une règle précédente.
  Finalement, la relation retourne le triplet $\langle\{\sigma\}, \emptyset, \rho'\rangle$:
  1. $\{\sigma\}$: la cellule filtrée et consommée ;
  2. $\emptyset$: aucune cellule n'est filtrée sans être consommée ;
  3. ρ' : le nouvel environnement pour la suite du filtrage.

  L'environnement ρ' montre une particularité ; la variable `^x` ne réfère pas à $\sigma^{\mathcal{K}}$, la cellule filtrée dans le complexe $\mathcal{K}$, mais à $\sigma^{\mathcal{K}'}$ la cellule filtrée dans le complexe

  $$\mathcal{K}' = (\{\sigma\}, \emptyset, \{\sigma \mapsto n\})$$

  Ce complexe cellulaire ne contient que la cellule σ. En effet, la spécification de la sous-collection reconstruite en partie droite de la règle peut faire référence à la variable `^x`. Or si celle-ci est prise dans le contexte du complexe cellulaire $\mathcal{K}$, la structure construite en partie droite pourrait contenir toutes les cellules de $\mathcal{K}$. Or les patches sont destinés à spécifier des modifications topologiques où des cellules peuvent disparaître. Utiliser $\mathcal{K}$ empêcherait toute suppression de cellule et rendrait les patches inutilisables.
- Règle 20 : Cette règle est exactement la même que la règle 19, excepté pour l'expression e_1 qui ne spécifie pas la dimension de la cellule filtrée. Dans ce cas, la dimension n'importe pas.
- Règle 21 : Cette règle diffère de la règle 19 en ce que la cellule filtrée n'est pas consommée. Elle ne doit toujours pas appartenir à l'ensemble S_{cur} des cellules filtrées par le filtrage en cours pour éviter qu'une même cellule soit filtrées par deux clauses différentes. En revanche, elle peut appartenir à S_{prec}, l'ensemble des cellules consommées par des applications de règles antérieures : elle n'est pas consommée par le filtrage en cours.
  La relation retourne le triplet $\langle\emptyset, \{\sigma\}, \rho'\rangle$:
  1. $\emptyset$: aucune cellule n'est à la fois filtrée et consommée ;
  2. $\{\sigma\}$: la cellule filtrée et non-consommée ;
  3. ρ' : le nouvel environnement pour la suite du filtrage.

$$\frac{\begin{array}{ll} v = C(\sigma^{\mathcal{K}}) & \\ v \neq 0_{\text{Abel(Val)}} & \rho' \vdash e_1 \rightarrow_{\mathbb{Z}} n \\ n = dim^{\mathcal{K}}(\sigma^{\mathcal{K}}) & \rho' \vdash e_2 \rightarrow_{\text{SEQ}(S^{\mathcal{K}})} [\ldots;\sigma_i^{\mathcal{K}};\ldots] \quad \forall i,\ \sigma_i^{\mathcal{K}} \in \texttt{faces}\ \sigma^{\mathcal{K}} \\ \mathcal{K}' = (\{\sigma\}, \emptyset, \{\sigma \mapsto n\}) & \rho' \vdash e_3 \rightarrow_{\text{SEQ}(S^{\mathcal{K}})} [\ldots;\sigma'^{\mathcal{K}}_j;\ldots] \quad \forall j,\ \sigma'^{\mathcal{K}}_j \in \texttt{cofaces}\ \sigma^{\mathcal{K}} \\ \rho' = \rho \uplus [\texttt{x} \mapsto v, \texttt{\^{}x} \mapsto \sigma^{\mathcal{K}'}] & \rho' \vdash e_4 \rightarrow \text{true} \\ \sigma^{\mathcal{K}} \notin S_{\text{prec}} \cup S_{\text{cur}} & \end{array}}{S_{\text{prec}}, S_{\text{cur}}, \rho \vdash \texttt{x:[dim =}\ e_1\texttt{,}\ e_2\ \texttt{in faces,}\ e_3\ \texttt{in cofaces,}\ e_4\ \texttt{]}(C) \hookrightarrow \langle \{\sigma\}, \emptyset, \rho' \rangle} \quad (19)$$

$$\frac{\begin{array}{ll} v = C(\sigma^{\mathcal{K}}) & \\ v \neq 0_{\text{Abel(Val)}} & \rho' \vdash e_1 \rightarrow \texttt{<undef>} \\ \mathcal{K}' = (\{\sigma\}, \emptyset, \{\sigma \mapsto dim^{\mathcal{K}}(\sigma^{\mathcal{K}})\}) & \rho' \vdash e_2 \rightarrow_{\text{SEQ}(S^{\mathcal{K}})} [\ldots;\sigma_i^{\mathcal{K}};\ldots] \quad \forall i,\ \sigma_i^{\mathcal{K}} \in \texttt{faces}\ \sigma^{\mathcal{K}} \\ \rho' = \rho \uplus [\texttt{x} \mapsto v, \texttt{\^{}x} \mapsto \sigma^{\mathcal{K}'}] & \rho' \vdash e_3 \rightarrow_{\text{SEQ}(S^{\mathcal{K}})} [\ldots;\sigma'^{\mathcal{K}}_j;\ldots] \quad \forall j,\ \sigma'^{\mathcal{K}}_j \in \texttt{cofaces}\ \sigma^{\mathcal{K}} \\ \sigma^{\mathcal{K}} \notin S_{\text{prec}} \cup S_{\text{cur}} & \rho' \vdash e_4 \rightarrow \text{true} \end{array}}{S_{\text{prec}}, S_{\text{cur}}, \rho \vdash \texttt{x:[dim =}\ e_1\texttt{,}\ e_2\ \texttt{in faces,}\ e_3\ \texttt{in cofaces,}\ e_4\ \texttt{]}(C) \hookrightarrow \langle \{\sigma\}, \emptyset, \rho' \rangle} \quad (20)$$

$$\frac{\begin{array}{ll} v = C(\sigma^{\mathcal{K}}) & \\ v \neq 0_{\text{Abel(Val)}} & \rho' \vdash e_1 \rightarrow_{\mathbb{Z}} n \\ n = dim^{\mathcal{K}}(\sigma^{\mathcal{K}}) & \rho' \vdash e_2 \rightarrow_{\text{SEQ}(S^{\mathcal{K}})} [\ldots;\sigma_i^{\mathcal{K}};\ldots] \quad \forall i,\ \sigma_i^{\mathcal{K}} \in \texttt{faces}\ \sigma^{\mathcal{K}} \\ \mathcal{K}' = (\{\sigma\}, \emptyset, \{\sigma \mapsto n\}) & \rho' \vdash e_3 \rightarrow_{\text{SEQ}(S^{\mathcal{K}})} [\ldots;\sigma'^{\mathcal{K}}_j;\ldots] \quad \forall j,\ \sigma'^{\mathcal{K}}_j \in \texttt{cofaces}\ \sigma^{\mathcal{K}} \\ \rho' = \rho \uplus [\texttt{x} \mapsto v, \texttt{\^{}x} \mapsto \sigma^{\mathcal{K}'}] & \rho' \vdash e_4 \rightarrow \text{true} \\ \sigma^{\mathcal{K}} \notin S_{\text{cur}} & \end{array}}{S_{\text{prec}}, S_{\text{cur}}, \rho \vdash \texttt{\~{}x:[dim =}\ e_1\texttt{,}\ e_2\ \texttt{in faces,}\ e_3\ \texttt{in cofaces,}\ e_4\ \texttt{]}(C) \hookrightarrow \langle \emptyset, \{\sigma\}, \rho' \rangle} \quad (21)$$

$$\frac{\begin{array}{ll} v = C(\sigma^{\mathcal{K}}) & \\ v \neq 0_{\text{Abel(Val)}} & \rho' \vdash e_1 \rightarrow \texttt{<undef>} \\ \mathcal{K}' = (\{\sigma\}, \emptyset, \{\sigma \mapsto dim^{\mathcal{K}}(\sigma^{\mathcal{K}})\}) & \rho' \vdash e_2 \rightarrow_{\text{SEQ}(S^{\mathcal{K}})} [\ldots;\sigma_i^{\mathcal{K}};\ldots] \quad \forall i,\ \sigma_i^{\mathcal{K}} \in \texttt{faces}\ \sigma^{\mathcal{K}} \\ \rho' = \rho \uplus [\texttt{x} \mapsto v, \texttt{\^{}x} \mapsto \sigma^{\mathcal{K}'}] & \rho' \vdash e_3 \rightarrow_{\text{SEQ}(S^{\mathcal{K}})} [\ldots;\sigma'^{\mathcal{K}}_j;\ldots] \quad \forall j,\ \sigma'^{\mathcal{K}}_j \in \texttt{cofaces}\ \sigma^{\mathcal{K}} \\ \sigma^{\mathcal{K}} \notin S_{\text{cur}} & \rho' \vdash e_4 \rightarrow \text{true} \end{array}}{S_{\text{prec}}, S_{\text{cur}}, \rho \vdash \texttt{\~{}x:[dim =}\ e_1\texttt{,}\ e_2\ \texttt{in faces,}\ e_3\ \texttt{in cofaces,}\ e_4\ \texttt{]}(C) \hookrightarrow \langle \emptyset, \{\sigma\}, \rho' \rangle} \quad (22)$$

$$\frac{\begin{array}{l} S_{\text{prec}}, S_{\text{cur}}, \rho \vdash \pi_1(C) \hookrightarrow \langle S_{\text{cons}}, S_{\overline{\text{cons}}}, \rho_1 \rangle \\ S_{\text{prec}}, S_{\text{cur}} \cup S_{\text{cons}} \cup S_{\overline{\text{cons}}}, \rho_1 \vdash \pi_2(C_1) \hookrightarrow \langle S'_{\text{cons}}, S'_{\overline{\text{cons}}}, \rho_2 \rangle \end{array}}{S_{\text{prec}}, S_{\text{cur}}, \rho \vdash \pi_1\ \pi_2(C) \hookrightarrow \langle S_{\text{cons}} \cup S'_{\text{cons}}, S_{\overline{\text{cons}}} \cup S'_{\overline{\text{cons}}}, \rho_2 \rangle} \quad (23)$$

RÈG. 4: Règles de sémantique pour le filtrage de patch.

$$\frac{S,\rho \vdash \mu(C) \hookrightarrow_{n,p} \langle l_\mu, \rho_\mu\rangle \qquad \rho_\mu \vdash e \rightarrow_{\text{SEQ(Val)}} l_e \qquad |l_\mu| = |l_e|}{S,\rho \vdash (\mu \Rightarrow e)(C) \rightharpoonup_{n,p} \langle l_\mu, l_e\rangle} \tag{24}$$

$$\frac{S_{\text{prec}},\emptyset,\rho \vdash \pi(C^{\mathcal{K}}) \hookrightarrow \langle S_{\text{cons}}, S_{\overline{\text{cons}}}, \rho_\pi\rangle \qquad \rho_\pi \vdash e \rightarrow_{\mathsf{CollType}} C_e}{S_{\text{prec}},\rho \vdash (\pi \Rightarrow e)(C^{\mathcal{K}}) \rightharpoonup \langle S_{\text{cons}}, C_e\rangle} \tag{25}$$

RÈG. 5: Règles de sémantique pour l'application d'une règle de réécriture mini-MGS.

- Règle 22 : Cette règle est la même que la règle 21 à ceci près que la dimension de la cellule filtrée n'est pas précisée.
- Règle 23 : La dernière règle 23 est très proche de la règle 13 (poursuite du filtrage dans les motifs de chemin). Deux sous-complexes sont filtrés par π_1 et π_2 ; on fait alors d'une part l'union des ensembles de cellules filtrées et consommées par π_1 et π_2, et d'autre part l'union des ensembles de cellules filtrées mais non-consommées pour définir la conclusion. Comme dans la règle 13, il faut marquer les cellules mises en jeu durant le filtrage de π_1 avant de filtrer le motif π_2 ; on met donc à jour l'ensemble S_{cur} des cellules filtrées par le filtrage en cours par : $S_{\text{cur}} \cup S_{\text{cons}} \cup S_{\overline{\text{cons}}}$.

3.4 Application d'une règle

L'application d'une règle consiste à exécuter le processus de filtrage du motif donné en partie gauche, puis à évaluer la partie droite de la règle. On retourne alors l'ensemble des cellules filtrées (et obligatoirement consommées en ce qui concerne les patches) par le motif, constituant ainsi la sous-collection et la valeur spécifiée par la partie droite. Le système de règles 5 définit formellement l'application d'une règle pour une transformation de chemins (règle 24) et pour un patch (règle 25). En voici la description :

- Règle 24 : elle concerne l'évaluation de l'application d'une règle de transformation de chemins. Le motif μ filtre le chemin l_μ dans la collection C, en considérant que les cellules de S ont déjà été filtrées par une application de règle précédente. L'étape de filtrage modifie l'environnement ρ en ρ' tenant compte des variables de filtre. L'expression e en partie droite de la règle s'évalue en la séquence l_e. L'application de la règle retourne donc le couple $\langle l_\mu, l_e\rangle$ (collection filtrée, collection calculée). On remarque dans les prémisses la garde $|l_\mu| = |l_e|$: les deux séquences doivent être de même longueur. Cette condition assure que toutes les positions filtrées seront décorées après reconstruction. Aucune modification de la topologie n'est autorisée par les transformations de chemin en mini-MGS. Ce n'est pas le cas dans l'interprète MGS où les collections leibniziennes autorisent de tels changements. On peut par exemple réécrire un sous-ensemble d'un ensemble en un sous-ensemble de cardinal différent, ou encore une sous-séquence par une séquence de longueur différente. En mini-MGS, les modifications topologiques ne peuvent être faites qu'à l'aide des patches.
- Règle 25 : elle effectue le même calcul pour les règles de réécriture de patch. Le motif π filtre et consomme les cellules de l'ensemble S_{cons}, filtre mais ne consomme pas les cellules de l'ensemble $S_{\overline{\text{cons}}}$ et retourne l'environnement ρ_π mis à jour pour les variables

de filtre. Dans les prémisses, le filtrage de π est initialisé : l'ensemble des cellules filtrées par des applications de règles antérieures est l'ensemble S_{prec}, fourni dans la conclusion. L'ensemble des cellules sélectionnées par le filtrage en cours est évidemment vide, le filtrage n'ayant pas encore été appliqué. L'expression e en partie droite de la règle est évalué dans l'environnement ρ_π pour fournir un nouveau complexe cellulaire C_e qui sera substitué en lieu et place de la sous-collection filtrée. Le couple $\langle S_{\text{cons}}, C_e \rangle$ (collection filtrée, collection calculée) est donc retourné.

3.5 Stratégies et reconstruction

Nous terminons cette sémantique par l'application des ensembles de règles que définissent les transformations. Cette application met en jeu :

- la politique d'application des règles durant une descente récursive, et
- la reconstruction de la collection résultat pendant la phase de remontée.

Pour commencer, nous mettons en place les stratégies d'application. La reconstruction, totalement orthogonale aux stratégies, sera décrite dans un second temps.

3.5.1 Les stratégies d'application

Nous définissons quatre ensembles de règles de réduction pour chacune des stratégies suivantes :

1. asynchrone sans priorité sur les règles ;
2. asynchrone avec priorité sur les règles ;
3. synchrone sans priorité sur les règles ;
4. synchrone avec priorité sur les règles.

Avant tout, la définition des règles d'inférence des stratégies étant récursive, nous explicitons dans un premier temps le cas de terminaison où aucune règle ne s'applique.

Terminaison. Le système de règles 8 est commun à toutes les stratégies d'application. Il existe deux cas d'application de ces règles :

1. l'ensemble des règles de réécriture à appliquer est vide, et
2. aucune des règles ne peut être appliquée.

Asynchrone sans priorité. Pour cette stratégie, au plus une des règles est appliquée une seule fois. Si aucune règle de réécriture ne convient, on utilisera les règles d'inférence du système 8. Sinon, les règles d'inférence pour cette stratégie sont données par le système de règles 6.

Ces deux règles suivent le même principe. On applique tout d'abord la i^{e} règle une seule fois pour obtenir d'une part les cellules filtrées (sous forme de chemin ou d'ensemble) et d'autre part la sous-collection à substituer en lieu et place des cellules filtrées (sous forme de séquence de valeurs ou de collection topologique). On applique alors un ensemble vide de règles de réécriture en prenant en compte les cellules filtrées pour préparer la reconstruction à l'aide du système de règles 8 (voir la section 3.5.2).

$$\frac{S,\rho \vdash \gamma_i(C) \rightharpoonup_{n,p} \langle l_\mu, l_e\rangle \qquad S \sqcup l_\mu, \rho \vdash \{\,\}(C) \rightsquigarrow_{n,p} C'}{S,\rho \vdash \{\ldots\ ;\ \gamma_i\ ;\ \ldots\}(C) \rightsquigarrow_{n,p} l_e.l_\mu + C'} \tag{26}$$

$$\frac{S,\rho \vdash \psi_i(C) \rightharpoonup \langle S_\pi, C_e\rangle \qquad S \cup S_\pi, \rho \vdash \{\,\}(C) \rightsquigarrow C'}{S,\rho \vdash \{\ldots\ ;\ \psi_i\ ;\ \ldots\}(C) \rightsquigarrow C_e + C'} \tag{27}$$

Règ. 6: Règles de sémantique pour la stratégie asynchrone sans priorité.

Asynchrone avec priorité. Pour cette stratégie, on applique une et une seule fois la première règle de réécriture qui peut être appliquée. Les règles d'inférence pour cette stratégie sont données par le système de règles 7.

Celles-ci ont un comportement identique à celui des règles du système 6. Néanmoins, une condition supplémentaire apparaît dans les prémisses : les $(i-1)$ premières règles de réécriture ne doivent pas pouvoir s'appliquer sur la collection pour pouvoir appliquer la règle γ_i. La priorité est donc donnée à la première règle pouvant s'appliquer. Encore une fois, si aucune règle ne peut être appliquée, le système de règles 8 est utilisé.

Synchrone sans priorité. Une stratégie synchrone consiste à appliquer en parallèle plusieurs règles de réécriture. Un sous-collection ne peut être filtrée que par une seule règle. Les règles d'inférence pour cette stratégie sont données par les règles du système 9. Les règles 32 et 33 ont la même structure que les règles 26 et 27 de la stratégie asynchrone. La différence se situe au niveau de l'appel récursif à la relation : pour une stratégie asynchrone, une seule règle doit être appliquée ; on termine donc l'application en rappelant la relation sur un ensemble de règles vide. Ici, on rappelle la relation avec le même ensemble de règles de telle sorte qu'une deuxième

$$\frac{\begin{array}{l} S,\rho \vdash \gamma_1(C) \not\rightharpoonup_{n,p} \\ \quad\vdots \qquad\qquad S,\rho \vdash \gamma_i(C) \rightharpoonup_{n,p} \langle l_\mu, l_e\rangle \quad S \sqcup l_\mu, \rho \vdash \{\,\}(C) \rightsquigarrow_{n,p} C' \\ S,\rho \vdash \gamma_{i-1}(C) \not\rightharpoonup_{n,p} \end{array}}{S,\rho \vdash \{\ldots\ ;\ \gamma_i\ ;\ \ldots\}(C) \rightsquigarrow_{n,p} l_e.l_\mu + C'} \tag{28}$$

$$\frac{\begin{array}{l} S,\rho \vdash \psi_1(C) \not\rightharpoonup \\ \quad\vdots \qquad\qquad S,\rho \vdash \psi_i(C) \rightharpoonup \langle S_\pi, C_e\rangle \quad S \cup S_\pi, \rho \vdash \{\,\}(C) \rightsquigarrow C' \\ S,\rho \vdash \psi_{i-1}(C) \not\rightharpoonup \end{array}}{S,\rho \vdash \{\ldots\ ;\ \psi_i\ ;\ \ldots\}(C) \rightsquigarrow C_e + C'} \tag{29}$$

Règ. 7: Règles de sémantique pour la stratégie asynchrone avec priorité.

$$\frac{\forall i \qquad S,\rho \vdash \gamma_i(C) \not\rightarrow_{n,p}}{S,\rho \vdash \{\ldots\ ;\ \gamma_i\ ;\ \ldots\}(C^{\mathcal{K}}) \rightsquigarrow_{n,p} C^{\mathcal{K}} - C^{\mathcal{K}}{\downarrow}(\mathcal{K} - (S^{\mathcal{K}} - S))} \tag{30}$$

$$\frac{\forall i \qquad S,\rho \vdash \gamma_i(C) \not\rightarrow}{S,\rho \vdash \{\ldots\ ;\ \psi_i\ ;\ \ldots\}(C^{\mathcal{K}}) \rightsquigarrow C^{\mathcal{K}}{\downarrow}(\mathcal{K} - S)} \tag{31}$$

RÈG. 8: Règles de sémantique pour les stratégies : aucune règle ne s'applique.

règle, puis une troisième, ... seront appliquées en série. Néanmoins, dans le résultat final de l'application de la transformation, les règles sembleront avoir été appliquées en parallèle.

Synchrone avec priorité. Il s'agit de la stratégie par défaut implantée dans l'interprète MGS. Les règles de réécriture sont appliquées dans l'ordre jusqu'à épuisement. Les règles d'inférence pour cette stratégie sont données par les règles du système 10. Les règles 34 et 36 sont très proches de celles du système 9 de la stratégie synchrone sans priorité. Les deux autres règles d'inférence correspondent à la consommation des règles de réécriture lorsqu'elles ne peuvent plus être appliquées. On teste par conséquent toutes les règles, dans l'ordre, jusqu'à ce que l'ensemble de règles de réécriture à appliquer soit vide. On utilise le système de règles 8 pour terminer l'appel récursif.

3.5.2 Reconstruction

Toutes les stratégies précédentes partagent les mêmes calculs pour construire le résultat d'une application d'un ensemble de règles :

- Les arbres de preuve des stratégies conduisent indubitablement à appliquer les axiomes 30 et 31. Soit S l'ensemble des cellules filtrées d'une collection $C^{\mathcal{K}}$. Les axiomes initialisent la reconstruction en retournant :
  - $C^{\mathcal{K}} - C^{\mathcal{K}}{\downarrow}(\mathcal{K} - (S^{\mathcal{K}} - S))$ pour les transformations de chemins, et
  - $C^{\mathcal{K}}{\downarrow}(\mathcal{K} - S)$ pour les patches.

$$\frac{S,\rho \vdash \gamma_i(C) \rightharpoonup_{n,p} \langle l_\mu, l_e\rangle \qquad S \sqcup l_\mu, \rho \vdash \{\ldots\ ;\ \gamma_i\ ;\ \ldots\}(C) \rightsquigarrow_{n,p} C'}{S,\rho \vdash \{\ldots\ ;\ \gamma_i\ ;\ \ldots\}(C) \rightsquigarrow_{n,p} l_e.l_\mu + C'} \tag{32}$$

$$\frac{S,\rho \vdash \psi_i(C) \rightharpoonup \langle S_\pi, C_e\rangle \qquad S \cup S_\pi, \rho \vdash \{\ldots\ ;\ \psi_i\ ;\ \ldots\}(C) \rightsquigarrow C'}{S,\rho \vdash \{\ldots\ ;\ \psi_i\ ;\ \ldots\}(C) \rightsquigarrow C_e + C'} \tag{33}$$

RÈG. 9: Règles de sémantique pour la stratégie synchrone sans priorité.

$$\frac{S,\rho \vdash \gamma_1(C) \rightharpoonup_{n,p} \langle l_\mu, l_e\rangle \qquad S \sqcup l_\mu, \rho \vdash \{\gamma_1 \ ; \ \ldots\}(C) \rightsquigarrow_{n,p} C'}{S,\rho \vdash \{\gamma_1 \ ; \ \ldots\}(C) \rightsquigarrow_{n,p} l_e.l_\mu + C'} \tag{34}$$

$$\frac{S,\rho \vdash \gamma_1(C) \not\rightharpoonup_{n,p} \qquad S,\rho \vdash \{\gamma_2 \ ; \ \ldots\}(C) \rightsquigarrow_{n,p} C'}{S,\rho \vdash \{\gamma_1 \ ; \ \gamma_2 \ ; \ \ldots\}(C) \rightsquigarrow_{n,p} C'} \tag{35}$$

$$\frac{S,\rho \vdash \psi_1(C) \rightharpoonup \langle S_\pi, C_e\rangle \qquad S \cup S_\pi, \rho \vdash \{\psi_1 \ ; \ \ldots\}(C) \rightsquigarrow C'}{S,\rho \vdash \{\psi_1 \ ; \ \ldots\}(C) \rightsquigarrow C_e + C'} \tag{36}$$

$$\frac{S,\rho \vdash \psi_1(C) \not\rightharpoonup \qquad S,\rho \vdash \{\psi_2 \ ; \ \ldots\}(C) \rightsquigarrow C'}{S,\rho \vdash \{\psi_1 \ ; \ \psi_2 \ ; \ \ldots\}(C) \rightsquigarrow C'} \tag{37}$$

Règ. 10: Règles de sémantique pour la stratégie synchrone avec priorité.

- Les cas récursifs (prenons par exemple les règles du système 6 de la stratégie asynchrone sans priorité) appliquent dans un premier temps une des règles de réécriture récupérant un couple $\langle l_\mu, l_e\rangle$ (resp. $\langle S_\pi, C_e\rangle$) pour les transformations de chemins (resp. pour les patches). L'appel récursif retourne ensuite une nouvelle collection C'. Le résultat du calcul est donné par
  - $l_e.l_\mu + C'$ pour les $\langle n,p\rangle$-transformations, et
  - $C_e + C'$ pour les patches.

Suivant ce raisonnement, la structure de batch annoncée dans le chapitre 2 (page 49) est effectivement présente. Les batches des $\langle n,p\rangle$-transformations et des patches sont présentés séparément.

Reconstruction des chemins. Soit la suite $(\langle l^i_\mu, l^i_e\rangle)$ des couples (collection filtrée, collection calculée) d'une transformation de chemins appliquée sur une collection $C^{\mathcal{K}}$; cette séquence correspond au batch de l'application de la transformation sur $C^{\mathcal{K}}$. Le résultat C' de l'application correspond alors à une réduction du batch. Les cellules topologiques de $\mathcal{K}$ filtrées sont données par :

$$S = \bigsqcup_i l^i_\mu$$

La collection résultat C' correspond alors à la réduction :

$$C' = \sum_i l^i_e.l^i_\mu + C^{\mathcal{K}} - C^{\mathcal{K}}{\downarrow}(\mathcal{K} - (S^{\mathcal{K}} - S))$$

Détaillons ce calcul. Tout d'abord, l'expression $C^{\mathcal{K}} - C^{\mathcal{K}}{\downarrow}(\mathcal{K} - (S^{\mathcal{K}} - S))$ permet d'annuler la valeur associée aux cellules de S dans $\mathcal{K}$. En effet, les transformations de chemins ne modifient pas la topologie ; il suffit d'annuler la valeur associée aux cellules filtrées, puis de sommer avec

une sous-collection adéquatement définie pour placer une nouvelle valeur. Prenons par exemple les objets suivants :

$$\begin{array}{lcl} \mathcal{K} & = & (\{\sigma_1,\sigma_2,\sigma_3\},\{\sigma_1 \prec \sigma_3, \sigma_2 \prec \sigma_3\},\{\sigma_1 \mapsto 0, \sigma_2 \mapsto 0, \sigma_3 \mapsto 1\}) \\ C^{\mathcal{K}} & = & \text{`v}.\sigma_1 + \text{`v}.\sigma_2 + \text{`e}.\sigma_3 \\ S & = & \{\sigma_3\} \\ \langle l_\mu, l_e\rangle & = & \langle[\sigma_3],[\text{`edge}]\rangle \end{array}$$

Il s'agit d'une collection topologique composée d'un arc et ses sommets. L'arc est filtré pour remplacer sa valeur actuelle `‘e` par `‘edge`.

Procédons au calcul. On a

$$\begin{array}{lcl} S^{\mathcal{K}} - S & = & \{\sigma_1,\sigma_2,\sigma_3\} - \{\sigma_3\} \\ & = & \{\sigma_1,\sigma_2\} \end{array}$$

puis

$$\mathcal{K} - (S^{\mathcal{K}} - S) \quad = \quad (\{\sigma_3\},\emptyset,\{\sigma_3 \mapsto 1\})$$

et

$$C^{\mathcal{K}}{\downarrow}(\mathcal{K} - (S^{\mathcal{K}} - S)) \quad = \quad \text{`e}.\sigma_3$$

On obtient finalement le résultat escompté

$$\begin{array}{lcl} C^{\mathcal{K}} - C^{\mathcal{K}}{\downarrow}(\mathcal{K} - (S^{\mathcal{K}} - S)) & = & \text{`v}.\sigma_1 + \text{`v}.\sigma_2 + (\text{`e} -_{\text{Abel(Val)}} \text{`e}).\sigma_3 \\ & = & \text{`v}.\sigma_1 + \text{`v}.\sigma_2 + 0_{\text{Abel(Val)}}.\sigma_3 \end{array}$$

où la valeur associée à σ_3 est annulée. La nouvelle valeur `‘edge` de σ_3 est alors obtenue par le calcul suivant :

$$\begin{array}{lcl} C'^{\mathcal{K}} & = & l_e.l_\mu + C^{\mathcal{K}} - C^{\mathcal{K}}{\downarrow}(\mathcal{K} - (S^{\mathcal{K}} - S)) \\ & = & [\text{`edge}].[\sigma_3] + \text{`v}.\sigma_1 + \text{`v}.\sigma_2 + 0_{\text{Abel(Val)}}.\sigma_3 \\ & = & \text{`edge}.\sigma_3 + \text{`v}.\sigma_1 + \text{`v}.\sigma_2 + 0_{\text{Abel(Val)}}.\sigma_3 \\ & = & \text{`v}.\sigma_1 + \text{`v}.\sigma_2 + (0_{\text{Abel(Val)}} +_{\text{Abel(Val)}} \text{`edge}).\sigma_3 \\ & = & \text{`v}.\sigma_1 + \text{`v}.\sigma_2 + \text{`edge}.\sigma_3 \end{array}$$

La valeur associée à σ_3 a bien été substituée par `‘edge`.

Reconstruction des patches. Bien que les calculs soient de nature différente, nous procédons de la même façon que pour la reconstruction de chemins. Soit la suite $(\langle S^i_\pi, C^i_e\rangle)$ des couples (collection filtrée, collection calculée) d'un patch appliqué sur une collection $C^{\mathcal{K}}$; cette séquence correspond au batch de l'application de la transformation sur $C^{\mathcal{K}}$. Le résultat C' de l'application correspond alors à une réduction du batch. Les cellules topologiques de $\mathcal{K}$ filtrées sont données par :

$$S = \bigcup_i S^i_\pi$$

La collection résultat C' correspond alors à la réduction :

$$C' = \sum_i C^i_e + C^{\mathcal{K}}{\downarrow}(\mathcal{K} - S)$$

Détaillons ce calcul. Pour les patches, l'initialisation de la reconstruction est plus simple : il suffit de restreindre la collection C aux cellules qui ne sont pas dans S, ce que calcule $C^{\mathcal{K}}{\downarrow}\mathcal{K} - S$. Ici, les cellules filtrées sont supprimées, il ne s'agit pas seulement d'annuler la valeur qui les décore. En effet, les patches modifient la topologie. Reprenons l'exemple précédent :

$$\begin{array}{lcl} \mathcal{K} & = & (\{\sigma_1,\sigma_2,\sigma_3\},\{\sigma_1 \prec \sigma_3, \sigma_2 \prec \sigma_3\},\{\sigma_1 \mapsto 0, \sigma_2 \mapsto 0, \sigma_3 \mapsto 1\}) \\ C^{\mathcal{K}} & = & \text{`v}.\sigma_1 + \text{`v}.\sigma_2 + \text{`e}.\sigma_3 \\ S & = & \{\sigma_3\} \\ \langle S_\pi, C_e\rangle & = & \langle\{\sigma_3\}, \text{`v}.\sigma_4^{\mathcal{K}_e}\rangle \\ \mathcal{K}_e & = & (\{\sigma_4\},\emptyset,\{\sigma_4 \mapsto 0\}) \end{array}$$

Cette fois-ci, l'arc σ_3 est filtré, supprimé et remplacé par un sommet isolé σ_4 décoré par le symbole `v. Le calcul auquel nous allons procéder construit donc une collection topologique $C'^{\mathcal{K}'}$ sur un nouveau complexe cellulaire abstrait $\mathcal{K}'$ contenant les trois sommets isolés σ_1, σ_2 et σ_4.

Procédons au calcul en commençant par l'initialisation de la reconstruction :

$$\begin{array}{lcl} \mathcal{K}_0 = \mathcal{K} - S & = & (\{\sigma_1,\sigma_2,\sigma_3\},\{\sigma_1 \prec \sigma_3, \sigma_2 \prec \sigma_3\},\{\sigma_1 \mapsto 0, \sigma_2 \mapsto 0, \sigma_3 \mapsto 1\}) - \{\sigma_3\} \\ & = & (\{\sigma_1,\sigma_2\},\emptyset,\{\sigma_1 \mapsto 0, \sigma_2 \mapsto 0\}) \end{array}$$

La restriction de $C^{\mathcal{K}}$ à $\mathcal{K} - S$ produit alors la collection topologique

$$C^{\mathcal{K}}{\downarrow}\mathcal{K}_0 = \text{`v}.\sigma_1^{\mathcal{K}_0} + \text{`v}.\sigma_2^{\mathcal{K}_0}$$

Cette fois-ci, σ_3 a totalement disparu. L'insertion du nouveau sommet est alors obtenu par le calcul suivant :

$$\begin{array}{lcl} C'^{\mathcal{K}'} & = & C_e + C^{\mathcal{K}}{\downarrow}\mathcal{K}_0 \\ & = & \text{`v}.\sigma_4^{\mathcal{K}_e} + \text{`v}.\sigma_1^{\mathcal{K}_0} + \text{`v}.\sigma_2^{\mathcal{K}_0} \\ & = & \text{`v}.\sigma_4^{\mathcal{K}'} + \text{`v}.\sigma_1^{\mathcal{K}'} + \text{`v}.\sigma_2^{\mathcal{K}'} \end{array}$$

où

$$\begin{array}{lcl} \mathcal{K}' & = & \mathcal{K}_e \cup \mathcal{K}_0 \\ & = & (\{\sigma_4\},\emptyset,\{\sigma_4 \mapsto 0\}) \cup (\{\sigma_1,\sigma_2\},\emptyset,\{\sigma_1 \mapsto 0, \sigma_2 \mapsto 0\}) \\ & = & (\{\sigma_1,\sigma_2,\sigma_4\},\emptyset,\{\sigma_1 \mapsto 0, \sigma_2 \mapsto 0, \sigma_4 \mapsto 0\}) \end{array}$$

L'arc σ_3 est bien supprimé et remplacé par un nouveau sommet σ_4 dans un nouveau complexe cellulaire $\mathcal{K}'$.

4 Exemples

Dans cette dernière section, trois exemples d'utilisation de notre sémantique pour le langage mini-MGS sont développés :

1. la subdivision d'arc,
2. l'utilisation des stratégies d'application de règles, et
3. la preuve qu'avec les stratégies synchrones, il ne reste aucune sous-collection non-filtrée pouvant être filtrée par l'une des règles.

pour illustrer la sémantique du langage mini-MGS définie dans la section précédente.

4.1 Subdivision d'arc

Ce premier exemple concerne l'évaluation d'un patch raffinant des arcs par l'insertion d'un sommet, appliqué sur un seul arc. Il s'agit de l'exemple décrit dans la présentation de l'interprète MGS.

Voici le patch insérant un sommet sur un arc écrit en mini-MGS :

$$\texttt{patch}\{\pi \Rightarrow e\}$$

composé d'une seule règle $\pi \Rightarrow e$ telle que

$$\begin{array}{rcl}
\pi & = & \texttt{~v1:[dim = 0, [] in faces, [] in cofaces, true]} \\
 & & \texttt{~v2:[dim = 0, [] in faces, [] in cofaces, true]} \\
 & & \texttt{e:[dim = 1, [\^{}v1 ;\^{}v2] in faces, [] in cofaces, true]} \\
e & = & \texttt{let v = PosGen(0)([])([]) in} \\
 & & \texttt{let e1 = PosGen(1)([\^{}v1,v])([]) in} \\
 & & \texttt{let e2 = PosGen(1)([v,\^{}v2])([]) in} \\
 & & \quad \texttt{\{| `nv @ v |\}, \{| `ne1 @ e1 |\}, \{| `ne2 @ e2 |\}}
\end{array}$$

À partir de l'arc filtré par `e`, on crée un nouveau sommet `v` incident à deux arcs `e1` et `e2` remplaçant `e`. Celui-ci étant incident à `v1` et à `v2`, les deux sommets filtrés mais non-consommés, `e1` (resp. `e2`) est incident à la cellule filtrée par `v1` (resp. `v2`).

Soit le complexe $\mathcal{K}$ composé d'un arc et de deux sommets :

$$\mathcal{K} = (\{\sigma_1, \sigma_2, \sigma_3\}, \{\sigma_1 \prec \sigma_3, \sigma_2 \prec \sigma_3\}, \{\sigma_1 \mapsto 0, \sigma_2 \mapsto 0, \sigma_3 \mapsto 1\})$$

On définit sur ce complexe la collection topologique :

$$C^{\mathcal{K}} = \texttt{`ov1}.\sigma_1 + \texttt{`ov2}.\sigma_2 + \texttt{`oe}.\sigma_3$$

Les valeurs associées illustreront l'apparition de nouveaux objets. On utilise le symbole `` `ox `` pour décorer un objet `x` présent avant l'application du patch, alors que le symbole `` `ny `` décore un objet `y` créé par le patch.

Soit ρ_0, l'environnement dans lequel aucune variable de $\overline{\mathcal{X}}$ n'a de valeur associée. On définit également l'environnement initial

$$\rho_{\text{init}} = [\texttt{C} \mapsto C^{\mathcal{K}}, \texttt{P} \mapsto \langle \texttt{patch}\{\pi \Rightarrow e\}, \rho_0 \rangle]$$

dans lequel on souhaite évaluer l'expression `P(C)`.

4.1.1 Application du patch

Soit $C'^{\mathcal{K}'}$, la collection topologique créée par l'application du patch. On peut commencer à construire l'arbre d'inférence de l'expression `P(C)` en utilisant la règle 10 :

$$\dfrac{\begin{array}{l}\rho_{\text{init}} \vdash \texttt{P} \rightarrow \langle \texttt{patch}\{\pi \Rightarrow e\}, \rho_0 \rangle \\ \rho_{\text{init}} \vdash \texttt{C} \rightarrow C^{\mathcal{K}}\end{array} \qquad \dfrac{A}{\emptyset, \rho_1 \vdash \{\pi \Rightarrow e\}(C^{\mathcal{K}}) \rightsquigarrow C'^{\mathcal{K}'}}\ 27}{\rho_{\text{init}} \vdash \texttt{P(C)} \rightarrow C'^{\mathcal{K}'}}\ 10$$

où $\rho_1 = \rho_0 \uplus [\texttt{self} \mapsto C^{\mathcal{K}}]$. Nous cherchons à développer le sous-arbre A. L'application des règles de réécriture étant initialisée, on choisit maintenant une stratégie ; nous optons pour la plus simple, la stratégie asynchrone sans priorité. En effet, le patch n'est composé que d'une règle et

il n'existe qu'une instance de motif. En premier lieu, la règle 27 est donc appliquée ; le filtrage et l'évaluation de la partie droite sont initiés à l'aide de la règle 25 ; l'axiome 31 est utilisé pour l'initialisation de la reconstruction. Le sous-arbre A est alors complété comme suit :

$$\dfrac{\dfrac{\dfrac{B}{\emptyset,\emptyset,\rho_1 \vdash \pi(C^{\mathcal{K}}) \hookrightarrow \langle S_\pi, T_\pi, \rho_4\rangle}\,{\scriptstyle 23} \qquad \dfrac{D}{\rho_4 \vdash e \to C_e^{\mathcal{K}_e}}\,{\scriptstyle 3}}{\emptyset,\rho_1 \vdash (\pi \Rightarrow e)(C^{\mathcal{K}}) \rightharpoonup \langle S_\pi, C_e^{\mathcal{K}_e}\rangle}\,{\scriptstyle 25} \qquad \dfrac{}{S_\pi,\rho_1 \vdash \{\}(C^{\mathcal{K}}) \rightsquigarrow C_0^{\mathcal{K}_0}}\,{\scriptstyle 31}}{\emptyset,\rho_1 \vdash \{\pi \Rightarrow e\}(C^{\mathcal{K}}) \rightsquigarrow C'^{\mathcal{K}'}}\,{\scriptstyle 27}$$

Dans les sections suivantes, nous voyons :

- le filtrage du motif π pour compléter le sous-arbre B,
- l'évaluation de la partie droite e pour construire le sous-arbre D[5], et
- la reconstruction avec le calcul de $\mathcal{K}_0$, $C_0^{\mathcal{K}_0}$ et finalement K' et $C'^{\mathcal{K}'}$.

4.1.2 Le filtrage de π, sous-arbre B

Le motif π est composé de trois sous-motifs élémentaires que nous nommerons respectivement π_1, π_2 et π_3. On utilise les abréviations suivantes :

$$\begin{aligned}
\pi_1 &= \texttt{\~{}v1:[dim=0]} \\
&= \texttt{\~{}v1:[dim = 0, [] in faces, [] in cofaces, true]} \\
\pi_2 &= \texttt{\~{}v2:[dim=0]} \\
&= \texttt{\~{}v2:[dim = 0, [] in faces, [] in cofaces, true]} \\
\pi_3 &= \texttt{e:[dim=1,[\^{}v1,\^{}v2] in faces]} \\
&= \texttt{e:[dim = 0, [\^{}v1,\^{}v2] in faces, [] in cofaces, true]}
\end{aligned}$$

Ainsi, le sous-arbre gauche B est complété en utilisant les règles de filtrage des patches 21 et 23 :

$$\dfrac{\dfrac{\rho_1 \vdash \texttt{0} \to 0}{\emptyset,\emptyset,\rho_1 \vdash \texttt{\~{}v1:[dim=0]}(C^{\mathcal{K}}) \hookrightarrow \langle \emptyset, \{\sigma_1\}, \rho_2\rangle}\,{\scriptstyle 21} \qquad \dfrac{B_1}{\emptyset,\{\sigma_1\},\rho_2 \vdash \pi_2\ \pi_3(C^{\mathcal{K}}) \hookrightarrow \langle S_\pi, T'_\pi, \rho_4\rangle}\,{\scriptstyle 23}}{\emptyset,\emptyset,\rho_1 \vdash \pi_1\ \pi_2\ \pi_3(C^{\mathcal{K}}) \hookrightarrow \langle S_\pi, \{\sigma_1\} \cup T'_\pi, \rho_4\rangle}\,{\scriptstyle 23}$$

On filtre ici le premier motif π_1. Deux cellules de $\mathcal{K}$ sont candidates : σ_1 et σ_2 qui sont toutes les deux de dimension 0. Aucune contrainte particulière n'est à vérifier ; la règle 21 peut alors être appliquée. On choisit arbitrairement la cellule σ_1 pour des questions de notation ; il aurait été tout à fait possible de choisir σ_2. Le retour $\langle \emptyset, \{\sigma_1\}, \rho_2\rangle$ indique bien que σ_1 a été filtrée mais n'est pas consommée. L'environnement ρ_1 est mis à jour pour tenir compte du nommage de la cellule filtrée par `v1` :

$$\begin{aligned}
\rho_2 &= \rho_1 \uplus [\texttt{v1} \mapsto \texttt{`ov1}, \texttt{\^{}v1} \mapsto \sigma_1^{\mathcal{K}_1}] \\
\mathcal{K}_1 &= (\{\sigma_1\}, \emptyset, \{\sigma_1 \mapsto 0\})
\end{aligned}$$

La suite du filtrage se poursuit de la même façon dans l'arbre B_1 pour filtrer sans consommer la cellule σ_2 par le motif π_2 :

$$\dfrac{\dfrac{\rho_2 \vdash \texttt{0} \to 0}{\emptyset,\{\sigma_1\},\rho_2 \vdash \texttt{\~{}v2:[dim=0]}(C^{\mathcal{K}}) \hookrightarrow \langle \emptyset, \{\sigma_2\}, \rho_3\rangle}\,{\scriptstyle 21} \qquad \dfrac{B_2}{\emptyset,\{\sigma_1,\sigma_2\},\rho_3 \vdash \pi_3(C^{\mathcal{K}}) \hookrightarrow \langle S_\pi, T''_\pi, \rho_4\rangle}\,{\scriptstyle 19}}{\emptyset,\{\sigma_1\},\rho_2 \vdash \pi_2\ \pi_3(C^{\mathcal{K}}) \hookrightarrow \langle S_\pi, \{\sigma_2\} \cup T''_\pi, \rho_4\rangle}\,{\scriptstyle 23}$$

[5] La variable C serait source de confusion avec les variables dénotant les collections topologiques

On définit alors l'environnement ρ_3 :

$$\begin{aligned}\rho_3 &= \rho_2 \uplus [\texttt{v2} \mapsto \texttt{`ov2}, \texttt{\^{}v2} \mapsto \sigma_2^{\mathcal{K}_2}]\\ \mathcal{K}_2 &= (\{\sigma_2\}, \emptyset, \{\sigma_2 \mapsto 0\})\end{aligned}$$

On termine le processus de filtrage dans B_3 en sélectionnant et en consommant σ_3, l'arc du complexe $\mathcal{K}$, à l'aide de la règle 19 :

$$\dfrac{\dfrac{\vdots}{\rho_3 \vdash \texttt{[\^{}v1,\^{}v2]} \rightarrow [\sigma_1^{\mathcal{K}_1}; \sigma_2^{\mathcal{K}_2}]}\ 4 \qquad \rho_3 \vdash \texttt{1} \rightarrow 1 \qquad \sigma_1, \sigma_2 \in \texttt{faces}_{\mathcal{K}}\ \sigma_3 = \{\sigma_1, \sigma_2\}}{\emptyset, \{\sigma_1, \sigma_2\}, \rho_3 \vdash \texttt{e:[dim=1,[\^{}v1,\^{}v2] in faces]}(C^{\mathcal{K}}) \hookrightarrow \langle \{\sigma_3\}, \emptyset, \rho_4 \rangle}\ 19$$

Cette fois-ci, la cellule filtrée est consommée : on retourne le triplet $\langle \{\sigma_3\}, \emptyset, \rho_4 \rangle$. Une fois de plus, on définit l'environnement de retour :

$$\begin{aligned}\rho_4 &= \rho_3 \uplus [\texttt{e} \mapsto \texttt{`oe}, \texttt{\^{}e} \mapsto \sigma_3^{\mathcal{K}_3}]\\ \mathcal{K}_3 &= (\{\sigma_3\}, \emptyset, \{\sigma_3 \mapsto 1\})\end{aligned}$$

Nous ne détaillons pas plus l'arbre évaluant l'expression `[^v1,^v2]` en $[\sigma_1^{\mathcal{K}_1}; \sigma_2^{\mathcal{K}_2}]$. Les variables S_π et T_π apparaissant maintenant dans l'arbre A où le sous-arbre B prend racine sont alors définies :

$$\begin{aligned}S_\pi &= \{\sigma_3\}\\ T_\pi &= \{\sigma_1\} \cup T'_\pi\\ &= \{\sigma_1\} \cup \{\sigma_2\} \cup T''_\pi\\ &= \{\sigma_1, \sigma_2\} \cup \emptyset\\ &= \{\sigma_1, \sigma_2\}\end{aligned}$$

4.1.3 L'évaluation de e, sous-arbre D

Comme pour le filtrage, nous utilisons un jeu d'abréviations pour alléger l'arbre de preuve :

$$\begin{aligned}e &= \texttt{let v = PosGen(0)([])([]) in } e_1\\ e_1 &= \texttt{let e1 = PosGen(1)([\^{}v1,v])([]) in } e_2\\ e_2 &= \texttt{let e2 = PosGen(1)([v,\^{}v2])([]) in } e_3\\ e_3 &= e_4\texttt{, } e_5\texttt{, } e_6\\ e_4 &= \{\texttt{| `nv @ v |}\}\\ e_5 &= \{\texttt{| `ne1 @ e1 |}\}\\ e_6 &= \{\texttt{| `ne2 @ e2 |}\}\end{aligned}$$

L'évaluation de e correspond au sous-arbre D de l'arbre A. Elle se fait dans le contexte de l'environnement ρ_4 issu du filtrage et retourne une collection topologique $\mathcal{C}_e^{\mathcal{K}_e}$. Trois premières définitions de variables permettent la construction de trois nouvelles cellules. Ces trois cellules sont décorées dans l'expression e_3 pour construire la collection topologique $\mathcal{C}_e^{\mathcal{K}_e}$.

Voyons dans un premier temps la construction des nouvelles cellules topologiques. La première cellule (un sommet) est créée :

$$\dfrac{\dfrac{\rho_4 \vdash \texttt{0} \rightarrow 0 \quad \rho_4 \vdash \texttt{[]} \rightarrow []}{\rho_4 \vdash \texttt{PosGen(0)([])([])} \rightarrow \sigma_4^{\mathcal{K}_4}}\ 4 \qquad \dfrac{D_1}{\rho_5 \vdash e_1 \rightarrow \mathcal{C}_e^{\mathcal{K}_e}}\ 3}{\rho_4 \vdash \texttt{let v = PosGen(0)([])([]) in } e_1 \rightarrow \mathcal{C}_e^{\mathcal{K}_e}}\ 3$$

avec l'environnement ρ_5 défini de la façon suivante :

$$\begin{aligned}\rho_5 &= \rho_4 \uplus [\texttt{v} \mapsto \sigma_4^{\mathcal{K}_4}]\\ \mathcal{K}_4 &= (\{\sigma_4\}, \emptyset, \{\sigma_4 \mapsto 0\})\end{aligned}$$

Nous utilisons ici la règle 3 d'évaluation de l'application d'une λ-expression ; rappelons en effet que le sucre syntaxique que nous utilisons est défini par :

$$\texttt{let x = } e_1 \texttt{ in } e_2 \equiv \texttt{(}\lambda\texttt{x}\cdot e_2\texttt{)(}e_1\texttt{)}$$

La cellule $\sigma_4^{\mathcal{K}_4}$ est le résultat de l'application de la fonction prédéfinie $fct_{\text{posGen}}(0)([])([])$. Nous ne détaillerons pas l'application des fonctions prédéfinies qui ont déjà été décrites plus haut dans le chapitre.

Deux nouveaux arcs sont alors générés ; voici la construction de l'arbre D_1 :

$$\dfrac{\dfrac{\rho_5 \vdash \texttt{1} \rightarrow 1 \quad \rho_5 \vdash \texttt{[]} \rightarrow [] \quad \rho_5 \vdash \texttt{[\^{}v1,v]} \rightarrow [\sigma_1^{\mathcal{K}_1}; \sigma_4^{\mathcal{K}_4}]}{\rho_5 \vdash \texttt{PosGen(1)([\^{}v1,v])([])} \rightarrow \sigma_5^{\mathcal{K}_5}}\,4 \qquad \dfrac{D_2}{\rho_6 \vdash e_2 \rightarrow \mathcal{C}_e^{\mathcal{K}_e}}\,3}{\rho_5 \vdash \texttt{let e1 = PosGen(1)([\^{}v1,v])([]) in } e_2 \rightarrow \mathcal{C}_e^{\mathcal{K}_e}}\,3$$

Cet arc est incident aux sommets $\sigma_1^{\mathcal{K}_1}$ et $\sigma_4^{\mathcal{K}_4}$. Le premier est issu du filtrage et le second correspond à la variable `v`. On construit σ_5 en utilisant la fonction prédéfinie posGen :

$$\begin{aligned}\text{predGen}(1, [\sigma_1^{\mathcal{K}_1}, \sigma_4^{\mathcal{K}_4}], []) &= dim^{\mathcal{K}_1}(\sigma_1) < 1 \wedge dim^{\mathcal{K}_4}(\sigma_4) < 1\\ &= 0 < 1 \wedge 0 < 1\\ &= \text{true}\\ \text{posGen}(1, [\sigma_1^{\mathcal{K}_1}, \sigma_4^{\mathcal{K}_4}], []) &= \sigma_5^{\mathcal{K}_5}\\ \mathcal{K}_5 &= (\{\sigma_5, \sigma_1, \sigma_4\}, \{\sigma_1 \prec \sigma_5, \sigma_4 \prec \sigma_5\}, \{\sigma_5 \mapsto 1, \sigma_1 \mapsto 0, \sigma_4 \mapsto 0\})\\ &\quad \cup \mathcal{K}_1 \cup \mathcal{K}_4\\ &= (\{\sigma_5, \sigma_1, \sigma_4\}, \{\sigma_1 \prec \sigma_5, \sigma_4 \prec \sigma_5\}, \{\sigma_5 \mapsto 1, \sigma_1 \mapsto 0, \sigma_4 \mapsto 0\})\end{aligned}$$

L'environnement ρ_6 est défini par :

$$\rho_6 = \rho_5 \uplus [\texttt{e1} \mapsto \sigma_5^{\mathcal{K}_5}]$$

On construit l'arc associé à `e2` par l'arbre D_2 :

$$\dfrac{\dfrac{\rho_6 \vdash \texttt{1} \rightarrow 1 \quad \rho_6 \vdash \texttt{[]} \rightarrow [] \quad \rho_6 \vdash \texttt{[v,\^{}v2]} \rightarrow [\sigma_4^{\mathcal{K}_4}; \sigma_2^{\mathcal{K}_2}]}{\rho_6 \vdash \texttt{PosGen(1)([v,\^{}v2])([])} \rightarrow \sigma_6^{\mathcal{K}_6}}\,4 \qquad \dfrac{D_3}{\rho_7 \vdash e_3 \rightarrow \mathcal{C}_e^{\mathcal{K}_e}}\,4}{\rho_6 \vdash \texttt{let e2 = PosGen(1)([v,\^{}v2])([]) in } e_3 \rightarrow \mathcal{C}_e^{\mathcal{K}_e}}\,3$$

où l'environnement ρ_7 est défini par :

$$\rho_7 = \rho_6 \uplus [\texttt{e2} \mapsto \sigma_6^{\mathcal{K}_6}]$$

La construction de $\sigma_6^{\mathcal{K}_6}$ est équivalente à celle de $\sigma_5^{\mathcal{K}_5}$. On définit directement $\mathcal{K}_6$ par :

$$\mathcal{K}_6 = (\{\sigma_6, \sigma_2, \sigma_4\}, \{\sigma_2 \prec \sigma_6, \sigma_4 \prec \sigma_6\}, \{\sigma_6 \mapsto 1, \sigma_2 \mapsto 0, \sigma_4 \mapsto 0\})$$

Après la construction des cellules définissant le complexe cellulaire pour le support de la partie droite, on construit la collection topologique $C_e^{\mathcal{K}_e}$ en développant D_3 jusqu'aux feuilles de l'arbre de preuve :

$$\dfrac{\rho_7 \vdash \{\!|\ \text{`nv} \ @\ \text{v}\ |\!\} \rightarrow \text{`nv}.\sigma_4^{\mathcal{K}_4} \qquad \dfrac{\begin{array}{r}\rho_7 \vdash \{\!|\ \text{`ne1} \ @\ \text{e1}\ |\!\} \rightarrow \text{`ne1}.\sigma_5^{\mathcal{K}_5} \\ \rho_7 \vdash \{\!|\ \text{`ne2} \ @\ \text{e2}\ |\!\} \rightarrow \text{`ne2}.\sigma_6^{\mathcal{K}_6}\end{array}}{\rho_7 \vdash e_5, e_6 \rightarrow \text{`ne1}.\sigma_5^{\mathcal{K}_5} + \text{`ne2}.\sigma_6^{\mathcal{K}_6}}\ 4}{\rho_7 \vdash e_4, e_5, e_6 \rightarrow \text{`nv}.\sigma_4^{\mathcal{K}_4} + \text{`ne1}.\sigma_5^{\mathcal{K}_5} + \text{`ne2}.\sigma_6^{\mathcal{K}_6}}\ 4$$

Nous terminons ainsi la construction du sous-arbre D de l'arbre A. On définit en particulier la collection topologique $C_e^{\mathcal{K}_e}$ de la façon suivante :

$$\begin{array}{lcl} C_e^{\mathcal{K}_e} & = & \text{`nv}.\sigma_4^{\mathcal{K}_4} + \text{`ne1}.\sigma_5^{\mathcal{K}_5} + \text{`ne2}.\sigma_6^{\mathcal{K}_6} \\ & = & \text{`nv}.\sigma_4^{\mathcal{K}_e} + \text{`ne1}.\sigma_5^{\mathcal{K}_e} + \text{`ne2}.\sigma_6^{\mathcal{K}_e} \\ \mathcal{K}_e & = & \mathcal{K}_4 \cup \mathcal{K}_5 \cup \mathcal{K}_6 \\ & = & (\{\sigma_1, \sigma_2, \sigma_4, \sigma_5, \sigma_6\}, \\ & & \{\sigma_1 \prec \sigma_5, \sigma_2 \prec \sigma_6, \sigma_4 \prec \sigma_5, \sigma_4 \prec \sigma_6\}, \\ & & \{\sigma_1 \mapsto 0, \sigma_2 \mapsto 0, \sigma_4 \mapsto 0, \sigma_5 \mapsto 1, \sigma_6 \mapsto 1\}) \end{array}$$

4.1.4 La reconstruction

À partir des résultats précédents et des règles 31 et 27 les calculs se font directement :

$$\begin{array}{lcl} C_0^{\mathcal{K}_0} & = & C^{\mathcal{K}_0} \\ & = & \text{`ov1}.\sigma_1^{\mathcal{K}_0} + \text{`ov2}.\sigma_2^{\mathcal{K}_0} \\ \mathcal{K}_0 & = & \mathcal{K} - S_\pi \\ & = & (\{\sigma_1, \sigma_2\}, \emptyset, \{\sigma_1 \mapsto 0, \sigma_2 \mapsto 0\}) \\ C'^{\mathcal{K}'} & = & C_e^{\mathcal{K}_e} + C_0^{\mathcal{K}_0} \\ & = & \text{`nv}.\sigma_4^{\mathcal{K}_e} + \text{`ne1}.\sigma_5^{\mathcal{K}_e} + \text{`ne2}.\sigma_6^{\mathcal{K}_e} + \text{`ov1}.\sigma_1^{\mathcal{K}_0} + \text{`ov2}.\sigma_2^{\mathcal{K}_0} \\ & = & \text{`nv}.\sigma_4^{\mathcal{K}'} + \text{`ne1}.\sigma_5^{\mathcal{K}'} + \text{`ne2}.\sigma_6^{\mathcal{K}'} + \text{`ov1}.\sigma_1^{\mathcal{K}'} + \text{`ov2}.\sigma_2^{\mathcal{K}'} \\ \mathcal{K}' & = & \mathcal{K}_e \cup \mathcal{K}_0 \\ & = & (\{\sigma_1, \sigma_2, \sigma_4, \sigma_5, \sigma_6\}, \\ & & \{\sigma_1 \prec \sigma_5, \sigma_2 \prec \sigma_6, \sigma_4 \prec \sigma_5, \sigma_4 \prec \sigma_6\}, \\ & & \{\sigma_1 \mapsto 0, \sigma_2 \mapsto 0, \sigma_4 \mapsto 0, \sigma_5 \mapsto 1, \sigma_6 \mapsto 1\}) \end{array}$$

Le nouveau complexe $\mathcal{K}'$ est composé de cinq cellules topologiques, dont trois sommets et deux arcs. Les deux anciens sommets σ^1 et σ^2 sont toujours décorés par les valeurs qui leur étaient associées dans $C^{\mathcal{K}}$. L'arc σ_3 est supprimé topologiquement et est remplacé par les deux arcs σ_5 et σ_6 incidents respectivement à σ_1 et σ_4, et à σ_2 et σ_4.

4.2 Utilisation des stratégies

Pour illustrer les différences de comportement des quatre stratégies d'application des règles, nous proposons de calculer les différents résultats de l'application d'une même transformation de chemin suivant chacune des stratégies. Cette transformation est spécifiée comme suit en mini-MGS :

```
trans<n,p>{
  x => [10*x] ;
  x => [-1*x]
}
```

Elle est composée de deux règles de réécriture que nous dénoterons respectivement par les variables γ_1 et γ_2. La règle γ_1 filtre un élément pour modifier sa décoration en dix fois sa valeur, et la règle γ_2 transforme la décoration des éléments qu'elle filtre en leur opposé.

Nous proposons d'appliquer cette transformation sur une collection très simple constituée de deux sommets isolés σ_1 et σ_2. On décore le sommet σ_1 par l'entier 1, et σ_2 par l'entier 2. Les cellules étant de dimension 0, nous utilisons le $\langle 0,1\rangle$-voisinage (dans cet exemple, la valeur de la dimension p n'a pas d'importance, les chemins filtrés étant élémentaires). Nous construisons donc le complexe cellulaire $\mathcal{K}$ et la collection topologique $C^{\mathcal{K}}$ correspondante :

$$\begin{array}{lcl} \mathcal{K} & = & (\{\sigma_1,\sigma_2\},\emptyset,\{\sigma_1\mapsto 0,\sigma_2\mapsto 0\}) \\ C^{\mathcal{K}} & = & 1.\sigma_1 + 2.\sigma_2 \end{array}$$

Comme dans l'exemple précédent, nous supposons l'existence d'un environnement initial ρ_{init} défini comme suit :

$$\rho_{\text{init}} \quad = \quad [\texttt{T}\mapsto\langle \texttt{trans<n,p>}\{\gamma_1\,;\gamma_2\},\emptyset\rangle, \texttt{C}\mapsto C^{\mathcal{K}}]$$

L'objet de cet exemple est l'évaluation dans l'environnement ρ_{init} de l'expression

$$\texttt{T<0,1>(C)}$$

Nous commençons par construire la base de l'arbre de réduction qui nous intéresse en utilisant la règle de réduction 9 :

$$\frac{\begin{array}{l}\rho_{\text{init}} \vdash \texttt{T} \rightarrow \langle \texttt{trans<n,p>}\{\gamma_1\,;\gamma_2\},\emptyset\rangle \\ \rho_{\text{init}} \vdash \texttt{0} \rightarrow 0 \\ \rho_{\text{init}} \vdash \texttt{1} \rightarrow 1 \\ \rho_{\text{init}} \vdash \texttt{C} \rightarrow C^{\mathcal{K}}\end{array} \qquad \frac{A}{\emptyset,\rho \vdash \{\gamma_1\,;\gamma_2\}(C^{\mathcal{K}}) \rightsquigarrow_{0,1} C'}}{\rho_{\text{init}} \vdash \texttt{T<0,1>(C)} \rightarrow C'}\ 9$$

où l'environnement ρ vaut :

$$\rho \quad = \quad [\texttt{self}\mapsto C^{\mathcal{K}}, \texttt{n}\mapsto 0, \texttt{p}\mapsto 1]$$

4.2.1 Application de chaque règle seule

L'objectif de cet exemple n'est pas de construire les arbres de preuve concernant le filtrage des règles γ_1 et γ_2. Nous allons simplement énumérer les éléments des relations :

$$S,\rho \vdash \gamma_i(C^{\mathcal{K}}) \rightharpoonup_{0,1} \langle l_\mu, l_e\rangle$$

L'ensemble des cellules déjà filtrées S peut prendre quatre valeurs différentes :

1. $S = \emptyset$: aucun processus de filtrage n'a encore été exécuté ;
2. $S = \{\sigma_1\}$: σ_1 a déjà été filtrée ;

3. $S = \{\sigma_2\}$: σ_2 a déjà été filtrée ;
4. $S = \{\sigma_1, \sigma_2\}$: σ_1 et σ_2 ont déjà été filtrées.

Voici les éléments des relations d'application de règle pour γ_1 et γ_2 :

$$\begin{array}{rl} \emptyset, \rho \vdash \gamma_1(C^{\mathcal{K}}) & \rightharpoonup_{0,1} \langle [\sigma_1], [10] \rangle \\ \emptyset, \rho \vdash \gamma_1(C^{\mathcal{K}}) & \rightharpoonup_{0,1} \langle [\sigma_2], [20] \rangle \\ \{\sigma_1\}, \rho \vdash \gamma_1(C^{\mathcal{K}}) & \rightharpoonup_{0,1} \langle [\sigma_2], [20] \rangle \\ \{\sigma_2\}, \rho \vdash \gamma_1(C^{\mathcal{K}}) & \rightharpoonup_{0,1} \langle [\sigma_1], [10] \rangle \\ \{\sigma_1, \sigma_2\}, \rho \vdash \gamma_1(C^{\mathcal{K}}) & \not\rightharpoonup_{0,1} \end{array} \qquad \begin{array}{rl} \emptyset, \rho \vdash \gamma_2(C^{\mathcal{K}}) & \rightharpoonup_{0,1} \langle [\sigma_1], [-1] \rangle \\ \emptyset, \rho \vdash \gamma_2(C^{\mathcal{K}}) & \rightharpoonup_{0,1} \langle [\sigma_2], [-2] \rangle \\ \{\sigma_1\}, \rho \vdash \gamma_2(C^{\mathcal{K}}) & \rightharpoonup_{0,1} \langle [\sigma_2], [-2] \rangle \\ \{\sigma_2\}, \rho \vdash \gamma_2(C^{\mathcal{K}}) & \rightharpoonup_{0,1} \langle [\sigma_1], [-1] \rangle \\ \{\sigma_1, \sigma_2\}, \rho \vdash \gamma_2(C^{\mathcal{K}}) & \not\rightharpoonup_{0,1} \end{array}$$

4.2.2 Les stratégies

Les paragraphes ci-dessous présentent le calcul de la relation :

$$\emptyset, \rho \vdash \{\gamma_1 ; \gamma_2\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} C'$$

pour chacune des stratégies.

Stratégie asynchrone sans priorité. Pour cette stratégie le choix de la règle à appliquer est non-déterministe. À cela, on ajoute le non-déterminisme du filtrage acceptant d'une part σ_1 et d'autre part σ_2. Il en résulte quatre arbres de preuve possibles pour A (voir le système de règles 11) :

1. Arbre 38 : application de γ_1 filtrant σ_1,

$$\dfrac{\emptyset, \rho \vdash \gamma_1(C^{\mathcal{K}}) \rightharpoonup_{0,1} \langle [\sigma_1], [10] \rangle \qquad \dfrac{}{\{\sigma_1\}, \rho \vdash \{\,\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} 2.\sigma_2}\,30}{\emptyset, \rho \vdash \{\gamma_1 ; \gamma_2\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} 10.\sigma_1 + 2.\sigma_2}\,26/28 \tag{38}$$

$$\dfrac{\emptyset, \rho \vdash \gamma_1(C^{\mathcal{K}}) \rightharpoonup_{0,1} \langle [\sigma_2], [20] \rangle \qquad \dfrac{}{\{\sigma_2\}, \rho \vdash \{\,\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} 1.\sigma_1}\,30}{\emptyset, \rho \vdash \{\gamma_1 ; \gamma_2\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} 1.\sigma_1 + 20.\sigma_2}\,26/28 \tag{39}$$

$$\dfrac{\emptyset, \rho \vdash \gamma_2(C^{\mathcal{K}}) \rightharpoonup_{0,1} \langle [\sigma_1], [-1] \rangle \qquad \dfrac{}{\{\sigma_1\}, \rho \vdash \{\,\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} 2.\sigma_2}\,30}{\emptyset, \rho \vdash \{\gamma_1 ; \gamma_2\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} (-1).\sigma_1 + 2.\sigma_2}\,26 \tag{40}$$

$$\dfrac{\emptyset, \rho \vdash \gamma_2(C^{\mathcal{K}}) \rightharpoonup_{0,1} \langle [\sigma_2], [-2] \rangle \qquad \dfrac{}{\{\sigma_2\}, \rho \vdash \{\,\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} 1.\sigma_1}\,30}{\emptyset, \rho \vdash \{\gamma_1 ; \gamma_2\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} 1.\sigma_1 + (-2).\sigma_2}\,26 \tag{41}$$

RÈG. 11: Exemple d'utilisation de la stratégie asynchrone avec (les deux premiers arbres uniquement) et sans priorité (les quatre arbres).

2. Arbre 39 : application de γ_1 filtrant σ_2,
3. Arbre 40 : application de γ_2 filtrant σ_1,
4. Arbre 41 : application de γ_2 filtrant σ_2.

Stratégie asynchrone avec priorité. Le degré de liberté laissé par la stratégie asynchrone sans priorité pour le choix de la règle de réécriture à appliquer disparaît ici forçant l'application de la règle γ_1. Cette règle peut filtrer soit σ_1 soit σ_2. Les règles décrivant ces deux réductions possibles correspondent alors aux arbres 38 et 39 de la stratégie asynchrone sans priorité sur les règles.

Stratégie synchrone avec priorité. Cette stratégie demande le plus d'appels récursifs. En effet, chaque règle doit être appliquée le plus de fois possible jusqu'à ce qu'elle ne puisse plus être appliquée. On passe alors à la règle suivante. Dans notre exemple, deux arbres différents peuvent être construits. Ils sont présentés par le système de règles 12. Les arbres 42 et 43 présentent le filtrage des cellules σ_1 et σ_2 (dans chacun des deux ordres possibles). Une fois ces cellules filtrées aucune règle ne peut plus s'appliquer. Les arbres partagent alors les appels récursifs rendant compte de ces échecs décrits par l'arbre 44.

Stratégie synchrone sans priorité. Il s'agit de la stratégie d'application offrant le plus de cas différents pour le sous-arbre A. Il peut en effet être appliqué :

- la règle γ_1 deux fois,
- la règle γ_1 puis la règle γ_2,
- la règle γ_2 puis la règle γ_1, et enfin

$$\dfrac{\emptyset, \rho \vdash \gamma_1(C^{\mathcal{K}}) \rightharpoonup_{0,1} \langle [\sigma_1], [10] \rangle \qquad \dfrac{\{\sigma_1\}, \rho \vdash \gamma_1(C^{\mathcal{K}}) \rightharpoonup_{0,1} \langle [\sigma_2], [20] \rangle \quad \textit{cf.}\ \text{règle 44}}{\{\sigma_1\}, \rho \vdash \{\gamma_1 ; \gamma_2\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} 20.\sigma_2}\ 34}{\emptyset, \rho \vdash \{\gamma_1 ; \gamma_2\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} 10.\sigma_1 + 20.\sigma_2}\ 34 \tag{42}$$

$$\dfrac{\emptyset, \rho \vdash \gamma_1(C^{\mathcal{K}}) \rightharpoonup_{0,1} \langle [\sigma_2], [20] \rangle \qquad \dfrac{\{\sigma_2\}, \rho \vdash \gamma_1(C^{\mathcal{K}}) \rightharpoonup_{0,1} \langle [\sigma_1], [10] \rangle \quad \textit{cf.}\ \text{règle 44}}{\{\sigma_2\}, \rho \vdash \{\gamma_1 ; \gamma_2\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} 10.\sigma_1}\ 34}{\emptyset, \rho \vdash \{\gamma_1 ; \gamma_2\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} 10.\sigma_1 + 20.\sigma_2}\ 34 \tag{43}$$

$$\dfrac{\{\sigma_1, \sigma_2\}, \rho \vdash \gamma_1(C^{\mathcal{K}}) \not\rightharpoonup_{0,1} \qquad \dfrac{\{\sigma_1, \sigma_2\}, \rho \vdash \gamma_2(C^{\mathcal{K}}) \not\rightharpoonup_{0,1} \quad \dfrac{}{\{\sigma_1, \sigma_2\}, \rho \vdash \{\ \}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} 0_{\mathsf{CollType}}}\ 30}{\{\sigma_1, \sigma_2\}, \rho \vdash \{\gamma_2\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} 0_{\mathsf{CollType}}}\ 35}{\{\sigma_1, \sigma_2\}, \rho \vdash \{\gamma_1 ; \gamma_2\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} 0_{\mathsf{CollType}}}\ 35 \tag{44}$$

RÈG. 12: Exemple d'utilisation de la stratégie synchrone avec priorité.

- la règle γ_2 deux fois.

Dans chaque cas, la première application de règle est non-déterministe : on peut filtrer σ_1 ou σ_2. La seconde application se fait alors sur la cellule restante. Ainsi 8 réductions peuvent être construites. Une fois les deux applications de règles faites, plus aucune cellule n'est disponible. Il ne peut donc y avoir une troisième application. C'est le cas de base de la récurrence associée à la réduction synchrone sans priorité : le nombre d'éléments pouvant être filtrés est fini à l'état initial et décroît à chaque application de règle. Les arbres de réduction pour la stratégie synchrone avec priorité sont donnés par le système de règles 13.

4.3 Exemple de preuve

La définition formelle de la sémantique d'un langage a deux objectifs : d'une part, elle permet une bonne compréhension des mécanismes développés dans le langage et, d'autre part, elle permet le développement de preuves mathématiques pour vérifier certaines propriétés des programmes écrits dans ce langage.

L'exemple qui suit n'est pas développé en détail mais simplement esquissé. Nous esquissons le raisonnement à suivre pour prouver une propriété. La stratégie maximale parallèle des L systèmes [PLH+90, PH92, LJ92, RS92] peut être caractérisée par le fait qu'*aucune sous-collection restant après le filtrage ne peut être filtrée par aucune des règles*. Notons P cette propriété. Nous souhaitons montrer que la stratégie synchrone sans priorité sur les règles vérifie cette propriété.

Prenons le cas des patches (celui des $\langle n, p\rangle$-transformations est identique). Soit un ensemble de règles $\{\ldots;\gamma_i;\ldots\}$ à appliquer sur une collection $C^{\mathcal{K}}$ dans le contexte d'un environnement ρ et sachant que les cellules de l'ensemble $S \subset \mathcal{K}$ ont déjà été filtrées. La propriété P doit être montrée pour toute collection résultat $C'^{\mathcal{K}'}$, c'est-à-dire

$$S, \rho \vdash \{\ldots;\gamma_i;\ldots\}(C^{\mathcal{K}}) \rightsquigarrow C'^{\mathcal{K}'} \Rightarrow P(S, \rho, \{\ldots;\gamma_i;\ldots\}, C^{\mathcal{K}}, C'^{\mathcal{K}'})$$

Dans le cadre de la stratégie synchrone sans priorité, seules les deux règles d'inférence 31 et 33 peuvent être appliquées.

4.3.1 Application de la règle 31

L'utilisation de la règle 31 construit l'arbre de preuve suivant :

$$\frac{\forall i \qquad S, \rho \vdash \gamma_i(C) \not\rightarrow}{S, \rho \vdash \{\ldots ; \psi_i ; \ldots\}(C^{\mathcal{K}}) \rightsquigarrow C'^{\mathcal{K}'}}\ {}^{31}$$

Les prémisses montrent qu'il n'existe pas de sous-collection pouvant être filtrée par l'un des motifs des règles γ_i.

La propriété $P(S, \rho, \{\ldots;\gamma_i;\ldots\}, C^{\mathcal{K}}, C'^{\mathcal{K}'})$ est donc vérifiée.

4.3.2 Application de la règle 33

L'utilisation de la règle 33 construit l'arbre de preuve suivant :

$$\frac{S, \rho \vdash \psi_i(C) \rightharpoonup \langle S_\pi, C_e^{\mathcal{K}_e}\rangle \qquad S \cup S_\pi, \rho \vdash \{\ldots ; \psi_i ; \ldots\}(C^{\mathcal{K}}) \rightsquigarrow C_s^{\mathcal{K}_s}}{S, \rho \vdash \{\ldots ; \psi_i ; \ldots\}(C^{\mathcal{K}}) \rightsquigarrow C_e^{\mathcal{K}_e} + C_s^{\mathcal{K}_s}}\ {}^{33}$$

avec $C'^{\mathcal{K}'} = C_e^{\mathcal{K}_e} + C_s^{\mathcal{K}_s}$.

Ainsi, pour toute sous-collection filtrée S_π, pour vérifier $P(S, \rho, \{\ldots;\gamma_i;\ldots\}, C^{\mathcal{K}}, C'^{\mathcal{K}'})$, il suffit que $P(S \cup S_\pi, \rho, \{\ldots;\gamma_i;\ldots\}, C^{\mathcal{K}}, C_s^{\mathcal{K}_s})$. Cette remarque conduit vers une preuve par récurrence

$$\dfrac{\emptyset, \rho \vdash \gamma_1(C^{\mathcal{K}}) \rightharpoonup_{0,1} \langle [\sigma_1], [10] \rangle \qquad \dfrac{\dfrac{}{\{\sigma_1, \sigma_2\}, \rho \vdash \{\gamma_1 ; \gamma_2\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} 0_{\mathsf{CollType}}}\,30 \quad \{\sigma_1\}, \rho \vdash \gamma_1(C^{\mathcal{K}}) \rightharpoonup_{0,1} \langle [\sigma_2], [20] \rangle}{\{\sigma_1\}, \rho \vdash \{\gamma_1 ; \gamma_2\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} 20.\sigma_2}\,32}{\emptyset, \rho \vdash \{\gamma_1 ; \gamma_2\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} 10.\sigma_1 + 20.\sigma_2}\,32 \tag{45}$$

$$\dfrac{\emptyset, \rho \vdash \gamma_1(C^{\mathcal{K}}) \rightharpoonup_{0,1} \langle [\sigma_2], [20] \rangle \qquad \dfrac{\dfrac{}{\{\sigma_2, \sigma_1\}, \rho \vdash \{\gamma_1 ; \gamma_2\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} 0_{\mathsf{CollType}}}\,30 \quad \{\sigma_2\}, \rho \vdash \gamma_1(C^{\mathcal{K}}) \rightharpoonup_{0,1} \langle [\sigma_1], [10] \rangle}{\{\sigma_2\}, \rho \vdash \{\gamma_1 ; \gamma_2\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} 10.\sigma_1}\,32}{\emptyset, \rho \vdash \{\gamma_1 ; \gamma_2\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} 20.\sigma_2 + 10.\sigma_1}\,32 \tag{46}$$

$$\dfrac{\emptyset, \rho \vdash \gamma_1(C^{\mathcal{K}}) \rightharpoonup_{0,1} \langle [\sigma_1], [10] \rangle \qquad \dfrac{\dfrac{}{\{\sigma_1, \sigma_2\}, \rho \vdash \{\gamma_1 ; \gamma_2\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} 0_{\mathsf{CollType}}}\,30 \quad \{\sigma_1\}, \rho \vdash \gamma_2(C^{\mathcal{K}}) \rightharpoonup_{0,1} \langle [\sigma_2], [(-2)] \rangle}{\{\sigma_1\}, \rho \vdash \{\gamma_1 ; \gamma_2\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} (-2).\sigma_2}\,32}{\emptyset, \rho \vdash \{\gamma_1 ; \gamma_2\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} 10.\sigma_1 + (-2).\sigma_2}\,32 \tag{47}$$

$$\dfrac{\emptyset, \rho \vdash \gamma_1(C^{\mathcal{K}}) \rightharpoonup_{0,1} \langle [\sigma_2], [20] \rangle \qquad \dfrac{\dfrac{}{\{\sigma_2, \sigma_1\}, \rho \vdash \{\gamma_1 ; \gamma_2\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} 0_{\mathsf{CollType}}}\,30 \quad \{\sigma_2\}, \rho \vdash \gamma_2(C^{\mathcal{K}}) \rightharpoonup_{0,1} \langle [\sigma_1], [(-1)] \rangle}{\{\sigma_2\}, \rho \vdash \{\gamma_1 ; \gamma_2\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} (-1).\sigma_1}\,32}{\emptyset, \rho \vdash \{\gamma_1 ; \gamma_2\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} 20.\sigma_2 + (-1).\sigma_1}\,32 \tag{48}$$

$$\dfrac{\emptyset, \rho \vdash \gamma_2(C^{\mathcal{K}}) \rightharpoonup_{0,1} \langle [\sigma_1], [(-1)] \rangle \qquad \dfrac{\dfrac{}{\{\sigma_1, \sigma_2\}, \rho \vdash \{\gamma_1 ; \gamma_2\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} 0_{\mathsf{CollType}}}\,30 \quad \{\sigma_1\}, \rho \vdash \gamma_1(C^{\mathcal{K}}) \rightharpoonup_{0,1} \langle [\sigma_2], [20] \rangle}{\{\sigma_1\}, \rho \vdash \{\gamma_1 ; \gamma_2\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} 20.\sigma_2}\,32}{\emptyset, \rho \vdash \{\gamma_1 ; \gamma_2\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} (-1).\sigma_1 + 20.\sigma_2}\,32 \tag{49}$$

$$\dfrac{\emptyset, \rho \vdash \gamma_2(C^{\mathcal{K}}) \rightharpoonup_{0,1} \langle [\sigma_2], [(-2)] \rangle \qquad \dfrac{\dfrac{}{\{\sigma_2, \sigma_1\}, \rho \vdash \{\gamma_1 ; \gamma_2\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} 0_{\mathsf{CollType}}}\,30 \quad \{\sigma_2\}, \rho \vdash \gamma_1(C^{\mathcal{K}}) \rightharpoonup_{0,1} \langle [\sigma_1], [10] \rangle}{\{\sigma_2\}, \rho \vdash \{\gamma_1 ; \gamma_2\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} 10.\sigma_1}\,32}{\emptyset, \rho \vdash \{\gamma_1 ; \gamma_2\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} (-2).\sigma_2 + 10.\sigma_1}\,32 \tag{50}$$

$$\dfrac{\emptyset, \rho \vdash \gamma_2(C^{\mathcal{K}}) \rightharpoonup_{0,1} \langle [\sigma_1], [(-1)] \rangle \qquad \dfrac{\dfrac{}{\{\sigma_1, \sigma_2\}, \rho \vdash \{\gamma_1 ; \gamma_2\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} 0_{\mathsf{CollType}}}\,30 \quad \{\sigma_1\}, \rho \vdash \gamma_2(C^{\mathcal{K}}) \rightharpoonup_{0,1} \langle [\sigma_2], [(-2)] \rangle}{\{\sigma_1\}, \rho \vdash \{\gamma_1 ; \gamma_2\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} (-2).\sigma_2}\,32}{\emptyset, \rho \vdash \{\gamma_1 ; \gamma_2\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} (-1).\sigma_1 + (-2).\sigma_2}\,32 \tag{51}$$

$$\dfrac{\emptyset, \rho \vdash \gamma_2(C^{\mathcal{K}}) \rightharpoonup_{0,1} \langle [\sigma_2], [(-2)] \rangle \qquad \dfrac{\dfrac{}{\{\sigma_2, \sigma_1\}, \rho \vdash \{\gamma_1 ; \gamma_2\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} 0_{\mathsf{CollType}}}\,30 \quad \{\sigma_2\}, \rho \vdash \gamma_2(C^{\mathcal{K}}) \rightharpoonup_{0,1} \langle [\sigma_1], [(-1)] \rangle}{\{\sigma_2\}, \rho \vdash \{\gamma_1 ; \gamma_2\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} (-1).\sigma_1}\,32}{\emptyset, \rho \vdash \{\gamma_1 ; \gamma_2\}(\mathcal{C}^{\mathcal{K}}) \rightsquigarrow_{0,1} (-2).\sigma_2 + (-1).\sigma_1}\,32 \tag{52}$$

Règ. 13: Exemple d'utilisation de la stratégie synchrone sans priorité.

nous ramenant dans le cas de l'application de la règle 31, qui devient notre cas de base. Cette récurrence se fait sur le nombre de sous-collections qu'il est possible de filtrer, sachant que celui-ci est fini. Intuitivement, le nombre de sous-collections restant à filtrer diminue à chaque application de la règle 33. Cette règle est appliquée jusqu'à ce qu'il ne reste plus de sous-collection à filtrer. Nous allons montrer que le nombre de sous-collections est fini et que le nombre de sous-collections filtrables décroît au fur et à mesure du filtrage.

Nombre de sous-collections fini. Il s'agit d'une preuve par induction sur la construction des motifs. Nous montrons que le nombre de sous-collections pour les motifs de patches est fini, la démarche étant exactement la même pour les motifs de chemins.

Les règles d'inférences axiomatiques du filtrage de patch (règles 19, 20, 21 et 22) imposent qu'une cellule σ d'une collection topologique $C^{\mathcal{K}}$ vérifie :

$$C^{\mathcal{K}}(\sigma) \neq 0_{\text{Abel(Val)}}$$

Les collections topologiques étant des fonctions nulles presque partout, seul un nombre fini de cellules $\sigma \in \mathcal{K}$ convient. Ces règles ne filtrent donc qu'un nombre fini de sous-collections (réduites à une seule cellule)

Le cas d'induction concerne la poursuite du filtrage, règle 23. Soit π_1 et π_2 deux motifs ne filtrant qu'un nombre fini de sous-collections. Montrons que le motif $\pi_1\pi_2$ ne filtre qu'un nombre fini de sous-collections. Soit S_1 (resp. S_2) une sous-collection filtrée par π_1 (resp. π_2). Ces deux sous-collections vérifient $S_1 \cap S_2 = \emptyset$ pour éviter qu'une même cellule ne soit filtrée à la fois par π_1 et π_2. La règle 23 définit $S_1 \cup S_2$ comme la sous-collection filtrée par le motif $\pi_1\pi_2$. Par hypothèse d'induction, il n'existe qu'un nombre fini de sous-collections S_1 (resp. S_2) possibles. Il n'existe donc qu'un nombre fini de sous-collections $S_1 \cup S_2$ filtrées par $\pi_1\pi_2$.

Nous montrons ainsi, que pour tout motif π, il n'existe qu'un nombre fini de sous-collections filtrées par π dans une collection $C^{\mathcal{K}}$.

Diminution du nombre de sous-collections non-filtrées. Dans l'application de la règle 33, l'ensemble S des cellules filtrées est augmenté de S_π, l'ensemble des cellules filtrées par l'application de l'une des règles γ_i. Nous supposons que les motifs utilisés sont bien construits. Ils vérifient en particulier le prédicat `OkPatch`, vérifiant qu'au moins une des clauses du motif filtre **et** consomme une cellule.

Suivant le même type de preuve que précédemment (induction sur la construction des motifs de patches), il est possible de montrer que si `OkPatch`(π) est vérifié pour un motif π, l'ensemble S_π ne peut pas être vide.

Ainsi, lors de l'appel récursif de la règle 33, le nombre de cellules augmente obligatoirement (passage de S à $S \cup S_\pi$ où S_π n'est pas vide). Le nombre de sous-collections filtrées étant fini, le nombre de cellules non-filtrées, et donc de sous-collections non-filtrées, diminue lors de l'appel récursif.

Retour à la propriété P. Le nombre de sous-collections pouvant être filtrées diminuant, il arrive un moment où la règle 33 ne peut plus être appliquée et où la règle 31 est utilisée pour finalement terminer l'arbre de preuve (l'arbre de preuve est donc de hauteur finie).

La récurrence se fait de la façon suivante :

- **Cas de base** : Supposons que S contienne l'ensemble des cellules filtrables (toutes les cellules décorées par une valeur différente de $0_{\text{Abel(Val)}}$ dans $C^{\mathcal{K}}$) ; aucune cellule ne peut donc être filtrée. La règle 31 est appliquée et la propriété

$$P(S, \rho, \{\ldots; \gamma_i; \ldots\}, C^{\mathcal{K}}, C'^{\mathcal{K}'})$$

est vérifiée comme nous l'avons fait remarquer précédemment.

- **Récurrence** : Supposons que la propriété P est vérifiée pour tout ensemble S' de cellules topologiques filtrables de $C^{\mathcal{K}}$ (il n'en existe qu'un nombre fini comme nous venons de le voir) tel que $S \subsetneq S'$. Nous supposons donc que :

$$P(S', \rho, \{\ldots;\gamma_i;\ldots\}, C^{\mathcal{K}}, C_s^{\mathcal{K}_s})$$

Rappelons que nous considérons l'application de la règle 33 :

$$\frac{S, \rho \vdash \psi_i(C) \rightharpoonup \langle S_\pi, C_e^{\mathcal{K}_e}\rangle \qquad S \cup S_\pi, \rho \vdash \{\ldots\ ;\ \psi_i\ ;\ \ldots\}(C^{\mathcal{K}}) \rightsquigarrow C_s^{\mathcal{K}_s}}{S, \rho \vdash \{\ldots\ ;\ \psi_i\ ;\ \ldots\}(C^{\mathcal{K}}) \rightsquigarrow C_e^{\mathcal{K}_e} + C_s^{\mathcal{K}_s}}\ {\scriptstyle 33}$$

Or comme S_π ne peut pas être vide, $S \cup S_\pi \subsetneq S$. Par hypothèse de récurrence,

$$P(S \cup S_\pi, \rho, \{\ldots;\gamma_i;\ldots\}, C^{\mathcal{K}}, C_s^{\mathcal{K}_s})$$

est vérifiée. La sous-collection S_π étant filtrée par l'application de la règle 33, on vérifie alors

$$P(S, \rho, \{\ldots;\gamma_i;\ldots\}, C^{\mathcal{K}}, C'^{\mathcal{K}'})$$

Finalement, pour tout ensemble S de cellules filtrées, on a :

$$S, \rho \vdash \{\ldots;\gamma_i;\ldots\}(C^{\mathcal{K}}) \rightsquigarrow C'^{\mathcal{K}'} \Rightarrow P(S, \rho, \{\ldots;\gamma_i;\ldots\}, C^{\mathcal{K}}, C'^{\mathcal{K}'})$$

ce qui est en particulier vrai pour $S = \emptyset$.

En conclusion, la stratégie synchrone sans priorité est donc une stratégie maximale parallèle.

Chapitre 5

Stratégies d'application probabilistes

Dans ce chapitre, nous nous intéressons à une modification de la sémantique du langage mini-MGS présentée dans le chapitre 4 pour tenir compte de stratégies d'application des règles probabilistes. On entend par *sémantique probabiliste* le calcul permettant d'obtenir une valeur donnée en résultat d'un programme mini-MGS.

L'approche utilisée dans ce chapitre, fondée sur l'association à une expression d'une densité de probabilité n'est pas nouvelle. Elle a été utilisée dans [Jon90] où la sémantique d'un langage fonctionnel a été développée. Par rapport à ce cadre nous restreignons les lois de probabilité utilisées à des densités de probabilité particulières correspondant à des fonctions nulles presque partout, associant une probabilité à un sous-ensemble fini des valeurs manipulées par le langage. Nous évitons ainsi la manipulation délicate et les difficultés techniques induites par l'utilisation de densités non-nulles presque partout. L'objectif de ce chapitre n'est donc pas le développement d'une nouvelle sémantique pour la formalisation de constructions stochastiques, mais plutôt de décrire formellement l'application stochastique de règles (alors que [Jon90] traite de la définition et de l'application des fonctions). Cela implique de formaliser le non-déterminisme du filtrage et les mécanismes (stochastiques) de choix d'une règle à appliquer.

1 Motivations

La présence de phénomènes stochastiques dans un grand nombre de systèmes physiques n'est plus à démontrer. Ces dernières années, un nombre croissant de chercheurs s'est attaché à l'étude de l'aléa et du bruit présent dans les systèmes biologiques. Le lecteur intéressé pourra se reporter à [MSD04] où sont détaillés de nombreux phénomènes stochastiques en biologie, et ce du niveau moléculaire au niveau cellulaire. De façon à pouvoir rendre compte de ces phénomènes, de nombreux formalismes dédiés à la simulation de systèmes dynamiques ont intégré des mécanismes afin de pouvoir exprimer des modèles stochastiques. On peut citer :

- Les L systèmes stochastiques [Jür76, ES80, Yok80, Nis80, Pru87, PH89, Cie06],
- Les P systèmes stochastiques [Obt03, Mad03, AC03, PBMZ06],
- Les automates cellulaires stochastiques [TN87, Fre90, Mar98].

Le langage MGS étant dédié à la simulation de modèles de systèmes dynamiques, une extension pour fournir les mécanismes nécessaires à l'introduction de la stochasticité dans l'application des transformations est naturel.

En ce qui concerne les systèmes de réécriture, l'application de règles paramétrées par des lois de probabilité a déjà été étudiée. Ces mécanismes d'application probabilistes de règles ont été introduits dans des environnements généraux de réécriture : en Maude par exemple, ce mécanisme passe par la notion de *théories de réécriture probabiliste* [KKMA03] ; un système de stratégies de réécriture probabiliste a également été proposé pour le système de réécriture Elan [BK02, BH03].

L'introduction de la stochasticité dans les systèmes de réécriture se traduit par la définition de nouvelles relations d'évaluation, et par conséquent de nouvelles stratégies d'application des règles obéissant aux contraintes imposées par les lois de probabilité associées au système de règles. Dans le cadre du langage MGS, nous nous sommes intéressés à deux nouveaux types de stratégies d'application pour les transformations :

1. *Stratégie stochastique standard* (ou stratégie StS) : il s'agit de l'extension de la stratégie asynchrone sans priorité. En effet, cette dernière permet l'application d'une des règles (dont on peut trouver une instance de motif dans la collection) de façon aléatoire. L'implantation de cette stratégie nécessite un générateur de nombres aléatoires avec une loi de probabilité uniforme[1] : il est utilisé pour choisir de façon uniforme quelle règle appliquer. La stratégie stochastique étend la stratégie asynchrone sans priorité en proposant de paramétrer la loi de probabilité pour le tirage au sort de la règle à appliquer. Chaque règle doit alors disposer de sa probabilité d'application. Les règles ainsi pondérées, la stratégie stochastique standard probabilise le déclenchement de l'application de l'une des règles.

2. *Stratégie pour les systèmes à évènements discrets* (ou stratégie SED) : alors que la stratégie stochastique standard permet de probabiliser le déclenchement de l'une des règles de la transformation, la stratégie SED probabilise le choix d'une occurrence de sous-collection filtrée et l'application de la règle correspondante. Ce type de stratégie permet de modéliser et de simuler un système dont les lois d'évolution sont décrites par des règles dont la probabilité dépend :
   - du nombre d'applications potentielles de cette règle, et
   - du nombre d'applications potentielles des autres règles.

   Par exemple, les systèmes chimiques font intervenir ce type de probabilité [Gil77]. En effet, même si les affinités chimiques sont très grandes, une réaction n'aura que peu de probabilité

[1] L'interprète MGS utilise le générateur de nombre aléatoire de la GSL (pour *GNU Scientific Library* [GDT+06]).

de se produire si l'un des réactants est rare. Les probabilités interviennent donc bien pour pondérer les occurrences possibles d'applications d'une règle, et non le choix de cette règle.

Relation entre stratégie SED et StS. En MGS, il est possible d'émuler une stratégie SED en utilisant une stratégie StS : l'idée est de paramétrer chaque règle par une expression faisant intervenir le nombre d'instances possibles de la règle. En effet, contrairement à d'autres systèmes de réécriture probabilistes, MGS permet d'associer une expression à chaque règle et non une simple valeur flottante, autorisant un calcul complexe pouvant faire intervenir la collection argument de la transformation (par la variable `self`). Ce calcul se fait à chaque application de la transformation.

Cependant, le calcul du nombre d'instances dépend de la « forme » du motif ainsi que de la collection argument dont la topologie et les décorations sont arbitraires. Un calcul *ad-hoc* est nécessaire. Par exemple, les calculs donnés par D.T. Gillespie dans [Gil77] correspondent aux nombres de sous-collections filtrées par des règles avec des motifs sans garde, de longueur fixe (*i.e.* sans itération ni choix) et dont les motifs élémentaires filtrent des constantes scalaires (*i.e.* sans joker), appliquées sur des multi-ensembles (voir le chapitre 10).

Il existe cependant un moyen de calculer de façon générique le nombre d'instances de motif en construisant l'ensemble des sous-collections filtrées par les règles pour finalement en déduire leur cardinal. La stratégie SED automatise ce calcul. Le programmeur paramètre chaque règle par la probabilité d'occurrence de l'évènement discret qu'elle modélise. Nous appellerons cette probabilité la *constante stochastique* de la règle, en référence au terme utilisé par D.T. Gillespie dans [Gil77].

L'application stochastique des résultats amène à considérer une évaluation probabiliste de n'importe quelle expression. Il nous faut donc modifier la sémantique en conséquence.

2 Outils mathématiques

2.1 Nécessité d'une densité de probabilité

O. Bournez présente dans [BH03] un système de réécriture probabiliste de termes. La réduction d'un terme M en un terme N est caractérisée par la relation :

$$M \Rightarrow_p N$$

où p correspond à la probabilité de réduction de M en N. Suivant ce même principe, il paraît envisageable de définir une nouvelle relation pour l'évaluation des expressions mini-MGS de la forme suivante :

$$\rho \vdash e \rightarrow_p v$$

se lisant naturellement : « l'expression e s'évalue avec une probabilité p en la valeur v dans l'environnement ρ ».

Définir une telle relation est possible, mais difficile à calculer. En effet, supposons qu'il existe exactement et uniquement deux arbres A_1 et A_2 pour la relation d'évaluation d'une expression e en une valeur v :

$$\frac{A_1}{\rho \vdash e \rightarrow v} \qquad \frac{A_2}{\rho \vdash e \rightarrow v}$$

Nous souhaitons maintenant définir l'évaluation *probabiliste* de l'expression e en v. Afin de réaliser cette évaluation, nous transformons les règles d'inférence mises en jeu dans la construction des arbres A_1 et A_2 pour faire intervenir les probabilités : supposons que l'arbre A_1 (resp.

A_2) est transformé en un arbre équivalent mais avec probabilité $A_1^{p_1}$ (resp. $A_2^{p_2}$) définie de telle sorte que :

$$\frac{A_1^{p_1}}{\rho \vdash e \rightarrow_{A_1,p_1} v} \qquad \frac{A_2^{p_2}}{\rho \vdash e \rightarrow_{A_2,p_2} v}$$

où la relation $\rho \vdash e \rightarrow_{A_i,p_i} v$ signifie que « l'expression e s'évalue en la valeur v dans l'environnement ρ avec une probabilité p_i *sachant que l'arbre de preuve est construit de façon identique à l'arbre* A_i ». Sachant que les seules réductions possibles sont A_1 et A_2, et que l'une et l'autre sont des évènements indépendants, on peut déduire de ces réductions que :

$$\rho \vdash e \rightarrow_{p_1+p_2} v$$

Cet exemple nous montre qu'il est indispensable de construire l'ensemble des réductions possibles de l'évaluation d'une expression e en une valeur v (c'est-à-dire, l'ensemble des arbres prouvant la relation $\rho \vdash e \rightarrow v$) pour pouvoir calculer la relation $\rho \vdash e \rightarrow_p v$.

Nous ne poursuivons pas dans la suite avec la relation $\rho \vdash e \rightarrow_p v$; en effet, elle ne répond pas réellement aux motivations exprimées dans la section précédente. Cette relation fournit un moyen de *probabiliser l'évaluation des expressions*. Or nous souhaitons définir un mécanisme calculant la *probabilité du résultat de l'évaluation*. Pour cela, il faut associer à chaque expression les valeurs possibles de son évaluation, avec pour chaque valeur sa probabilité d'obtention. Cela peut s'écrire :

$$\rho \vdash e \rightarrow p_1.v1 + p_2.v_2 + \dots$$

Le mécanisme d'évaluation établit que l'expression e est évaluée en la valeur v_1 avec la probabilité p_1, en v_2 avec la probabilité p_2, etc. Si nous comparons avec l'approche précédente, cela revient à établir : $\rho \vdash e \rightarrow_{p_1} v_1$, $\rho \vdash e \rightarrow_{p_2} v_2$, ... Nous appelons l'expression $p_1.v1 + p_2.v_2 + \dots$ une *densité de probabilité* sur les valeurs de Val.

2.2 Densité de probabilité

Voici la définition de la densité de probabilité.

Définition 34 (Densité de probabilité) *Soit E un ensemble, on définit une* densité de probabilité sur E *comme une fonction totale $P : E \rightarrow [0, 1]$ nulle presque partout telle que*

$$\sum_{e \in E} P(e) = 1$$

On note $\mathscr{P}(E)$ l'ensemble des densités de probabilité sur E.

Nous utilisons alors le symbole $\sum$ au lieu du signe intégral $\int$ pour rappeler la nature discrète des densités de probabilité (nous discutons ce point dans la section suivante). En tant que fonction nulle presque partout, les densités de probabilité peuvent être représentées par des sommes formelles finies. Soit P une densité de probabilité sur un ensemble E, P est dénotée :

$$P = \sum_{e \in E} P(v).v$$

Dans les équations des prochaines sections, nous utilisons les notations suivantes ; soit P, P_1 et P_2 trois densités de probabilité sur un ensemble E et r un réel de $[0, 1]$:

- $rP = \sum_{e \in E}(rP(v)).v$
- $P_1 + P_2 = \sum_{e \in E}(P_1(v) + P_2(v)).v$

Les fonctions retournées par ces deux calculs ne sont pas des densités de probabilité. Par exemple dans le cas d'une densité P multipliée par le réel $\frac{1}{2}$, on a :

$$\sum_{e\in E}(\frac{1}{2}P)(e) = \sum_{e\in E}\frac{1}{2}(P(e)) = \frac{1}{2}\sum_{e\in E}P(e) = \frac{1}{2}$$

La fonction $\frac{1}{2}P$ n'est donc pas une densité. En revanche, pour tout $r \in [0,1]$, on vérifie trivialement que $rP + (1-r)P$ est toujours une densité.

Nous terminons cette description des densités de probabilité en définissant un opérateur permettant de combiner deux densités de probabilité. Soient A et B deux ensembles arbitraires sur lesquels est défini un opérateur op combinant un élément $a \in A$ avec un élément $b \in B$ en un nouvel élément $c = a\ op\ b$ d'un ensemble C. Soient deux densités de probabilité $P_A \in \mathscr{P}(A)$ et $P_B \in \mathscr{P}(B)$. On définit la densité $P_A \odot_{op} P_B$, par $\forall c = a\ op\ b \in C$

$$(P_A \odot_{op} P_B)(c) = P_A(a)\, P_B(b)$$

pouvant également s'écrire :

$$P_A \odot_{op} P_B = \sum_{a\in A}\sum_{b\in B}(P_A(a)\, P_B(b)).(a\ op\ b)$$

Prenons par exemple les deux densités de probabilité suivantes sur l'ensemble de booléens $\mathcal{B}$:

$$P_1 = \frac{1}{4}.\text{true} + \frac{3}{4}.\text{false} \qquad\qquad P_2 = \frac{2}{5}.\text{true} + \frac{3}{5}.\text{false}$$

On calcule la conjonction de P_1 et P_2 par

$$\begin{aligned} P_1 \odot_{\wedge} P_2 &= \frac{1}{4}\frac{2}{5}.(\text{true}\wedge\text{true}) + \frac{1}{4}\frac{3}{5}.(\text{true}\wedge\text{false}) + \frac{3}{4}\frac{2}{5}.(\text{false}\wedge\text{true}) + \frac{3}{4}\frac{3}{5}.(\text{false}\wedge\text{false}) \\ &= \frac{1}{10}.\text{true} + (\frac{3}{20} + \frac{3}{10} + \frac{9}{20}).\text{false} \\ &= \frac{1}{10}.\text{true} + \frac{9}{10}.\text{false} \end{aligned}$$

2.3 Critique de la notion de densité de probabilité

Les densités de probabilité ne sont pas nécessairement des fonctions nulles presque partout. Nous nous restreignons ici à ce cas moins général afin d'éviter les difficultés soulignées dans [Jon90].

Dans sa thèse [Jon90], C. Jones précise que l'utilisation des densités de probabilité discrète apparaît limitée, notamment pour représenter des distributions de probabilité d'ensembles non-dénombrables. Par exemple, pour le tirage aléatoire d'un nombre réel entre 0 et 1, chaque réel a une probabilité nulle d'être tiré. Pour résoudre ce problème, on considère des distributions de probabilité associant une probabilité à certains sous-ensembles de résultats. C. Jones développe dans sa thèse une étude poussée et qui fait office de référence quant à la formalisation des langages non-déterministes.

L'ambition de la sémantique que nous proposons dans ce chapitre est plus modeste : on cherche à introduire les concepts liés à la stochasticité des modèles spécifiés par des transformations et des collections topologiques dans le cadre de la simulation des systèmes dynamiques. Dans ces problèmes, les systèmes sont représentés de façon finie et l'ensemble de leurs avenirs possibles est également fini : le non-déterminisme de MGS provenant des transformations et le

$\mathcal{C}$

cte ::= n | f | `true` | `false` | s | `<undef>` | {||} | fct

$\mathcal{F}$

fct ::= `PosGen` | `Extension` | `Concat` | `Cons` | ...

Σ

e ::= cte
| `x`
| `^x`
| e(`)`e
| e`<`e`,`e`>(`e`)`
| λ`x·`e
| `if` e `then` e `else` e
| `trans<x,x>{` ... `;` γ `;` ... `}`

Γ

γ ::= μ =$\{e\}\Rightarrow$ e

M

μ ::= _
| μ `,` μ
| μ `as x`
| μ `/` e

GRAM. 2: Grammaire restreinte du langage mini-MGS pour la sémantique probabiliste.

nombre de sous-collections d'une collection étant fini (comme nous l'avons précisé dans le chapitre 4), il n'existe qu'un nombre fini de résultats possibles à l'application d'une transformation. Dans cette optique, nous n'avons pas besoin de manipuler des ensembles infinis de résultats d'évaluation. C'est pourquoi, la suite du chapitre est à lire sous l'hypothèse que les arbres de preuve sont finis :

- en hauteur : tout programme ne terminant pas n'est pas considéré,
- en largeur : un programme ne peut être évalué avec une probabilité non-nulle qu'en un nombre fini de valeurs.

Nous précisons cependant que l'intuition nous pousse à penser que l'extension de ces notions à des considérations plus proches de celles de [Jon90] est possible. Le lecteur est invité à lire [Jon90, Mon00, PPT05] pour plus de détails sur les langages et les sémantiques stochastiques.

3 Une nouvelle sémantique

Cette section présente comment la sémantique du chapitre 4 est modifiée pour prendre en compte les densités de probabilité :

- la syntaxe est modifiée pour réduire le nombre de règles d'inférence,
- les domaines sont modifiés pour prendre en compte les densités de probabilité des résultats,
- les règles d'inférence sont modifiées et les nouvelles stratégies d'application des règles présentées.

3.1 Nouvelle syntaxe

La grammaire 1 (page 99) du langage mini-MGS est réduite :

- les patches ne sont pas considérés, les transformations de chemins étant suffisantes pour illustrer nos propos,
- les règles de construction des motifs sont réduites à 4 :
  1. le joker `_`,
  2. la poursuite μ_1 `,` μ_2,
  3. le nommage μ `as x`, et
  4. la garde μ `/` e.
- une expression supplémentaire décore chaque règle. Celles-ci sont de la forme $\mu = \{e_1\} \Rightarrow e_2$ où e_1 spécifie soit la probabilité de l'application de la règle dans le cadre de la stratégie stochastique, soit la constante stochastique de la règle dans le cadre de la stratégie SED.

Ces restrictions sont formellement définies dans la nouvelle grammaire 2.

3.2 Nouveaux domaines

Les modifications apportées aux domaines concernent la définition des relations d'évaluation. En effet, celles-ci doivent retourner des densités de probabilité sur Val (dont la définition reste inchangée) et non une simple valeur de Val. Notons que les environnements associent à chaque variable de $\overline{\mathcal{X}}$ une valeur unique de Val. Les évaluations se font en effet dans des contextes déterminés (c'est-à-dire non probabilisés). Les différents résultats possibles sont regroupés ensuite en densité. Il est important de conserver à l'esprit que seules les parties droites des relations sont modifiées. Par exemple, de façon informelle, la relation d'évaluation des expressions $\rho \vdash e \rightarrow v$ est simplement transformée en $\rho \vdash e \rightarrow P$.

Nous conservons les mêmes notations que celles introduites dans le chapitre précédent ; seules les signatures des relations sont modifiées. La modification touchant la relation d'évaluation du filtrage est légèrement différente ; le filtrage retourne l'ensemble des chemins possibles.

Evaluation des expressions. Une expression s'évalue maintenant en une densité de probabilité.

Définition 35 (Evaluation des expressions) *L'évaluation des expressions est donnée par une relation ternaire :*

$$\rho \vdash e \rightarrow P$$

de signature

$$\mathcal{E} \times \Sigma \times \mathscr{P}(\mathrm{Val})$$

La relation typée est également redéfinie :

$$\rho \vdash e \rightarrow_T P$$

qui signifie que pour la densité P, pour toute valeur $v \in \mathrm{Val} - T$, qui n'est pas de type T, $P(v) = 0$. Nous exprimons ici le fait que toutes les valeurs issues de l'évaluation de e doivent être de type T.

Application probabiliste d'un ensemble de règles. L'application de plusieurs règles retourne une densité de probabilité sur l'ensemble des collections.

Définition 36 (Stratégie et reconstruction) *L'évaluation de l'application d'un ensemble de règles d'une $\langle n, p\rangle$-transformation est donnée par la relation :*

$$\rho \vdash \{\ldots ; \gamma_i ; \ldots\}(C) \rightsquigarrow_{n,p} P_{C'}$$

de signature

$$\mathcal{E} \times \textsc{Seq}(\Gamma) \times \mathsf{CollType} \times \mathbb{Z} \times \mathbb{Z} \times \mathscr{P}(\mathsf{CollType})$$

On notre l'absence de l'ensemble S des cellules déjà filtrées. En effet, les stratégies que nous décrivons sont asynchrones ; l'ensemble S est inutile.

Application probabiliste d'une règle. Comme nous le verrons plus loin, le filtrage retourne une densité sur des ensembles de chemins filtrables. Tout d'abord, simplifions le raisonnement en considérant que le filtrage retourne un seul ensemble de chemins filtrables par le motif μ, $\{\langle l_\mu, \rho'\rangle \mid l_\mu$ est filtré par $\mu\}$. Pour chacun des chemins l_μ, un environnement ρ' contenant la définition des variables de motif est également fourni. L'évaluation de la partie droite de la règle dans le contexte de l'environnement ρ' produit une densité de probabilité de séquences P_e. L'évaluation de l'application d'une règle produit un ensemble de couples (collection filtrée, densité de collections résultats calculée) : $\{\langle l_\mu, P_e\rangle \mid l_\mu$ est filtré par $\mu\}$.

Il faut cependant considérer que les gardes à évaluer pendant le filtrage sont des expressions et que, en tant que telles, leur évaluation retourne également une densité de probabilité sur l'ensemble des booléens. En d'autres termes, il faut considérer qu'un chemin l_μ présente également une probabilité d'être filtré ou non.

Généralisons cela à une densité sur des ensembles de chemins filtrables :

$$\sum p_i.\{\langle l_\mu, \rho'\rangle \mid l_\mu \text{ est filtré par } \mu\}$$

L'application d'une règle retourne donc une densité d'ensemble de couples :

$$\sum p_i.\{\langle l_\mu, P_e\rangle \mid l_\mu \text{ est filtré par } \mu\}$$

Ces considérations motivent la définition suivante :

Définition 37 (Application probabiliste d'une règle) *L'évaluation probabiliste de l'application d'une règle de $\langle n, p\rangle$-transformation est donnée par la relation :*

$$\rho \vdash \gamma(C) \rightharpoonup_{n,p} \sum p_i.\{\langle l_\mu, P_e\rangle, \ldots\}$$

de signature

$$\mathcal{E} \times \Gamma \times \mathsf{CollType} \times \mathbb{Z} \times \mathbb{Z} \times \mathscr{P}(\mathcal{P}(\textsc{Seq}(\mathcal{S}) \times \mathscr{P}(\textsc{Seq}(\mathrm{Val}))))$$

Filtrage. Le filtrage n'est pas un processus stochastique. Il permet en effet, à partir d'un motif, de sélectionner un ensemble de sous-collections : aucune décision probabiliste n'est prise au cours du filtrage, tout chemin valable est considéré. Néanmoins, comme nous venons de le voir, les gardes jalonnant le motif associent à chaque chemin filtrable une probabilité d'être filtré ou non. Un motif μ filtre par conséquent dans une collection topologique C les chemins l_μ avec une probabilité p_μ, générant un environnement de filtrage ρ' pour chacun d'eux. Ainsi, le résultat du filtrage de μ dans C est de la forme :

$$\{\langle l_\mu, \rho', p_\mu\rangle \mid l_\mu \text{ est filtré par } \mu\}$$

Définition 38 (Filtrage) *L'évaluation du filtrage pour des motifs de chemins est donnée par la relation :*

$$S, \rho \vdash \mu(C) \hookrightarrow_{n,p} \{\langle l_\mu, \rho', p_\mu \rangle, \ldots\}$$

de signature

$$\mathcal{P}(\mathcal{S}) \times \mathcal{E} \times \mathrm{M} \times \mathsf{CollType} \times \mathbb{Z} \times \mathbb{Z} \times \mathcal{P}(\textsc{Seq}(\mathcal{S}) \times \mathcal{E} \times [0,1])$$

se lisant : « dans l'environnement ρ, sachant que les cellules de S ont déjà été filtrées, le motif μ filtre dans la collection C un ensemble de $\langle n,p \rangle$-chemins désignés par des triplets $\langle l_\mu, \rho', p_\mu \rangle$ où l_μ est un $\langle n,p \rangle$-chemin, ρ' est l'environnement ρ dans lequel les variables de motif sont définies, et p_μ est la probabilité que les gardes de μ soient toutes vérifiées. »

3.3 Nouvelles règles

Cette section est dédiée à la redéfinition des règles d'inférence pour la construction des arbres de preuve. Nous commençons naturellement par l'évaluation des expressions. Nous poursuivons ensuite par le filtrage et l'application d'une règle, et nous terminons par la définition des nouvelles stratégies.

3.3.1 Expressions

Le système de règles 14 donne la sémantique d'évaluation des expressions. La définition des règles est inspirée des travaux de C. Jones [Jon90]. Voici une description détaillée de chaque règle :

- Règle 1 : une constante *cte* s'évalue en une densité de probabilité associant la probabilité 1 à elle-même : $1.cte$.
- Règle 2 : une variable de l'environnement retourne la valeur qui lui est associée à l'aide d'une densité de probabilité associant la probabilité 1 à cette valeur.
- Règle 3 : une λ-expression crée une nouvelle clôture retournée à l'aide d'une densité de probabilité associant la probabilité 1 à cette clôture.
- Règle 4 : une transformation crée une nouvelle clôture retournée à l'aide d'une densité de probabilité associant la probabilité 1 à cette clôture.
- Règle 5 : l'application d'une expression nous permet d'introduire sur un exemple les notations que nous utilisons. Soit e_1 une expression à appliquer sur une autre expression e_2. Deux types de valeurs peuvent être appliquées par la construction syntaxique $e_1(e_2)$ (le cas des $\langle n,p \rangle$-transformations est capturé par la règle 6) : les clôtures de λ-expression et les clôtures opaques (les fonctions prédéfinies supposées déterministes). L'évaluation typée de l'expression e_1 produit donc une somme en deux parties distinctes :

  $$\sum p_i.\langle \lambda \mathtt{x} \cdot e_i, \rho_i' \rangle + \sum q_j.fct_j$$

  avec à gauche la partie de la densité de probabilité concernant les λ-expressions et à droite celle des clôtures opaques. L'expression argument e_2 en revanche fournit une densité de probabilité sur les éléments de Val :

  $$\sum r_k.v_k$$

  Il est important de noter que les sommes sont faites sur les indices :

  $$\sum c_{xyz}^{abc} \equiv \sum_{xyz} c_{xyz}^{abc}$$

$$\frac{}{\rho \vdash cte \rightarrow 1.cte} \tag{1}$$

$$\frac{}{\rho \vdash \mathtt{x} \rightarrow 1.\rho(\mathtt{x})}\ \mathtt{x} \in \mathcal{D}(\rho) \tag{2}$$

$$\frac{}{\rho \vdash \lambda \mathtt{x}{\cdot}e \rightarrow 1.\langle \lambda \mathtt{x}{\cdot}e, \rho\rangle} \tag{3}$$

$$\frac{}{\rho \vdash \texttt{trans<n,p>\{ ... ; } \gamma_i \texttt{ ; ... \}} \rightarrow 1.\langle \texttt{trans<n,p>\{ ... ; } \gamma_i \texttt{ ; ... \}}, \rho\rangle} \tag{4}$$

$$\frac{\begin{array}{l}\rho \vdash e_1 \rightarrow_{\mathsf{ClType} \cup \mathsf{OpClType}} \sum p_i.\langle \lambda \mathtt{x}{\cdot}e_i, \rho_i'\rangle + \sum q_j.fct_j \\ \rho \vdash e_2 \rightarrow \sum r_k.v_k\end{array} \qquad \rho_i' \uplus [\mathtt{x} \mapsto v_k] \vdash e_i \rightarrow \sum s_l^{ik}.v_l'}{\rho \vdash e_1\texttt{(}e_2\texttt{)} \rightarrow \sum (p_i r_k s_l^{ik}).v_k' + \sum (q_j r_k).(fct_j(v_k))} \tag{5}$$

$$\frac{\begin{array}{l}\rho \vdash e_1 \rightarrow \sum p_i.\langle \texttt{trans<n,p>\{...\}}, \rho_i'\rangle \\ \rho \vdash e_2 \rightarrow_{\mathbb{Z}} \sum q_j.n_j \\ \rho \vdash e_3 \rightarrow_{\mathbb{Z}} \sum r_k.p_k \\ \rho \vdash e_4 \rightarrow_{\mathsf{CollType}} \sum s_l.C_l\end{array} \qquad \rho_i' \uplus \left[\begin{array}{l}\mathtt{self} \mapsto C_l, \\ \mathtt{n} \mapsto n_j, \\ \mathtt{p} \mapsto p_k\end{array}\right] \vdash \{\ldots\}(C_l) \rightsquigarrow_{n_j, p_k} \sum t_m^{ijkl}.C_m'}{\rho \vdash e_1\texttt{<}e_2, e_3\texttt{>(}e_4\texttt{)} \rightarrow \sum (p_i q_j r_k s_l t_m^{ijkl}).C_m'} \tag{6}$$

$$\begin{array}{c}\dfrac{\rho \vdash e_1 \rightarrow_{\mathcal{B}} 1.\mathrm{true} \qquad \rho \vdash e_2 \rightarrow P_1}{\rho \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \rightarrow P_1} \qquad \dfrac{\rho \vdash e_1 \rightarrow_{\mathcal{B}} 1.\mathrm{false} \qquad \rho \vdash e_3 \rightarrow P_2}{\rho \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \rightarrow P_2} \\ \\ \dfrac{\rho \vdash e_1 \rightarrow_{\mathcal{B}} p.\mathrm{true} + (1-p).\mathrm{false} \qquad \rho \vdash e_2 \rightarrow P_1 \qquad \rho \vdash e_3 \rightarrow P_2}{\rho \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 \rightarrow pP_1 + (1-p)P_2}\end{array} \tag{7}$$

RÈG. 14: Sémantique probabiliste : évaluation des expressions.

Les exposants expriment également une dépendance à une somme mais celle-ci n'est pas capturée par le signe $\sum$:

$$\sum c_{xyz}^{abc} \not\equiv \sum_{abcxyz} c_{xyz}^{abc}$$

Observons dans un premier temps l'application des λ-expressions. Pour chaque couple $(\langle \lambda \mathtt{x}{\cdot}e_i, \rho_i'\rangle, v_k)$ nous appliquons la λ-expression sur l'argument v_k pour obtenir une nouvelle densité de probabilité sur Val notée

$$\sum s_l^{ik}.v_l'$$

La variable v_l' dénote un parcours de Val à travers l'indice l. Pour chaque valeur, une probabilité s_l^{ik} est donnée. La notation indique qu'il s'agit de la probabilité d'obtenir la valeur v_l' (par l'indice l), **sachant que** le couple $(\langle \lambda \mathtt{x}{\cdot}e_i, \rho_i'\rangle, v_k)$ est à l'origine du calcul de v_l' (exposants i et k). La probabilité que v_l' soit le résultat du calcul impliquant le couple

$(\langle \lambda \mathtt{x}{\cdot}e_i, \rho'_i \rangle, v_k)$ est donc $p_i r_k s_l^{ik}$. Cette probabilité apparaît effectivement dans le résultat final de l'évaluation.
L'évaluation de l'application des clôtures opaques suit la même démarche. Soit le couple (fct_j, v_k). La probabilité d'obtenir $fct_j(v_k)$, **sachant qu**'on utilise le couple (fct_j, v_k), est égale à 1 (les fonctions prédéfinies sont déterministes). La probabilité non-conditionnée d'avoir $fct_j(v_k)$ est donc donnée par le calcul $q_j r_k$ qui apparaît également dans le résultat final de l'évaluation.
La densité de probabilité totale est obtenue par la somme des résultats des deux types d'application.

- Règle 6 : il s'agit de l'application des transformations. Cette règle ne pose aucun problème à partir des explications données pour la règle 5.
- Règle 7 : ces dernières règles évaluent les expressions conditionnelles. Sachant que l'expression booléenne e_1 a une probabilité p d'être vraie (et donc $(1-p)$ d'être fausse), l'évaluation de l'expression e_2 est pondérée par p et l'évaluation de l'expression e_3 est pondérée par $(1-p)$. Si $p = 0$ (resp. $1-p=0$), la branche e_2 (resp. e_3) de la conditionnelle n'est pas évaluée.

3.3.2 Filtrage d'un motif

Rappelons tout d'abord que le filtrage d'un motif μ dans une collection C génère un ensemble de triplet :

$$\{\langle l_\mu, \rho', p_\mu \rangle \mid l_\mu \text{ est filtré par } \mu\}$$

où

- l_μ est un des chemins filtrés par μ dans C,
- ρ' est l'environnement issu du filtrage de l_μ, et

$$\frac{E = \{\sigma \in \mathcal{K}_{n_1} - S \mid C^{\mathcal{K}}(\sigma) \neq 0_{\mathrm{Abel(Val)}}\}}{S, \rho \vdash \_(C^{\mathcal{K}}) \hookrightarrow_{n_1,n_2} \{\langle [\sigma], \rho, 1\rangle \mid \sigma \in E\}} \tag{8}$$

$$\frac{S, \rho \vdash \mu_1(C) \hookrightarrow_{n_1,n_2} \{\langle l_i, \rho_i, p_i\rangle\} \qquad S \sqcup l_i, \rho_i \vdash \mu_2(C) \hookrightarrow_{n_1,n_2} E_i}{S, \rho \vdash (\mu_1\,,\, \mu_2)(C) \hookrightarrow_{n_1,n_2} \bigcup \langle l_i, \rho_i, p_i\rangle \bullet_{n_1,n_2} E_i} \tag{9}$$

$$\frac{S, \rho \vdash \mu(C) \hookrightarrow_{n_1,n_2} E}{S, \rho \vdash (\mu \texttt{ as x})(C) \hookrightarrow_{n_1,n_2} \{\langle l, \rho' \uplus [\mathtt{x} \mapsto C(l), \hat{}\mathtt{x} \mapsto l], p\rangle \mid \langle l, \rho', p\rangle \in E\}} \tag{10}$$

$$\frac{S, \rho \vdash \mu(C) \hookrightarrow_{n_1,n_2} E \qquad \rho' \vdash e \rightarrow_{\mathcal{B}} p_e.\mathrm{true} + (1-p_e).\mathrm{false}}{S, \rho \vdash (\mu \,/\, e)(C) \hookrightarrow_{n_1,n_2} \{\langle l, \rho', p\, p_e\rangle \mid \langle l, \rho', p\rangle \in E\}} \tag{11}$$

Règ. 15: Sémantique probabiliste : filtrage de chemin.

- p_μ est la probabilité que toute garde de μ soit vraie pendant le filtrage de μ.

Le système de règles 15 donne la nouvelle sémantique du filtrage de chemin. Ces règles sont décrites comme suit :

- Règle 8 : Il s'agit de l'axiome pour le filtrage de chemins. Nous définissons l'ensemble E des cellules σ de dimension n, pouvant être filtrées par le motif joker et n'appartenant pas à S. A partir de chacune des cellules, le triplet $\langle [\sigma], \rho, 1\rangle$ est construit ; l'environnement reste inchangé, et toutes les gardes (il n'y en a pas) sont vérifiées.
- Règle 9 : Il s'agit de la poursuite. Nous cherchons à déterminer l'ensemble des chemins filtrés par le motif μ_1 , μ_2. Dans un premier temps, le filtrage de μ_1 produit l'ensemble de chemins filtrables $\{\langle l_i, \rho_i, p_i\rangle\}$. Le filtrage de μ_2 retourne l'ensemble de chemins E_i lorsque les positions du chemin l_i sont invalidées dans S par $S \sqcup l_i$. Notons néanmoins que tous les $\langle n, p\rangle$-chemins de l'ensemble E_i ne conviennent pas pour prolonger le $\langle n, p\rangle$-chemin l_i : le dernier élément de l_i doit être $\langle n, p\rangle$-voisin du premier élément des chemins filtrés par μ_2. Cette restriction de E_i est donnée par l'opérateur $\bullet_{n,p}$:

  $$\langle l_i, \rho_i, p_i\rangle \bullet_{n,p} E_i = \{\langle l_i @ l', \rho', p_i p'\rangle \mid \langle l', \rho', p'\rangle \in E_i \wedge (\text{last } l_i) ,^n_p (\text{hd } l')\}$$

  Cette opération concatène les chemins l' de E_i respectant la condition de $\langle n, p\rangle$-voisinage à l_i par $l_i @ l'$. L'environnement à l'issue du filtrage de ce chemin est le même que l'environnement à l'issue du filtrage de l', c'est-à-dire ρ'. Finalement, la probabilité que les gardes de μ_1 et μ_2 soient vraies ensemble est donnée en multipliant p_i par p'. Notons que cette multiplication convient car les gardes sont indépendantes entre elles. Le langage mini-MGS est fonctionnel pur : en l'absence d'effet de bord, le calcul d'une garde de μ_1 n'affecte en rien l'environnement dans lequel les gardes de μ_2 sont évaluées.
- Règle 10 : cette règle permet de lier un nom de variable à un chemin filtré. Le $\langle n, p\rangle$-voisinage est remplacé par un $\langle n_1, n_2\rangle$-voisinage pour éviter la confusion avec la probabilité p.
- Règle 11 : l'évaluation d'une garde produit la densité de probabilité $p_e.\text{true} + (1 - p_e).\text{false}$. Ainsi, pour tout chemin l filtré par μ, la probabilité que les gardes soient toutes vérifiées doit être multipliée par p_e, ce qui est fait par le calcul $p\,p_e$ où p est la probabilité que les gardes de μ soient vérifiées au cours du filtrage de l.

3.3.3 Application d'une règle

Lors de la définition des relations nous avons vu que le filtrage d'un motif μ générait un ensemble de chemins de la forme :

$$\{\langle l_\mu, \rho', p_\mu\rangle \mid l_\mu \text{ est filtré par } \mu\}$$

À partir de cet ensemble, nous allons calculer la densité de probabilité suivante :

$$\sum p_i.\{\langle l_\mu, \rho'\rangle \mid l_\mu \text{ est filtré par } \mu\}$$

Illustrons la démarche par un exemple. Supposons que le filtrage du motif μ retourne l'ensemble :

$$\{\ \langle l_1, \rho_1, \frac{1}{3}\rangle, \langle l_2, \rho_2, \frac{1}{2}\rangle\ \}$$

Deux chemins peuvent être filtrés l_1 et l_2. Chacun d'eux voit ses gardes vérifiées avec une certaine probabilité ($\frac{1}{2}$ pour l_1, et $\frac{1}{3}$ pour l_2). Quatre possibilités sont à considérer :

$$\frac{\emptyset, \rho \vdash \mu(C) \hookrightarrow_{n,p} \{\langle l_i, \rho'_i, p_i\rangle\} \qquad \rho'_i \vdash e \rightarrow_{\text{SEQ(Val)}} P_i}{\rho \vdash (\mu =\{\_\}\Rightarrow e)(C) \rightarrow_{n,p} \texttt{Den}(\bigcup\{\langle l_i, P_i, p_i\rangle\})} \quad (12)$$

RÈG. 16: Sémantique probabiliste : application d'une règle de réécriture mini-MGS.

1. les deux chemins conviennent : il faut donc que les gardes de l'un **et** de l'autre soient vérifiées, ce qui est le cas avec une probabilité $\frac{1}{2}\frac{1}{3} = \frac{1}{6}$ (les gardes sont considérées comme des évènements indépendants pour les mêmes raisons que celles évoquées ci-dessus) ;
2. le chemin l_1 filtre mais pas l_2 : les gardes de l_1 sont vérifiées et celles de l_2 ne le sont pas avec une probabilité $\frac{1}{2}(1-\frac{1}{3}) = \frac{1}{3}$;
3. le chemin l_2 filtre mais pas l_1 : les gardes de l_2 sont vérifiées et celles de l_1 ne le sont pas avec une probabilité $(1-\frac{1}{2})\frac{1}{3} = \frac{1}{6}$;
4. aucun chemin ne filtre : les gardes ne sont pas vérifiées avec une probabilité $(1-\frac{1}{2})(1-\frac{1}{3}) = \frac{1}{3}$.

La densité de probabilité qu'on souhaite calculer est donc :

$$\frac{1}{6}.\{\langle l_1, \rho_1\rangle, \langle l_2, \rho_2\rangle\} + \frac{1}{3}.\{\langle l_1, \rho_1\rangle\} + \frac{1}{6}.\{\langle l_2, \rho_2\rangle\} + \frac{1}{3}.\emptyset$$

Supposons maintenant que l'expression e donnée en partie droite s'évalue en la densité de probabilité P_{e1} (resp. P_{e2}) dans l'environnement ρ_1 (resp. ρ_2). L'application de la règle μ=>e produit la densité de probabilité suivante :

$$\frac{1}{6}.\{\langle l_1, P_{e1}\rangle, \langle l_2, P_{e2}\rangle\} + \frac{1}{3}.\{\langle l_1, P_{e1}\rangle\} + \frac{1}{6}.\{\langle l_2, P_{e2}\rangle\} + \frac{1}{3}.\emptyset$$

La fonction `Den` permet de calculer la densité désirée. Elle est définie par induction sur l'ensemble des chemins filtrables :

$$\begin{aligned} \texttt{Den}(\emptyset) &= 1.\emptyset \\ \texttt{Den}(\{\langle l, P, p\rangle\} \cup E) &= p\sum p_i.(\{\langle l, P\rangle\} \cup L_i) + (1-p)\sum p_i.L_i \end{aligned}$$

où $\texttt{Den}(E) = \sum p_i.L_i$. Dans le cas où il n'y a aucun chemin qui puisse être filtré, la densité retournée associe la probabilité 1 à l'ensemble vide. Supposons maintenant que l'ensemble des chemins filtrés soit composé d'au moins un chemin $\langle l, P, p\rangle$ et du reste des chemins E. Un appel récursif de la fonction `Den` sur E produit la densité $\sum p_i.L_i$ correspondant à l'ensemble E. Pour chaque ensemble de chemins L_i, le couple $\langle l, P\rangle$ est ajouté avec une probabilité p dans L_i. D'autre part, L_i ne contiendra pas le chemin l avec une probabilité $(1-p)$.

Cette fonction est directement utilisée pour définir l'unique règle (voir la règle 12) gérant l'application d'une règle de réécriture mini-MGS.

3.4 La stratégie StS

L'application d'un ensemble de règles définit la stratégie d'application que l'on souhaite utiliser. Le système de règles 17 donne la sémantique de la stratégie stochastique standard.

$$\frac{\rho \vdash e'_i \rightarrow_{\mathbb{F}} 1.f_i \qquad \rho \vdash \mu_i = \{e'_i\} \Rightarrow e_i(C^{\mathcal{K}}) \rightharpoonup_{n_1,n_2} \sum p^i_j . \overbrace{\{ \langle l^{ij}_k, \sum q^{ijk}_l . l'^{ijk}_l \rangle \}}^{E^i_j} \qquad C'^{ijkl} = l^{ij}_k . l'^{ijk}_l + C^{\mathcal{K}} - C^{\mathcal{K}}{\downarrow}(\mathcal{K} - (S^{\mathcal{K}} - l^{ij}_k))}{\rho \vdash \{\ldots; \mu_i = \{e'_i\} \Rightarrow e_i; \ldots\}(C^{\mathcal{K}}) \rightsquigarrow_{n_1,n_2} \sum \frac{\widetilde{f}_i\, p^i_j\, q^{ijk}_l}{|E^i_j|} . C'^{ijkl} + (1 - \sum \frac{\widetilde{f}_i\, p^i_j\, q^{ijk}_l}{|E^i_j|}) . C^{\mathcal{K}}} \quad (13)$$

RÈG. 17: Sémantique probabiliste : stratégie StS.

Pour cette stratégie *asynchrone* (une seule règle est appliquée une seule fois lorsque cela est possible), chaque règle de réécriture fournit un paramètre supplémentaire (dénoté e'_i pour la i^{e} règle) évaluant un flottant. Nous appelons ce flottant le *poids* de la règle. Par exemple, pour l'ensemble

$$\{\, \mu_1 = \{\texttt{2.0}\} \Rightarrow e_1, \mu_2 = \{\texttt{3.0}\} \Rightarrow e_2 \,\}$$

la première (resp. la seconde) règle a un poids de 2 (resp. de 3). Ces poids sont normalisés afin d'associer à chaque règle la probabilité de son application. On définit pour cela le poids de l'ensemble de règles comme la somme des poids de chaque règle ; la probabilité des règles est alors obtenue en divisant leur poids par le poids de l'ensemble. Dans notre exemple, la probabilité d'application de la première règle (resp. la seconde règle) est $\frac{2}{2+3} = \frac{2}{5}$ (resp. $\frac{3}{5}$). Par cette normalisation, nous vérifions que la somme des probabilités d'application des règles est égale à 1. Cette propriété est capitale lorsqu'on manipule des densités de probabilité.

3.4.1 Les règles sémantiques pour StS

La stratégie StS est donnée par la règle 13. Considérons l'application de la i^{e} règle de réécriture. L'évaluation de e'_i retourne le poids de la règle de réécriture. Pour simplifier notre description, nous supposons que les paramètres des règles sont des expressions déterministes. Aussi, l'expression e'_i évalue le poids f_i de la i^{e} règle. La probabilité $\widetilde{f}_i$ de la règle i est donnée par

$$\widetilde{f}_i = \frac{f_i}{\sum_j f_j}$$

comme décrit ci-dessus.

Le choix de la i^{e} règle étant effectué, la densité sur les ensembles de chemins filtrés par μ_i est calculée. Cette densité est dénotée par

$$\sum p^i_j . \{\, \langle l^{ij}_1, \sum q^{ij1}_l . l'^{ij1}_l \rangle, \langle l^{ij}_2, \sum q^{ij2}_l . l'^{ij2}_l \rangle, \ldots \} = \sum p^i_j . \{\, \langle l^{ij}_k, \sum q^{ijk}_l . l'^{ijk}_l \rangle \,\}$$

L'ensemble $\{\, \langle l^{ij}_k, \sum q^{ijk}_l . l'^{ijk}_l \rangle \,\}$ est noté E^i_j. Chaque chemin de l'ensemble E^i_j a la même probabilité d'être filtré. Ainsi, en notant $|E^i_j|$ le cardinal de cet ensemble, chaque chemin a la probabilité $\frac{1}{|E^i_j|}$ d'être transformé.

Finalement, une fois la règle i choisie ainsi que l'un des couples (collection filtrée l^{ij}_k, collection calculée l'^{ijk}_l) de l'ensemble E^i_j (nous nous plaçons donc dans le contexte où les indices i, j, k et

l sont fixés), la collection résultat C'^{ijkl} est calculée par :

$$C'^{ijkl} = l_k^{ij}.l_l'^{ijk} + C^{\mathcal{K}} - C^{\mathcal{K}}{\downarrow}(\mathcal{K} - (S^{\mathcal{K}} - (S \sqcup l_k^{ij})))$$

On reconnaît ici le calcul d'une collection dans laquelle les valeurs décorant les cellules sélectionnées par le chemin l_k^{ij} sont annulées, puis substituées par les valeurs de la séquence $l_l'^{ijk}$ issue de l'évaluation de la partie droite.

Finalement, la probabilité que le résultat final de l'application soit C'^{ijkl} est donnée par

$$\frac{\widetilde{f}_i\, p_j^i\, q_l^{ijk}}{|E_j^i|}$$

où les probabilités fixant chaque indice sont multipliées les unes avec les autres.

Pour finir, on remarque que ces probabilités ne concernent que les cas où une des règles de réécriture peut être appliquée. Le cas où E_j^i est vide n'est pas traité. En fait, si aucune règle ne peut être appliquée la collection retournée est simplement $C^{\mathcal{K}}$. C'est pourquoi la densité de probabilité fait intervenir la collection $C^{\mathcal{K}}$ avec la probabilité

$$1 - \sum \frac{\widetilde{f}_i\, p_j^i\, q_l^{ijk}}{|E_j^i|}$$

3.4.2 Exemple d'utilisation de la stratégie stochastique.

Pour illustrer cette stratégie, nous reprenons l'exemple de transformation de chemins développé dans le chapitre précédent (voir page 4.2). Nous étudions la transformation

```
trans<n,p>{
  _ as x ={0.25}=> [10*x] ;
  _ as x ={0.75}=> [-1*x]
}
```

Les règles sont nommées respectivement γ_1 et γ_2. Cette transformation est appliquée sur la collection $C^{\mathcal{K}}$ définie par :

$$\begin{array}{lcl} \mathcal{K} & = & (\{\sigma_1, \sigma_2\}, \emptyset, \{\sigma_1 \mapsto 0, \sigma_2 \mapsto 0\}) \\ C^{\mathcal{K}} & = & 1.\sigma_1 + 2.\sigma_2 \end{array}$$

Soit ρ_0, l'environnement vide. Nous cherchons à développer l'arbre de preuve permettant de calculer la densité de collections topologiques $P_{C'}$ telle que :

$$\rho_0 \ \vdash\ \{\gamma_1 ; \gamma_2\}(C^{\mathcal{K}}) \ \rightsquigarrow_{0,1}\ P_{C'}$$

Pour cela, nous commençons par appliquer la règle 13 puis la règle 12 pour calculer l'ensemble des chemins filtrés par les règles μ_1 et μ_2 de γ_1 et γ_2 :

$$\dfrac{\rho_0 \vdash \texttt{0.25} \rightarrow \frac{1}{4} \qquad \rho_0 \vdash \texttt{0.75} \rightarrow \frac{3}{4} \qquad \dfrac{\dfrac{A}{\emptyset, \rho_0 \vdash \mu_i(C^{\mathcal{K}}) \hookrightarrow_{0,1} F_i} \qquad \rho_j^i \vdash e_i \rightarrow P_e^{ij}}{\rho_0 \vdash \gamma_i(C^{\mathcal{K}}) \rightarrow_{0,1} D_i}\ 12}{\rho_0 \vdash \{\gamma_1 ; \gamma_2\}(C^{\mathcal{K}}) \rightsquigarrow_{0,1} P_{C'}}\ 13$$

où

- D_1 (resp. D_2) est la densité des ensembles de couples (collection filtrée, densité de collections calculées) de la règle γ_1 (resp. γ_2), et
- F_1 (resp. F_2) est l'ensemble des chemins filtrés par μ_1 (resp. μ_2).

Ainsi, la densité D_i est calculée en appliquant la fonction `Den` sur l'ensemble de chemin F_i où les environnements de filtrage ρ_j^i sont substitués par la densité P_e^{ij} (correspondant à l'évaluation de la partie droite e_i de la règle γ_i dans l'environnement ρ_j^i).

Dans le paragraphe suivant, nous développons le sous-arbre A. Notons que les motifs μ_1 et μ_2 sont les mêmes ; nous factorisons donc les arbres concernant les deux règles γ_1 et γ_2 de telle sorte que $F_1 = F_2$.

Sous-arbre A. Nous développons l'arbre A en utilisant les règles 8 et 10 :

$$\dfrac{\dfrac{}{\emptyset, \rho_0 \ \vdash \ \texttt{(\_)}(C^{\mathcal{K}}) \ \hookrightarrow_{0,1} \ \{\langle [\sigma_1], \rho_0, 1\rangle, \langle [\sigma_2], \rho_0, 1\rangle\}}\ {\scriptstyle 8}}{\emptyset, \rho_0 \ \vdash \ \texttt{(\_ as x)}(C^{\mathcal{K}}) \ \hookrightarrow_{0,1} \ \{\langle [\sigma_1], \rho_1, 1\rangle, \langle [\sigma_2], \rho_2, 1\rangle\}}\ {\scriptstyle 10}$$

où les environnements ρ_1 et ρ_2 sont définis par :

$$\begin{array}{rcl} \rho_1 & = & \rho_0 \uplus [\texttt{\^{}x} \mapsto \sigma_1, \texttt{x} \mapsto 1] \\ \rho_2 & = & \rho_0 \uplus [\texttt{\^{}x} \mapsto \sigma_2, \texttt{x} \mapsto 2] \end{array}$$

Ainsi, l'ensemble F des chemins filtrés vaut :

$$F = F_1 = F_2 = \{\langle [\sigma_1], \rho_1, 1\rangle, \langle [\sigma_2], \rho_2, 1\rangle\}$$

Parties droites. Dans l'ensemble F, les environnements ρ_1 et ρ_2 sont substitués par les densités P_e^{ij} issues de l'évaluation des parties droites de règle e_1 et e_2. Dans notre exemple, l'évaluation de ces expressions est déterministe ; les densités sont donc composées d'une seule valeur dont la probabilité est 1. S'agissant d'une transformation de chemins, ces valeurs sont des séquences :

$$\begin{array}{rcl} P_e^{11} & = & 1.[10] \\ P_e^{12} & = & 1.[20] \\ P_e^{21} & = & 1.[-1] \\ P_e^{22} & = & 1.[-2] \end{array}$$

Ainsi, F_1 devient $\{\langle [\sigma_1], 1.[10], 1\rangle, \langle [\sigma_2], 1.[20], 1\rangle\}$ et F_2 devient $\{\langle [\sigma_1], 1.[-1], 1\rangle, \langle [\sigma_2], 1.[-2], 1\rangle\}$.

Calculs des densités D_i. Comme nous l'avons dit plus haut, les densités D_i sont calculées en appliquant la fonction `Den` sur les ensembles que nous venons de définir :

$$\begin{array}{rcl} D_1 & = & \texttt{Den}(\{\langle [\sigma_1], 1.[10], 1\rangle, \langle [\sigma_2], 1.[20], 1\rangle\}) \\ & = & 1.\{\langle [\sigma_1], 1.[10]\rangle, \langle [\sigma_2], 1.[20]\rangle\} \\ D_2 & = & \texttt{Den}(\{\langle [\sigma_1], 1.[-1], 1\rangle, \langle [\sigma_2], 1.[-2], 1\rangle\}) \\ & = & 1.\{\langle [\sigma_1], 1.[-1]\rangle, \langle [\sigma_2], 1.[-2]\rangle\} \end{array}$$

Calculs des collections résultats. Les densités D_1 et D_2 contiennent les informations suffisantes pour construire les collections topologiques apparaissant dans la densité finale $P_{C'}$. Ces collections correspondent à l'application d'une des deux règles en considérant une substitution particulière. Dans notre exemple, quatre collections topologiques sont construites :

$$\begin{array}{lcl} C'_{\gamma_1,\sigma_1} & = & 10.\sigma_1 + 2.\sigma_2 \\ C'_{\gamma_1,\sigma_2} & = & 1.\sigma_1 + 20.\sigma_2 \\ C'_{\gamma_2,\sigma_1} & = & -1.\sigma_1 + 2.\sigma_2 \\ C'_{\gamma_2,\sigma_2} & = & 1.\sigma_1 + (-2).\sigma_2 \end{array}$$

La collection C'_{γ_i,σ_j} signifie que la règle appliquée est γ_i et que la cellule filtrée est σ_j.

Calcul du résultat $P_{C'}$. Pour finaliser le calcul de $P_{C'}$, il suffit de calculer les probabilités associées aux collections C'_{γ_i,σ_j}. La règle 13 donne pour calculer les probabilités la formule :

$$\frac{\widetilde{f_i}\, p_j^i\, q_l^{ijk}}{|E_j^i|}$$

Dans notre exemple, il n'y a qu'un ensemble de chemins par règle ($p_j^i = 1$), il n'y a qu'une seule valeur évaluée par partie droite de règles ($q_l^{ijk} = 1$), le poids normalisé de chaque règle est donné directement par le paramètre de la règle ($\widetilde{f_1} = f_1 = \frac{1}{4}$ et $\widetilde{f_2} = f_2 = \frac{3}{4}$), et finalement, les ensembles de chemins filtrables ont un cardinal égal à 2 ($|E_j^i| = 2$). Il en résulte la valeur de $P_{C'}$:

$$P_{C'} = \frac{1}{8}.(10.\sigma_1 + 2.\sigma_2) + \frac{1}{8}.(1.\sigma_1 + 20.\sigma_2) + \frac{3}{8}.(-1.\sigma_1 + 2.\sigma_2) + \frac{3}{8}.(1.\sigma_1 + (-2).\sigma_2)$$

3.5 La stratégie SED

La stratégie SED est proche de la stratégie StS. Il s'agit également d'une stratégie asynchrone où l'une des règles est choisie pour être appliquée. La différence réside dans la signification du paramètre de chaque règle. Pour la stratégie StS, le paramètre caractérise la probabilité que la règle soit appliquée. Par exemple, soit a la probabilité associée à une règle μ. On suppose que la règle μ permet de filtrer h sous-collections différentes. Ainsi, la probabilité que la règle s'applique sur l'une de ces occurrences est $\frac{a}{h}$. Avec la stratégie SED, nous souhaitons paramétrer la règle par la probabilité que celle-ci s'applique sur une des occurrences. La raison provient de la modélisation des systèmes à évènements discrets où chaque évènement est caractérisé par la probabilité de se produire. Si nous supposons la même règle μ dans ce nouveau contexte, paramétrée par la probabilité c (on appelle c la *constante stochastique* de la règle, voir la section suivante), cela signifie que la probabilité que la règle μ s'applique sur l'une des h occurrences possibles est donnée par c. Il est donc possible de programmer ce comportement avec la stratégie standard stochastique en posant :

$$c = \frac{a}{h}$$

La probabilité de déclenchement de la règle vaut donc $h.c$. Ce résultat est cohérent avec les systèmes à évènements discrets : plus un évènement peut se produire (c'est-à-dire plus h est grand), plus la probabilité qu'il se produise est grande (c'est-à-dire plus $h.c$ est grand).

Le calcul du nombre h d'instances de filtrage d'une règle dépend de la collection donnée en argument et du motif de la règle. Utiliser la stratégie StS demande donc d'exprimer ce calcul explicitement dans le paramètre de chaque règle. En revanche, un moyen générique et implicite

$$\rho \vdash \mu_i = \{e'_i\} \Rightarrow e_i(C^{\mathcal{K}}) \rightharpoonup_{n_1,n_2} \sum p^i_j . \overbrace{\{ \langle l^{ij}_k , \sum q^{ijk}_l . l'^{ijk}_l \rangle \}}^{E^i_j}$$
$$\rho \vdash e'_i \rightarrow 1.c_i$$
$$a^i_j = c_i \, |E^i_j|$$
$$a_i = \sum p^i_j . (c_i \, |E^i_j|)$$
$$A^i_j = a^1 \odot_+ \ldots \odot_+ a^{i-1} \odot_+ 1.a^i_j \odot_+ a^{i+1} \odot_+ \ldots$$
$$\widetilde{a}^i_j = 1.a^i_j \odot_/ A^i_j = \sum r^{ij}_m . \widetilde{a}^{ij}_m$$
$$C'^{ijklm} = l^{ij}_k . l'^{ijk}_l + C^{\mathcal{K}} - C^{\mathcal{K}}{\downarrow}(\mathcal{K} - (S^{\mathcal{K}} - l^{ij}_k))$$

$$\rho \vdash \{\ldots ; \mu_i = \{e'_i\} \Rightarrow e_i ; \ldots\}(C^{\mathcal{K}}) \rightsquigarrow_{n_1,n_2} \sum \frac{r^{ij}_m \, \widetilde{a}^{ij}_m \, p^i_j \, q^{ijk}_l}{|E^i_j|} . C'^{ijklm} + (1 - \sum \frac{r^{ij}_m \, \widetilde{a}^{ij}_m \, p^i_j \, q^{ijk}_l}{|E^i_j|}) . C^{\mathcal{K}} \quad (14)$$

RÈG. 18: Sémantique probabiliste : stratégie SED.

de le calculer consiste à construire l'ensemble des instances de motif et à en prendre le cardinal. La stratégie SED effectue ce calcul.

La stratégie SED correspond aux mécanismes suivants :

- Pour chaque règle de réécriture, la constante stochastique donnée en paramètre de la règle est évaluée ; nous supposons que ce calcul est déterministe pour simplifier l'écriture de la règle d'inférence.
- Pour chaque règle nous évaluons la densité des ensembles de couples (collection filtrée, densité de collections calculées). Le cardinal de chacun de ces ensembles correspond au nombre de fois où la règle peut être appliquée, et par conséquent, à la variable h utilisée ci-dessus.
- Le poids a de la règle est calculé pour chacun de ces ensembles, puis normalisé ; il est finalement représenté par une densité de probabilité $\widetilde{a}^i_j$ (les itérateurs i et j fixent respectivement la règle appliquée et l'ensemble des chemins filtrables). Nous appelons cette variable la *propension de la règle* (voir la section suivante).
- Un couple particulier de l'ensemble est alors désigné comme résultat. La probabilité de ce résultat est obtenue en suivant le même principe que celui de la stratégie stochastique, où le poids normalisé $\widetilde{f}_i$ est remplacé par la propension normalisée $\widetilde{a}^i_j$ de la règle.

3.5.1 Les règles sémantiques pour SED

La règle d'inférence 14 décrit ce calcul. Soit γ_i la i^{e} règle de réécriture ; $\gamma_i = (\mu_i = \{e'_i\} \Rightarrow e_i)$. La densité de probabilité des ensembles de chemins filtrés par γ_i est donnée par

$$\sum p^i_j . E^i_j \quad \text{où} \quad E^i_j = \{ \langle l^{ij}_k , \sum q^{ijk}_l . l'^{ijk}_l \rangle \}$$

Rappelons que nous cherchons à déterminer la probabilité que l'application de la règle γ_i sur la collection $C^{\mathcal{K}}$ calcule la collection C'^{ijklm} dans la cadre de la stratégie SED ; chaque exposant fixe un des éléments du calcul :

- i : la règle γ_i,
- j : l'ensemble E^i_j des chemins filtrés par γ_i,
- k : le chemin filtré l^{ij}_k,
- l : la sous-collection calculée l'^{ijk}_l, et
- m : la propension normalisée $\widetilde{a}^{ij}_m$, dont nous allons maintenant détailler le calcul.

La constante stochastique de γ_i étant c_i, on déduit le calcul de la propension de la règle μ_i, considérant l'ensemble de chemins filtrés E^i_j

$$a^i_j = c_i \times |E^i_j|$$

où le nombre de chemins de chaque ensemble E^i_j est multiplié par la constante stochastique pour obtenir la propension. Dans le contexte où l'on se place, les itérateurs i et j sont fixés. Par conséquent, la propension de la règle γ_i est également fixée à a^i_j. En revanche, les propensions des autres règles, intervenant dans le calcul de la somme des propensions, ne sont pas fixées par i et j. Il nous faut donc manipuler directement les densités de propension des autres règles $\gamma_{i'}$, $i \neq i'$. Nous définissons alors la densité de propension d'une règle $\gamma_{i'}$ par :

$$a_{i'} = \sum p^{i'}_{j'}.a^{i'}_{j'}$$

où l'itérateur j' parcourt les ensembles $E^{i'}_{j'}$ des chemins filtrés par $\gamma_{i'}$. Nous pouvons alors évaluer la densité A^i_j de propension totale des règles par :

$$A^i_j = a_1 \odot_+ \ldots \odot_+ a_{i-1} \odot_+ 1.a^i_j \odot_+ a_{i+1} \odot_+ \ldots$$

Nous considérons dans cette somme les densités de propension des autres règles, mais nous fixons la propension de la règle γ_i à a^i_j. La densité de propension normalisée $\widetilde{a}^i_j$ de la règle γ_i pour l'ensemble E^i_j des chemins filtrés est finalement obtenue en divisant à l'aide de l'opérateur $\odot$ la propension a^i_j par la propension totale A^i_j :

$$\widetilde{a}^i_j = 1.a^i_j \odot_/ A^i_j = \sum r^{ij}_m.\widetilde{a}^{ij}_m$$

De cette densité, il ressort que la règle γ_i a une propension normalisée de $\widetilde{a}^{ij}_m$ avec la probabilité r^{ij}_m. Cette propension normalisée est finalement utilisée de façon équivalente au poids normalisé des règles dans la stratégie StS. La probabilité que l'application de la transformation s'évalue en C'^{ijklm}, est donc donnée par

$$\frac{r^{ij}_m \widetilde{a}^{ij}_m p^i_j q^{ijk}_l}{|E^i_j|}$$

3.5.2 La stratégie SED pour programmer la méthode de D.T. Gillespie

L'algorithme de D.T. Gillespie, ainsi que des exemples, sont donnés dans le chapitre 10. Il nous semble néanmoins intéressant de rapprocher ici la stratégie SED de l'algorithme de D.T. Gillespie.

La théorie mathématique (détaillée chapitre 10) explique les méthodes suivies par D.T. Gillespie pour simuler des réactions chimiques. Nous nous concentrons ici sur la méthode directe.

Soit un ensemble de réactions chimiques μ_i. Chaque réaction μ_i fournit en paramètre la constante stochastique de réaction c_i. Supposons qu'il existe dans la solution chimique h_i combinaisons moléculaires pouvant être à l'origine d'une occurrence de la réaction μ_i.

Soit la réaction $X + Y \rightarrow Z$, faisant réagir une molécule d'une espèce chimique X avec une molécule de l'espèce chimique Y pour donner une molécule d'une nouvelle espèce Z dans une solution composée de $\#X$ molécules X et $\#Y$ molécules Y. Le nombre de combinaisons moléculaires pour μ_i est $h_i = \#X.\#Y$.

À partir de c_i et de h_i, D.T. Gillespie [Gil77] définit la fonction de *propension* a_i de la réaction μ_i par

$$a_i = h_i \, c_i$$

Considérant la fonction de propension de chaque réaction, la *méthode directe* consiste à tirer au sort de façon indépendante deux réels de $[0, 1]$, r_1 et r_2, pour déterminer le temps d'inter-réaction τ (c'est-à-dire le temps qu'il faut attendre avant que la prochaine réaction ait lieu)

$$\tau = \frac{1}{\sum_{i=0}^{M} a_i} \ln \frac{1}{r_1} \tag{15}$$

(où M est le nombre de réactions chimiques μ_i différentes), et la prochaine réaction μ_i qui aura lieu, en résolvant l'équation

$$\sum_{j=1}^{i-1} a_j < r_2 \sum_{j=1}^{M} a_i \leq \sum_{j=1}^{i} a_j \tag{16}$$

On reconnaît dans l'équation 16 le tirage au sort d'une réaction μ_i parmi les M réactions possibles, avec pour chacune d'elle la probabilité

$$\frac{a_i}{\sum_{j=1}^{M} a_j}$$

Cette équation correspond aux calculs de la stratégie SED que nous venons de présenter. Les notions de constante stochastique et de propension coïncident entre l'algorithme de D.T. Gillespie et la stratégie SED lorsque celle-ci est utilisée sur un multi-ensemble représentant une solution chimique.

Malheureusement, en ce qui concerne l'équation 15, la loi de probabilité du temps d'inter-réaction τ ne peut pas être prise en compte dans notre modèle. En effet, τ peut prendre une infinité de valeurs parmi les éléments de $\mathbb{R}$. Afin de représenter des densités de probabilité sur des espaces infinis et indénombrables comme $\mathbb{R}$, il faut développer des mécanismes plus sophistiqués que ceux que nous utilisons. Néanmoins, le tirage au sort de la réaction μ_i à appliquer et le tirage au sort du temps d'inter-réaction sont indépendants. Il est tout a fait possible de considérer le tirage de la réaction sans prendre en compte τ. Cette restriction entraîne l'absence de repère temporel dans les simulations qui pourront être programmées en mini-MGS. Notre sémantique fournit l'ensemble de tous les résultats possibles pondérés par une probabilité ; dans l'implantation réelle que nous avons faite du langage MGS et de la stratégie `'gillespie` (voir les chapitres 2 et 10 pour plus de détails), un seul des résultats est calculé, mais le temps d'inter-réaction est pris en compte. Les simulations successives d'un même modèle fournissent alors un échantillon de résultats à partir duquel les lois de probabilité peuvent être étudiées.

Chapitre 6

Analogie avec les formes différentielles

Dans ce chapitre, nous adoptons un point de vue différent pour la description des transformations. Notre objectif est d'introduire une analogie forte entre le concept de transformation MGS et ceux issus du calcul différentiel. La formalisation des collections topologiques comme des chaînes topologiques (voir chapitre 3), amène naturellement vers les concepts de cochaîne et de forme différentielle discrète.

Nous commençons par une présentation des formes différentielles et de l'intégration dans le cas continu (le plus standard), puis dans le cas discret. Nous faisons ensuite le rapprochement entre forme différentielle discrète et transformation pour définir un bestiaire d'opérations standards en physique discrète naturellement traduites pour l'application des transformations. Ce rapprochement donne, de façon informelle, une sémantique dénotationnelle des transformations, amenant la mise en place d'une algèbre des collections topologiques, des transformations et de l'application de ces dernières aux premières.

1 Calcul différentiel

1.1 Généralités

Une définition informelle des formes différentielles est donnée dans [Fla63] et dans [DKT06] : une *forme différentielle* est ce qui apparaît sous le signe intégrale. En d'autres termes, une forme différentielle correspond à un objet qui peut être intégré. Par exemple, dans l'équation suivante :

$$\int x \, \mathbf{d}x$$

$x \, \mathbf{d}x$ est une forme différentielle. Elle est caractérisée par une dimension, celle de l'espace sur lequel elle peut être intégrée. Dans notre exemple, la forme différentielle est de dimension 1 (on peut par exemple l'intégrer sur un intervalle $[a, b] \subset \mathbb{R}$) ; on dira qu'il s'agit d'une 1-forme différentielle et, plus généralement, une n-forme pour une forme différentielle de dimension n.

Plus formellement, soit P un point de $\mathcal{M} \subset R^n$, une région ouverte de $\mathbb{R}^n$. Les 1-formes au point P s'écrivent :

$$\sum_{i=1}^{n} a_i \, \mathbf{d}x^i, \quad a_i \text{ des constantes}$$

Intuitivement, les quantités $\mathbf{d}x^i$ correspondent à des « déplacements infinitésimaux » suivant les n dimensions de R^n au voisinage de P. En considérant la suite (x^i) comme une base de $\mathbb{R}^n$, l'ensemble des 1-formes au point P sur $\mathcal{M}$ définit un espace vectoriel, noté $T_P\mathcal{M}$, des vecteurs tangents au point P, pouvant être identifié à $\mathbb{R}^n$ lui-même, et dont la suite $(\mathbf{d}x^i)$ est une base. Les formes différentielles au point P de dimension $n > 1$ sont construites à l'aide du *produit extérieur* $\wedge$ (à ne pas confondre avec la conjonction des formules logiques) : soient ω et η une p-forme et une q-forme, $\omega \wedge \eta$ définit une $(p+q)$-forme. Cet opérateur est l'opération principale d'une construction mathématique appelée *algèbre extérieure* [MLB71]. Le produit extérieur est une opération bilinéaire, associative telle que :

$$\omega \wedge \eta = (-1)^{pq} \eta \wedge \omega$$

La définition des formes différentielles de dimension p en un point P de $\mathcal{M}$ est alors étendue à tout $\mathcal{M}$ en choisissant pour chaque point P de $\mathcal{M}$ une p-forme et cela de façon continue sur $\mathcal{M}$. Une p-forme sur $\mathcal{M}$ peut être représentée par la somme formelle suivante :

$$\omega = \sum a_H(x^1, \ldots, x^n) \, \mathbf{d}x^{h_1} \wedge \cdots \wedge \mathbf{d}x^{h_p}$$

où chaque fonction a_H est continue (l'indice H correspond à la suite $(h_i)_{1 \leq i \leq p}$). L'ensemble des p-formes sur $\mathcal{M}$, dénoté $\bigwedge^p T_{\mathcal{M}}$, est donc l'ensemble des p-vecteurs de $T_{\mathcal{M}}$. L'espace $\bigwedge^0 T_{\mathcal{M}}$ est simplement l'ensemble des fonctions continues sur $\mathcal{M}$.

L'opérateur différentiel extérieur d. Bien que l'opérateur $\mathbf{d}$ n'ait été utilisé jusqu'ici que pour former les symboles formels $\mathbf{d}x^i$ servant de base pour la construction des formes différentielles, il s'agit en fait d'un opérateur à part entière, permettant la construction d'une $(p+1)$-forme à partir d'une p-forme. Il est appelé la *différentielle extérieure*. Dans le cas des 0-formes, nous retrouvons le calcul de la dérivée d'une fonction continue. Soit f une fonction continue, on a

$$\mathbf{d}f = \sum_i \frac{\partial f}{\partial x^i} \, \mathbf{d}x^i$$

La différentielle extérieure vérifie les propriétés suivantes :

- $\mathbf{d}(\omega + \eta) = \mathbf{d}\omega + \mathbf{d}\eta$
- $\mathbf{d}(\omega \wedge \eta) = \mathbf{d}\omega \wedge \eta + (-1)^{\dim \omega}\omega \wedge \mathbf{d}\eta$
- $\mathbf{dd}w = 0$
- $\mathbf{d}$ coïncide avec la dérivée des fonctions continues dans le cas des 0-formes

La preuve d'existence et d'unicité de la dérivée extérieure est donnée dans [Fla63].

Le *calcul extérieur* (calcul fondé sur la manipulation des formes différentielles et de la différentielle extérieure) fournit une liste d'équivalences dont l'utilité n'est plus à démontrer en physique. En effet, les opérateurs standards (gradient, divergence et rotationnel) correspondent simplement aux passages d'une p-forme à une $(p+1)$-forme. Prenons le cas de l'opérateur gradient souvent dénoté $\overrightarrow{\nabla}$. Soit f un champ scalaire défini sur un domaine $\mathcal{M}$ de $\mathbb{R}^3$, $\overrightarrow{\nabla} f$ construit un champ vectoriel sur $\mathcal{M}$ tel que

$$\overrightarrow{\nabla} f = \begin{pmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \\ \frac{\partial f}{\partial z} \end{pmatrix}$$

Ce vecteur peut être transformé en une forme différentielle (le calcul extérieur fournit pour cela les opérateurs $\sharp$ et $\flat$), retrouvant alors le calcul de la différentielle extérieure de f :

$$\mathbf{d}f = \frac{\partial f}{\partial x}\,\mathbf{d}x + \frac{\partial f}{\partial y}\,\mathbf{d}y + \frac{\partial f}{\partial z}\,\mathbf{d}z = \overrightarrow{\nabla} f.\begin{pmatrix} \mathbf{d}x \\ \mathbf{d}y \\ \mathbf{d}z \end{pmatrix}$$

En notant en exposant la dimension de la forme différentielle sur laquelle la différentielle extérieure s'applique, on établit de la même façon que les opérateurs rotationnel (noté $\overrightarrow{\nabla}\wedge$) et divergence ($\overrightarrow{\nabla}.$) correspondent exactement à $\mathbf{d}^1$ et $\mathbf{d}^2$ (le gradient correspondant à $\mathbf{d}^0$). Sachant que $\mathbf{dd}$ retourne une forme différentielle nulle, on retrouve naturellement que :

$$\begin{array}{lcl} \mathbf{d}^1\mathbf{d}^0 = 0 & \Leftrightarrow & \overrightarrow{\nabla} \wedge \overrightarrow{\nabla} = \overrightarrow{0} \quad \Leftrightarrow \quad \overrightarrow{\mathrm{rot}}(\overrightarrow{\mathrm{grad}}) = \overrightarrow{0} \\ \mathbf{d}^2\mathbf{d}^1 = 0 & \Leftrightarrow & \overrightarrow{\nabla}.\overrightarrow{\nabla}\wedge = 0 \quad \Leftrightarrow \quad \mathrm{div}(\overrightarrow{\mathrm{rot}}) = 0 \end{array}$$

D'autres équivalences sont données dans [DKT06].

L'intégration de p-formes. Les p-formes sont les quantités pouvant être intégrées sur un domaine de dimension p. Soient ω une p-forme et $\mathcal{M}$ de $\mathbb{R}^n$ un ouvert de dimension p, l'intégration de ω sur $\mathcal{M}$ s'écrit

$$\int_{\mathcal{M}} \omega$$

Le théorème de Stokes est le résultat principal de l'intégration des formes différentielles. Soit ω une p-forme, et $\mathcal{M}$ un ouvert de $\mathbb{R}^n$ de dimension $p+1$, ce théorème établit l'égalité suivante :

$$\int_{\mathcal{M}} \mathbf{d}\omega = \int_{\partial\mathcal{M}} \omega$$

où $\partial\mathcal{M}$ est le bord de $\mathcal{M}$.

En se plaçant dans le cas des fonctions continues, ce théorème se traduit comme le *théorème fondamental du calcul différentiel et intégral* liant primitive et dérivée. Soit f une fonction continue sur l'intervalle $[a, b]$, la fonction F définie sur $[a, b]$ par :

$$F(x) = \int_a^x f(t)\,\mathbf{d}t$$

est dérivable sur $[a, b]$ et sa dérivée est exactement f. F est l'unique primitive de f s'annulant en a.

L'intégration et le théorème de Stokes peuvent également être appliqués aux opérateurs physiques soulignés dans le paragraphe précédent. Différentes égalités d'intégrales sont alors mises en évidence, notamment les théorèmes de Green liant l'intégrale sur une courbe fermée à l'intégrale sur la surface délimitée par cette courbe, et de Green-Ostrogradsky (également dit de flux-divergence) liant le flux à travers une surface fermée aux quantités dans le volume délimité par cette surface qui génèrent ce flux (comme la charge électrique pour le flux électrique).

1.2 Calcul différentiel discret

Le *calcul différentiel discret* (également appelé *calcul extérieur discret*) est l'équivalent discret du calcul différentiel présenté ci-dessus. Le lecteur intéressé par ce sujet trouvera dans [DKT06, Hir03, Leo04] une description approfondie du calcul extérieur discret.

L'enjeu principal de l'équivalent discret du calcul différentiel concerne l'implantation et la simulation numérique pour la modélisation. Les méthodes d'analyse numérique sont pour la plupart fondées sur une discrétisation des modèles continus décrivant les systèmes. Ces modèles (généralement constitués d'équations aux dérivées partielles) ne peuvent être résolus symboliquement, d'autant plus qu'ils s'appliquent sur des espaces complexes. Les méthodes des différences finies et des éléments finis (standards de l'analyse numérique) proposent une discrétisation de l'espace et du temps pour l'expression des modèles, l'approximation découlant de la discrétisation étant d'autant plus élevée que la discrétisation est grossière. De plus, ces méthodes ne garantissent pas le maintien d'invariants géométriques et/ou topologiques de tous les systèmes (des exemples sont donnés dans l'introduction de [DKT06]). Pour pallier ces difficultés, de nouvelles méthodes émergent, fondées sur les résultats des travaux de recherche dans le domaine de la physique discrète. On pense en particulier à la *cell method* d'E. Tonti [Ton01] et aux travaux de J. A. Chard et V. Shapiro sur le développement de structure de données dédiées à l'implantation des formes différentielles [CS00].

Formes différentielles discrètes. Le domaine $\mathcal{M}$ sur lequel les formes différentielles décrites précédemment sont définies, est discrétisé à l'aide d'un complexe cellulaire $\mathcal{K}$. La définition des CW-complexes donnée dans la chapitre 3 répond à cette discrétisation. Dans [DKT06] et [CS00], les complexes cellulaires utilisés pour l'implantation sont respectivement des complexes simpliciaux et des *starplexes* (pour *star pseudo-complexes*).

Les formes différentielles discrètes sont définies à partir des formes différentielles continues, et des chaînes combinatoires définies sur les complexes $\mathcal{K}$ issus de la discrétisation des domaines $\mathcal{M}$. Soient ω une p-forme, $\mathcal{M}$ un ouvert de $\mathbb{R}^n$, $\mathcal{K}$ la discrétisation de $\mathcal{M}$ et c une p-chaîne de $C_p(\mathcal{K}, \mathbb{Z})$ représentant $\mathcal{K}$:

$$\int_{\mathcal{M}} \omega = \int_{\sum_\sigma c(\sigma).\sigma} \omega = \sum_{\sigma \in \mathcal{K}} c(\sigma) \int_\sigma \omega$$

Dans cette égalité, le domaine $\mathcal{M}$ étant partitionné par sa discrétisation $\mathcal{K}$, l'intégrale de la p-forme ω est linéarisé sur chaque cellule de $\mathcal{K}$ à l'aide de la chaîne c. Le résultat de ce calcul ressemble fortement à la description de c par une somme formelle, où les cellules topologiques σ sont remplacés par les intégrales $\int_\sigma \omega$:

$$\sum_{\sigma \in \mathcal{K}} c(\sigma).\sigma \quad \text{devient} \quad \sum_{\sigma \in \mathcal{K}} c(\sigma) \int_\sigma \omega$$

La forme discrète de ω sur $\mathcal{K}$ correspond alors à la fonction qui associe à chaque cellule σ de $\mathcal{K}$, la valeur de l'intégrale de ω sur σ. Cette association permet de définir formellement la p-forme discrète ω^{d} de ω :

$$\begin{array}{rccc} \omega^{\mathrm{d}} : & \mathcal{K} & \to & \mathbb{R} \\ & \sigma & \mapsto & \displaystyle\int_{\sigma} \omega \end{array}$$

qui peut également être représentée par la somme formelle suivante :

$$\omega^{\mathrm{d}} = \sum_{\sigma \in \mathcal{K}} \left(\int_{\sigma} \omega \right) . \sigma$$

L'intuition pousse à définir ω^{d} comme une p-chaîne ; néanmoins, il n'apparaît pas que ω^{d} est nulle presque partout, condition vérifiée par les chaînes combinatoires. En fait, ω^{d} appartient à l'espace *dual*, l'espace des *cochaînes*. Il s'agit de l'ensemble des homomorphismes de $C_p(\mathcal{K}, \mathbb{Z})$ dans $\mathbb{R}$, dénoté $C^p(\mathcal{K}, \mathbb{Z}, \mathbb{R})$. Soient ω une p-cochaîne et c une p-chaîne, l'application $\omega(c)$ est notée $[\omega, c]$. La forme $[,]$ est une forme bilinéaire.

Pour retrouver les résultats précédents sur les formes différentielles, la différentielle extérieure discrète est définie de telle sorte que la version discrète du théorème de Stokes est vérifiée :

$$[\mathbf{d}\omega, c] = [\omega, \partial c]$$

Nous arrêtons ici la présentation du calcul différentiel, les principaux concepts abordés étant suffisants pour comprendre l'analogie que nous souhaitons mettre en place. Pour plus de détails à propos du calcul extérieur discret, le lecteur est invité à se référer à la thèse d'A. Hirani [Hir03], fournissant un texte de référence sur ce sujet.

2 Formalisation

Dans cette section, nous formalisons les concepts issus de la présentation du calcul différentiel (discret) en vue de les appliquer aux transformations MGS.

2.1 Les cochaînes

La présentation du calcul différentiel discret que nous venons de faire n'est pas exhaustive. Néanmoins, elle nous amène à considérer la notion de cochaîne. Dans le cadre de la physique discrète, ces objets sont restreints au domaine d'application de la modélisation des systèmes physiques, manipulant des ensembles tels que $\mathbb{Z}$ et $\mathbb{R}^n$. Avec les définitions qui suivent, nous cherchons à compléter les définitions du chapitre 3 avec les concepts apportés par le calcul différentiel discret, en les étendant à des groupes arbitraires.

Définition 39 (Cochaîne topologique, p-cochaîne) *Soient $\mathcal{K}$ un complexe cellulaire abstrait de dimension n, un entier $p \leq n$, G et G' deux groupes abéliens. L'ensemble $C^p(\mathcal{K}, G, G')$ dénote l'ensemble des homomorphismes de $C_p(\mathcal{K}, G)$ dans G' appelés les* p-cochaînes *sur $C_p(\mathcal{K}, G)$ à valeur dans G'. Pour la différencier de l'application des chaînes, l'application des cochaînes est notée par la forme $[\_, \_]$. Soient $T \in C^p(\mathcal{K}, G, G')$ et $c \in C_p(\mathcal{K}, G)$ l'application de T sur c s'écrit $[T, c]$.*

On définit l'addition $+_{C^p(\mathcal{K},G,G')}$ entre deux cochaînes T_1 et T_2 de $C^p(\mathcal{K}, G, G')$, l'opération qui construit une nouvelle cochaîne de $C^p(\mathcal{K}, G, G')$ telle que, $\forall c \in C_p(\mathcal{K}, G)$:

$$\left[T_1 +_{C^p(\mathcal{K},G,G')} T_2, c\right] = [T_1, c] +_{G'} [T_2, c]$$

L'addition de cochaînes permet d'établir le théorème suivant :

Théorème 7 *L'ensemble $C^p(\mathcal{K}, G, G')$ muni de la loi $+_{C^p(\mathcal{K},G,G')}$ est un groupe abélien, et la forme $[,]$ est bilinéaire.*

Nous utilisons alors cette propriété pour représenter les cochaînes par des sommes formelles. Soient $T \in C^p(\mathcal{K}, G, G')$ et $c = \sum g.\sigma \in C_p(\mathcal{K}, G)$:

$$\begin{aligned}[T, c] &= \left[T, \sum_{\sigma \in \mathcal{K}_p} g.\sigma\right] \\ &= \sum_{\sigma \in \mathcal{K}_p} [T, g.\sigma] \\ &= \sum_{\sigma \in \mathcal{K}_p} T_\sigma(g)\end{aligned}$$

où nous posons $T_\sigma(g) = [T, g.\sigma]$. Dans la dernière expression, seule la variable g dépend de la collection c : elle correspond à la valeur associée à σ par c. En substituant g par la cellule σ elle-même, nous obtenons une nouvelle expression indépendante de c et qui caractérise complètement la cochaîne T. Nous dénotons alors la cochaîne T par :

$$T = \sum_{\tau \in \mathcal{K}_p} T_\tau.\tau$$

Cette représentation n'est pas sans rappeler la décomposition de T sur chacune des cellules de $\mathcal{K}_p$. Il est facile d'en déduire que chaque fonction T_τ est un homomorphisme. Nous utilisons la variable τ plutôt que σ pour dénoter qu'il s'agit d'une cochaîne. Nous utilisons cette convention dans la suite du chapitre : la variable σ représente les cellules pour la définition des chaînes, et la variable τ représente les cellules pour les cochaînes. Notons néanmoins que l'application de la fonction T_τ est faite uniquement sur la valeur associée à cette même cellule dans la collection :

$$[T_\tau.\tau, g_\sigma.\sigma] = \begin{cases} T_\sigma(g_\sigma) & \text{si } \tau = \sigma \\ 0'_G & \text{sinon} \end{cases}$$

Il arrivera donc souvent dans les calculs que lors de l'application d'une cochaîne sur une chaîne, les sommes itérant d'une part sur σ et d'autre part sur τ soient transformées en une unique somme sur σ, les termes où σ et τ diffèrent s'annulant. Par exemple

$$\left[\sum_\tau T_\tau.\tau, \sum_\sigma g_\sigma.\sigma\right] = \sum_{\tau\sigma} [T_\tau.\tau, g_\sigma.\sigma] = \sum_\sigma T_\sigma(g_\sigma)$$

Les chaînes représentées par des cochaînes. Dans ce paragraphe, nous remarquons que toute p-chaîne de $C_p(\mathcal{K}, G)$ peut être représentée par une p-cochaîne équivalente de $C^p(\mathcal{K}, \mathbb{Z}, G)$. Le point de départ de cette équivalence provient de la représentation d'une cellule σ de $\mathcal{K}$ par la chaîne canonique $1.\sigma$ de $C_p(\mathcal{K}, G)$.

Soit c, une p-chaîne de $C_p(\mathcal{K}, G)$. Elle associe à chaque cellule σ de $\mathcal{K}$ une valeur $g \in G$ telle que $c(\sigma) = g$. Pour représenter la chaîne c par une cochaîne équivalente T de $C^p(\mathcal{K}, \mathbb{Z}, G)$, nous posons

$$[T, 1.\sigma] = c(\sigma)$$

Par conséquent, l'homomorphisme $T_\sigma : \mathbb{Z} \to G$ associé par la cochaîne T à la cellule σ, doit vérifier $T_\sigma(1) = g$. De part sa nature d'homomorphisme, T_σ doit être linéaire. La seule définition possible est

$$\begin{array}{rccc} T_\sigma & : & \mathbb{Z} & \longrightarrow & G \\ & & n & \longmapsto & n.c(\sigma) \end{array}$$

où l'expression $n.c(\sigma)$ (que l'on pourra également écrire $nc(\sigma)$ pour simplifier les équations) correspond à la n-somme $c(\sigma) +_G \cdots +_G c(\sigma)$. Ainsi, pour toute chaîne c, il existe une unique cochaîne T équivalente définie par :

$$T = \sum_\tau T_\tau.\tau = \sum_\tau (n \mapsto n.c(\tau)).\tau$$

2.2 Les complexes de cochaînes

Les p-cochaînes sont la représentation discrète des p-formes. Dans cette sous-section, nous définissons la différentielle extérieure discrète, fondée sur le théorème de Stokes. Pour cela, nous commençons par observer l'opérateur de bord des complexes de chaînes comme une cochaîne.

2.2.1 Opérateur de bord

Un complexe de chaîne $C(\mathcal{K}, G, \partial)$ définit une suite d'homomorphismes ∂_p tels que $\partial_p \in \mathrm{Hom}(C_p(\mathcal{K}, G), C_{p-1}(\mathcal{K}, G))$. Par définition, ces opérateurs de bord sont des cochaînes envoyant les p-chaînes de $C_p(\mathcal{K}, G)$ dans le groupe abélien $C_{p-1}(\mathcal{K}, G)$. Ainsi, $\forall p$,

$$\partial_p \in C^p(\mathcal{K}, G, C_{p-1}(\mathcal{K}, G))$$

À partir de la représentation d'une cochaîne par une somme formelle, on déduit qu'il existe une suite d'homomorphismes $\partial_p^\tau \in \mathrm{Hom}(G, C_{p-1}(\mathcal{K}, G))$ tels que :

$$\partial_p = \sum_{\tau \in \mathcal{K}_p} \partial_p^\tau.\tau$$

En tant que fonctions retournant des $(p-1)$-chaînes, les homomorphismes ∂_p^τ peuvent être développés. Ainsi, $\forall g \in G$, $\forall \sigma^p \in \mathcal{K}_p$:

$$\partial_p^\tau(g) = \sum_{\sigma \in \mathcal{K}_{p-1}} g_{\tau\sigma}.\sigma \qquad \text{où } g_{\tau\sigma} \in G$$

Nous appelons les fonctions envoyant g sur $g_{\tau\sigma}$ les fonctions d'*orientation* associées à l'opérateur de bord d'un complexe $C(\mathcal{K}, G, \partial)$ que nous notons $\mathrm{o}_{\tau\sigma}$. Soit la suite de fonctions d'orientation $(\mathrm{o}_{\sigma\tau})_{\sigma \in \mathcal{K}_p, \tau \in \mathcal{K}_{p-1}}$ de G dans G, on a $\forall g \in G, \sigma \in \mathcal{K}_p$

$$\mathrm{o}_{\sigma\tau}(g) = g_{\sigma\tau}$$

Il est facile de vérifier que ces fonctions sont des homomorphismes de G dans G. De façon générale, l'opérateur de bord et le concept de chaîne apporte une information supplémentaire sur l'organisation des cellules d'un complexe cellulaire abstrait que la relation d'incidence seule ne permet pas d'exprimer (voir le chapitre 3, page 75). Nous ne considérons donc que des fonctions d'orientation telles que $\mathrm{o}_{\sigma\tau}$ n'est pas nul que si σ et τ sont incidentes : $\tau < \sigma$.

Ainsi, le calcul du bord d'une p-chaîne est entièrement déterminé par la suite des fonctions d'orientation. Soit $C(\mathcal{K}, G, \partial)$ un complexe de chaînes et c une p-chaîne, on a

$$\partial(c) = \sum_{\tau \in \mathcal{K}_{p-1}} \left(\sum_{\sigma \in \mathcal{K}_p} \mathrm{o}_{\sigma\tau}(c(\sigma)) \right) .\tau$$

Le nom de fonction « d'orientation » provient de la notion d'*orientation relative*. Celle-ci est utilisée dans [Ale82] pour définir l'opérateur de bord sur des complexes simpliciaux. Dans ce document, en supposant $\mathcal{K}$ un complexe simplicial, les p-chaînes sont des éléments de $C_p(\mathcal{K}, \mathbb{Z})$ et l'opérateur de bord ∂ est défini pour chaque cellule topologique de $\sigma \in \mathcal{K}_p$ de telle sorte que :

$$\partial\sigma = \sum_{\tau \in \mathcal{K}_{p-1}} t_\tau^\sigma .\sigma$$

où t_τ^σ vaut :

- 0 si σ et τ ne sont pas incidents,
- 1 si τ est une face de σ et que les deux cellules sont de même orientation,
- -1 si τ est une face de σ et que les deux cellules sont d'orientations opposées.

L'orientation d'une cellule étant arbitraire, la donnée de cet entier pour chaque couple (σ, τ) définit entièrement l'orientation des cellules pourvu que l'on fixe l'orientation de l'une d'entre elles. Ne se restreignant pas au groupe $\mathbb{Z}$ (bien que dans la pratique, cet ensemble soit majoritairement utilisé), nous avons étendu cet entier à un homomorphisme quelconque $\mathrm{o}_{\sigma\tau}$, qui dans le cas des complexes simpliciaux que nous venons de décrire est $\mathrm{o}_{\sigma\tau}(n) = t_\tau^\sigma .n$.

Notons que l'écriture de l'application de l'opérateur de bord ne respecte pas les notations définies plus haut à propos de l'application des cochaînes. Cependant, pour en faciliter les calculs nous utilisons les deux notations

$$\partial(c) \equiv [\partial, c]$$

En mélangeant les deux notations, nous obtenons par exemple que, pour toute cochaîne T, $[T \circ \partial, c]$ est équivalent à $[T, \partial(c)]$.

2.2.2 Différentielle extérieure discrète et théorème de Stokes

Les homomorphismes ∂_p d'un complexe de chaînes $C(\mathcal{K}, G, \partial)$ donnent naissance à une suite d'opérateurs duaux sur l'ensemble des cochaînes. Cette notion de dualité est la généralisation directe de la dualité des espaces vectoriels : intuitivement, si les chaînes sont considérées comme des « vecteurs », les cochaînes sont des « formes linéaires ». En approfondissant cette analogie, la notion d'homomorphisme dual peut également être ramenée au cas des groupes. Nous notons $\mathrm{Hom}(A, B)$, l'ensemble des homomorphismes de groupes de A dans B.

Définition 40 (Homomorphisme dual) *Un homomorphisme de groupe $\theta \in \mathrm{Hom}(A, B)$ produit un* homomorphisme dual $\widetilde{\theta}$

$$\mathrm{Hom}(A, G) \xleftarrow{\widetilde{\theta}} \mathrm{Hom}(B, G)$$

de direction opposée, défini par $\widetilde{\theta}(\phi) = \phi \circ \theta$

La définition du dual de l'opérateur ∂ en découle directement

Définition 41 (Opérateur de bord dual) *Soit $C(\mathcal{K}, G, \partial)$ un complexe de chaînes. Pour chaque homomorphisme ∂_p de $C_p(\mathcal{K}, G)$ dans $C_{p-1}(\mathcal{K}, G)$, nous définissons l'*opérateur de bord dual *$\widetilde{\partial}_p$, comme l'homomorphisme dual de ∂_p de $C^{p-1}(\mathcal{K}, G, G')$ dans $C^p(\mathcal{K}, G, G')$.*

Ainsi, pour toute cochaîne T de $C^{p-1}(\mathcal{K}, G, G')$ et pour toute chaîne c de $C_p(\mathcal{K}, G)$, on a :

$$\left[\widetilde{\partial}_p T, c\right] = [T \circ \partial_p, c] = [T, \partial_p(c)]$$

Cette équation correspond au théorème de Stokes discret. Nous identifions alors l'opérateur $\widetilde{\partial}_p$ à la différentielle extérieure discrète.

Définition 42 (Différentielle extérieure) *Soit un complexe de chaîne $C(\mathcal{K}, G, \partial)$, on définit la* différentielle extérieure *comme l'opérateur $\mathbf{d}$ adjoint à ∂ tel que $\mathbf{d} = \widetilde{\partial}$.*

Calcul de la différentielle. Nous avons vu que l'opérateur ∂ était entièrement défini par la donnée des fonctions d'orientation $\mathrm{o}_{\sigma\tau}$. Nous proposons dans ce paragraphe de vérifier la même propriété pour la différentielle extérieure $\mathbf{d}$. Soient f, un homomorphisme de groupes abéliens de G dans G', $\tau \in \mathcal{K}_p$ et $c = \sum g_\sigma.\sigma \in C_{p+1}(\mathcal{K}, G)$. Il suit

$$\begin{aligned}
\left[\mathbf{d}f.\tau, \sum_{\sigma\in\mathcal{K}_{p+1}} g_\sigma.\sigma\right] &= \left[f.\tau, \partial \sum_{\sigma\in\mathcal{K}_{p+1}} g_\sigma.\sigma\right] \\
&= \left[f.\tau, \sum_{\sigma\in\mathcal{K}_{p+1}} \partial(g_\sigma.\sigma)\right] \\
&= \left[f.\tau, \sum_{\sigma\in\mathcal{K}_{p+1}} \sum_{\sigma'\in\mathcal{K}_p} \mathrm{o}_{\sigma\sigma'}(g_\sigma).\sigma'\right] \\
&= \left[f.\tau, \sum_{\sigma'\in\mathcal{K}_p} \left(\sum_{\sigma\in\mathcal{K}_{p+1}} \mathrm{o}_{\sigma\sigma'}(g_\sigma)\right).\sigma'\right] \\
&= f\left(\sum_{\sigma\in\mathcal{K}_{p+1}} \mathrm{o}_{\sigma\tau}(g_\sigma)\right) \\
&= \sum_{\sigma\in\mathcal{K}_{p+1}} f(\mathrm{o}_{\sigma\tau}(g_\sigma)) \\
&= \sum_{\sigma\in\mathcal{K}_{p+1}} (f \circ \mathrm{o}_{\sigma\tau})(g_\sigma) \\
&= \left[\sum_{\tau'\in\mathcal{K}_{p+1}} (f \circ \mathrm{o}_{\tau'\tau}).\tau', \sum_{\sigma\in\mathcal{K}_{p+1}} g_\sigma.\sigma\right]
\end{aligned}$$

On identifie ainsi l'égalité suivante : $\forall f \in \mathrm{Hom}(G, G), \tau \in \mathcal{K}$

$$\mathbf{d}(f.\tau) = \sum_{\sigma\in\mathcal{K}_{p+1}} (f \circ \mathrm{o}_{\sigma\tau}).\sigma$$

En étendant ce résultat à une cochaîne T non-réduite à une seule cellule, on obtient la forme générale du calcul de la différentielle extérieure

$$\mathbf{d}(T) = \sum_{\sigma\in\mathcal{K}_{p+1}} \left(\sum_{\tau\in\mathcal{K}_p} T(\tau) \circ \mathrm{o}_{\sigma\tau}\right).\sigma$$

entièrement déterminé par la donnée des fonctions d'orientation. On notera notamment sa ressemblance avec l'équation calculant le bord lorsqu'on utilise les homomorphismes duaux des fonctions d'orientation $\widetilde{\mathrm{o}_{\sigma\tau}}(f) = f \circ \mathrm{o}_{\sigma\tau}$:

$$\partial(c) = \sum_{\tau \in \mathcal{K}_{p-1}} \left(\sum_{\sigma \in \mathcal{K}_p} \mathrm{o}_{\sigma\tau}(c(\sigma)) \right) .\tau \qquad \mathbf{d}(T) = \sum_{\sigma \in \mathcal{K}_{p+1}} \left(\sum_{\tau \in \mathcal{K}_p} \widetilde{\mathrm{o}_{\sigma\tau}}(T(\tau)) \right) .\sigma$$

Représentation géométrique de d. Nous avons vu au chapitre 3 que l'opérateur de bord ∂ permettait le transport des valeurs des p-cellules aux $(p-1)$-cellules d'un complexe. Soient deux cellules σ_1 et σ_2 d'un complexe $\mathcal{K}$ telles que $\sigma_2 < \sigma_1$. On note $\partial(g.\sigma_1).\sigma_2$, la valeur associée à la cellule σ_2 par le calcul du bord de la chaîne $g.\sigma_1$. Par les définitions qui précèdent, il suit que $\partial(g.\sigma_1).\sigma_2 = \mathrm{o}_{\sigma_1\sigma_2}(g)$. Ainsi, la valeur g est transportée de σ_1 vers σ_2 sous la forme $\mathrm{o}_{\sigma_1\sigma_2}(g)$.

De façon duale, si $\tau_1 < \tau_2$ sont deux cellules de $\mathcal{K}$, un homomorphisme h est transporté de τ_1 vers τ_2 par $\mathbf{d}(h.\tau_1).\tau_2 = \widetilde{\mathrm{o}_{\tau_2\tau_1}}(h)$. La différence fondamentale avec le transport calculé par l'opérateur de bord ∂ concerne les dimensions des cellules : ∂ transmet les valeurs des cellules vers leurs faces (de dimension inférieure), alors que $\mathbf{d}$ transmet les homomorphismes des cellules vers leurs cofaces (de dimension supérieure).

Nous cherchons alors à définir formellement le transport dans une chaîne des valeurs des cellules vers leurs cofaces. Pour cela nous utilisons l'équivalence entre chaîne et cochaîne. Soient deux cellules $\sigma_1 < \sigma_2$, nous cherchons à transmettre une valeur $g \in G$ de σ_1 à σ_2 (l'opérateur ∂ transporte g de σ_2 à σ_1). La chaîne $g.\sigma_1$ est transformée en la cochaîne $(n \mapsto n.g).\tau_1$ (où $\tau_1 = \sigma_1$). À la lumière de ce qui vient d'être dit, l'opérateur $\mathbf{d}$ transporte $(n \mapsto n.g)$ vers τ_2 (choisie telle que $\tau_2 = \sigma_2$) en la valeur $\widetilde{\mathrm{o}'_{\tau_2\tau_1}}(n \mapsto n.g)$. L'orientation que nous utilisons ici est celle des cochaînes de $C(\mathcal{K}, \mathbb{Z}, G, \mathbf{d})$ et n'a pas encore été définie. Cependant, celle-ci doit être en accord avec l'orientation définie par $C(\mathcal{K}, G, \partial)$. Pour cela, nous posons :

$$\widetilde{\mathrm{o}'_{\tau_2\tau_1}}(n \mapsto n.g) = (n \mapsto n.g) \circ \mathrm{o}'_{\tau_2\tau_1} = (n \mapsto \mathrm{o}^{-1}_{\tau_2\tau_1}(n.g)) = (n \mapsto n\, \mathrm{o}^{-1}_{\tau_2\tau_1}(g))$$

où $\mathrm{o}^{-1}_{\tau_2\tau_1}$ représente l'orientation relative de τ_1 par rapport à τ_2 ($\mathrm{o}_{\tau_2\tau_1}$ étant l'orientation relative de τ_2 par rapport à τ_1)[1]. Ainsi, la cellule τ_2 reçoit l'homomorphisme $(n \mapsto n\, \mathrm{o}^{-1}_{\tau_2\tau_1}(g))$. En transformant la cochaîne ainsi obtenue en une chaîne (la transformation inverse n'est possible que lorsque le complexe $\mathcal{K}$ est localement fini, cas dans lequel nous nous plaçons), la valeur finalement associée à σ_2 est $(n \mapsto n\, \mathrm{o}^{-1}_{\tau_2\tau_1}(g))(1) = \mathrm{o}^{-1}_{\tau_2\tau_1}(g)$. Le transport inverse d'une cellule vers ses cofaces est alors défini : la cellule σ_1 recevant $\mathrm{o}_{\tau_2\tau_1}(g)$ par ∂, retourne finalement $\mathrm{o}^{-1}_{\tau_2\tau_1}(\mathrm{o}_{\tau_2\tau_1}(g))$.

2.2.3 Les complexes de cochaînes

Pour clore cette section, nous définissons les complexes de cochaînes, adjoints des complexes de chaînes :

[1]Pour établir cette égalité, nous supposons en effet qu'il existe une suite de fonctions d'orientation $\mathrm{o}^{-1}_{\tau_2\tau_1}$. Intuitivement, si $\mathrm{o}_{\tau_2\tau_1}$ correspond à l'orientation de τ_1 par rapport τ_2, $\mathrm{o}^{-1}_{\tau_2\tau_1}$ correspond à l'orientation de τ_2 par rapport à τ_1. Nous utilisons l'exposant -1 pour signifier que si $\mathrm{o}_{\tau_2\tau_1}$ est inversible il s'agit effectivement de son inverse de telle sorte que si g est transporté de σ_1 vers σ_2 en la valeur $\mathrm{o}_{\sigma_2\sigma_1}(g)$ par ∂, alors l'opération inverse que nous définissons ici transporte $\mathrm{o}_{\sigma_2\sigma_1}(g)$ de σ_2 vers σ_1 en effectuant le calcul $\mathrm{o}^{-1}_{\tau_2\tau_1}(\mathrm{o}_{\sigma_2\sigma_1}(g))$ dont le résultat est finalement g lui-même. Les équations mènent finalement au fait que si $\mathrm{o}_{\sigma_2\sigma_1}$ est un isomorphisme, le cadre des transports de valeurs impose qu'il s'agisse d'une involution. Ce résultat est tout à fait cohérent dans le sens où la notion d'orientation implique une orientation positive et négative ; les fonctions d'orientation reviennent donc à retourner l'opposé de son argument dans le cas négatif, ou à retourner l'argument sans modification dans le cas positif.

Définition 43 (Groupe de cochaînes, complexe de cochaînes) *Soient $\mathcal{K}$ un complexe cellulaire abstrait de dimension N, et G et G' deux groupes abéliens arbitraires. Le* groupe des cochaînes *associé à $\mathcal{K}$, G et G' est défini par :*

$$C(\mathcal{K}, G, G') = \bigoplus_{i \in \mathbb{N}} C^i(\mathcal{K}, G, G') = C^0(\mathcal{K}, G, G') \oplus C^1(\mathcal{K}, G, G') \oplus \cdots \oplus C^N(\mathcal{K}, G, G') \oplus \dots$$

où $\oplus$ est la somme directe de groupes abéliens.

Un complexe de cochaînes $C(\mathcal{K}, G, G', \mathbf{d})$ *est une suite $(C^p(\mathcal{K}, G, G'), \mathbf{d}^p)_{p \in \mathbb{N}}$ de groupes abéliens et d'homomorphismes $\mathbf{d}^p : C^p(\mathcal{K}, G, G') \to C^{p+1}(\mathcal{K}, G, G')$.*

Les complexes de cochaînes construisent avec leurs complexes de chaînes adjoints le diagramme suivant :

$$\begin{array}{ccccccc}
\dots & \xrightarrow{\mathbf{d}} & C^p(\mathcal{K}, G, G') & \xrightarrow{\mathbf{d}} & C^{p+1}(\mathcal{K}, G, G') & \xrightarrow{\mathbf{d}} & \dots \\
\dots & \xleftarrow{\partial} & C_p(\mathcal{K}, G) & \xleftarrow{\partial} & C_{p+1}(\mathcal{K}, G) & \xleftarrow{\partial} & \dots
\end{array}$$

L'étoile de Hodge, une autre idée du transport. Nous avons vu que l'opérateur $\mathbf{d}$ permettait le transport d'une valeur d'une cellule vers ses cofaces dans une chaîne : pour cela nous somme passés par la représentation équivalente d'une chaîne par une cochaîne. Un autre point de vue peut être adopté pour définir ce transport. Soit un complexe de chaînes $C(\mathcal{K}, G, \partial)$. L'opérateur de bord transportant les valeurs des cellules vers leurs faces dans $\mathcal{K}$, il transporterait les valeurs vers leurs cofaces dans le même complexe $\mathcal{K}$ mais où la relation d'incidence serait renversée. Un tel complexe existe, il s'agit du complexe dual de $\mathcal{K}$.

Définition 44 (Complexe dual) *Soit $\mathcal{K} = (S, \prec, dim)$ un complexe cellulaire abstrait de dimension n. On définit le complexe $*\mathcal{K} = (S, *\prec, *dim)$* dual *de $\mathcal{K}$ par :*

- $\tau \prec \sigma \;\Rightarrow\; \sigma *\prec \tau$ *(autrement dit, $*\prec = \succ$)*
- $dim(\sigma) = p \Rightarrow *dim(\sigma) = n - p$ *(autrement dit, $*dim = n - dim$)*

À partir du complexe de chaînes $C(\mathcal{K}, G, \partial)$, nous définissons un nouveau complexe de chaînes $C(*\mathcal{K}, G, *\partial)$, où l'opérateur de bord $*\partial$ coïncide avec ∂ de telle sorte que le transport des valeurs soit effectué des cellules vers les cofaces. Soient $\sigma_1 < \sigma_2$ deux cellules topologiques de $\mathcal{K}$; en particulier, on note que $\sigma_2 *\!\!\ll \sigma_1$. Si $\partial(g.\sigma_2).\sigma_1$ envoie la valeur $\mathrm{o}_{\sigma_2\sigma_1}(g)$ vers σ_1, l'expression $*\partial(g.\sigma_1).\sigma_2$ associe $*\mathrm{o}_{\sigma_1\sigma_2}(g)$ de σ_1 vers σ_2, où $*\mathrm{o}_{\sigma_1\sigma_2}$ correspond aux fonctions d'orientation dans le complexe de chaînes $C(*\mathcal{K}, G, *\partial)$. Pour que le transport calculé par $*\partial$ soit le transport inverse de ∂, nous posons simplement $*\mathrm{o}_{\sigma_1\sigma_2} = \mathrm{o}^{-1}_{\sigma_2\sigma_1}$, l'orientation inverse de $\mathrm{o}_{\sigma_1\sigma_2}$ que nous avons introduit précédemment. Dans sa thèse [Hir03], A. Hirani donne un algorithme pour calculer l'orientation du dual en fonction de celle définie dans le primal.

Le complexe de chaînes dual $C(*\mathcal{K}, G, *\partial)$ induit également une différentielle extérieure duale, dénotée par $*\mathbf{d}$, définissant ainsi, les complexes de cochaînes duaux sur des groupes G' arbitraires $C(*\mathcal{K}, G, G', *\mathbf{d})$. L'ensemble de ces complexes est regroupé sur le diagramme suivant :

$$\begin{array}{ccccccc}
\dots & \xrightarrow{\mathbf{d}} & C^p(\mathcal{K}, G, G') & \xrightarrow{\mathbf{d}} & C^{p+1}(\mathcal{K}, G, G') & \xrightarrow{\mathbf{d}} & \dots \\
\dots & \xleftarrow{\partial} & C_p(\mathcal{K}, G) & \xleftarrow{\partial} & C_{p+1}(\mathcal{K}, G) & \xleftarrow{\partial} & \dots \\
 & & * \downarrow \uparrow * & & * \downarrow \uparrow * & & \\
\dots & \xrightarrow{\partial} & C_{n-p}(*\mathcal{K}, G) & \xrightarrow{*\partial} & C_{n-p-1}(*\mathcal{K}, G) & \xrightarrow{*\partial} & \dots \\
\dots & \xleftarrow{*\mathbf{d}} & C^{n-p}(*\mathcal{K}, G, G') & \xleftarrow{*\mathbf{d}} & C^{n-p-1}(\mathcal{K}, G, G') & \xleftarrow{*\mathbf{d}} & \dots
\end{array}$$

La première et la dernière ligne du diagramme sont liées par un nouvel opérateur *géométrique* (il s'oppose aux opérateurs précédents qui sont purement algébriques et topologiques) appelé l'*étoile de Hodge*. Le complexe dual a une grande importance pour le calcul différentiel discret à travers une forte signification géométrique : si $\mathcal{K}$ représente une partition d'une variété $\mathcal{M}$, $*\mathcal{K}$ découpe $\mathcal{M}$ de façon différente. On associe alors à chaque cellule une norme issue d'une métrique définie sur $\mathcal{M}$. En notant par $|\sigma|$ la norme associée à la cellule σ d'un complexe $\mathcal{K}$, l'étoile de Hodge d'une forme différentielle ω, dénotée $\star\omega$, est définie par :

$$\frac{1}{|\sigma|}\int_{\sigma}\omega = \frac{1}{|*\sigma|}\int_{*\sigma}\star\omega$$

Cet opérateur complète alors le diagramme précédent en

$$\begin{array}{ccccccc}
\ldots & \xrightarrow{\mathbf{d}} & C^p(\mathcal{K},G,G') & \xrightarrow{\mathbf{d}} & C^{p+1}(\mathcal{K},G,G') & \xrightarrow{\mathbf{d}} & \ldots \\
 & & \star\downarrow\uparrow\star & & \star\downarrow\uparrow\star & & \\
\ldots & \xleftarrow{*\mathbf{d}} & C^{n-p}(*\mathcal{K},G,G') & \xleftarrow{*\mathbf{d}} & C^{n-p-1}(\mathcal{K},G,G') & \xleftarrow{*\mathbf{d}} & \ldots
\end{array}$$

L'étoile de Hodge permet notamment de définir la codifférentielle discrète δ, un nouvel opérateur s'appliquant sur des p-cochaînes, en fonction de la différentielle discrète par :

$$\delta = (-1)^{n(p-1)+1}\ \star\ *\mathbf{d}\ \star$$

Cet opérateur est par exemple utilisé pour définir le laplacien discret par $\Delta = \delta\mathbf{d} + \mathbf{d}\delta$. Afin de comprendre comment celui-ci fonctionne, les différentes étapes du calcul du laplacien d'une p-cochaîne peuvent être décrites par le diagramme suivant :

$$\begin{array}{ccccc}
C^{p-1}(\mathcal{K},G,G') & \xrightarrow{\mathbf{d}} & C^p(\mathcal{K},G,G') & \xrightarrow{\mathbf{d}} & C^{p+1}(\mathcal{K},G,G') \\
\star\uparrow & & \star\downarrow\uparrow\star & & \star\downarrow \\
C^{n-p+1}(*\mathcal{K},G,G') & \xleftarrow{*\mathbf{d}} & C^{n-p}(*\mathcal{K},G,G') & \xleftarrow{*\mathbf{d}} & C^{n-p-1}(*\mathcal{K},G,G')
\end{array}$$

Le laplacien correspond donc à un mouvement des valeurs associées par une chaîne à une p-cellule σ d'un complexe $\mathcal{K}$, vers les p-cellules $(p-1)$-voisines et $(p+1)$-voisines de σ.

3 Travaux en cours

Dans cette section, nous souhaitons faire part au lecteur des travaux en cours que nous menons sur l'utilisation des concepts apportés par les cochaînes et le formalisme que nous venons de présenter dans le cadre du langage MGS. En effet, nous avons utilisé les chaînes topologiques pour représenter formellement les collections topologiques (voir chapitre 3) et ainsi définir une sémantique du langage (voir chapitres 4 et 5). Nous présentons dans ces sémantiques comment les transformations opèrent pour calculer de nouvelles collections à partir d'un ensemble de règles de déduction. Néanmoins, nous ne donnons pas directement un objet mathématique correspondant aux transformations.

Nous avons vu plus haut que les cochaînes étaient des fonctions de chaînes. Par notre formalisation des collections topologiques, il est naturel de se pencher sur les liens existant entre les notions de cochaîne et de transformation. Nous commençons par poser les bases de cette analogie que nous illustrons ensuite en programmant à l'aide des transformations les opérateurs que nous venons de voir. Pour finir, nous développons quelques perspectives que ce rapprochement entre transformations et cochaînes invite à suivre accompagnées d'exemples de programmes.

3.1 Rappels sur les domaines

Pour élaborer nos propos, nous nous plaçons dans le cadre défini par la sémantique du chapitre 4 auquel nous apportons les quelques modifications suivantes.

Les scalaires. Les scalaires que nous avons considérés (les entiers, les flottants, les booléens, les symboles, les positions de $\mathcal{S}$ et les clôtures) ne permettent pas de manipuler les éléments de Abel(Val). Nous considérons désormais qu'ils appartiennent à l'ensemble des scalaires MGS. Ils correspondent à des fonctions de Val dans $\mathbb{Z}$; il s'agit donc d'un nouveau type de valeurs d'ordre supérieur (en plus des clôtures et des clôtures opaques) pouvant être appliquées sur n'importe quel élément de Val. Leur manipulation est proche de celle des clôtures opaques : les règles de la sémantique concernant ces dernières fonctionnent également pour les éléments de Abel(Val). La valeur spéciale `<undef>` est alors identifiée à $0_{\text{Abel(Val)}}$.

Les collections topologiques. Nous reprenons l'ensemble des collections topologiques construites sur les positions de l'ensemble de symboles $\mathcal{S}$. Une collection topologique est une chaîne de $C(\mathcal{K}, \text{Abel(Val)})$ où $\mathcal{K}$ est un complexe cellulaire abstrait construit sur $\mathcal{S}$ ($S^{\mathcal{K}} \subset \mathcal{S}$). Nous pouvons remarquer que l'ensemble des éléments Val possède une structure de groupe hérité des éléments de Abel(Val) ; tout élément n'appartenant pas à Abel(Val) est automatiquement transformé en utilisant l'injection canonique Inj_{Val}. Ainsi, nous simplifions l'ensemble des collections topologiques en $\mathsf{CollType}(\text{Val}) = C(\mathcal{K}, \text{Val})$.

Pour résumer, l'ensemble des valeurs MGS Val correspond au plus petit ensemble solution de l'équation :

$$\text{Val} = \mathsf{ScalType} \cup \text{Abel(Val)} \cup \mathsf{CollType}(\text{Val})$$

3.2 Fondement de l'analogie transformation/forme différentielle

Les collections topologiques étant des chaînes de $C(\mathcal{K}, \text{Val})$, nous posons la question de la représentation d'une cochaîne de $C(\mathcal{K}, \text{Val}, \text{Val})$ à l'aide d'une transformation. Pour cela nous considérons la cochaîne $T = f_{\tau_1}.\tau_1 + f_{\tau_2}.\tau_2$ associant l'homomorphisme de $f_{\tau_1} \in \text{Hom}(\text{Val}, \text{Val})$ (resp. f_{τ_2}) à la cellule topologique τ_1 (resp. τ_2) du complexe $\mathcal{K}$. Il est sous entendu que pour toutes les autres cellules topologiques la cochaîne T associe l'homomorphisme nul envoyant tout argument sur 0_{Val}.

Il est intuitif de représenter la cochaîne T par la transformation `T` définie comme suit :

```
trans T = {
  x / ^x == τ1 => f_τ1(x) ;
  x / ^x == τ2 => f_τ2(x) ;
  x            => <undef>
} ;;
```

Bien entendu, les parties du programme faisant intervenir des variables mathématiques sont à remplacer par le programme MGS adéquat (par exemple, f_{τ_1} peut être substituée par la variable `f_tau1` définie ailleurs et encodant le comportement de f_{τ_1}).

3.2.1 Paramétrisation de la reconstruction

Dans l'état actuel des choses (c'est-à-dire l'état de la sémantique du chapitre 4), si la transformation `T` est appliquée sur une collection représentée par la chaîne $\sum g_\sigma.\sigma$, la valeur retournée

est la collection topologique $f_{\tau_1}(g_{\tau_1}).\tau_1 + f_{\tau_2}(g_{\tau_2}).\tau_2$. Ce résultat ne correspond pas tout à fait au calcul effectué par la cochaîne T :

$$\left[T, \sum g_\sigma.\sigma\right] = f_{\tau_1}(g_{\tau_1}) +_{\text{Val}} f_{\tau_2}(g_{\tau_2})$$

Pour retrouver ce résultat, il suffirait de remplacer l'addition des chaînes par l'addition dans Val. Ce « remplacement » correspond à un homomorphisme de chaînes. L'application d'une forme à une chaîne peut donc être programmée en MGS par :

```
fold((\x,y.x +_val y), <undef>, T(c)) ;;
```

où la variable `c` est associée à $\sum g_\sigma.\sigma$ et l'opérateur `+_val` dénote $+_{\text{Val}}$ l'addition du groupe Val. L'opération `fold` est appelée parfois « réduction » dans les langage fonctionnels. Or, une forme de réduction est présente dans une transformation : la reconstruction du résultat après la phase de filtrage, c'est-à-dire le passage de la structure de batch (voir les chapitres 2 et 4, pages 58 et 124) au résultat. Sur notre exemple, le batch construit par la transformation `T` appliquée sur la collection `c` est le suivant :

$$[\langle \tau_1, f_{\tau_1}(g_{\tau_1})\rangle, \langle \tau_2, f_{\tau_2}(g_{\tau_2})\rangle, \ \ldots]$$

où l'ellipse correspond aux cellules σ de $\mathcal{K} - \{\tau_1, \tau_2\}$, pour lesquels, le couple (collection filtrée, collection calculée) est de la forme $\langle \sigma, 0_{\text{Val}}\rangle$.

Pouvoir paraméter la reconstruction effectuée à partir du batch, i.e. choisir de l'homomorphisme qui interprètera l'addition des chaînes, offre un mécanisme très puissant qui généralise la réduction que l'on trouve dans les langages fonctionnels pour les types algébriques. Aussi, nous proposons de laisser la spécification de la reconstruction au programmeur, de façon similaire à la spécification de la réduction que nous venons de voir. Les transformations sont alors dotées de deux nouveaux paramètres optionnels :

1. `zero` : il s'agit de la valeur initialisant l'accumulateur de la réduction. La valeur par défaut, de cet argument optionnel est 0_{Val}.
2. `addition` : il s'agit de la fonction de réduction à trois arguments : l'ensemble des positions filtrées `S` (représentant la collection filtrée), l'évaluation de la partie droite de la règle `e`, et la valeur de l'accumulateur `acc`. Par défaut, la fonction de réduction est : `\(S,e,acc).e +_val acc`.

Suivant cette nouvelle paramétrisation de la reconstruction, le résultat de l'application `T(c)` évalue la valeur $f_{\tau_1}(g_{\tau_1}) +_{\text{Val}} f_{\tau_2}(g_{\tau_2})$ et non la collection topologique $f_{\tau_1}(g_{\tau_1}).\tau_1 + f_{\tau_2}(g_{\tau_2}).\tau_2$. Ce dernier résultat est obtenu par l'application paramétrée :

```
T[ addition = \(S,e,acc).({| e@oneof(S) |}, acc),
   zero     = {||}
 ](c)
```

L'expression `oneof(S)` retourne un élément de l'ensemble `S`, qui est dans cet exemple un singleton.

3.2.2 Dictionnaire de l'analogie

Les transformations avec la paramétrisation de la reconstruction[2] deviennent un concept extrêmement proche des cochaînes. Ce rapprochement amène à constater l'analogie suivante

[2]Bien que la paramétrisation ait été introduite dans le but de rapprocher le concept de transformation à celui de cochaîne, la paramétrisation de la reconstruction est également motivée, de façon totalement indépendante du rapprochement avec la topologie algébrique, par la généralisation des transformations à tout type de données. En effet, à l'heure actuelle, chaque type de collection topologique possède sa propre reconstruction.

entre calcul différentiel, topologie algébrique et les concepts issus du langage MGS :

	Calcul différentiel	Topologie algébrique	MGS
Application	$\int_{\_} \_$	$[\_, \_]$	`_(_)`
Argument	$\mathcal{M}$ un ouvert de $\mathbb{R}^n$	c une p-chaîne	`c` collection topologique
Objet appliqué	$\omega = \sum a_H(\mathbf{x})\, \mathbf{d}x^{h_1} \wedge \cdots \wedge \mathbf{d}x^{h_p}$	$T = \sum_{\tau \in \mathcal{K}_p} f_\tau . \tau$	`trans T = { x => f(x) }`
Itérateur/position	$\mathbf{d}x^{h_1} \wedge \cdots \wedge \mathbf{d}x^{h_p}$	τ	`^x`
Valeur	$\mathbf{x} = (x^{h_1}, \ldots, x^{h_p})$	$c(\tau)$	`x`
Fonction locale	a_H	f_τ	`f`
Syntaxe complète	$\int_{\mathcal{M}} \omega$	$[T, c]$	`T(c)`

Ce tableau résume sous la forme d'un dictionnaire les différents concepts dont nous avons parlé. Les trois colonnes de droite correspondent respectivement au calcul d'une intégrale, à l'application d'un cochaîne sur une chaîne, et à l'application d'une transformation sur une collection topologique. Les lignes établissent l'analogie suivante :

Application : cette ligne décrit la forme syntaxique dénotant l'application. L'application consiste dans tous les cas à appliquer localement une fonction en chaque élément d'un domaine. La syntaxe doit permettre de fournir une fonction et un domaine à parcourir. Du point de vue du calcul à effectuer, cette syntaxe spécifie un « itérateur » parcourant les éléments du domaine pour appliquer localement une fonction et combiner les résultats.

Argument : cette ligne indique l'objet qui décrit le domaine de l'application. Le domaine exprime une structure spatiale organisée où il est possible d'itérer sur les éléments et d'en récupérer la valeur associée.

Objet appliqué : il s'agit de la spécification des fonctions à appliquer localement sur chaque élément du domaine.

Itérateur : l'itérateur est une référence à l'objet courant du calcul local qu'on souhaite effectuer (l'élément indexé par l'itérateur).

Valeur : il s'agit de la décoration associée par l'argument à la position pointée par l'itérateur.

Fonction locale : elle spécifie le calcul à appliquer localement ; elle prend en entrée la valeur associée par l'argument à la position pointée par l'itérateur.

Syntaxe complète : donne l'ensemble des constituants de l'application suivant la syntaxe spécifique à chacun des trois champs d'application.

3.3 Implantation des opérateurs discrets

L'analogie entre calcul différentiel et topologie combinatoire a été décrit au début de ce chapitre. L'analogie liant la topologie combinatoire et MGS est fondée essentiellement sur la formalisation mathématique des collections sous forme de chaînes topologiques. Les concepts élaborés pour le langage MGS sont donc adaptés à la programmation d'opérateurs liés aux deux premiers champs d'application de notre analogie.

Positions et orientation. Les collections topologiques sont des chaînes construites sur des complexes cellulaires abstraits dont les positions sont issues de $\mathcal{S}$. Comme nous l'avons déjà dit dans le chapitre 4, une position prise hors du contexte d'un complexe cellulaire abstrait n'a pas de sens. Nous utilisons pour cela la notation $\sigma^{\mathcal{K}}$ pour signifier que la cellule topologique était considérée dans le complexe $\mathcal{K}$.

Un complexe $\mathcal{K}$ permet de définir des collections topologiques, et par conséquent des chaînes topologiques. Afin de pouvoir définir l'opérateur de bord de ces collections, les fonctions d'orientation doivent être précisées. Nous supposons l'existence d'une fonction MGS `orient` disponible dans les règles de réécriture d'une transformation. Elle fait alors référence à la fonction d'orientation associée à l'opérateur de bord défini par la forme de la collection. Il s'agit d'une fonction à trois arguments ; soient σ_1 et σ_2 deux cellules d'un même complexe $\mathcal{K}$ et g une valeur de Val :

$$\texttt{orient}(\sigma_1,\sigma_2,g) = \begin{cases} \mathrm{o}_{\sigma_1\sigma_2}(g) & \text{si } dim^{\mathcal{K}}(\sigma_1) > dim^{\mathcal{K}}(\sigma_2) \\ \mathrm{o}^{-1}_{\sigma_1\sigma_2}(g) & \text{sinon} \end{cases}$$

L'opérateur de bord. Les fonctions d'orientation définissent l'opérateur de bord. Soit une collection topologique construite sur le complexe $\mathcal{K}$. Cet opérateur transmet une valeur g associée à une cellule σ_1 de $\mathcal{K}_p$ à une cellule σ_2 de $\mathcal{K}_{p-1}$, sous la forme $\mathrm{o}_{\sigma_1\sigma_2}(g)$. Les fonctions d'orientation sont nulles lorsque σ_1 et σ_2 ne sont pas incidentes. Ainsi, pour une cellule de σ de $\mathcal{K}$ les seules valeurs non nulles qui la décoreront proviennent de ses cofaces. Il suffit par conséquent, pour chaque cellule de la collection d'itérer sur la liste des valeurs associées à ses cofaces pour calculer le transport de valeurs. La primitive `cofacefold` fournit par MGS (voir chapitre 2 page 51) permet cela :

```
trans boundary[ addition = \(S,e,acc).({| e@oneof(S) |}, acc),
                zero     = {||}
              ] =
{  x => cofacefold(\(y,acc).(orient(^y,^x,y) +_val acc), <undef>, x)  } ;;
```

L'opérateur dual de bord. Comme nous l'avons vu, l'opérateur de bord dual agit de façon opposée à l'opérateur ; il suit naturellement que pour l'implantation, l'itération n'est plus sur les cofaces mais sur les faces des cellules :

```
trans coboundary[ addition = \(S,e,acc).({|e@oneof(S)|}, acc),
                  zero     = {||}
                ] =
{  x => facefold(\(y,acc).(orient(^y,^x,y) +_val acc), <undef>, x)  } ;;
```

La différentielle extérieure. La définition de cet opérateur est directement liée au théorème de Stokes : $[\mathbf{d}T, c] = [T, \partial c]$. Il est important de noter que cet opérateur agit sur les transformations et non sur les collections. Soit `T` une transformation MGS. La différentielle extérieure de `T` produit une nouvelle transformation ; en utilisant la transformation `boundary` définie précédemment, l'implantation de la différentielle est directe :

```
fun derivative[zero=<undef>, addtion=\S,e,acc.(e +_val acc)](T) =
  \c.T[zero=zero,addition=addition](boundary(c))
;;
```

La différentielle d'une transformation `T` prend en argument une collection `c`, lui applique la composée de `T` et de la transformation de bord et retourne une transformation (i.e. une fonction). Les paramètres pour la reconstruction sont transmis via des arguments optionnels éponymes associés à la fonction `derivative`. Les valeurs par défaut qui leur sont associées correspondent aux paramètres de reconstruction standards. La différentielle extérieure duale s'exprime de façon similaire, à l'aide de l'opérateur de bord dual.

Les opérateurs gradient, rotationnel et divergence. Nous avons vu plus haut que la différentielle extérieure avait une représentation géométrique lorsqu'elle était appliquée sur des cochaînes issues de la traduction d'une chaîne en cochaîne. De ce point de vue, cet opérateur coïncide entièrement avec l'implantation de l'opérateur dual de bord que nous venons de voir. Cependant, on peut illustrer directement la différentielle extérieure. Pour cela nous allons nous intéresser aux opérateurs gradient, rotationnel et divergence de la physique discrète. Ils correspondent à des applications de la différentielle extérieure pour des dimensions particulières ; en effet, nous rappelons que le gradient (resp. le rotationnel, la divergence) calcule une 1-forme (resp. une 2-forme, une 3-forme) à partir d'une 0-forme (resp. une 1-forme, une 2-forme). Comme il ne s'agit que d'une dépendance à la dimension, nous allons nous concentrer sur l'opérateur gradient.

Considérons l'équation suivante liant le potentiel électrique V au champ électrique $\overrightarrow{E}$:

$$\overrightarrow{E} = -\overrightarrow{\nabla} V$$

Connaissant le potentiel électrique V, un opérateur `gradient` permettrait de calculer le champ électrique. Nous avons vu plus haut que cet opérateur s'applique sur une 0-forme et produit une 1-forme. La champ scalaire V correspond par conséquent à une 0-forme ; il s'agit donc d'une fonction qui associe à chaque point de l'espace et de façon continue, la valeur du potentiel électrique. Soit $\mathcal{K}$, la structure discrète sur laquelle nous souhaitons appliquer l'équation ; on représente V par une 0-cochaîne de $C^0(\mathcal{K}, \mathbb{Z}, \mathbb{R})$:

$$V = \sum_{\tau^0 \in \mathcal{K}_0} V_{\tau^0}.\tau^0$$

où V_{τ^0} est un homomorphisme de $\mathbb{Z}$ dans $\mathbb{R}$ qui associe à $\tau^0 \in \mathcal{K}_0$ la valeur $V(1.\tau^0) = v_{\tau^0}$ du potentiel électrique en ce point. Suivant la logique que nous avons suivie jusqu'ici, la forme V doit être implantée par une transformation. Cependant, lorsque le domaine $\mathcal{K}_0$ est fini, ce qui est souvent le cas en simulation, il est tout aussi envisageable de représenter la cochaîne V par la chaîne $V' \in C_0(\mathcal{K}, \mathbb{R})$ équivalente :

$$V' = \sum_{\sigma^0 \in \mathcal{K}_0} V(1.\sigma^0).\sigma^0 = \sum_{\sigma^0 \in \mathcal{K}_0} v_{\sigma^0}.\sigma^0$$

Nous représentons alors la « forme différentielle » V' par une collection topologique. L'opérateur gradient, correspondant à l'application de la différentielle extérieure sur la forme V, est représenté par son équivalent sur les chaînes : l'opérateur de bord dual. Ainsi, nous définissons un opérateur gradient similaire directement sur les collections topologiques par :

```
fun grad(V) = coboundary<1>(V) ;;
```

L'application de la transformation `coboundary` est restreinte aux cellules de dimension `1` (dans une syntaxe proche des transformations de chemins), dimension des cellules de la collection résultat, ici une 1-forme. Ainsi, en supposons que le potentiel électrique est encodée dans une collection topologique `V`, le champ électrique se calcule par :

```
E = map(\x.-x, grad(V)) ;;
```

Les opérateurs rotationnel et divergence sont implantés de la même façon par

```
fun rot(c) = coboundary<2>(c) ;;
fun div(c) = coboundary<3>(c) ;;
```

Le laplacien. Le laplacien est calculé à partir de la différentielle et de la codifférentielle par : $\Delta = \delta\mathbf{d} + \mathbf{d}\delta$. Comme nous l'avons constaté précédemment, le laplacien permet, à une cellule σ^p d'un complexe $\mathcal{K}$, d'accéder aux valeurs associées aux cellules topologique $p-1$-voisines et $p+1$-voisines. Nous illustrons ce mouvement par deux exemples, dont l'application directe de la formule du laplacien permet de retrouver des résultats connus.

Soit un graphe linéaire infini $\mathcal{K}$ où les sommets sont placés de façon régulière dans un espace de dimension 1 de telle sorte que la longueur de chaque arc soit unitaire. Nous décidons d'étiqueter les sommets par un entier de telle sorte que $\sigma_i \in \mathcal{K}_0$ ait pour voisins σ_{i-1} et $\sigma_{i+1} \in \mathcal{K}_0$. Soit $q \in C^0(\mathcal{K}, \mathbb{Z}, \mathbb{R})$ une cochaîne

$$q = \sum_{\sigma_i \in \mathcal{K}_0} q_i.\sigma_i$$

En appliquant le laplacien on calcule :

$$\Delta q = \sum_{\sigma_i \in \mathcal{K}_0} ((q_{i-1} - q_i) + (q_{i+1} - q_i)).\sigma_i$$

Ce résultat apparaît par exemple lorsque la méthode des différences finies est utilisée pour résoudre l'équation de la chaleur dans un espace de dimension 1 :

$$\frac{\partial u}{\partial t} = a^2 \Delta u \qquad \Longrightarrow \qquad \frac{u_i(t+\Delta t) - u_i(t)}{\Delta t} = a^2((u_{i-1}(t) - u_i(t)) + (u_{i+1}(t) - u_i(t))$$

Le temps est discrétisé avec un pas fixe Δt et l'espace est discrétisé en sections régulières, dénotées par un entier i, sur lesquelles la forme u à l'instant t est estimée à $u_i(t)$. La constante a est un coefficient caractérisant le diffusivité de la chaleur.

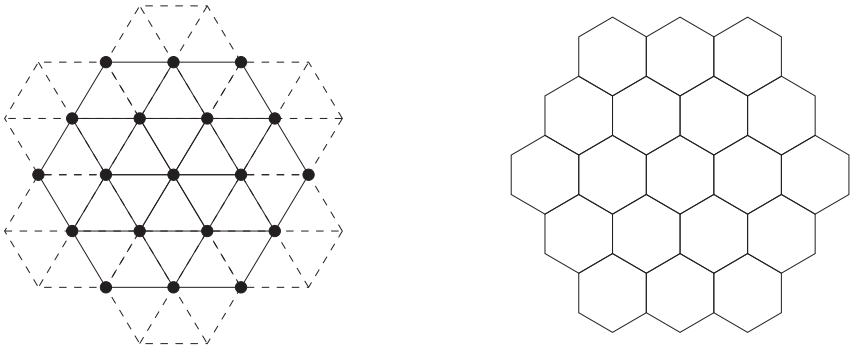


FIG. 1 – Maillage hexagonal : un maillage hexagonal du plan est obtenu en construisant un graphe régulier où chaque sommet possède six 1-voisins. Ce graphe triangule le plan (figure de gauche). Le dual du graphe construit le maillage hexagonal : chaque sommet du primal correspond à un hexagone du dual.

Dans l'exemple du graphe linéaire, l'effet de l'étoile de Hodge faisant intervenir la norme et la norme duale des cellules de $\mathcal{K}$, n'apparaît pas, celle-ci étant fixée à 1 par l'intervalle régulier qui sépare les sommets. Nous remplaçons maintenant le graphe linéaire $\mathcal{K}$ par un maillage hexagonal régulier : chaque sommet de $\mathcal{K}_0$ possède six 1-voisins placés régulièrement autour de lui à une distance unitaire. L'espace en dimension 2 est par conséquent triangulé comme le montre la figure 1. Les hexagones ont une aire de $\frac{\sqrt{3}}{2}$ et les arcs duaux (bordant les hexagones) ont pour longueur $\frac{1}{\sqrt{3}}$. En considérant cette métrique, le laplacien correspond à l'équation suivante

$$\Delta q = \sum_{\sigma_i \in \mathcal{K}_0} (\frac{2}{3} \sum_{j=0}^{6} (q_j - q_i)).\sigma_i$$

où nous supposons que les quantités q_j sont celles portées par les voisins de σ_i dans le maillage. Là encore, ce résultat est déjà utilisé pour simuler une diffusion dans les automates cellulaires [Rei05].

Nous oublions pour l'instant l'aspect géométrique du laplacien pour nous concentrer sur une définition combinatoire ; nous considérons pour cela que toutes les cellules ont une norme de 1 dans le primal et le dual. Dans ce cas, la codifférentielle correspond à la différentielle duale, qui peut être utilisée directement pour la programmation du laplacien.

```
fun coderivative[zero=<undef>, addtion=\(S,e,acc).(e +_val acc)](T) =
  \c.T[zero=zero,addition=addition](coboundary(c))
;;

fun laplacian[zero=<undef>, addtion=\(S,e,acc).(e +_val acc)](T) =
  (derivative(coderivative(T)))(c), (coderivative(derivative(T))(c))
;;
```

Le laplacien ainsi obtenu effectue les mouvements de données que nous venons de décrire.

Comme pour la différentielle, le laplacien peut également être programmé pour prendre en argument une collection topologique et non une transformation. L'intérêt de cette seconde programmation est l'utilisation de la primitive `neighborfold` (voir chapitre 2, page 51). En effet, nous proposons de programmer un opérateur quelque peu différent du laplacien, nommé `Delta`, calqué sur les formules issues des calculs précédents :

```
trans <n,p> Delta[ zero_nf = <undef>,
                   add_nf  = +_val,
                   sub_nf  = -_val
                 ] =
{ x => neighborfold(\(y,acc).( add_nf(sub_nf(y,x),acc) ), zero_nf, x) } ;;
```

Cette transformation est constituée d'une seule règle s'appliquant sur les `n`-cellules de la collection argument. La primitive `neighbordfold` itère sur les `n`-cellules `p`-voisines de `^x`. En plus des arguments optionnels de la reconstruction, la transformation `Delta` fournit 3 nouveaux arguments permettant la paramétrisation de l'itération sur les voisins :

1. `zero_nf` : initialisation de l'itération et valeur de retour si `^x` n'a pas de $\langle \mathtt{n}, \mathtt{p} \rangle$-voisines,
2. `sub_nf` : permet de combiner les valeurs de `x` et de `y`,
3. `add_nf` : addition de la réduction de la liste des voisins de `^x`.

Nous verrons une utilisation de cet opérateur dans les paragraphes suivants.

3.4 Ouvertures et perspectives

La mise en place de l'analogie ci-dessus a permis de programmer un grand nombre d'opérateurs. La mise en place de ces derniers invite à définir une algèbre de calcul autour des transformations et des collections topologiques. Les paragraphes suivants montrent deux exemples d'utilisation pouvant orienter le développement de cette algèbre.

Algèbres de programme. Les sommes formelles apparaissant dans la définition des collections topologiques et des transformations décomposent de façon canonique ces objets. Cette décomposition apparaît de surcroît dans le mécanisme de filtrage partitionnant une collection en sous-collections. La reconstruction est alors définie comme la réduction d'une liste, le batch, issue du filtrage. Il est intéressant de comparer cette situation avec l'approche des algèbres de Bird-Meertens [Bir87] et des algèbres de power-list [Mis94, Kor97]. Ces théories développent les fondements d'une définition (récursive) des listes et des tableaux. La décomposition repose alors sur la concaténation : concaténer deux listes (resp. deux tableaux de structures homogènes) produit une autre liste (resp. un nouveau tableau de structure homogène), amenant à considérer des stratégies de calcul « diviser pour régner ».

Ces travaux sont également à l'origine de l'étude des homomorphismes de listes, et plus généralement des catamorphismes [Mal90, MFP91] : un catamorphisme est une fonction agissant sur des structures de données de type algébrique (des arbres formels), calculant suivant une stratégie de bas en haut (des feuilles vers les racines) en remplaçant systématiquement les constructeurs par une fonction à évaluer dont les arguments sont le résultat de l'application du catamorphisme sur les feuilles. Par exemple, dans la figure 2, la somme des entiers décorant les feuilles d'un arbre se calcule par un catamorphisme : chaque constructeur de feuille est substitué par la fonction identité, alors que les constructeurs de nœud sont remplacés par l'addition des entiers. Ces catamorphismes sont élégamment programmés en utilisant le filtrage des structures de données algébriques comme nous l'avons présenté dans le chapitre 2.

L'approche offerte par les transformations étend le filtrage à des structures de données plus générales que les termes ; mais, là aussi, la partition par filtrage des collections en sous-collections construit une nouvelle structure à partir de la structure décrite par la relation d'incidence de la collection argument. Il est donc envisageable de réduire la collection ainsi partitionnée en remplaçant la relation d'incidence liant les sous-collections par des fonctions, suivant le même principe de substitution des constructeurs des types de données algébriques. C'est fait avec la paramétrisation du batch d'une transformation, induisant ainsi un homomorphisme de collections topologiques. Il reste à présent à étudier les lois de composition qui en résulte, de la même manière que Bird et Merteens ont pu étudier les homomorphisme de listes.

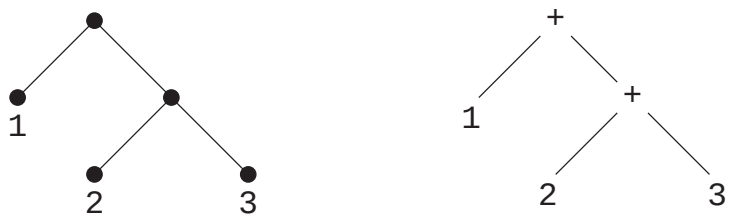


Fig. 2 – Exemple de catamorphisme : la somme des valeurs des feuilles d'un arbre est un catamorphisme. Le constructeur des feuilles est remplacé par la fonction identité, et le constructeur des nœuds est substitué par l'opérateur d'addition.

Equations différentielles discrètes. Nous entendons par *équations différentielles discrètes* un paradigme de programmation où les programmes ne sont plus des séquences d'instructions (des langages impératifs), ni des composées de fonctions (des langages fonctionnels), mais l'expression d'équations, à la façon de la logique équationnelle. Ce paradigme est évidemment inspiré des équations différentielles et de leur utilisation pour la modélisation physique ou biologique. L'idée est d'exprimer sous la forme d'une ou plusieurs équations les propriétés que le système modélisé doit respecter. Par exemple la définition d'une *énergie* peut pousser un système à se transformer pour que la totalité de l'énergie qu'il possède diminue jusqu'à atteindre un état minimisant (*a priori* localement) cette quantité. La trajectoire du système peut alors être contrainte : entre chaque modification diminuant un peu la quantité d'énergie qu'il contient, le système doit vérifier des propritétés de conservation d'autres quantités.

Ces propos semblent fortement liés à la simulation en physique. Les travaux de Tonti [Ton74, Ton75, Ton76, Ton01] expriment largement ces idées dans le cadre de la physique discrète où les opérateurs discrets que nous avons présentés dans ce chapitre sont utilisés pour exprimer des lois de conservation, de balance, etc. Néanmoins, la modélisation des systèmes n'est pas le seul domaine d'application de cette algèbre d'opérateurs. Pour illustrer ce point de vue, nous proposons de programmer par équation le calcul du plus court chemin dans un graphe.

Soit $\mathcal{K}$ un complexe cellulaire de dimension 1 fini, représentant le graphe pour lequel nous souhaitons exprimer le calcul du plus court chemin. Nous considérons pour ce problème que les arcs ont un poids égal à un. Nous cherchons à exprimer en chaque sommet de la structure le nombre minimum d'arcs qui faudra passer pour atteindre un sommet $\sigma \in \mathcal{K}_0$. On note en particulier qu'une fois le résultat calculé, les 1-voisins de σ étant distants d'un arc de σ, sont décorés par la valeur 1. Prenons un sommet arbitraire du graphe $\sigma' \neq \sigma$; il existe forcément parmi les 1-voisins de σ', un sommet plus proche que lui de σ. En effet, si σ' est à une distance n de σ, ce voisin sera à une distance $n-1$ de ce sommet. Cette propriété découle directement du principe de Bellman. Il est alors facile de vérifier que pour tout sommet de la structure sauf σ, le minimum de la différence de sa décoration et de celle d'un de ces voisins est toujours égale à 1. Nous pouvons donc utiliser l'opérateur `Delta` que nous avons introduit précédemment :

```
Delta[ addition = \(S,e,acc).{| e@oneof(S) |}, acc,
       zero     = {||}
       zero_nf  = -1
       sub_nf   = \(y,x).(y-x)
       add_nf   = min
     ]
```

Les deux premiers arguments permettent de reconstruire une collection topologique, et les trois derniers paramètrent l'appel à la primitive `neighborfold`. L'expression `sub_nf(y,x)` soustrait le valeur décorant la cellule 1-voisine, dénotée par `y`, à la valeur associée au sommet considéré, dénotée par `x`. On calcule alors le minimum de ces différences. Ce minimum doit être égal à -1 pour tout sommet différent de σ. En revanche, le minimum des différences est forcément 1 pour σ. Nous initialisons donc la réduction de la liste des voisins à -1 pour que tous les sommets, y compris σ, vérifie la même propriété. En dénotant Δ_{nf} la valeur de l'expression MGS que nous venons de définir, et c la collection résultat de notre problème, le système doit vérifier l'équation :

$$\Delta_{\mathrm{nf}} c = \sum_{\sigma \in \mathcal{K}_0} -1.\sigma$$

Cette simple équation caractérise le résultat du calcul que nous souhaitons effectuer, mais ne spécifie pas le calcul lui même. Pour cela, nous supposons une collection c_0 dont chaque sommet

surévalue sa distance à σ, excepté σ qui est décoré par 0. La surévaluation est donnée par exemple en associant à chaque sommet le cardinal $|\mathcal{K}_0|$; en effet, il ne faudra pas travers plus de sommet qu'il n'y en a dans le graphe pour atteindre σ.

$$c_0 = 0.\sigma + \sum_{\sigma' \neq \sigma} |\mathcal{K}_0|.\sigma'$$

Soit $c'_0 = \Delta_{\text{nf}} c_0$. Si une cellule σ' présente un coefficient différent de -1 dans c'_0, ce coefficient est obligatoirement inférieur à -1 ; soit $n = |c'_0(\sigma')|$, la valeur absolue de la décoration de σ'. La valeur $n-1$ correspond alors à combien est surévaluée la distance (dans le cas d'une sous-évaluation, notre modèle diverge). Il suffit donc de retrancher $n-1$ à la valeur associée à σ' pour obtenir le résultat. La modification entraîne peut être des changements vis à vis des voisins de σ'. Le calcul doit être alors réitéré. Il vient donc le calcul de la suite :

$$c_{i+1} = c_i + \Delta_{\text{nf}} c_i + \sum_{\sigma \in \mathcal{K}_0} 1.\sigma$$

On retrouve une expression de calcul proche des équations différentielles :

$$\frac{dc}{dt} = \Delta_{\text{nf}} c + 1$$

où l'expression $\frac{dc}{dt}$ est un abus de notation pour $c_{i+1} - c_i$, la variation temporelle de la suite. La figure 3 donne un exemple d'exécution de notre algorithme.

Cet exemple est sans doute naïf mais il illustre bien l'approche que l'on pourrait développer et qui consiste à exprimer sous la forme d'une équation, le résultat à calculer. L'équation caractérise les propriétés du résultat. Les opérateurs intervenant dans cette équation ont une interprétation locale même s'il correspondent à une opération globale (qui agit en parallèle sur chacun des éléments d'une certaine structure de données).

Un des challenge est de déterminer une famille intéressante d'opérateurs (et nous avons vu que les opérateurs de la physique avait une interprétation en terme de mouvement de valeurs dans la structure). Nous postulons que de nombreux problèmes peuvent s'exprimer avec ce type d'opérateurs. Par exemple un algorithme de tri, le bead-sort [Coh04], correspond aussi à un transport simple de valeurs. Les problèmes d'optimisation peuvent aussi sans doute s'exprimer simplement.

Un autre challenge est de savoir résoudre les équations/programmes (et donc incidemment de déterminer si une solution existe, si elle est unique, etc.). Dans l'exemple que nous venons d'esquisser, la résolution peut se faire par une méthode explicite qui ne pose pas de difficultés et qui correspond *in fine* à un calcul de point-fixe.

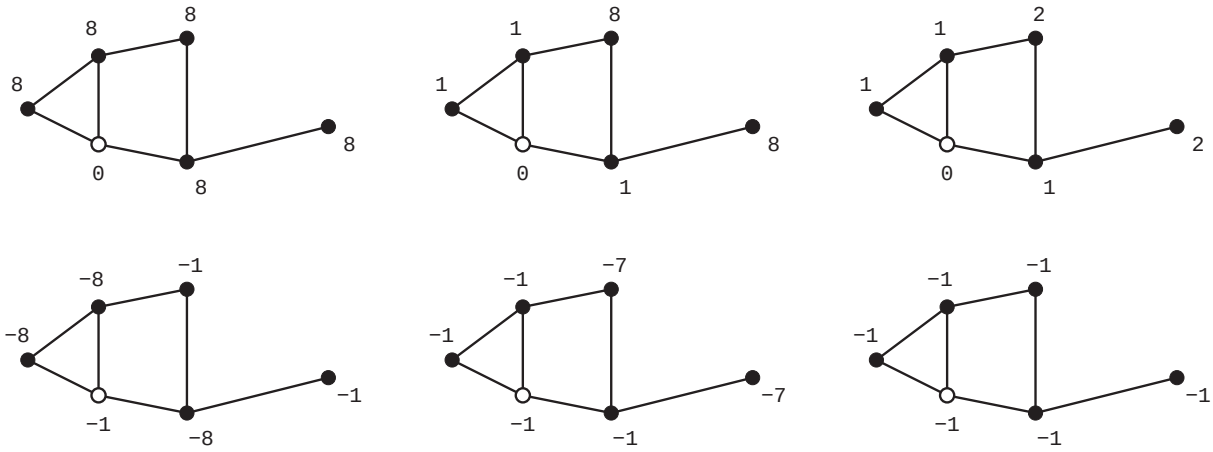


FIG. 3 – Calcul du plus court chemin : en haut, la suite des graphes c_i calculés par l'algorithme, avec de gauche à droite l'état initial où 0 est associé au sommet σ (représenté par un cercle vide), et l'entier 8 (le nombre de sommets dans le graphe) est associé aux autres, l'état intermédiaire où les sommets distants de 1 sont décorés correctement, et l'état final. En bas chaque graphe représente l'application de l'opérateur Δ_{nf} sur le graphe placé au-dessus. Pour l'état final, tous les sommets sont décorés par -1.

Troisième partie

Exemples et applications

Chapitre 7

Introduction aux exemples d'applications

Le matériel présenté dans ce chapitre synthétise des éléments largement présentés dans les articles [GM01b, GM01a, GGMP02b, MGC02, GM02a, Gia03, GMCS04, Coh04, SMG04a, GS06a].

Comme nous l'avons évoqué au début de cette thèse, les motivations du projet MGS sont doubles : répondre aux problèmes posés par la simulation informatique de systèmes dynamiques (SD) et introduire des notions topologiques dans un langage de programmation. Ces deux motivations ne sont pas étrangères l'une à l'autre : un SD correspond à un phénomène qui évolue dans le temps et dans l'espace. Il est donc nécessaire de développer des représentations informatiques de relations spatiales et temporelles. Or le fondement théorique de ces relations repose le plus souvent sur des notions topologiques (notion de trajectoire, de bassin d'attraction, de localité, etc.).

Une des hypothèses du projet MGS est donc que la prise en compte de relations topologiques dans un langage de programmation permet de simplifier la spécification et la simulation des SD et plus particulièrement des SD à structure dynamique (abrégé en (SD)²) [GGMP02b].

Les chapitres qui suivent mettent en évidence que *l'hypothèse a été effectivement validée* : ils sont dédiés à la présentation d'applications non-triviales qui ont été développées en MGS. Les programmes obtenus sont concis et lisibles, comme le lecteur pourra s'en rendre compte par lui-même. Le code MGS « colle » à l'expression naturelle du problème ; les collections topologiques offertes par MGS sont bien adaptées à la représentation d'états complexes dont la structure est

dynamique et les transformations sont un outil puissant et expressif pour spécifier simplement des lois d'évolution locales de manière déclarative.

Mis à part le chapitre 8 qui présente des applications qui relèvent du domaine de l'informatique graphique, de la modélisation géométrique et de la morphogenèse formelle, les chapitres suivants sont dédiés à des applications de simulation de processus physiques et surtout biologiques[1].

Plan du chapitre. Dans ce chapitre, nous allons décrire succinctement le domaine d'application privilégié du projet MGS et nous présentons *l'analyse a priori* qui nous a amenés à concevoir, développer et formaliser les notions de collection topologique et de transformation pour la simulation des (SD)[2].

Nous terminons ce chapitre par les conclusions que nous avons tirées des exemples que nous avons développés.

1 Les besoins de la simulation dans le domaine de la biologie intégrative

La biologie constitue à la fois un domaine d'application croissant (par exemple avec les outils informatiques pour le séquençage du génome et le traitement des données issues de la post-génomique) et une source d'inspiration pour l'informatique [Pat94] : réseaux de neurones, automates cellulaires, algorithmes génétiques, algorithmes évolutionnistes, vie artificielle, calcul par ADN, calcul aqueux, calcul chimique, calcul par membranes... on se reportera au volume [MBFG04] pour un panorama des modèles de programmation récemment inspirés par la biologie.

Un des domaines où l'interaction entre la biologie et l'informatique est la plus riche est celui de la modélisation et de la simulation. Dans le domaine de la *biologie des systèmes*[2], la simulation informatique constitue un formidable outil d'analyse qui permet de vérifier la cohérence du modèle formel avec les hypothèses dérivées des données expérimentales, de rechercher les propriétés particulières de certains ensembles d'interactions, d'étudier la décomposition en sous-systèmes intégrés, de prédire le résultat de perturbations ou de situations nouvelles (mutant, modification de l'environnement, perturbation du métabolisme, ...), et de découvrir de nouveaux modes de régulation.

La simulation n'est pas le seul outil informatique qui peut contribuer aux avancées de la biologie des systèmes : la modélisation logique [TK01], les systèmes de réécriture [GMM04, EKL+02], le model-checking [Ric06], les techniques de validation et de test de systèmes [FMP00], la spécification des systèmes concurrents [Car04], l'optimisation... sont autant de domaines informatiques qui apportent des contributions importantes. Mais dans le cadre du projet MGS, nous ne considérons que la simulation.

[1] Une application importante programmée en MGS n'est pas présentée ici car elle a été développée par Pierre Barbier de Reuille, un chercheur extérieur au projet MGS lors de son travail de thèse [BdRBCL+06, BdR05]. Cette application modélise le transport actif de l'auxine, une hormone végétale, dans le méristème apical de l'arabette (une plante des chemins) et utilise intensivement les collections de Delaunay.

[2] La Biologie des Systèmes est « l'étude itérative et intégrative des systèmes biologiques en tant que systèmes, en réponse à des perturbations » [AIRRH03]. Ce domaine utilise des méthodes mathématiques et informatiques pour reconstruire un modèle physiologique et biochimique intégré à partir des connaissances parcellaires de ses fonctions et de leurs interactions, connaissances hétérogènes issues notamment (mais pas seulement) de la génomique fonctionnelle (glossaire INRA `http://www.jouy.inra.fr/glossaire`).

Les problèmes informatiques qui sont posés par ces nouvelles applications de simulation sont très difficiles et nécessitent de nouveaux concepts pour la représentation des diverses entités biologiques, leur construction modulaire, la construction incrémentale des programmes de simulation, leur validation et l'exploitation des résultats. Une stratégie possible pour faire face à ces problèmes est la conception et le développement d'un *langage de programmation dédié*.

Le développement d'un langage spécifique à un domaine d'application (ou DSL pour *Domain Specific Language*) se justifie par la plus grande facilité de programmation et une meilleure réutilisation et capitalisation des programmes face aux problèmes spécifiques posés par le domaine d'application. On peut donc espérer une meilleure productivité, sûreté, maintenabilité, évolutivité et flexibilité qu'un langage de programmation généraliste.

Un langage dédié aux besoins de la simulation dans le domaine de la biologie intégrative doit permettre l'exploitation de l'énorme masse de données produite par les méthodes de la *biologie à grande échelle* afin de modéliser, de relier et d'intégrer les nombreux réseaux d'interactions intracellulaires aux structures cellulaires et supra-cellulaires (cellules, tissus, organes, compartiments sanguins, etc.) dans le but de rendre compte, comprendre et maîtriser les nombreuses fonctions biologiques et physiologiques. On peut lister plus précisément les problèmes qui se posent de manière particulièrement aiguë dans ce domaine :

- Il y a une explosion combinatoire des entités à spécifier, chacune exhibant de nombreux attributs et comportements différents.
- La spécification de chaque entité biologique regroupe des aspects hétérogènes mais qui peuvent interagir : structure physique, état, spécification de son évolution propre et des interactions avec les autres entités, géométrie (localisation et voisinage)... De plus, ces aspects dépendent dynamiquement de l'état de l'entité biologique elle-même.
- Le système ne peut pas être décrit simplement de manière globale (par exemple à travers un modèle numérique), mais uniquement par un ensemble d'interactions locales entre des entités plus élémentaires.
- La description du système ne peut pas se structurer simplement en termes de hiérarchie. De plus, cette structure est souvent dynamique et doit être calculée conjointement à l'évolution du système.

Les deux derniers points nécessitent de développer des concepts et des techniques de programmation permettant de représenter des processus *dont la structure est dynamique et spatialement distribuée*. Nous avons qualifier ce type de processus de *système dynamique à structure dynamique* ou, en abrégé, $(SD)^2$. La notion de $(SD)^2$ est au cœur du projet MGS et a été exposée par exemple dans [GM01b, GGMP02a, GM02b, GM03, Gia03]. Un exemple paradigmatique de $(SD)^2$ est le développement d'un embryon, ou plus généralement, tous les modèles de morphogenèse un tant soit peu réaliste. Ces applications sont décrites et analysées dans [GS06a] et nous invitons le lecteur à s'y reporter.

2 Les systèmes dynamiques à structure dynamique

Intuitivement, un système dynamique est une façon formelle de décrire comment un point (l'état du système) se déplace dans un *espace des phases* (l'espace de tous les états possibles du système). Il faut donc spécifier une règle, la *fonction d'évolution*, exprimant où doit se trouver le point après sa position courante. Il existe de nombreux formalismes qui sont utilisés pour décrire un SD : équations différentielles ordinaires (EDO), équations différentielles partielles (EDP), couplage discret d'équations différentielles, itérations de fonctions, automates cellulaires... suivant la nature discrète ou continue du temps et de l'espace ainsi que des valeurs utilisées lors de la

modélisation. Des exemples de formalismes, classés suivant le caractère discret ou continu du temps et des variables d'état, sont listés à la table 1. De nombreux SD sont structurés, c'est-à-dire

C : continu, D : discret.	EDO	Itérations de fonctions	Automate à états finis
Temps	C	D	D
Etat	C	C	D

TAB. 1 – Quelques formalismes utilisés pour spécifier les SD suivant le caractère discret ou continu du temps et des variables d'état (repris de [GGMP02b]).

qu'ils peuvent êtres décomposés en multiples parties. De plus, on peut *parfois* exprimer l'état complet s du système comme la simple composition des états de chaque partie. L'évolution de l'état du système entier est alors vu comme le résultat des changements des états de chaque partie. Dans ce cas, la fonction d'évolution h_i de l'état d'une partie o_i dépend uniquement d'un sous-ensemble $\{o_{i_j}\}$ des variables d'état du système complet. Dans ce contexte, nous disons que le SD exhibe une *structure statique* si :

1. l'état du système est décrit statiquement par l'état d'un ensemble fini de ses parties et que cet ensemble ne change pas au cours du temps ;
2. les relations entre les états des parties, spécifiées à travers la définition de la fonction h_i entre les o_i et les arguments o_{i_j}, sont aussi fixées et ne changent pas au cours du temps.

De plus, on dit que les o_{i_j} sont les *voisins logiques* des o_i (et très souvent, deux parties d'un système interagissent quand ils sont *spatialement* voisins). Cette situation est simple et apparaît souvent en physique élémentaire. Par exemple, la chute d'une pierre est décrite statiquement par une position et une vitesse, et cet ensemble de variables ne change pas au cours du temps (même si la *valeur* de la position et la *valeur* de la vitesse changent au cours du temps). Le calcul de la vitesse ne dépend pas de la position alors que le calcul de la position dépend de la position et de la vitesse. Ces dépendances ne changent pas au cours du temps (la fonction d'évolution du système est toujours la même).

Suivant l'analyse développée dans [GGMP02b], on peut remarquer que de nombreux systèmes biologiques peuvent être vus comme des systèmes dynamiques dans lesquels non seulement la valeur des variables d'état, mais aussi l'*ensemble* de ces variables d'état *et/ou* la fonction d'évolution, changent au cours du temps. Ce sont des $(SD)^2$ suivant la terminologie introduite dans [GM01b, GM01a].

Un exemple immédiat de $(SD)^2$ en biologie est donné par le développement d'un embryon. Initialement, l'état du système est décrit par le seul état chimique o_0 de l'oeuf (peu importe la complexité de cet état chimique). Après de nombreuses divisions, l'état de l'embryon n'est plus spécifié par le seul état chimique o_i des cellules, mais aussi par leur arrangement spatial[3]. Le nombre des cellules, leur organisation spatiale et leurs interactions évoluent constamment au cours du développement et ne sont pas le fait d'une unique structure figée $\mathcal{O}$. Au contraire, l'espace des phases $\mathcal{O}(t)$ utilisé pour caractériser la structure de l'état du système au temps t

[3]Le voisinage de chaque cellule est d'une extrême importance dans l'évolution du système du fait de la dépendance entre la forme du système et l'état des cellules. La forme du système a un impact sur la diffusion des signaux physico-chimiques et par conséquent sur l'état des cellules. Réciproquement, l'état de chaque cellule détermine, par exemple en se divisant, l'évolution de la forme de tout le système.

doit être calculé *en même temps* que l'état courant du système. Autrement dit, l'espace des phases doit être conçu comme *une observable* du système.

La notion de $(SD)^2$ est particulièrement évidente en biologie du développement. Mais cette notion est centrale dans toute la biologie et a reçu plusieurs noms différents : *hyper-cycle* dans l'étude des réseaux auto-catalytiques par Eigen et Schuster [ES79], *autopoïese* dans les travaux de Varela sur les systèmes autonomes [Var79], *variable structure system* dans la théorie du contrôle [Itk76, HGH93], *developmental grammar* dans les travaux de Mjolsness [MSR91] ou encore *organisation* dans les travaux de Fontana et Buss sur l'émergence de structures stables dans les systèmes ouverts [FB94].

Cependant, ce type de systèmes ne se rencontre pas uniquement en biologie : la modélisation de réseaux dynamiques (internet, réseaux mobiles), les phénomènes de morphogenèse en physique (croissance dans un médium dynamique), la mécanique des milieux élastiques ou des systèmes déformables, la croissance des villes ou encore les réseaux sociaux regorgent d'exemples de $(SD)^2$.

3 La structure topologique des interactions d'un système

La spécification informatique des $(SD)^2$ pose un problème : *quel est le langage adapté à la définition d'une fonction d'évolution h qui porte sur un état dont on ne peut décrire la structure à l'avance ?*

Le système ne peut pas être décrit simplement de manière globale mais uniquement par un ensemble d'**interactions** locales entre des entités plus élémentaires qui composent le système. Notre problème est de définir ces entités élémentaires et leurs interactions. Nous allons voir que ces interactions exhibent une structure topologique et que l'on peut en tirer parti pour les définir.

Le point de départ de cette analyse[4] est la décomposition d'un système *en fonction de son évolution.* À un instant t donné, nous découpons un système S en plusieurs sous-systèmes $S_1, \ldots, S_n$ disjoints tels que le prochain état $s_i(t+1)$ du sous-système S_i ne dépend uniquement que du précédent état $s_i(t)$. Autrement dit, chaque sous-système S_i évolue indépendamment entre un instant t et l'instant $t+1$ suivant. On parle parfois de *boîte* pour désigner les S_i : une boîte regroupe l'ensemble des éléments en interaction dans une évolution locale [Rau03]. Cette notion de boîte rend compte d'une encapsulation : lorsque l'évolution se réalise, seul ce qui est dans la boîte a besoin d'être connu.

La décomposition de S en les S_i est une partition *fonctionnelle* qui peut correspondre, mais pas nécessairement, à un découpage structurel du système en composants. Remarquons que nous prenons ici le contre-pied d'une « approche objet » qui commence par décrire les constituants structurels d'un système avant de définir leurs interactions ; ici notre point de départ sont les activités du système, et nous tentons d'en déduire une décomposition.

La décomposition fonctionnelle de S en S_i doit dépendre du temps. En effet, si la partition en S_i ne dépendait pas du temps, on aurait une collection de systèmes parallèles, complètement autonomes et n'interagissant pas. Il n'y aurait alors aucune nécessité à les considérer simultanément pour constituer un système intégré. Par conséquent, on écrit $S_1^t, S_2^t, \ldots, S_{n_t}^t$ pour la décomposition du système S au temps t et on a : $s_i(t+1) = h_i^t(s_i(t))$ où les h_i^t sont les « fonctions d'évolution locales » des S_i^t. Par commodité, on suppose qu'à un instant t donné, l'un des S_i^t représente la partie du système « qui n'évolue pas » (et la fonction d'évolution associée est l'identité).

L'état « global » $s(t)$ du système S peut être retrouvé à partir des états « locaux » des sous-systèmes : il existe une fonction φ^t telle que $s(t) = \varphi^t(s_1(t), \ldots, s_{n_t}(t))$ qui induit une relation

[4] Cette analyse a été présentée dans [GMCS04], une version écourtée a été publiée dans [Gia04] et une reformulation, dont nous reprenons en grande partie les éléments, a été développée dans [Coh04].

entre la fonction d'évolution « globale » h et les fonctions d'évolution locale :

$$s(t+1) = h(s(t)) = \varphi^{t+1}(h_1^t(s_1(t)), \ldots, h_{n_t}^t(s_{n_t}(t)))$$

Si l'on suit cette analyse, la spécification d'un (SD)² doit passer par la définition de trois entités :

- la partition dynamique de S en S_i^t,
- les fonctions h_i^t,
- la fonction φ^t.

La description des décompositions successives $S_1^t, S_2^t, \ldots, S_{n_t}^t$ peut s'appuyer sur la notion de *parties élémentaires* du système : un sous-système arbitraire S_i^t sera composé de parties élémentaires. Plusieurs partitions de S en parties élémentaires sont possibles ; aussi, on s'intéresse ici à une décomposition induite naturellement par l'ensemble des S_i^t.

Deux sous-systèmes quelconques S' et S'' de S interagissent (au temps t) s'il existe un S_j^t tel que $S', S'' \in S_j^t$. Deux sous-systèmes S' et S'' sont *séparables* s'il existe un S_j^t tel que $S' \in S_j^t$ et $S'' \notin S_j^t$ ou réciproquement. Cela nous amène à considérer l'ensemble $\mathcal{S}$ défini comme le plus petit ensemble clos par intersection contenant les S_j^t (voir figure 1). Nous nommerons cet ensemble la *structure d'interaction* de S. Les éléments de $\mathcal{S}$ sont des ensembles. Les éléments de $\mathcal{S}$ qui ne contiennent aucun autre élément de $\mathcal{S}$ correspondent aux parties élémentaires recherchées.

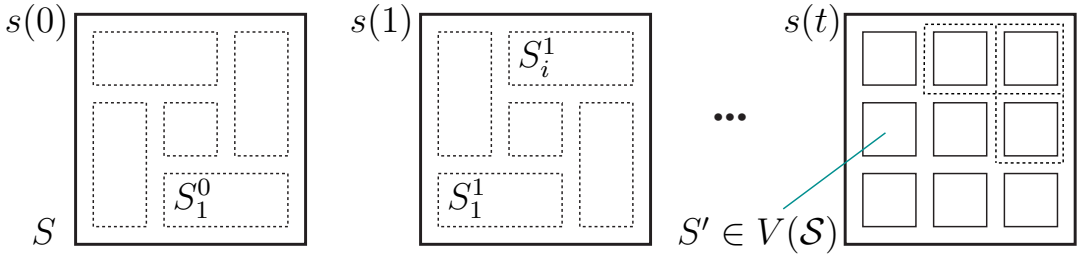


FIG. 1 – La structure d'interaction d'un système S résultant des sous-systèmes des éléments en interactions à un pas de temps donné.

L'ensemble $\mathcal{S}$ possède une *structure topologique* naturelle : $\mathcal{S}$ correspond à un *complexe simplicial abstrait*. Cette notion est une spécialisation de la notion de complexe cellulaire présentée dans le chapitre 3. Un complexe simplicial abstrait [Hen94] est une collection $\mathcal{C}$ d'ensembles finis non-vides tels que si A est un élément de $\mathcal{C}$, alors il en est de même pour tout sous-ensemble non-vide de A. L'élément A de $\mathcal{C}$ est appelé un *simplexe* de $\mathcal{C}$; sa *dimension* est égale au nombre de ses éléments moins un. La dimension de $\mathcal{C}$ est la plus grande dimension d'un de ses simplexes. Tout sous-ensemble non-vide de A est appelé une *face*. On définit aussi l'*ensemble des sommets* $V(\mathcal{C})$, comme l'union des ensembles à un élément de $\mathcal{C}$.

La correspondance entre $\mathcal{S}$ et un complexe $\mathcal{C}$ est la suivante : un élément de $\mathcal{S}$ est un simplexe et les parties élémentaires de $\mathcal{S}$ correspondent aux sommets de $\mathcal{C}$. Nous identifions donc $\mathcal{S}$ avec un complexe abstrait. La correspondance entre le complexe abstrait $\mathcal{S}$ et la décomposition fonctionnelle de S est la suivante : les sommets de $\mathcal{S}$ correspondent aux parties élémentaires du système S. Les éléments de $\mathcal{S}$ qui ne sont pas une face d'un autre élément, appelés simplexes maximaux, correspondent aux S_i^t. La dimension d'un simplexe maximal correspond au nombre (diminué de 1) de parties élémentaires impliquées dans une interaction.

La topologie est l'étude de la connexité : deux objets sont isomorphes d'un point de vue topologique si on peut mettre en bijection leurs parties tout en conservant les relations de connections

entre ces parties. Ici, deux parties sont connectées si elles interagissent lors d'une évolution du système. C'est donc une notion très abstraite de voisinage qui est capturée par la structure topologique de $\mathcal{S}$. Par définition, seules des entités « spatialement proches » interagissent. Cependant, très souvent, l'espace abstrait des interactions correspond à l'espace physique concret (il n'y a pas d'action à distance). Il n'est donc pas étonnant que le voisinage logique décrit par $\mathcal{S}$ recoupe souvent le voisinage spatial des parties physiques du système, comme on le verra dans les nombreux exemples des chapitres suivants.

4 Structures de données et structures de contrôle pour la simulation des (SD)²

L'analyse précédente nous montre qu'il est possible de spécifier un (SD)² S en spécifiant :

- les S_i^t *comme une composition de simplexes* de $\mathcal{S}$,
- en associant à chaque S_i^t une fonction h_i^t,
- et en combinant les applications de chaque h_i^t à travers une fonction φ^t.

Cette approche peut sembler inutilement abstraite mais elle s'interprète très simplement en terme de programmation :

- L'idée est de définir directement l'ensemble $\mathcal{S}$ comme une structure de données. Une structure de données doit donc se caractériser par la relation de voisinage qui organise ses éléments (les sommets de $\mathcal{S}$).
- Une fonction h_i^t permet alors de définir le devenir d'une partie de la structure de données. L'association d'un S_i^t et d'une fonction d'évolution locale h_i^t peut naturellement s'écrire par une règle :
$$S_i^t \Rightarrow h_i^t(S_i^t)$$
- Pour être générique, la partie gauche de la règle ne doit pas correspondre à une partie fixée du système, mais doit spécifier les parties élémentaires qui interagissent pour évoluer suivant h_i^t.
- Les différentes applications de règles à un instant donné doivent être contrôlées et les différents résultats doivent être recombinés pour construire le nouvel état du système. De ce point de vue, la fonction φ^t correspond à la fois à une stratégie d'application des règles et à la notion de substitution utilisée pour appliquer les règles (mécaisme que nous avons aussi appelé *reconstruction* dans les chapitres précédents).

On retrouve les notions de collection topologique et de transformation que nous avons précédemment introduites : une collection topologique correspond à une structure $\mathcal{S}$ et une transformation topologique à la définition des h_i^t, à leur domaine d'application et à la fonction de substitution φ^t.

Notons que dans une transformation, la spécification des motifs et les fonctions h_i^t ne varient pas au cours du temps. En revanche, les parties filtrées S_i^t peuvent varier. De plus, le passage d'un pas de temps élémentaire correspond à l'application d'une transformation et il est possible de changer de transformation pour l'application suivante.

5 L'approche MGS pour la simulation des (SD)²

Nous pouvons à présent synthétiser l'approche MGS pour la simulation des (SD)².

Une collection topologique représente l'état d'un système dynamique à un instant donné. Les éléments de la collection peuvent représenter des entités (un sous-système ou une partie atomique d'un système dynamique) ou des messages (signaux, commandes, informations, actions, ...) adressés à d'autres entités. Une sous-collection représente un sous-ensemble d'entités en interaction ainsi que des messages du système.

L'évolution du système est spécifiée par une transformation, où typiquement les parties gauches d'une règle filtrent une entité et un message qui lui est destiné, et où la partie droite spécifie le changement d'état de l'entité, ainsi que possiblement d'autres messages adressés à d'autres entités.

La relation de voisinage permet de prendre en compte plusieurs types d'interaction. Par exemple, si l'on utilise une organisation de multi-ensemble pour la collection, les entités interagissent de façon non-structurée : tout élément de la collection est susceptible d'interagir avec tout autre élément. Un multi-ensemble réalise donc une espèce de « soupe chimique ». Des collections topologiques plus organisées sont utilisées pour représenter des organisations spatiales et des interactions plus sophistiquées (voir les exemples dans les chapitres suivants).

Plus généralement, de nombreux modèles mathématiques d'objets et de processus sont fondés sur une notion d'état qui spécifie l'objet ou le processus en attribuant certaines données à chaque point d'un espace physique ou abstrait. Le langage de programmation MGS facilite cette approche en offrant de nombreux mécanismes permettant la construction d'espaces complexes et en évolution ainsi que la gestion des relations entre ces espaces et ces données.

6 Bilan des exemples de programmes MGS

Nous présentons ici une conclusion que nous avons tirée des exemples développés dans les chapitres suivants. Un des enjeux de ces exemples était en effet de valider les langages de motifs disponibles en MGS. Les langages de motifs permettent de décrire les interactions entre parties élémentaires d'un système. L'implantation du filtrage est une partie lourde. Certains motifs résultent en un filtrage intrinsèquement inefficace tandis que d'autres peuvent être optimisés. Il était donc important d'analyser les motifs utilisés dans les exemples, ainsi que leur expressivité. Il apparaît que les exemples peuvent se classer en trois catégories :

Applications algorithmiques. Ces exemples nécessitent des motifs de chemins complexes (utilisant par exemple l'itération `*`). Ils relèvent de « l'algorithmique classique ».

Ces applications ne sont pas présentées dans cette thèse. Nous avons développés plusieurs algorithmes sur les graphes en MGS lors de notre stage de DEA [Spi03] : calcul de flot maximal, de chemin, etc. On trouvera aussi dans les références [GM01a, GM01b, Coh04, MJG04, MJ05] des exemples de calculs sur les multi-ensembles (calcul de factorielle, calcul de l'enveloppe convexe d'un ensemble de points), le crible d'Eratosthène, la mise en forme conjonctive ou disjonctive de formules logiques, le tri à bulle ou par boulier, la recherche de chemin dans un labyrinthe, l'analyse de protocoles (recherche de solutions dans un grand espace d'états). Le spectre des applications algorithmiques couvertes est donc très large.

Applications de simulation à structure statique. Ces applications utilisent dans leur grande majorité un motif très simple. En fait, les règles sont très souvent de la forme :

`x =>` φ`(Neighborfold(f, zero, x)`$, \ldots$`)`

Ces applications correspondent à des simulations explicites de systèmes dynamiques à structure statique. La fonction d'évolution est locale. On a simplement besoin de combiner la valeur des voisins avec la valeur du point courant. Ces motifs peuvent se complexifier un

peu afin de modéliser l'évolution d'une partie finie connexe et simple du système. C'est le cas par exemple des modèles de croissance comme le modèle d'Eden ou de gaz sur réseau. On trouvera des exemples de ce type dans [Coh04] : par exemple, un processus de diffusion et de diffusion-réaction. Nous avons développé des exemples de diffusion limitée par agrégation et d'automates cellulaires sur des maillages arbitraire publiés dans [SMG04a].

Application de simulation à structure dynamique. Enfin, la dernière classe d'applications est mixte. C'est dans cette classe que se rangent les applications présentées dans les chapitres suivants. Ce sont des applications qui correspondent souvent à des simulations de $(SD)^2$ ou bien à des algorithmes qui impliquent un changement de structure (e.g. subdivision de surface). Dans cette classe :

- certaines règles sont simples et correspondent à l'évolution explicite du système n'impliquant pas de réorganisation ;
- les règles restantes sont compliquées et décrivent un changement de structure topologique ;
- pour ces règles, le langage des motifs de chemin est généralement insuffisant pour décrire les parties en interaction du système.

Cette classification est bien sûr à affiner mais elle montre bien qu'un langage de motifs sophistiqués est nécessaire. Si le langage de motifs de chemin est suffisant pour beaucoup d'applications algorithmiques, il ne suffit plus pour la spécification des changements de structures, ce qui a motivé le développement du langage de patches. Cependant, on peut remarquer qu'une classe importante d'applications se contente de motifs très simples : il faut donc savoir repérer et traiter efficacement ces motifs. Il est d'ailleurs possible de capitaliser ces « patrons de traitements » récurrents dans des opérateurs spécialisés (ce qui motive d'ailleurs l'étude des opérateurs du chapitre 6). Nous reviendrons sur ces constations dans la conclusion de cette thèse.

Chapitre 8

Modifications topologiques

Les exemples développés dans ce chapitre correspondent aux principales motivations du développement des collections topologiques et des transformations de dimension arbitraire. Ils illustrent l'utilisation des patches pour la spécification d'opérations topologiques sur des maillages tridimensionnels. En particulier, ils répondent par la positive à la question d'un langage déclaratif spécialisé dans la description locale de tels processus posée dans [SPS04].

Le premier exemple propose une implantation en MGS de plusieurs algorithmes classiques de subdivision de maillage. Il s'agit d'une opération importante dans le domaine de la modélisation géométrique et du rendu graphique qui consiste à raffiner des objets géométriques représentés par des maillages. La subdivision est une opération souvent simplement décrite informellement : elle est expliquée sous la forme d'insertions de sommets sur des arcs et de créations de nouveaux arcs. Les sommets sont également plongés dans l'espace de telle sorte que le raffinement affiche une courbure cohérente (les algorithmes de subdivision sont étudiés pour faire tendre le raffinement vers des objets continus). Cette opération n'est pas globale et demande une manipulation souvent compliquée et *ad-hoc* de la structure de données représentant le maillage. Avec MGS, les collections topologiques fournissent un point de vue suffisamment général de la structure pour que la description informelle de la subdivision soit traduite quasi-automatiquement en transformations. Nous illustrons notamment le fait que l'implantation sous forme de transformations des algorithmes de subdivision est indépendant du type de collection. La transformation ainsi obtenue s'applique indifféremment aussi bien à des chaînes abstraites qu'à des G-cartes.

Le second exemple s'intéresse à la programmation en MGS de la construction d'une fractale : le triangle de Sierpinski. Cet exemple est un prétexte pour comparer deux approches du concept d'auto-assemblage, qui nous permet ici de générer la fractale. Dans un premier temps, nous proposons une construction du triangle sur un espace prédéfini, une matrice, à travers l'auto-assemblage d'un motif à la façon d'un automate cellulaire. Nous illustrons ici le concept d'*auto-assemblage par agrégation*. Nous utilisons les collections GBF qui se prêtent à la spécification d'espaces homogènes. Dans notre seconde approche, le problème est inversé : nous construisons l'espace correspondant à la fractale en en supprimant des parties. Il s'agit d'*auto-assemblage par découpage*. Nous utilisons alors les patches et les chaînes abstraites pour représenter la structure du triangle de Sierpinski.

1 Subdivision de maillage

La définition et la génération de courbes et de surfaces lisses spécifiées à partir d'un petit nombre de points de contrôle est un problème fondamental en modélisation géométrique. Une approche possible se fonde sur l'idée de *subdivision* qui consiste à remplacer itérativement une représentation grossière par une représentation plus fine. Introduits par Chaikin en 1974 [Cha74] les algorithmes de subdivision pour les courbes et les surfaces se sont depuis multipliés. Si ces algorithmes peuvent se décrire de manière très intuitive par des opérations locales agissant sur un point et ses voisins, leur formulation et leur implantation sont souvent compliquées par les notations indexées (vecteurs et tableaux) habituellement utilisées dans les programmes. Cette complication justifie le développement d'approches implicites, *i.e.* qui ne font pas référence à des séquences indexées de points. Cependant la définition d'un cadre à la fois déclaratif et implicite n'a réellement abouti que pour la subdivision de courbes [PSSK03].

Nous nous intéressons ici à la subdivision de surfaces. Les algorithmes de subdivision génèrent à la limite des surfaces lisses en itérant des subdivisions de maillages polygonaux. Dans [Zor00], on trouve une description détaillée des processus de subdivision utilisés pour la modélisation et l'animation. Nous nous sommes inspirés de ce chapitre pour organiser notre exemple.

Les algorithmes de subdivision sont spécifiés localement à l'aide de *masques* [Zor00] ; il s'agit de parties de complexe cellulaire décrivant une partie d'un maillage centrée sur un élément à raffiner (un arc ou une face) pour lequel un nouveau sommet est créé. Les coordonnées de ce sommet sont déterminées par une combinaison affine des positions des sommets apparaissant dans le masque. Les propriétés de continuité de la surface obtenue en itérant jusqu'à la limite l'algorithme de subdivision, diffèrent selon le masque utilisé. La qualité du masque dépend de ces propriétés : on cherche à construire des surfaces aussi lisses que possible (C^1-continues ou C^2-continues partout par exemple). Il est difficile de réaliser cette propriété sur des maillages arbitraires, ceux-ci présentant des irrégularités situées en des points singuliers. Un masque est donc choisi suivant le type de maillage ou de propriétés à vérifier. De façon plus précise, on trouve dans [Zor00] une classification des algorithmes de subdivision. Ils sont caractérisés par

- **le type du raffinement** :
  - *par insertion de sommet*, où un sommet est inséré sur chaque arc du maillage pour le diviser en deux arcs ; les anciens sont conservés, et les nouveaux sommets sont liés entre eux. La figure 1 montre cette opération pour des maillages triangulaires et quadrangulaires (on note que pour ce dernier type de maillage, un nouveau sommet est également inséré sur chaque quadrilatère).
  - *par arrondi des coins et des arêtes*, où chaque face est remplacée par une face équivalente en forme et en nombre de côtés, plus petite, placée en son milieu. Cette opération a pour effet de transformer chaque arc et sommet en une nouvelle face : les parties anguleuses

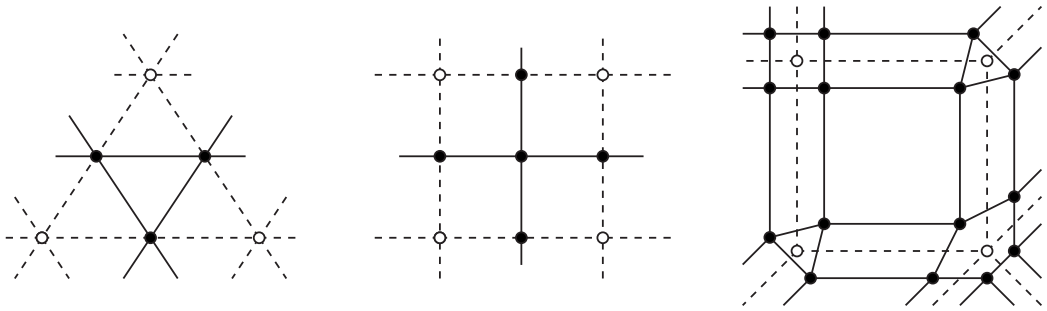


FIG. 1 – Subdivision : à gauche, par insertion de sommets sur un maillage triangulaire. Au centre par insertion de sommets sur un maillage quadrangulaire ; les traits en pointillés et les cercles vides correspondent au maillage initial, les traits et cercles pleins aux nouveaux objets. À droite, par arrondi des sommets et des arêtes. Les traits en pointillés et les cercles vides correspondent au maillage initial, seuls les traits et cercles pleins composent le nouvel objet.

de l'objet raffiné sont donc arrondies. La figure 1 décrit l'arrondi.

- **le type de maillage généré** : *triangulaire*, composé de triangles, ou *quadrangulaire*, composé de carrés ;
- **le type de masque** ; cette caractéristique concerne les subdivisions conservant les sommets du maillage initial pour former le maillage raffiné :
  - *masque approximant* : les anciens sommets voient leur position dans l'espace modifiée et les sommets insérés sont placés en fonction de la position des anciens sommets dans le maillage initial.
  - *masque interpolant* : les anciens sommets conservent les mêmes coordonnées et les sommets insérés sont placés pour créer une courbure.

Cette classification donne naissance au tableau suivant associant un exemple d'algorithme typique pour chacune de ces caractéristiques :

	Insertion de sommet	
	Maillage triangulaire	*Maillage quadrangulaire*
Approximant	Loop	Catmull-Clark
Interpolant	Butterfly modifié	Kobbelt

Arrondi des coins
Doo-Sabin

Les sous-sections suivantes développent l'utilisation de MGS et des patches pour programmer ces cinq algorithmes. Nous détaillons tout d'abord la représentation des objets en MGS, puis nous décrivons en particulier la programmation de la subdivision Loop, les masques utilisés étant les plus simples et les méthodes d'implantation des autres algorithmes étant similaires. Enfin, nous simplifions également nos exemples en nous restreignant au développement des algorithmes pour des objets convexes et sans bord.

1.1 Représentation des objets en MGS

Les objets géométriques sont représentés en MGS aussi bien par des chaînes abstraites (de type `achain`) que par des G-cartes (de type `qmf`). Ces structures de données MGS sont présentées dans le chapitre 2, respectivement pages 36 et 38.

Nous souhaitons représenter des surfaces composées de polygones. Il s'agit donc de complexes cellulaires de dimension 2. Les 2-cellules (appelées également *faces* dans ce chapitre) sont décorées

par le symbole `'face`. Les arcs (1-cellules) sont décorés par le symbole `'edge` ; on remarque en particulier que la manipulation de solides impose que chaque arc soit bordé par deux faces (les arcs situés aux bords sont incidents à une seule face ; nous ne les considérons pas dans notre implantation).

Les sommets portent une information plus structurée, représentée par l'enregistrement suivant :

```
record vertex = { x:float,
                  y:float,
                  z:float,
                  d:int,
                  n:int
                } ;;
```

dont les champs `x`, `y` et `z` codent les coordonnées du sommet qu'ils décorent, l'entier `d` est le degré du sommet (c'est-à-dire le nombre d'arcs incidents ; cette valeur est conservée pour ne pas être recalculée à chaque itération ; elle correspond au cardinal de l'ensemble des cofaces pouvant être calculé directement pour un sommet `v` par l'expression `size(cofaces(v))`), et l'entier `n` encode la génération à laquelle le nœud a été créé. En effet, certains algorithmes demandent à distinguer les anciens et les nouveaux sommets. On utilise alors ce champ : soit `gen`, l'entier codant la génération courante, un ancien sommet aura un champ `n` strictement inférieur à `gen`.

Pour simplifier la description des programmes MGS, nous définissons une fonction `addCoord` faisant la somme des coordonnées de deux points :

```
fun addCoord(p1,p2) = (
  { x = p1.x + p2.x,
    y = p1.y + p2.y,
    z = p1.z + p2.z
  }
) ;;
```

ainsi qu'une variable globale dénotant le neutre de cette opération :

```
0 := { x = 0.0, y = 0.0, z = 0.0 } ;;
```

1.2 Subdivision Loop

La subdivision Loop [Loo87] raffine un maillage triangulaire par insertion de sommet : la modification du maillage est fondée sur la subdivision polyédrique présentée figure 2. Un sommet est inséré sur chaque arc ; les triangles sont raffinés en 4 triangles plus petits. Tous les nouveaux sommets créés possèdent six 1-voisins.

Les masques. Les masques de Loop fournissent les informations nécessaires pour positionner les nouveaux sommets en fonction des coordonnées des anciens, et pour déplacer les anciens sommets (la subdivision Loop est approximante) :

- **insertion des nouveaux sommets :** la figure 3 décrit le calcul de la position du nouveau sommet. Soit un arc e à subdiviser ; e est bordé par les sommets v_1 et v_2. Les sommets v_3 et v_4 sont les deux sommets 1-voisins de v_1 et de v_2. Un nouveau sommet v est créé dont les coordonnées sont données par la somme pondérée :

$$v = \frac{3}{8}(v_1 + v_2) + \frac{1}{8}(v_3 + v_4) \quad (1)$$

```
trans <0,0> even_vertex = {
  v => (
    let mask = ccellsfold(addCoord, 0, v, 1) in
    let k    = v.d in
    let beta = β in
      v + { ox = v.x,
            oy = v.y,
            oz = v.z,
            x  = (1-k*beta)*v.x + beta*mask.x,
            y  = (1-k*beta)*v.y + beta*mask.y,
            z  = (1-k*beta)*v.z + beta*mask.z
          }
  )
} ;;


patch odd_vertex[gen] = {
  ~v1 < e:[dim=1, (^f1,^f2) in cofaces] > ~v2
  ~v1 < ~e13 > ~v3 < ~e23 > ~v2
  ~v1 < ~e14 > ~v4 < ~e24 > ~v2
  => (
    ‘v:[ dim = 0, cofaces = (‘e1,‘e2), value = { n = gen, d = 6 } + v ]
    ‘e1:[ dim = 1, faces = (‘v,^v1), cofaces=(^f1,^f2), value = ‘edge ]
    ‘e2:[ dim = 1, faces = (‘v,^v2), cofaces=(^f1,^f2), value = ‘edge ]
  )
} ;;


patch subdivideFace[gen] = {
  f:[ dim = 2, (^e1,^e2,^e3,^e4,^e5,^e6) in faces ]
  ~v1 < ~e1 > ~v2:[ v2.n == gen ] < ~e2 >
  ~v3 < ~e3 > ~v4:[ v4.n == gen ] < ~e4 >
  ~v5 < ~e5 > ~v6:[ v6.n == gen ] < ~e6 > ~v1
  => (
    ‘a1:[ dim = 1, faces = (^v2,^v4), cofaces = (‘f4,‘f1), value = ‘edge ]
    ‘a2:[ dim = 1, faces = (^v4,^v6), cofaces = (‘f4,‘f2), value = ‘edge ]
    ‘a3:[ dim = 1, faces = (^v6,^v2), cofaces = (‘f4,‘f3), value = ‘edge ]

    ‘f1:[ dim = 2, faces = (‘a1,^e2,^e3), value = ‘face ]
    ‘f2:[ dim = 2, faces = (‘a2,^e4,^e5), value = ‘face ]
    ‘f3:[ dim = 2, faces = (‘a3,^e6,^e1), value = ‘face ]
    ‘f4:[ dim = 2, faces = (‘a1,‘a2,‘a3), value = ‘face ]
  )
}
```

Prog. 1: La subdivision Loop en MGS : les variables β et v sont données par les équations 1 et 3.

L'arc e est raffiné en deux nouveaux arcs e_1 et e_2.

- **déplacement des anciens sommets :** pour chaque sommet v, possédant k 1-voisins v_i, v est déplacé aux coordonnées v' données par :

$$v' = (1 - k\beta)v + \beta \sum_{i=1}^{k} v_i \tag{2}$$

$$\beta = \frac{1}{k}\left(\frac{5}{8} - \left(\frac{3}{8} + \frac{1}{4}\cos\frac{2\pi}{k}\right)^2\right) \tag{3}$$

L'implantation en MGS. Avant toute chose, la subdivision Loop nécessite un déplacement des sommets du maillage. Afin de réaliser l'insertion des nouveaux sommets, il faut conserver la position des anciens sommets avant leur déplacement. Nous sauvegardons donc dans l'enregistrement associé à chaque sommet, sa position dans le maillage courant et nous mettons à jour sa position dans le nouveau maillage, et cela au moyen de la transformation `even_vertex`[1] (voir le programme 1) : celle-ci est composée d'une seule règle MGS, réécrivant chaque sommet `v` de telle sorte que les nouvelles coordonnées soient placées dans les champs `x`, `y` et `z` de l'enregistrement et les anciennes soient conservées dans les champs `ox`, `oy` et `oz`. L'accès aux coordonnées des sommets 1-voisins à `v` est fait par l'expression `ccellsfold(addCoord,0,v,1)` équivalente à :

```
fold(addCoord, 0, ccells(self,^v,1))
```

La liste des enregistrements associés aux cellules 1-voisines est calculée par `ccells(self,^v,1)` (on rappelle que l'expression `ccells(^v,1)` fournit les positions 1-voisines et non les valeurs qui leur sont associées) ; cette liste est ensuite réduite à l'aide de la fonction d'ordre supérieur `fold` effectuant la somme des coordonnées.

La sauvegarde et le déplacement étant effectués, il suffit de programmer la subdivision polyédrique. Cependant, bien qu'elle soit aisément dessinée *localement* (voir figure 2), la modification topologique *déborde* sur les faces voisines d'une face subdivisée. Elles doivent donc également être traitées. Nous supposons dans cet exemple que l'objet entier est raffiné[2]. Les règles MGS décrivant des opérations locales, il est impossible de programmer la règle schématique de

[1] L'expression *even vertex* provient de la nomenclature standard utilisée dans la description d'algorithme de subdivision, et désigne un arc nouvellement créé. On trouve dans [Zor00] une justification pour cette expression.

[2] Pour plus de détails sur les subdivisions partielles ou adaptatives, nous invitons le lecteur à s'intéresser à [Zor00, Kob00].

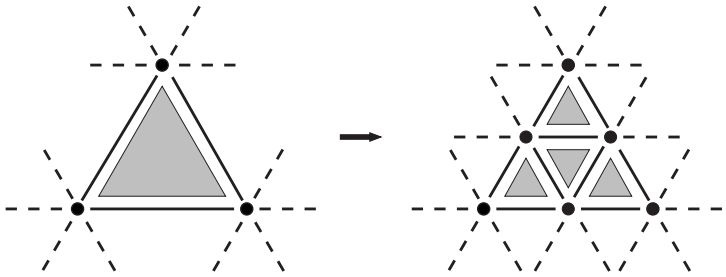


FIG. 2 – Subdivision polyédrique, un patron pour les algorithmes Loop et Butterfly modifié.

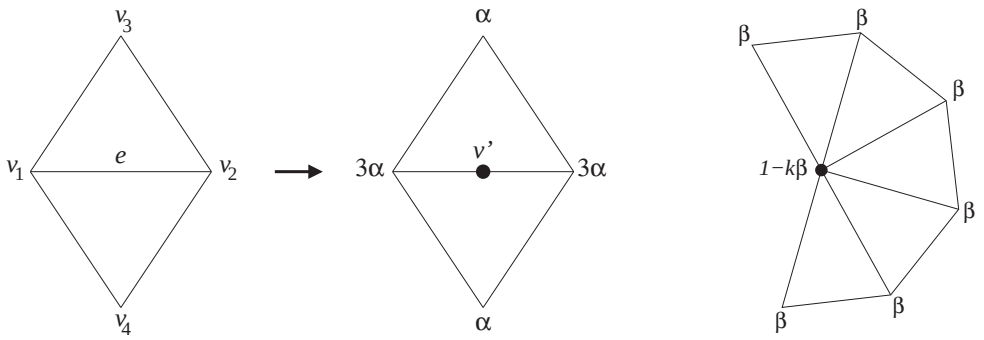


FIG. 3 – Masque pour la subdivision Loop : à gauche, l'insertion des nouveaux sommets ($\alpha = \frac{1}{8}$), à droite le déplacement des anciens sommets (β est donné par l'équation 3).

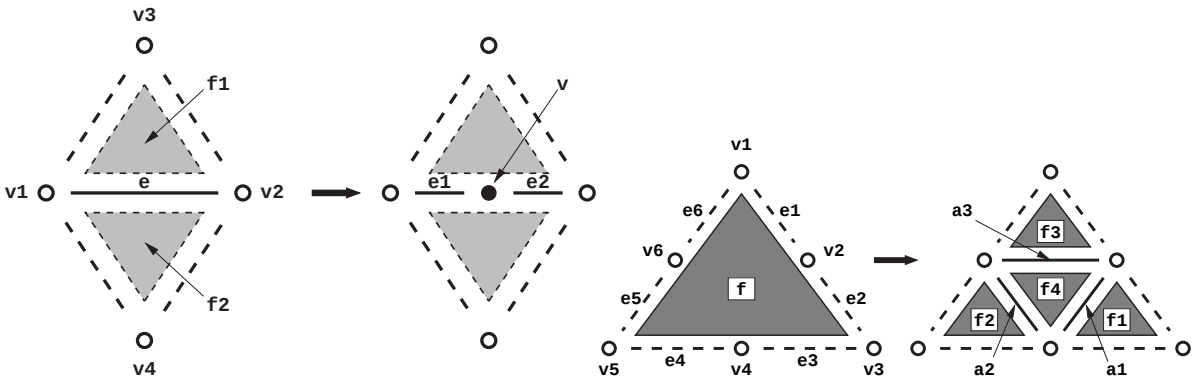


FIG. 4 – Subdivision Loop : représentation schématique des règles de réécriture des patches `odd_vertex` et `face` (voir le programme 1). En haut, l'insertion des sommets respectant le masque de Loop ; en bas, la création des 4 triangles raffinés. Les 0-cellules, 1-cellules et 2-cellules consommées ou reconstruites sont respectivement des cercles pleins, des traits pleins et des polygones foncés avec un bord plein ; les 0-cellules, 1-cellules et 2-cellules non-consommées sont respectivement des cercles vides, des traits pointillés et des polygones clairs avec un bord pointillé.

la figure 2. C'est pourquoi, comme dans l'implantation de la subdivision polyédrique de [SPS04], nous procédons suivant deux étapes :

1. L'insertion de sommet sur chaque arc suivant le masque de Loop est réalisé par le patch `odd_vertex` (voir le programme 1) : il filtre chaque arc comme décrit figure 4. Les coordonnées du nouveau sommet `'v` sont données à l'aide de l'équation 1 pour laquelle on prend soin d'utiliser les champs de sauvegarde `ox`, `oy` et `oz` pour les coordonnées de v_1, v_2, v_3 et v_4. On associe également au nouveau sommet sa génération avec la définition du champ `n` dans l'enregistrement prenant la valeur de la génération courante `gen`, et son degré `d` de valeur 6 ; en effet, bien que chaque sommet inséré ne soit incident pour l'instant qu'à deux voisins (v_1 et v_2), l'étape qui suit en rajoutera quatre. Cette opération transforme chaque triangle en hexagone.

2. La subdivision de chaque hexagone en quatre triangles est décrite figure 4 et réalisée par le patch `subdivideFace` du programme 1. Ce patch est composé d'une règle filtrant et consommant une 2-cellule, et filtrant sans consommer son bord (on remarque l'utilisation

de l'opérateur `~`), en distinguant les sommets nouvellement insérés des anciens par un prédicat sur leur génération : les sommets `^v2`, `^v4` et `^v6` vérifient la condition `v.n == gen` où `gen` est la génération courante. En partie gauche, la 2-cellule est remplacée par les trois arcs internes `‘a1`, `‘a2` et `‘a3` dont les bords sont `^v2`, `^v4` et `^v6`, ainsi que par quatre nouvelles faces triangulaires `‘f1`, `‘f2`, `‘f3` et `‘f4`.

1.3 La subdivision Butterfly

La modification topologique de la subdivision Butterfly [DLG90] est la même que pour Loop : un sommet est inséré sur chaque arc, puis, les triangles raffinés sont construits à partir des hexagones ainsi créés. Les deux subdivisions diffèrent par le placement des sommets du nouveau maillage. Pour la subdivision Butterfly, les anciens sommets ne sont pas déplacés (il s'agit d'un schéma interpolant). Le placement des nouveaux sommets suit des masques différents de ceux de Loop. Ces masques sont présentés figure 5.

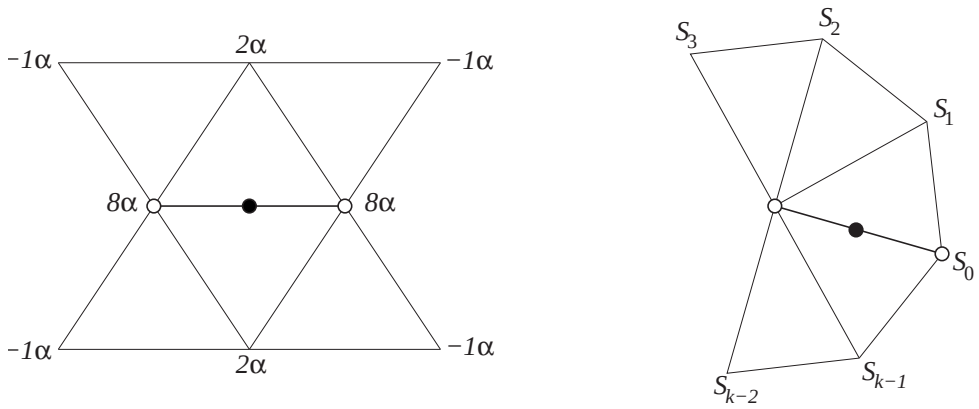


FIG. 5 – Masques pour la subdivision Butterfly modifiée : à gauche pour les sommets réguliers ($\alpha = \frac{1}{16}$), à droite pour les sommets singuliers (les S_i sont donnés par les équations 6). L'arc en gras bordé par deux cercles vides représente l'arc à raffiner. Le sommet ajouté est figuré par un cercle plein.

L'implantation de la subdivision Butterfly requiert une attention particulière par rapport à la régularité du maillage, ce qui n'est pas le cas pour la subdivision Loop[3] : dans un maillage triangulaire régulier, les sommets sont de degré 6 (à l'exception des bords) ; ces sommets sont dits *réguliers*. On les oppose aux sommets *singuliers* possédant un nombre de 1-voisins différent de 6. Si le maillage initial possède des points singuliers, ceux-ci doivent être traités spécialement pour conserver les propriétés de continuité désirées. On programme donc l'insertion par plusieurs règles de réécriture selon les cas suivants :

- Arcs aux sommets réguliers : le masque de Loop est étendu ; soient v_5, v_6, v_7 et v_8 les quatres sommets supplémentaires pris en compte (voir figure 5). Les coordonnées du sommet v créé entre v_1 et v_2 sont données par :

$$v = \frac{1}{2}(v_1 + v_2) + \frac{1}{8}(v_3 + v_4) - \frac{1}{16}(v_5 + v_6 + v_7 + v_8) \tag{4}$$

[3]Dans [Zor00], on précise tout de même que le masque ne peut être utilisé sur des maillages possédant des sommets de degré supérieur à 7 sans perdre la propriété de C^1-continuité assurée pour les autres maillages. Un masque différent est proposé pour ces cas particuliers et pour ainsi rétablir la propriété, mais les différences ne sont pas visibles sur les rendus graphiques.

- Arcs dont l'un des sommets est singulier : soient v le sommet singulier, v_0 l'extrémité opposée de l'arc à décomposer, et v_i $(1 \leq i < k)$ les autres sommets 1-voisins de v. L'entier k représente donc le degré de v ($k \neq 6$ pour que v soit singulier). On associe à chaque sommet v_i $(0 \leq i < k)$ un coefficient S_i (voir figure 5) permettant le calcul de la position du nouveau sommet à créer entre v et v_0 :

$$v' = (1 - \sum_{i=0}^{k} S_i)v + \sum_{i=0}^{k} S_i v_i \tag{5}$$

$$\begin{cases} S_0 = \frac{5}{12},\ S_{1,2} = -\frac{1}{12} & \text{si } k = 3 \\ S_0 = \frac{3}{8},\ s_2 = -\frac{1}{8},\ S_{1,3} = 0 & \text{si } k = 4 \\ S_i = \frac{1}{k}\left(\frac{1}{4} + \cos\frac{2i\pi}{k}\right) + \frac{1}{2}\cos\frac{4i\pi}{k}\right) & \text{si } k \geq 5 \end{cases} \tag{6}$$

- Arcs dont les deux sommets sont singuliers : dans ce cas, le calcul du point précédent est fait indépendamment pour chacun des sommets ; les coordonnées finales sont la moyenne de ces deux résultats intermédiaires.

Bien que plus longue, l'implantation de ces masques en MGS est très proche du programme 1. Nous ne la donnons donc pas. Ces deux approches sont comparées sur un exemple figure 6 ; ces images sont réalisées par MGS. Toutes les sorties graphiques que nous présentons sont générées par le programme MGS que nous décrivons. Une fonction, non donnée dans ce manuscrit, permet d'exporter dans un format dédié la structure des collections : les sommets sont positionnés selon les coordonnées calculées par l'algorithme. L'affichage et l'exportation au format EPS sont assurés par un logiciel compagnon développé en parallèle du projet MGS, appelé Imoview.

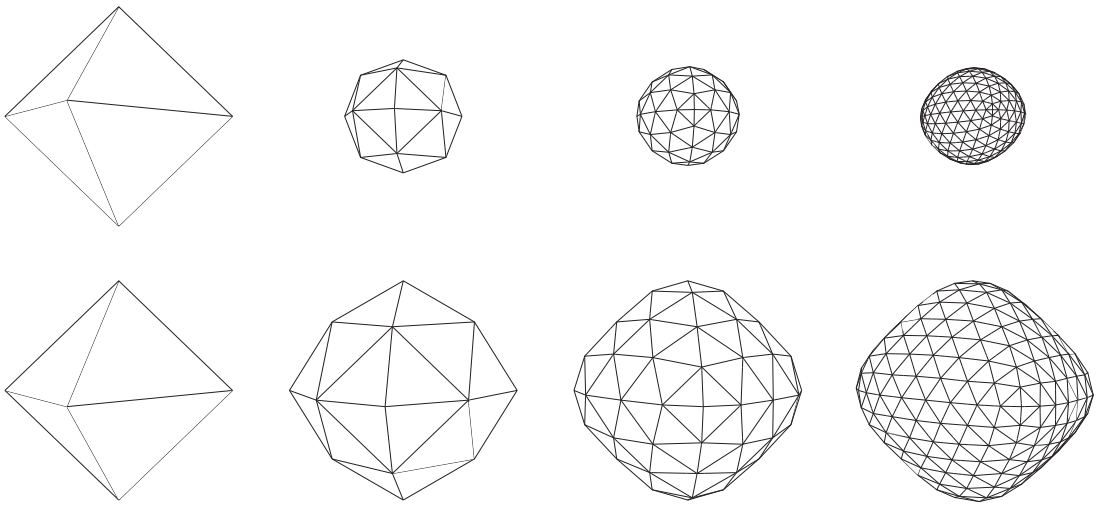


FIG. 6 – Résultats d'applications des algorithmes de Loop (en haut) et Butterfly (en bas) : de gauche à droite, l'objet initial puis raffiné jusqu'à 3 itérations de l'algorithme.

1.4 Subdivision Catmull-Clark et Kobbelt

Ces deux schémas [CC78, Kob96] raffinent un maillage quadrangulaire par insertion de sommet pour chaque face et chaque arc ; la modification du maillage est décrite figure 7. Les techniques d'implantation en MGS sont proches de celles pour les subdivisions Loop et Butterfly.

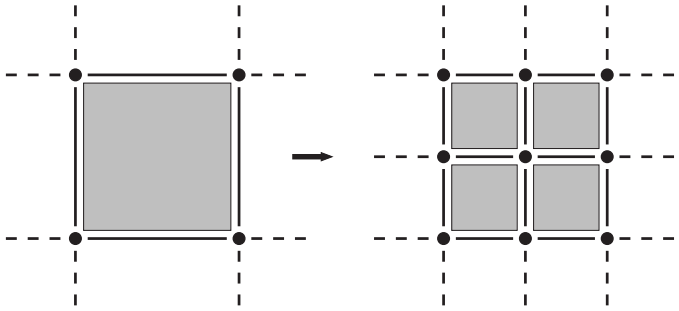


FIG. 7 – Modification topologique de maillage pour les algorithmes Catmull-Clark et Kobbelt.

```
trans <2,2> face_vertex = {
  f => let G = icellsfold(..., 0, f, 0) in G + { faceval = f }
} ;;

patch edge_vertex[gen] = {
  ~v1 < e:[dim=1, (^f1,^f2) in cofaces] > ~v2
  ... le masque correspondant
  => (  ‘v:[ dim = 0, value = ... ]
        ‘e1:[ dim = 1, faces = (‘v,^v1), cofaces=(^f1,^f2), value = ‘edge ]
        ‘e2:[ dim = 1, faces = (‘v,^v2), cofaces=(^f1,^f2), value = ‘edge ]
     )
} ;;

patch subdivideFace[gen] = {
  f:[dim=2 , (^e1,^e2,^e3,^e4,^e5,^e6,^e7,^e8) in faces ]
  ~v1 < ~e1 > ~v2:[ dim=0 , (v2.n==idx) ] < ~e2 >
  ...
  ~v7 < ~e7 > ~v8:[ dim=0 , (v8.n==idx) ] < ~e8 > ~v1
  => ( ‘v:[ dim = 0, value = { x = f.x, y = f.y, z = f.z, n = gen, d = 4} ]
       ‘e2:[ dim = 1 , face=(‘v,^v2), value = ‘edge ]
       ‘e4:[ dim = 1 , face=(‘v,^v4), value = ‘edge ]
       ‘e6:[ dim = 1 , face=(‘v,^v6), value = ‘edge ]
       ‘e8:[ dim = 1 , face=(‘v,^v8), value = ‘edge ]
       ‘f1:[ dim = 2 , face=(^e1,‘e2,‘e8,^e8), value = ... ]
       ‘f2:[ dim = 2 , face=(^e2,‘e2,‘e4,^e3), value = ... ]
       ‘f4:[ dim = 2 , face=(^e4,‘e4,‘e6,^e5), value = ... ]
       ‘f3:[ dim = 2 , face=(^e6,‘e6,‘e8,^e7), value = ... ]
     )
} ;;
```

PROG. 2: Les subdivisions Catmull-Clark et Kobbelt en MGS.

Encore une fois, le modification de maillage schématisée figure 7 n'est pas strictement locale (la frontière des faces adjacentes à la face d'intérêt sont modifiés). Plusieurs étapes sont donc nécessaires à sa réalisation. Nous ne détaillons pas les différents masques (une fois de plus, le lecteur intéressé se référera à [Zor00]) mais nous présentons les différents patches réalisant la modification topologique :

- **Insertion des sommets**. Deux types de nouveaux sommets apparaissent :
  1. pour chaque face : un sommet est créé en son milieu. Les coordonnées de ce sommet sont calculées par la transformation `face_vertex` du programme 2. Nous utilisons pour cela l'itérateur `icellsfold` qui donne accès aux coordonnées des sommets incidents à la face `^f` filtrée et qui est paramétré par une fonction particulière dépendant du masque utilisé. Créer seul ce sommet n'aurait pas de sens. Ses coordonnées sont donc conservées en les associant à la face. Nous construisons pour cela l'enregistrement `G + { faceval = f }` contenant, en plus des coordonnées `G`, l'ancienne valeur `f` associée à la face `^f`. Le sommet sera réellement construit par le patch `subdivideFace` du programme 2.
  2. pour chaque arc : un sommet est créé pour le diviser en deux. Nous définissons pour cela le patch `edge_vertex` du programme 2 qui, suivant le masque utilisé, construit et place dans l'espace un nouveau sommet. Cette étape transforme chaque carré du maillage en octogone.
- **Subdivision des octogones**. De façon totalement équivalente au patch `subdivideFace` du programme 1, le patch `subdivideFace` du programme 2 construit pour chaque octogone `f` filtré, le nouveau sommet `'v` en son centre en récupérant les coordonnées calculées par l'application de la transformation `face_vertex`, les arcs `'e1`, `'e2`, `'e3` et `'e4` liant `'v` aux sommets de `^f`, et finalement, les quatre carrés raffinés `'f1`, `'f2`, `'f3` et `'f4`.

La figure 8 compare l'utilisation des subdivisions Catmull-Clark et Kobbelt ; ces images sont réalisées par MGS.

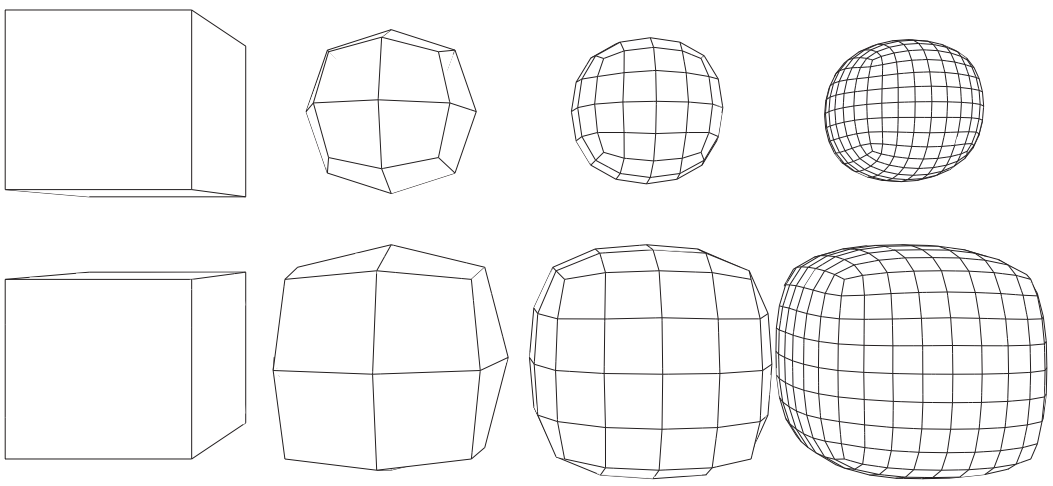
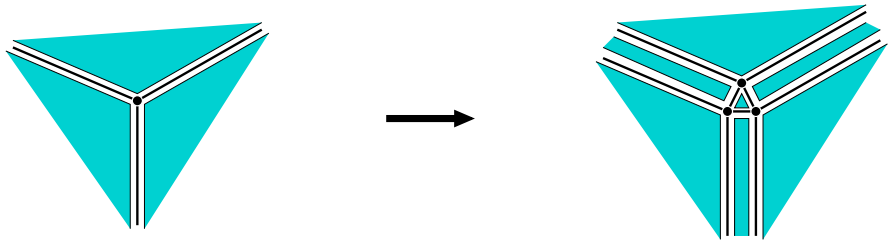


FIG. 8 – Résultats d'applications des algorithmes Catmull-Clark (en haut) et Kobbelt (en bas) : de gauche à droite, l'objet initial puis raffiné jusqu'à 3 itérations de l'algorithme.

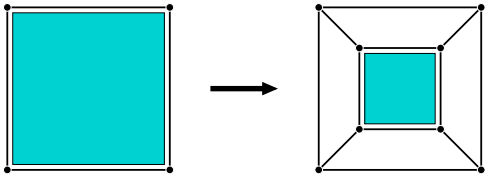
1.5 La subdivision Doo-Sabin

Il s'agit du dernier algorithme de la classification. Il s'apparente à l'arrondi d'objets géométriques, appelé aussi communément *chanfreinage* [Led02]. Dans cette subdivision [DS78], chaque cellule (de n'importe quelle dimension) est transformée en une 2-cellule. Pour un cube par exemple, les arêtes et les sommets sont respectivement biseautés en rectangles et en triangles :

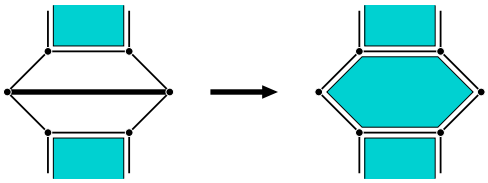


Afin de réaliser cette modification du maillage, trois étapes s'appliquent sur chaque dimension de telle sorte que tout élément du maillage initial est remplacé ; l'objet final est composé de cellules « fraîches » :

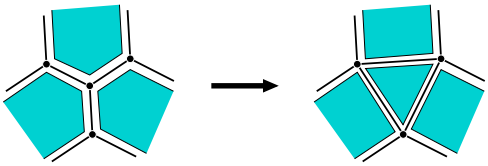
- Suppression des 2-cellules : chaque face est remplacée par un polygone identique de taille inférieure. Les nouveaux sommets de ce polygone sont liés par un arc aux anciens sommets.



- Suppression des 1-cellules : dans le maillage initial, chaque arc est au bord de deux faces. Sachant que l'opération précédente construit une copie des arcs pour chacune des faces, chaque arc initial dans le maillage intermédiaire est isolé comme figuré ci-contre (en gras). Ces arcs sont alors détruits et remplacés par un hexagone.



- Suppression des 0-cellules : soit un sommet du maillage initial de degré n ; après les deux premières étapes, le sommet est commun à n hexagones. Le sommet est alors détruit pour être remplacé par un polygone à n côtés. La figure suivante illustre cette suppression pour un sommet de degré 3 (ce qui est le cas dans la première application de l'algorithme Doo-Sabin sur un cube).



La figure 9 illustre l'utilisation de la subdivision Doo-Sabin pour chanfreiner un cube ; ces images sont réalisées par MGS.

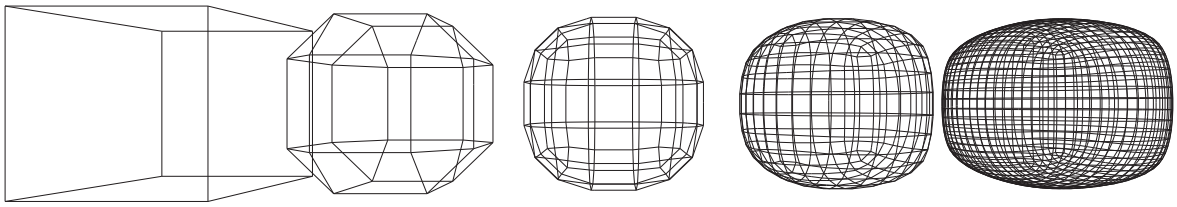


FIG. 9 – Résultats de l'application de l'algorithme Doo-Sabin : de gauche à droite, l'objet initial et ses raffinements jusqu'à 4 applications de l'algorithme.

2 Auto-assemblage de fractales

L'*auto-assemblage* (*self-assembling* en anglais) est un processus qui crée de façon incrémentale des structures spatiales complexes. On trouve dans la nature un grand nombre d'exemples de systèmes auto-assemblés : de la cristallisation en physique jusqu'aux processus de morphogenèse en biologie. Il n'existe aucune théorie générale et unifiée concernant l'auto-assemblage ; néanmoins, en comprendre les principes permettrait d'élargir nos possibilités technologiques, en particulier dans le domaine des nanotechnologies.

Dans cette section, les systèmes auto-assemblés sont vus comme des agrégats de briques élémentaires (molécules, cellules, etc.) dont le comportement est généralement simple, et qui se coordonnent localement pour former une entité dont la structure et le comportement sont complexes.

Du point de vue informatique, les processus d'auto-assemblage sont une source d'inspiration particulière. L'organisation dynamique de leurs constituants émerge de façon décentralisée à partir d'interactions uniquement locales, se produisant en parallèle et à différentes échelles de temps et d'espace. Ils ont en particulier inspiré de nouveaux modèles de calcul comme le *calcul amorphe* [AAC+00] ou le *calcul autonome* [Hor01].

L'*émergence* d'une structure globale d'un système auto-organisé ne peut être induit à partir des propriétés de ses composants élémentaires. La simulation permet bien souvent, même si ce n'est pas le seul moyen, d'obtenir des informations détaillées sur la construction de ces systèmes complexes. Néanmoins, la modélisation et la simulation de phénomènes d'auto-assemblage sont difficiles à réaliser, à cause de la représentation de l'espace et de la manipulation de structures spatiales dans cet espace.

Un thème central de la recherche sur les processus d'auto-assemblage concerne l'étude des principes organisationnels qui peuvent être utilisés pour structurer une population à partir de briques élémentaires. Dans cette partie, nous nous concentrons sur deux types d'auto-assemblage fondés sur l'ajout ou la suppression d'éléments. Nous ne considérons pas l'auto-organisation par déplacement de matière ou par déformation (avec étirement par exemple). Enfin, notre point de vue sur l'auto-assemblage est de nature *combinatoire*. Nous distinguons deux formes de processus d'auto-assemblage : par *accrétion* et par *découpage*.

Auto-assemblage par accrétion. Il s'agit de l'une des formes d'auto-assemblage les plus fondamentale où les briques élémentaires s'unissent pour former une structure complexe par *processus de croissance*. Un processus de croissance incrémental peut être décrit par l'itération

d'agrégations élémentaires : à chaque étape de la croissance, de nouvelles entités sont ajoutées en fonction de l'état de la croissance à l'issue de l'étape précédente [Kaa92]. L'agrégation dépend des entités disponibles et de la forme de la structure en construction à chaque étape.

L'expression « croissance par accrétion » caractérise en particulier les processus de croissance localisés aux frontières du système. Ce type de croissance s'oppose à une « croissance intercalaire » située à l'intérieur du système.

Auto-assemblage par découpage. Manca *et al.* ont introduit une forme de calcul non-conventionnel appelée *calcul par découpage*[4] [MMVP99], dont l'idée est de générer un (large) ensemble de solutions à un problème à partir d'un ensemble duquel sont supprimés les candidats ne résolvant pas le problème. Ce principe de suppression des éléments non désirables permet également de construire des espaces complexes par découpage. Sur ce principe, [KS00] propose par exemple un algorithme de construction d'un volume en accord avec un ensemble de photos d'une forme tridimensionnelle. Transposé au domaine de l'auto-ensemblage, cela conduit à supprimer de façon itérative, des parties d'une structure à partir d'une forme initiale simple. Peut être le terme d'*auto-désassemblage* serait-il plus approprié.

Pour illustrer ces deux formes d'auto-assemblage, nous proposons de construire de façon itérative et suivant ces deux concepts, un espace fractal : le triangle de Sierpinski. On profite également de cette dualité pour comparer l'utilisation de deux types différents de collections topologiques : les collections GBF fournissant un espace homogène et les chaînes abstraites pour la construction d'espaces et de structures arbitraires.

2.1 Croissance par accrétion du triangle de Sierpinski

Le triangle de Sierpinski (TS dans la suite du document) est une figure fractale décrite formellement par Waclaw Sierpinski (mathématicien polonais 1882-1969) en 1915 mais qui est apparue dans l'art italien dès le XIII^{e} siècle (par exemple, il est représenté dans les mosaïques Cosmati de la cathédrale d'Anagni en Italie). L'appellation « dentelle de Sierpinski » est également utilisée (*Sierpinski gasket* ou *Sierpinski sieve* en anglais [Ste95]). Le TS peut être produit en itérant le morphisme bidimensionnel défini sur $\{0,1\}$ par $0 \longrightarrow \begin{smallmatrix}0&0\\0&0\end{smallmatrix}$ et $1 \longrightarrow \begin{smallmatrix}1&0\\1&1\end{smallmatrix}$. Partant de 1, on obtient :

$$1 \longrightarrow \begin{smallmatrix}1&0\\1&1\end{smallmatrix} \longrightarrow \begin{smallmatrix}1&0&0&0\\1&1&0&0\\1&0&1&0\\1&1&1&1\end{smallmatrix} \longrightarrow \begin{smallmatrix}1&0&0&0&0&0&0&0\\1&1&0&0&0&0&0&0\\1&0&1&0&0&0&0&0\\1&1&1&1&0&0&0&0\\1&0&0&0&1&0&0&0\\1&1&0&0&1&1&0&0\\1&0&1&0&1&0&1&0\\1&1&1&1&1&1&1&1\end{smallmatrix} \longrightarrow \ldots$$

De façon équivalente, cette matrice peut être construite à partir du triangle de Pascal dont les coefficients sont considérés modulo 2 (voir figure 10). La formule des coefficients binômiaux est $P(0,j) = 1$, $P(i,j) = 0$ si $i > j$ et $P(i,j) = P(i-1,j-1) + P(i-1,j)$ sinon.

Nous construisons le GBF à deux dimensions suivant (voir chapitre 2 page 33 pour plus de détails sur les collections GBF)

```
gbf Grid2 = < sud, ouest >
```

pour traduire cette formule « modulo 2 » avec la transformation

```
trans ST1 = { <undef> |sud> x |ouest> y => (x+y) mod 2, x, y }
```

[4] pour *computation by carving* en anglais.

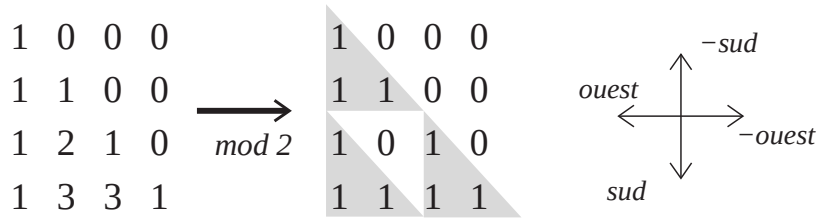


FIG. 10 – Prendre les coefficients binômiaux modulo 2 produit la forme du TS.

Dans cette règle, la virgule est spécialisée à l'aide d'un générateur de `Grid2` : `a |sud> b` signifie que `b` est voisin de `a` suivant la direction `sud`. Cette transformation peut être itérée sur une maillage initial où les positions $(0, j)$ sont décorées par l'entier 1 et les positions $(i, 0)$ par 0. Le résultat est donné figure 11.

Cependant, cette transformation utilise des opérateurs arithmétiques (`+` et `mod`). Ce modèle ne correspond pas à un processus de croissance physique : les entiers 0 et 1 doivent être considérés comme des symboles et non comme des valeurs sur lesquelles un calcul arithmétique peut être appliqué. Pour éviter ces opérations, l'unique règle précédente peut être capturée par quatre règles différentes correspondant aux quatre possibilités de calcul que peut décliner la règle de la transformation `ST1`. La nouvelle transformation s'écrit :

```
trans ST2 = {
    <undef> |sud> 0 |ouest> 0  =>  0, 0, 0
    <undef> |sud> 0 |ouest> 1  =>  1, 0, 1
    <undef> |sud> 1 |ouest> 0  =>  1, 1, 0
    <undef> |sud> 1 |ouest> 1  =>  0, 1, 1
}
```

Ce calcul plus élémentaire est proche de la spécification de la construction du TS par un processus de pavage où les tuiles sont des fragments d'ADN, présentée dans [RPW04]. Ce travail introduit quatre tuiles correspondant aux deux valeurs booléennes qu'une cellule (i, j) du maillage reçoit des cellules $(i-1, j-1)$ et $(i-1, j)$. Ce pavage est facilement implanté en MGS : nous utilisons quatre symboles `'T00`, `'T10`, `'T01` et `'T11` référant aux quatre tuiles. La tuile `'T`xy à la position (i, j) signifie que x est la valeur $P(i-1, j)$ et y est celle $P(i-1, j-1)$. Ainsi, l'entier 0 (resp. 1) précédent est encodé soit par `'T00` ou `'T11` (resp. `'T10` ou `'T01`). Finalement, les quatre règles de la transformation deviennent :

```
trans ST3 = {
    <undef> |south> ('T00|'T11) as x |west> ('T01|'T10) as y
    => 'T01, x, y

    <undef> |south> ('T00|'T11) as x |west> ('T00|'T11) as y
    => 'T11, x, y

    ... deux règles symétriques supplémentaires ...
}
```

Dans [RPW04], les tuiles correspondent à des molécules et le processus d'auto-assemblage est la cristallisation. Dans ce type de phénomène, à un moment donné, seule une partie des sites acceptent effectivement une molécule entraînant la croissance de la structure. La stratégie maximale parallèle de MGS ne correspond pas à ce type de modèle.

Le caractère non-déterministe de [RPW04] est dû aux erreurs et à la cinétique chimique de la cristallisation. Ce non-déterminisme peut être modélisé à l'aide de la stratégie probabiliste de MGS. Comme le montre la figure 11, l'utilisation de l'une ou l'autre des stratégies n'affecte pas l'état final de la simulation, mais les états intermédiaires diffèrent durant la croissance du TS.

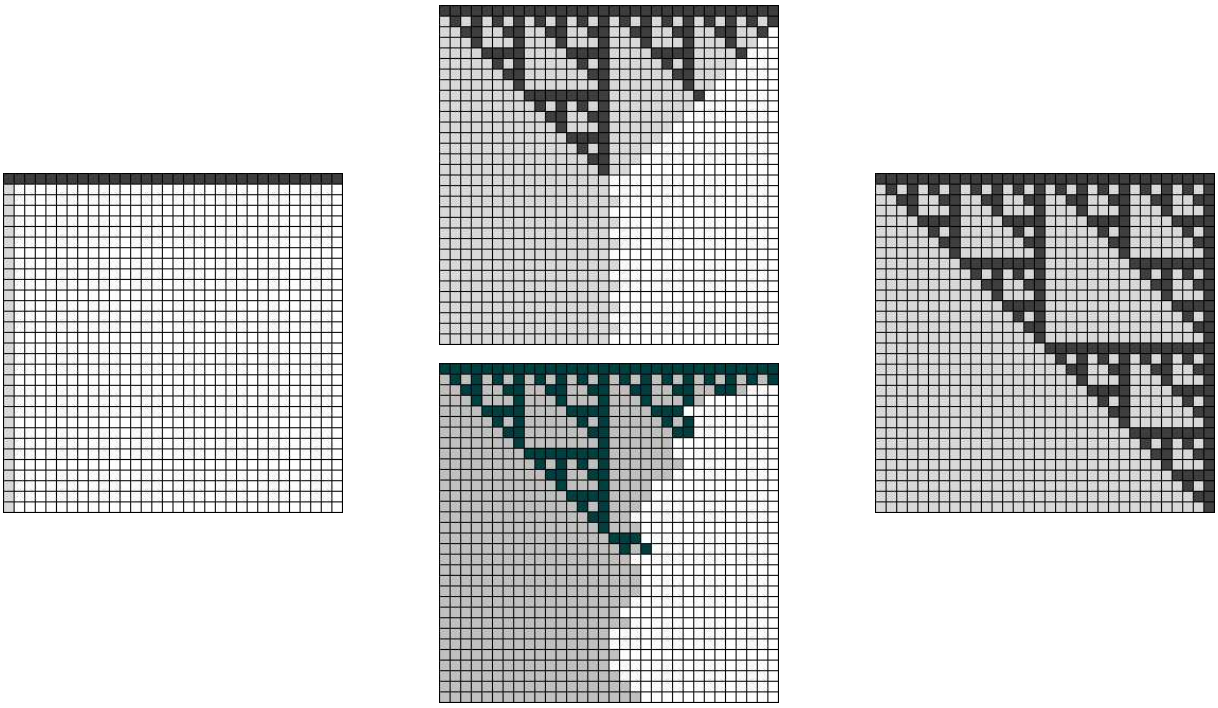


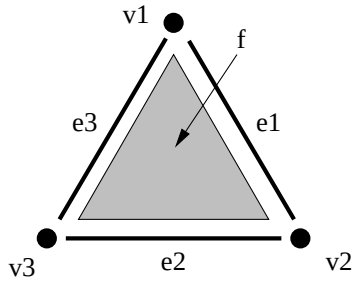
FIG. 11 – Croissance du TS sur une collection GBF : les quatre figures présentent la croissance de la structure sur une grille de type `Grid2`. *À gauche*, l'état initial ; *à droite*, l'état final ; *au centre en haut*, une étape intermédiaire pour une simulation utilisant la stratégie maximale parallèle de MGS ; *au centre en bas*, utilisation de la stratégie stochastique (on note la frontière irrégulière de la structure). Sur les grilles, les carrés blancs correspondent à la valeur `<undef>`, les carrés gris clair et gris foncé sont respectivement les entiers 0 et 1. L'état initial est composé d'une ligne horizontale de 1 et d'une colone verticale de 0 représentant les « bords » sur lesquels va s'appuyer le pavage.

2.2 Croissance par découpage du triangle de Sierpinski

Dans cette seconde implantation, nous utilisons pour représenter le TS, un complexe cellulaire abstrait de dimension 2 où les sommets sont décorés par des coordonnées afin de plonger la structure dans le plan. Le principal avantage des complexes cellulaires est qu'ils permettent la manipulation de cellules topologiques de dimension arbitraire représentant ainsi tous les éléments constituant le TS. En fait, dans la représentation précédente, le TS est un motif apparaissant sur une matrice de 0 et de 1, c'est-à-dire un espace prédéfini. Ici, la structure concrète et géométrique du TS est spécifiée et il constitue également l'espace dans lequel le TS est embarqué. Construire le TS par découpage signifie alors : itérer la suppression de la partie centrale d'un triangle. La limite du processus conduit à la structure fractale du TS (voir figure 12).

L'état initial consiste en un triangle. Il correspond à l'entier 1 de la représentation matricielle du TS et est composé de trois 0-cellules, trois 1-cellules et une 2-cellule :

```
let v1 = add_cell(0,[],[])
and v2 = add_cell(0,[],[])
and v3 = add_cell(0,[],[])
and e1 = add_cell(1,[v1,v2],[])
and e2 = add_cell(1,[v2,v3],[])
and e3 = add_cell(1,[v3,v1],[])
and f  = add_cell(2,[e1,e2,e3],[]) ;;
```



Le découpage, seule étape du processus de construction, correspond à la *chirurgie topologique* décrite figure 13. Le patch MGS correspondant est donné par le programme 3.

La fonction `average` qui n'est pas détaillée ici, calcule les coordonnées du milieu de chaque arc. Dans ce patch, tous les éléments sont filtrés et consommés, à l'exception des sommets faisant toujours partie de la structure après l'application de la règle. Il est important de noter qu'à n'importe quelle étape, chaque arc possède une seule coface ; cette propriété est facilement vérifiable suivant un raisonnement inductif sur chaque étape : en effet, la règle crée 9 nouveaux arcs possédant chacun une seule coface. Ce type de preuve (qui représente une problématique que nous n'adressons pas dans ce document) est standard sur les systèmes de réécriture et permet une vérification simple d'un certain nombre de propriétés des processus auto-assemblés modélisés par de tels systèmes.

Le patch `CRV` est long ce qui rend sa lecture difficile (cf. programme 3 page 215). De plus, il dépend fortement du fait que chaque arc a une unique coface. En effet, si nous appliquons `CRV` sur un autre état initial comme présenté figure 14, le résultat obtenu n'est pas forcément le résultat attendu. Dans cet exemple, deux triangles partagent un arc. Si nous appliquons `CRV` sur l'un des deux triangles, l'autre triangle est également modifié en un quadrilatère.

Pour s'assurer que le résultat ne dépend pas de l'état initial, le processus de construction est divisé en deux transformations distinctes, proches des transformations de la subdivision polyédrique (voir figure 15) :

```
patch AV [gen] = {
  ~v1 < e:[dim = 1] > ~v2
  => ‘v:[dim=0, cofaces=(‘e1,‘e2), val=average(v1,v2,gen)]
     ‘e1:[dim=1, faces=(^v1,‘v), val=‘edge]
     ‘e2:[dim=1, faces=(^v2,‘v), val=‘edge]
}
```

L'étape suivante remplace tous les hexagones par trois triangles (voir figure 15) :

```
patch RF [gen] = {
  f:[dim=2, (e1,e2,e3,e4,e5,e6) in faces]
  ~v1 < ~e1 > ~v2:[v2.n == gen] < ~e2 >
  ~v3 < ~e3 > ~v4:[v4.n == gen] < ~e4 >
  ~v5 < ~e5 > ~v6:[v6.n == gen] < ~e6 > ~v1
  => ‘e24:[dim=1, faces=(v2,v4)]
     ‘e46:[dim=1, faces=(v4,v6)]
     ‘e62:[dim=1, faces=(v6,v2)]
     ‘f1:[dim=2, faces=(e6,e1,‘e62)]
     ‘f2:[dim=2, faces=(e2,e3,‘e24)]
     ‘f3:[dim=2, faces=(e4,e5,‘e46)]
}
```

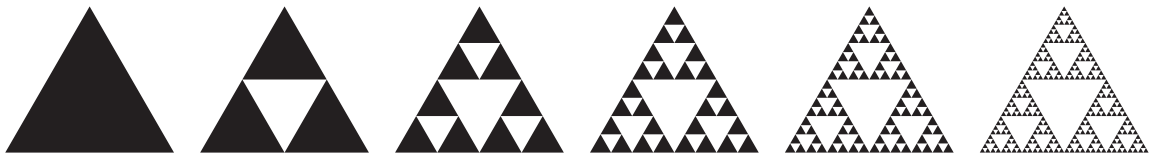


FIG. 12 – Le TS peut également être produit par itérations successives du découpage d'un triangle dans un autre.

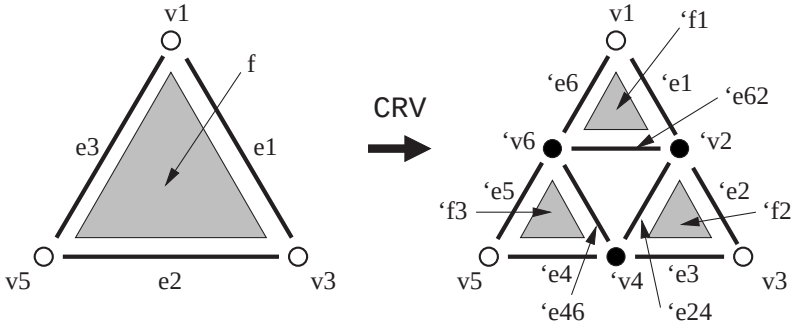


FIG. 13 – Modification topologique du TS : le patch divise chaque triangle en 3 plus petits, laissant un trou triangulaire au milieu de la structure.

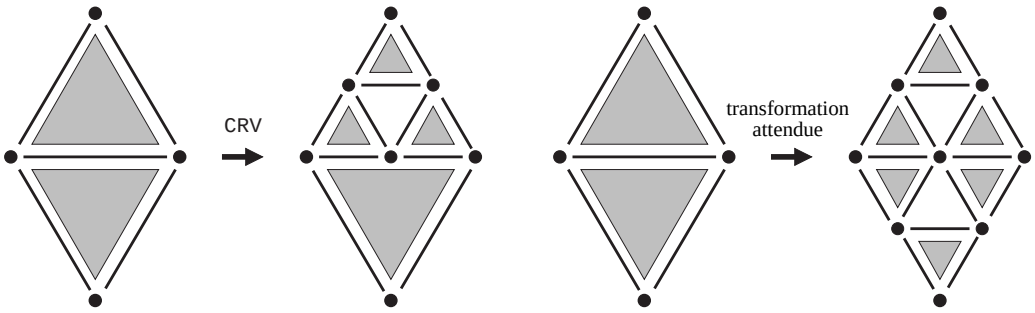


FIG. 14 – À gauche, l'application du patch `CRV` sur un état initial différent. À droite, le résultat attendu.

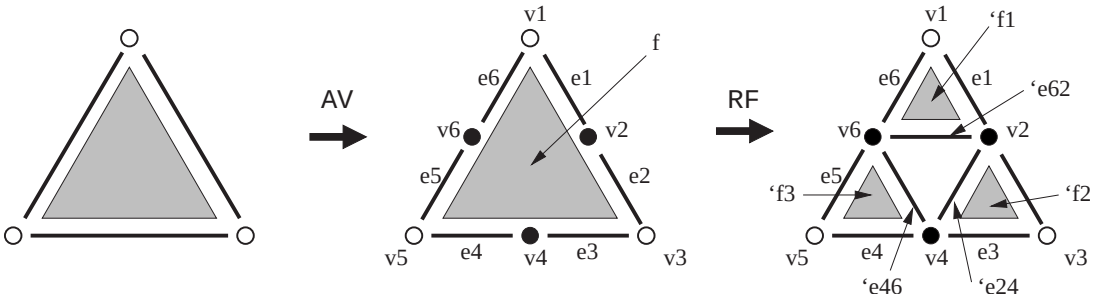


FIG. 15 – Découpage d'un triangle. La première transformation `AV` ajoute un sommet au milieu de chaque arc. La seconde, `RV`, raffine l'hexagone obtenu en 3 triangles.

```
patch CRV = {
  ~v1 < e1:[dim = 1] >
  ~v3 < e2:[dim = 1] >
  ~v5 < e3:[dim = 1] > ~v1
  f:[dim = 2, (e1,e2,e3) in faces]
  =>‘v2[dim = 0, value = average(v1,v3)]
    ‘v4[dim = 0, value = average(v3,v5)]
    ‘v6[dim = 0, value = average(v5,v1)]
    ‘e1[dim = 1, faces = (^v1,‘v2), value = ‘edge]
    ‘e2[dim = 1, faces = (‘v2,^v3), value = ‘edge]
    ‘e3[dim = 1, faces = (^v3,‘v4), value = ‘edge]
    ‘e4[dim = 1, faces = (‘v4,^v5), value = ‘edge]
    ‘e5[dim = 1, faces = (^v5,‘v6), value = ‘edge]
    ‘e6[dim = 1, faces = (‘v6,^v1), value = ‘edge]
    ‘e24:[dim = 1, faces = (‘v2,‘v4), value = ‘face]
    ‘e46:[dim = 1, faces = (‘v4,‘v6), value = ‘face]
    ‘e62:[dim = 1, faces = (‘v6,‘v2), value = ‘face]
    ‘f1:[dim = 2, faces = (‘e6,‘e1,‘e62), value = ‘face]
    ‘f2:[dim = 2, faces = (‘e2,‘e3,‘e24), value = ‘face]
    ‘f3:[dim = 2, faces = (‘e4,‘e5,‘e46), value = ‘face]
}
```

PROG. 3: Première version de la construction du TS par découpage en MGS.

Le TS est un objet de dimension 2. La fractale correspondante en 3 dimensions s'appelle l'*éponge de Sierpinski* (voir figure 16 page 216). Ces images ont été générées par MGS. Le programme est pratiquement le même qu'en dimension 2. En fait, nous utilisons les patches `AV` et `RV` pour « perforer » les faces du tétraèdre. Un dernier patch (qui n'est pas détaillé ici) est utilisé pour créer les quatre tétraèdres raffinés. Il est intéressant de noter que ce processus de découpage est récursif avec la dimension : pour une éponge de dimension n, on utilise les patches de dimension $n-1$, puis une dernière étape traite l'objet de dimension n. La figure 17 montre une autre construction de fractale : l'éponge de Menger.

3 Travaux apparentés

Dans cette section, nous voulons revenir sur le lien entre certains travaux menés dans le domaine de la modélisation géométrique et les notions de collection topologique et de transformation en MGS.

La recherche en modélisation géométrique et en infographie est à l'origine d'un nombre considérable de structures de données pour représenter des solides cherchant à équilibrer efficacité (représentation de données la plus compacte possible avec possibilité d'en parcourir les éléments) et expressivité (spécification d'opérations géométriques standards comme des opérations booléennes, des extrusions, des triangulations, des raffinements, etc). Avec la modélisation des (SD)^2, nous nous sommes intéressés en particulier aux structures offrant des mécanismes supplémentaires facilitant la modification de l'organisation topologique de la structure.

- Les triangulations de Delaunay [Aur91, OBSC00] ont été présentées dans les chapitres 2 et 3. Elles permettent la représentation d'un espace de dimension n à partir d'un nombre fini de sommets. La partition de l'espace est calculée à partir de la position des sommets choisis. Les modifications topologiques correspondent simplement à l'insertion ou la

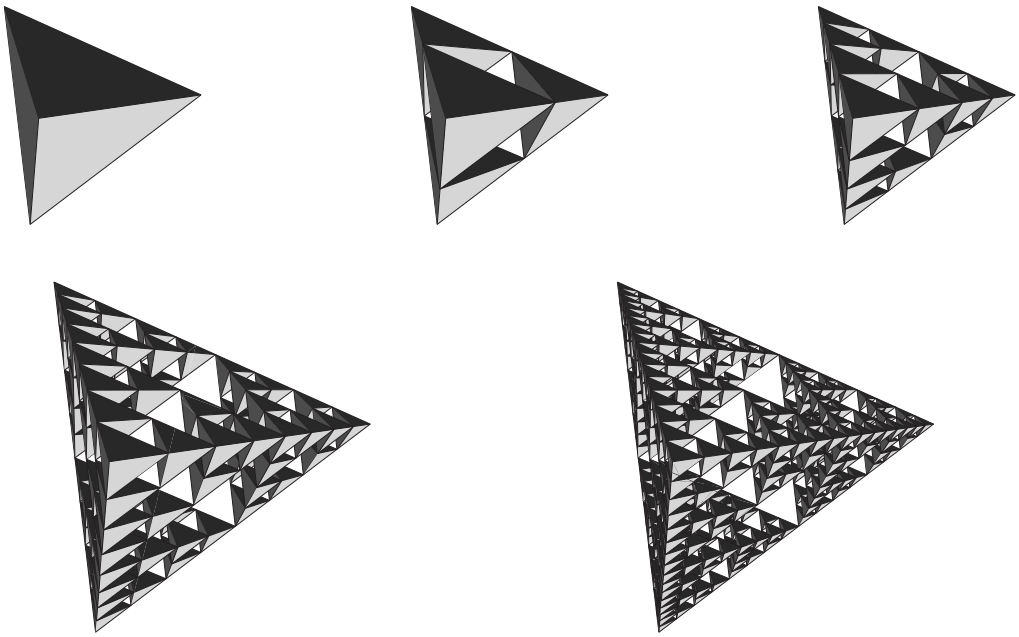


FIG. 16 – Construction de l'éponge de Sierpinski : état initial et étapes 1, 2, 3 et 4.

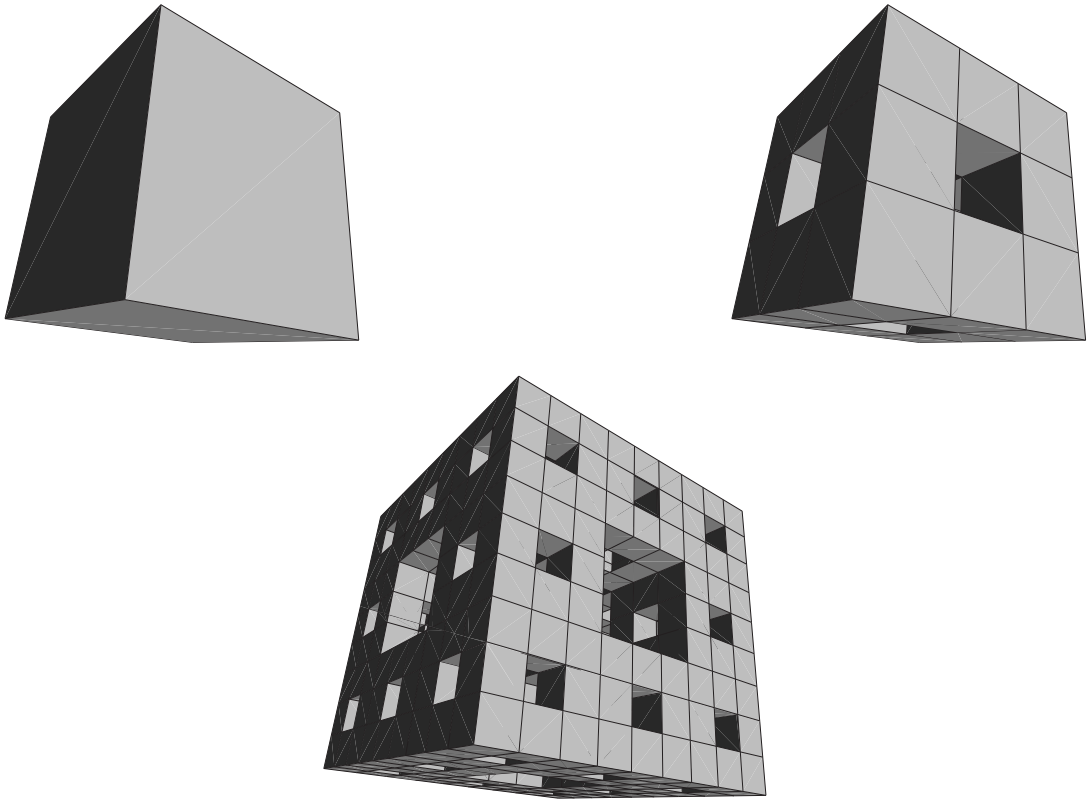


FIG. 17 – Construction de l'éponge de Menger : état initial et étapes 1 et 2.

suppression de sommets. Le désavantage de cette structure provient de l'expression implicite des cellules topologiques autres que les sommets : il n'est pas facile d'associer une information à un arc par exemple.

- Les L systèmes sont une grande réussite dans le domaine de la représentation de modèles de croissance. Leur utilisation a mené à des implantations à la fois efficaces mais également expressives.
  L'utilisation des L systèmes est restreinte à la représentation de structures linéaires ou arborescentes. Bien que conservant le cadre déclaratif des L systèmes, leurs extensions à la représentation de maillages polygonaux manquent des dispositifs nécessaires à l'application d'opérations topologiques complexes (comme par exemple la subdivision) [SPS04].

- Les vertex-vertex systèmes permettent de représenter des maillages polygonaux. Ils sont fondés sur la notion de *système de rotation de graphes*. Dans ces systèmes, chaque sommet d'un maillage polygonal fournit la liste *ordonnée* de ses 1-voisins. Une algèbre d'opérations sur les vertex-vertex systèmes a été définie pour décrire des opérations topologiques locales. Enfin, une implantation, appelée VV, a été développée pour manipuler la structure des vertex-vertex systèmes comme une extension du langage de programmation C++ [SPS04]. L'implantation très efficace est actuellement utilisée pour la simulation de croissance surfacique des plantes. Cette structure présente néanmoins deux inconvénients. D'une part, le langage VV ne fournit pas un cadre déclaratif pour spécifier les opérations (par exemple dans un style se rapprochant des grammaires de graphes) : les manipulations restent impératives. D'autre part, les vertex-vertex systèmes sont une structure de dimension 1 (un graphe) et permettent de représenter des surfaces en typant certains sommets comme représentant une 2-cellule et non une 0-cellule. Le passage à la dimension 3 semble plus délicat encore.

- Les G-cartes sont des modèles combinatoires définis pour la représentation de la topologie de subdivisions de quasi-variétés orientables ou non, avec ou sans bord. Les quasi-variétés sont définies comme une sous-classe des pseudo-variétés[5], objets connus en topologie algébrique. Il a été montré qu'il existe une correspondance bijective entre les G-cartes et les quasi-variétés, illustrant le pouvoir de représentation, en terme d'objets géométriques, de cette structure de données [Lie94].
  Cette structure satisfait également les contraintes d'efficacité requise en modélisation du solide. Les définitions des modèles et des opérations topologiques et géométriques sont simples et peuvent être implantées facilement dans des modeleurs géométriques [LM99].
  Par la rigueur mathématique liant la structure de données à la classe d'objets représentés, le développement des G-cartes a également conduit vers la formalisation des opérations topologiques et géométriques, passant d'algorithmes ordinaires à des spécifications prouvées de ces opérations :
  - L'implantation d'opérations topologiques et géométriques complexes sur les G-cartes ont fait l'objet d'une spécification formelle, offrant alors à la fois une description plus abstraite, locale, affranchie de l'ordre de parcours des éléments, et généralisable à des dimensions arbitraires [LAGB01, LMA+00, LAB00].
  - La construction de G-cartes peut se paramétrer en augmentant un λ-calcul. Le langage obtenu est très puissant et offre toute l'expressivité d'un calcul fonctionnel pour composer et itérer la construction de G-cartes (*via* la composition de fonctions et la

[5] Un complexe simplicial $\mathcal{K}$ de dimension n fortement connexe est appelé une *pseudo-variété* de dimension n si tout $(n-1)$-simplexe de $\mathcal{K}$ est une face de précisément deux n-simplexes de $\mathcal{K}$.

récursivité) dans un cadre mathématique bien maîtrisé [DL02]. Ces constructions sont bien sûr aussi disponibles en MGS, puisque celui-ci est un langage fonctionnel. Ce point n'est pas développé dans cette thèse, mais le lecteur peut se référer à [SM04a].

La question des modifications topologiques pour simulation de solide en croissance a également été étudiée dans le cadre des G-cartes. L'évolution de la constitution même des objets (pas seulement la déformation de leur forme) est appelée *métamorphose*. Dans [Ter94], il est proposé d'une part, une méthode hiérarchique pour représenter les objets (en ajoutant des involutions « virtuelles » permettant de représenter par exemple la filiations entre objets) et un mécanisme général associant à chaque brique élémentaire d'une G-carte un fragment de programme encodant le traitement à appliquer à cet élément au cours de l'évolution de l'objet. Il s'agit là d'une spécification *procédurale* de l'évolution qui doit être rapprochée du paradigme multi-agents : chaque cellule est un agent autonome interagissant avec les autres. Il est néanmoins difficile de spécifier par cette approche une notion générale d'interaction : chaque cellule étant indépendante, il n'existe pas de support pour l'interaction.

Avec les exemples de ce chapitre, nous mettons en avant la capacité des transformations à fonctionner sur tout type de structure de données. En effet, comme cela a déjà été précisé, les exemples que nous avons développés dans ce chapitre fonctionnent aussi bien sur des chaînes abstraites que sur des G-cartes. La spécification de modifications topologiques à l'aide des transformations est indépendante de la structure de données utilisée *in fine*. Nous insistons sur le fait que les différentes structures de données spécialisées en modélisation géométrique et informatique graphique sont autant d'implantations possibles, avec plus ou moins de contraintes, de la notion abstraite de collection topologique manipulée par MGS. Cette affirmation est validée par l'intégration des G-cartes comme un type particulier de collections topologiques dans l'interprète MGS. Ensuite, nous offrons un cadre abstrait et déclaratif dans lequel les opérations topologiques sont représentées localement par des règles de réécriture. La sémantique du chapitre 4 permet alors de raisonner sur les programmes MGS. Finalement, en séparant la description de la structure et de l'évolution du système (contrairement à une approche procédurale où les lois d'évolution sont portées par chaque cellule topologique à la façon des multi-agents), nous fournissons un support naturel, explicite et déclaratif à la notion d'interaction à travers des règles.

Chapitre 9

Quelques aspects de simulation numérique

Dans ce chapitre, nous souhaitons mettre en avant l'utilisation du langage MGS pour résoudre des problèmes posés lors de simulations numériques. Nous proposons pour cela deux exemples.

Le premier exemple montre comment programmer un problème d'intégration numérique en MGS. Pour cela, nous proposons d'implanter un modèle de couplage d'oscillateurs dont l'origine est l'étude du plasmodium de *Physarum polycephalum*, un organisme utilisant la fréquence d'oscillateurs pour encoder et propager de l'information.

Nous proposons ensuite la programmation d'un modèle du déplacement cellulaire du spermatozoïde du nématode *Ascaris suum*. Ce second exemple est intéressant car il implique une structure dynamique, une modélisation qui fait appel à la résolution numérique d'équations aux dérivées partielles, et la modélisation de lois mécaniques et chimiques. Il apparaît donc comme un excellent test pour vérifier l'expressivité et l'adéquation de notre approche sur un problème « en vraie grandeur ».

1 *Physarum polycephalum* : intégration numérique

Le plasmodium[1] de *Physarum polycephalum* est un organisme unicellulaire amiboïde[2] qui, malgré l'absence de système nerveux, présente une forme de transmission d'informations entraînant la migration cellulaire quand il est stimulé.

Diverses formes d'oscillations sont observables dans le plasmodium : chimiques, thermiques ou mécaniques (vagues cytoplasmiques). Ces oscillations sont interdépendantes et encodent l'information des stimuli dans leur fréquence. Celle-ci augmente sous l'effet d'un attracteur, et diminue sous l'effet d'un répulseur.

Dans [TTNT97], les auteurs proposent un modèle de couplage de fréquences afin d'étudier la propagation d'un stimulus entre plusieurs oscillateurs. La topologie (l'organisation des oscillateurs) du système est une séquence d'oscillateurs, chacun d'eux influençant et étant influencé par son voisin gauche et son voisin droit (à l'exception des extrémités bien entendu).

Dans cette section, nous implantons ce modèle en MGS. Nous profitons du contexte pour illustrer l'utilisation des transformations pour résoudre un problème d'intégration numérique. En effet, les oscillateurs sont couplés par deux équations non-linéaires qu'il est difficile de résoudre symboliquement. Nous programmons cette résolution par des intégrations numériques de types Euler et Runge-Kutta d'ordres 2 et 4.

1.1 Description du système

Le système est une séquence de n oscillateurs. Pour chacun d'eux, la dynamique de l'oscillation est encodée dans une variable complexe x_i dont la partie réelle représente la quantité oscillante (une concentration chimique dans [TTNT97]) et la fréquence de l'oscillation est caractérisée par la fréquence intrinsèque ω_i telle que :

$$\begin{aligned} x_i &= x_i^0 + r\exp^{j\theta_i} \\ \dot{x}_i &= F(x_i, \omega_i) \end{aligned}$$

où la valeur de la fréquence intrinsèque ω_i est proportionnelle à la valeur x_i^0 représentant la composante lente de l'oscillation, et où θ_i est la phase de la composante rapide de l'oscillation.

La fonction F est capturée par deux équations aux dérivées partielles faisant intervenir les constantes de couplage K_1 et K_2 :

$$\dot{\theta}_i = \omega_i + K_1 \sum_{j \in Vois(i)} H(\theta_j - \theta_i) \quad (1)$$

$$\dot{\omega}_i = K_2 \sum_{j \in Vois(i)} ((\omega_j - \dot{\theta}_j) - (\omega_i - \dot{\theta}_i)) \quad (2)$$

Nous supposons pour la suite que la fonction H est implantée par une fonction MGS `H`.

Nous proposons ici une implantation de l'intégration numérique de ces deux équations. Pour plus de détails sur la mise en place de ces équations, le lecteur intéressé se référera à [TTNT97].

État du système. Le système est représenté en MGS par une séquence de longueur n dont les éléments sont des enregistrements référant aux quantités ω_i et θ_i définies ci-dessus ainsi qu'à leurs variations :

[1] Masse multinucléaire de cytoplasme formé par l'agrégation de plusieurs cellules amiboïdes.
[2] de ou ressemblant à une amibe.

```
    constraint state      = [oscillator]seq(n)
and record     oscillator = { w  :  float,
                              dw :  float,
                              T  : float,
                              dT : float
                            } ;;
```

Initialisation. L'état initial du système est déterminé aléatoirement. Dans la pratique, la longueur de la séquence est fixée à $n = 20$, les phases θ_i des oscillateurs à 0.0, et les fréquences intrinsèques ω_i sont choisies suivant une distribution gaussienne de moyenne 0.1 et de variance 0.0001. L'interprète MGS fournit la fonction `random` permettant le tirage uniforme d'un réel entre 0 et 1, et qui est utilisée pour programmer le tirage suivant la loi gaussienne.

Expérience. L'expérience proposée dans [TTNT97] est de simuler l'oscillation pour 1000 unités de temps, puis de perturber à cette date et pour le reste de l'expérience le premier oscillateur de la séquence en fixant la variable ω_0 à une valeur ω_s (pour stimulus). Afin de réaliser une simulation en accord avec les propos de l'article de référence, nous optons pour $K_1 = 0.10$, $K_2 = 0.10$ et $\omega_s = 0.08$. La gestion du temps est donnée par les méthodes d'intégration : on suppose l'existence d'une variable globale `t` correspondant à la date courante ; à chaque pas d'intégration, cette date est incrémentée de `dt`, une seconde variable globale encodant la longueur d'un pas d'intégration.

Nous comparons les implantations des différentes méthodes d'intégration numérique sur un même état initial. Cette comparaison est faite sur les variations de ω_i en fonction du temps.

Traduction des équations en MGS. Les équations 1 et 2 se traduisent de la façon suivante en MGS :

```
fun dtheta(xi, Theta, Omega) =
  Omega(xi) + K1*(neighborfold( (\xj.\y.(y + H(Theta(xj) - Theta(xi)))),
                                0.0,
                                xi))
;;

fun domega(xi, Theta, Omega) =
  K2*(neighborfold( (\xj.\y.(y + ((Omega(xj) - xj.dT) -
                                  (Omega(xi) - xi.dT)))),
                    0.0,
                    xi))
;;
```

Ces deux fonctions sont destinées à être appelées en partie droite d'une règle de transformation. En effet, la primitive `neighborfold` n'a de sens que dans une transformation alors que `xi` correspond à une variable de filtre. Les fonctions `Theta` et `Omega` données en arguments sont utilisées pour paramétrer les valeurs des champs `T` et `w` de `xi` pour les méthodes de Runge-Kutta.

On remarque que ces fonctions sont locales et par conséquent ne dépendent pas de la topologie du voisinage (les oscillateurs peuvent avoir un nombre arbitraire de voisins). La primitive `neighborfold` retourne en effet les valeurs associées aux voisins de `xi` quel que soit le type de collection sur laquelle la règle est appliquée. Les oscillateurs pourraient tout aussi bien présenter

un voisinage plus complexe (ce qui n'était pas considéré dans [TTNT97]) sans qu'il n'y ait à modifier la traduction en MGS des équations : il s'agit d'un exemple de *polytypisme* (voir chapitre 2, page 45).

1.2 Méthode d'Euler

Il s'agit de la méthode la plus simple. Soit l'équation différentielle suivante

$$\frac{dx(t)}{dt} = f(x,t) \tag{3}$$

la méthode d'intégration d'Euler est une méthode itérative calculant à partir d'une condition de type $x(t_0) = X_0$, la suite :

$$X_{i+1} = x(t_{i+1}) = x(t_i + \Delta t) = X_i + \Delta t \times f(X_i, t_i) \tag{4}$$

Les équations 1 et 2 sont traduites par la transformation `Euler`, calculant les suites pour les quantités θ et ω de chaque oscillateur. Elle est déclinée par le programme 4. Tous les

```
trans Euler = {
  x / (^x==0 && t>=1000) => (
    let Theta = \x.(x.T)
    and Omega = \x.(x.w) in
      x + { T  = x.T + dt*dtheta(x, Theta, Omega),
            dT = dtheta(x, Theta, Omega),
            w  = ws,
            dw = 0 }
  ) ;
  x => (
    let Theta = \x.(x.T)
    and Omega = \x.(x.w) in
      x + { T  = x.T + dt*dtheta(x, Theta, Omega),
            dT = dtheta(x, Theta, Omega),
            w  = x.w + dt*domega(x),
            dw = domega(x, Theta, Omega) }
  )
} ;;
```

PROG. 4: Méthode d'Euler : intégration des équations 1 et 2.

champs sont mis à jour en appelant les fonctions `dtheta` et `domega`. Elles retournent la variation des quantités correspondantes. Les valeurs retournées sont alors utilisées comme spécifié par la méthode d'intégration d'Euler. Deux règles sont décrites : la seconde correspond au cas général alors que la première (prioritaire sur la suivante d'après la stratégie standard de MGS) décrit le stimulus appliqué au premier oscillateur (`^x == 0`). Sa fréquence intrinsèque est fixée à `ws` à partir de la date 1000.

1.3 Méthodes de Runge-Kutta

Les méthodes de Runge-Kutta construisent le même type de suites que celle d'Euler, mais propose des calculs moins approximatifs ; le pas de temps peut alors être plus grand, accélérant

l'intégration.

La méthode de Runge-Kutta est caractérisée par l'*ordre* à laquelle on l'applique. Cet ordre va correspondre au nombre d'étapes nécessaires pour effectuer le calcul. En règle général, les ordres utilisés sont 2 et 4 ; il faut savoir que le calcul à l'ordre n introduit $n-1$ inconnues dans le calcul. Voici les équations de cette méthode pour l'ordre 2 considérant l'équation différentielle 3 :

$$\begin{aligned} X_{i+1} &= X_i + \Delta t \times (\mathrm{A} f_1 + \mathrm{B} f_2) \\ f_1 &= f(X_i, t_i) \\ f_2 &= f(X_i + \frac{h}{2\mathrm{B}} f_1, t_i + \frac{h}{2\mathrm{B}}) \\ \mathrm{A} &= 1 - \mathrm{B} \end{aligned}$$

Le coefficient B est indéterminé. On prend en général $\mathrm{B} = \frac{1}{2}$ ou $\mathrm{B} = 1$ retrouvant respectivement la méthode de Heun et la méthode d'Euler améliorée.

Le calcul de f_2 étant dépendant de la valeur de f_1 (notamment pour les voisins), cette méthode est encodée en deux étapes, la première pour le calcul de f_1 et la seconde pour celui de f_2 et de l'intégration proprement dite. Le programme 5 décrit ces deux étapes avec les transformations `RK2_f1` et `RK2_f2`.

Les équations et leur traduction en MGS pour la méthode de Runge-Kutta d'ordre 4 ne sont pas données ici ; quatre étapes sont nécessaires. Elles restent en effet proches de la méthode de Runge-Kutta d'ordre 2 et il n'est pas nécessaire de les développer. La figure 1 présente le résultat de l'application de ces transformations sur une même séquence initiale ; on y trouve l'intégration par les méthodes numériques d'Euler (avec `dt` = 2.3), de Runge-Kutta d'ordre 2

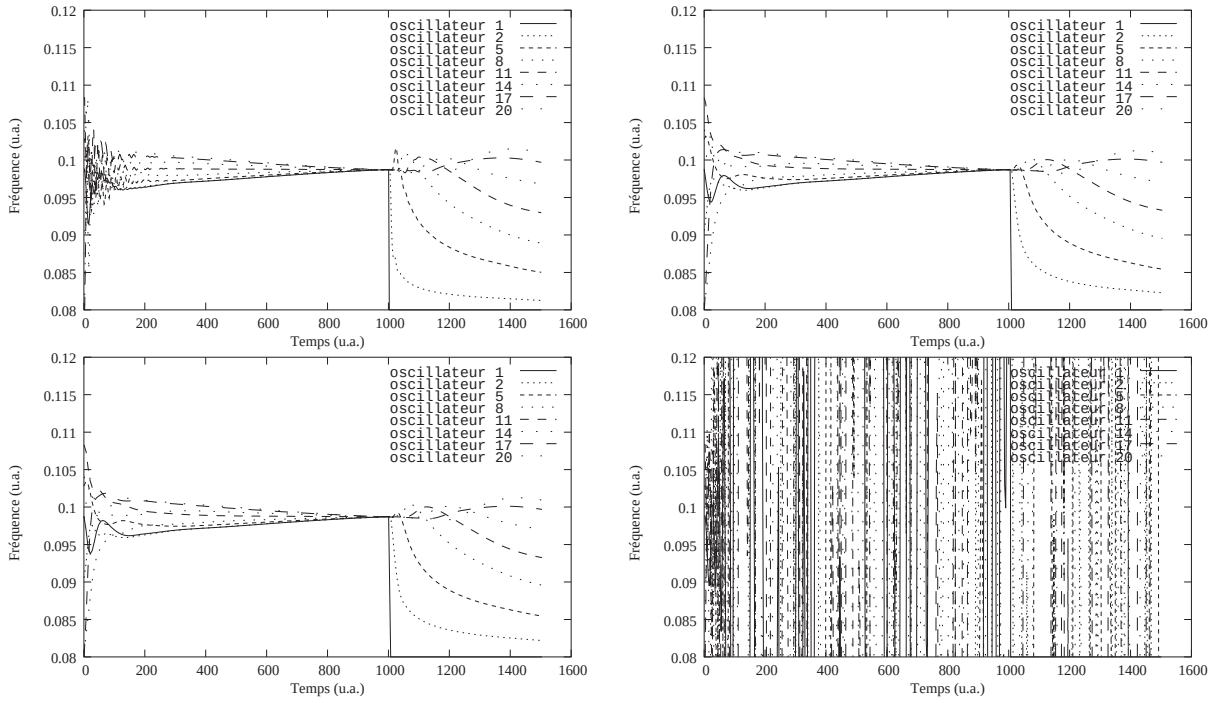


FIG. 1 – Comparatif des résultats d'intégration numérique suivant la méthode utilisée : en haut à gauche la méthode d'Euler avec `dt` = 2.3, en haut à droite la méthode de Runge-Kutta d'ordre 2 avec `dt` = 4.9, en bas à gauche la méthode de Runge-Kutta d'ordre 4 avec `dt` = 6.9, et en bas à droite la méthode d'Euler avec `dt` = 3.0.

```
trans RK2_f1 = {
  x / (^x==0 && t>=1000) => (
    let Theta = \x.(x.T)
    and Omega = \x.(x.w) in
      x + { dT_f1 = dtheta(x, Theta, Omega),
            dw_f1 = 0 }
  ) ;
  x => (
    let Theta = \x.(x.T)
    and Omega = \x.(x.w) in
      x + { dT_f1 = dtheta(x, Theta, Omega),
            dw_f1 = domega(x, Theta, Omega) }
  )
} ;;


trans RK_f2 = {
  x / (^x==0 && t>=1000) => (
    let Theta = \x.(x.T + (dt/(2*B))*x.dT_f1)
    and Omega = \x.(x.w + (dt/(2*B))*x.dw_f1) in
    let dT_f2 = dtheta(x, Theta, Omega) in
      x + { T  = x.T + dt*(A*x.dT_f1 + B*dT_f2),
            dT = A*x.dT_f1 + B*dT_f2,
            w  = ws,
            dw = 0 }
  ) ;
  x => (
    let Theta = \x.(x.T + (dt/(2*B))*x.dT_f1)
    and Omega = \x.(x.w + (dt/(2*B))*x.dw_f1) in
    let dT_f2 = dtheta(x, Theta, Omega)
    and dw_f2 = domega(x, Theta, Omega) in
      x + { T  = x.T + dt*(A*x.dT_f1 + B*dT_f2),
            dT = A*x.dT_f1 + B*dT_f2,
            w  = x.w + dt*(A*x.dw_f1 + B*dw_f2),
            dw = A*x.dw_f1 + B*dw_f2 }
  )
} ;;
```

Prog. 5: Méthode de Runge-Kutta d'ordre 2 : intégration des équations 1 et 2. Deux étapes sont nécessaires pour réaliser cette intégration.

(avec `dt` = 4.9) et de Runge-Kutta d'ordre 4 (avec `dt` = 6.9). Au-delà des valeurs proposées pour les pas de temps, les erreurs d'approximation commencent à être visibles. Le dernier graphique présente un exemple de résultat où le même calcul, avec la méthode d'Euler et un pas de temps fixé à `dt` = 3.0, diverge. On trouve dans ces graphiques les résultats attendus : de la date 0 à la date 1000, les oscillateurs se stabilisent pour finalement vibrer à la même fréquence (la fréquence intrinsèque ω est représentée), puis le stimulus de l'oscillateur 1 forcé à vibrer à la fréquence $w_s = 0.8$ entraîne une modification de la fréquence intrinsèque des oscillateurs voisins, modification qui se répercute de proche en proche pour finalement atteindre le dernier oscillateur. En attendant assez longtemps, les oscillateurs finissent par se resynchroniser de nouveau.

2 Déplacement cellulaire : éléments finis

Dans cette section, nous nous intéressons à l'implantation d'un modèle biologique proposé dans [BMR+02]. Ce modèle simule la motilité du spermatozoïde du nématode *Ascaris suum*, un ver parasite habituellement trouvé chez le porc. Nous commençons par décrire le modèle et sa discrétisation en 2 dimensions. Nous verrons ensuite comment ce modèle peut être implanté en utilisant les constructions offertes par le langage MGS.

2.1 Description du modèle

Le spermatozoïde d'*Ascaris suum* est une cellule biologique séparée en deux parties distinctes. À gauche de la figure 2, un diagramme décrit l'organisation de cette cellule vue du dessus. A gauche, la région péri-nucléaire contient entre autres le noyau et les mitochondries de la cellule, et à droite, s'étend la partie de la cellule où les mécanismes chimiques et mécaniques permettant à la cellule de se déplacer ont lieu, la région *lamellipodale*. Sur ce schéma, la cellule se déplace vers la droite.

Le spermatozoïde se déplace suivant un cycle *protusion-adhésion-rétraction*. Dans ce modèle, le système correspond à la partie de la membrane de la région lamellipodale accrochée à la matrice extra-cellulaire (environnement de la cellule) sous la cellule. Tout d'abord, une polymérisation fibreuse apparaît sur le front de la cellule créant des *protusions*. Ces protusions poussent alors la membrane vers l'avant qui se prolonge alors par des « bras » cherchant à prendre contact avec la matrice extracellulaire. Durant la phase d'adhésion, la protusion se colle à la matrice. Un mécanisme de traction permet alors au corps de la cellule de se déplacer vers l'avant. À mesure que la cellule avance, les contraintes mécaniques entre la matrice et la cellule emmagasine de l'énergie sous forme d'énergie élastique. L'étape finale a lieu près de la frontière péri-nucléaire où la dépolymérisation du gel fibreux cause la *désadhésion*, ou *rétraction*, de la membrane de la matrice extérieure. L'énergie élastique stockée est alors libérée, tirant la région péri-nucléaire vers l'avant. Ce mécanisme est régulé par un gradient de pH s'étendant du front de la cellule à la région nucléaire où les mitochondries sont une source d'acidité.

Les équations continues considérées pour modéliser le fonctionnement de la machinerie lamellipodale, correspondent d'une part à un modèle mécanique où les forces élastiques et de contraction s'appliquent sur la partie de la membrane en contact avec la matrice extra-cellulaire, et d'autre part à un modèle chimique calculant la distribution des protons dans cette même zone afin de prendre en compte le gradient de pH.

Forces mécaniques

L'équation donnée par Bottino *et al.* déterminant les forces mécaniques est la suivante :

$$\mu(u)\frac{\partial u}{\partial t} = \nabla.\sigma(u)$$

Cette équation est valable sur l'ensemble de la membrane : u est un vecteur position. La partie droite calcule la somme des forces agissant en un point u de la membrane ; les contraintes mécaniques étant de deux sortes, le paramètre σ se divise en deux parties :

$$\sigma = \text{contraintes élastiques } - \text{ contraintes de tension}$$

En effet, les propriétés mécaniques du gel fibreux ne se restreignent pas à une contrainte élastique simple ; il permet notamment le stockage d'énergie élastique à mesure qu'il se polymérise. La contrainte de tension modélise la dilatation due à la pression osmotique du gel, couplée au phénomène de stockage d'énergie. Les contraintes élastiques et de tension seront respectivement associées aux variables κ et τ par la suite. En partie gauche, on calcule le frottement visqueux qui agit en chaque point u de la membrane. Celui-ci est proportionnel à la vitesse de déplacement locale de la membrane en ce point. L'égalité entre ces deux membres correspond au calcul de l'équilibre des forces en tout point de la membrane. Les valeurs des coefficients (dans les expressions de μ et de σ) apparaissant dans cette équation dépendent de la position car elles sont régulées par la distribution du pH le long de la membrane.

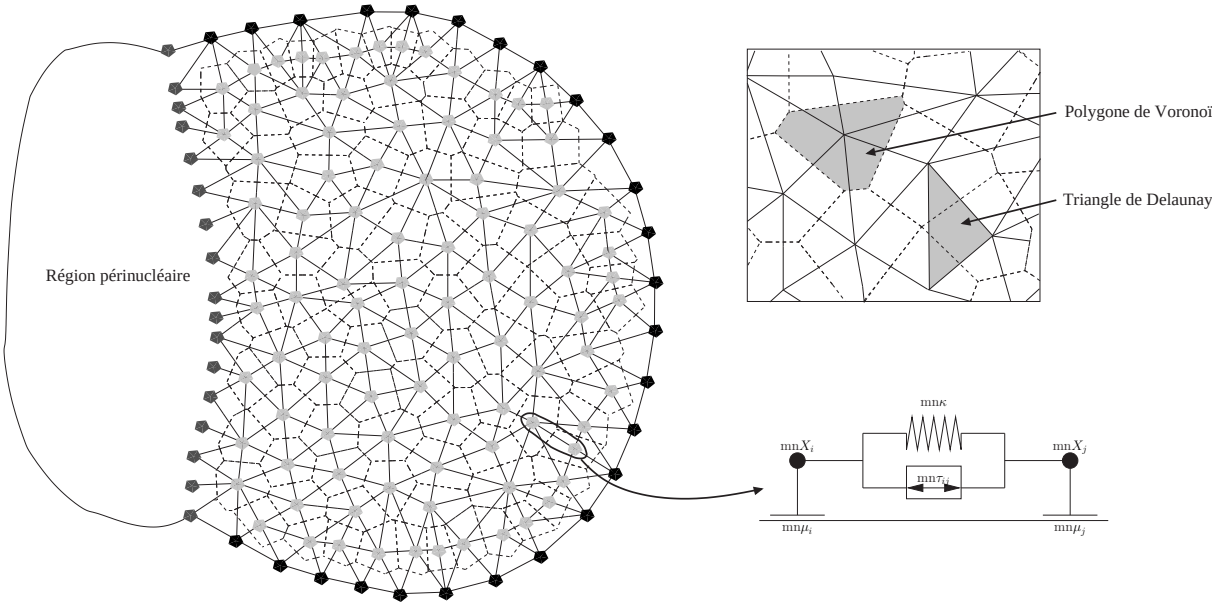


FIG. 2 – Le spermatozoïde du nématode. À gauche, un diagramme schématique montrant l'organisation de la cellule : sur la gauche, la région nucléaire, avec à sa droite la région lamellipodale. La discrétisation est donnée par les sommets. Les arcs pleins correspondent au voisinage de Delaunay. Les arcs en pointillés sont les frontières des polygones de Voronoï. En haut à droite, figure une partie zoomée de la discrétisation. En bas à droite, un arc de Delaunay reliant deux sommets représente un ressort de module κ en parallèle avec un élément de tension τ. Une friction de coefficient μ apparaît quand un sommet est en contact avec le tissu extérieur (ces diagrammes sont inspirés des figures prises de [BMR$^+$02], mais sont générés par MGS).

Distribution de pH

La seconde équation qui nous intéresse rend compte de la distribution en proton le long de la membrane. Elle permet donc de calculer le gradient de pH :

$$D\nabla^2[H^+] = P([H^+] - [H^+]_{ext})$$

où $[H^+]$ est la concentration en protons à une position donnée, et $[H^+]_{ext}$ est la concentration externe en protons. Cette équation est celle d'une diffusion avec fuite. Elle relate également un équilibre. En effet, elle exprime que la diffusion des protons en chaque point (partie gauche de l'équation) est compensée par la fuite des protons vers l'extérieur, due à la perméabilité de la membrane (partie droite de l'équation). Le phénomène de diffusion a lieu à une échelle de temps plus rapide que l'échelle de temps des forces mécaniques, et cet équilibre correspond à une approximation quasi-statique. Les paramètres D et P sont respectivement les coefficients de diffusion et de perméabilité de la cellule.

2.2 Le modèle à éléments finis

Dans [BMR+02], les équations précédentes sont résolues directement en dimension 1. En ce qui concerne la résolution en deux dimensions, les auteurs proposent l'utilisation d'une méthode numérique fondée sur l'utilisation d'éléments finis pour approximer le comportement de la surface à simuler.

Cette méthode suit plusieurs étapes successives. La première consiste à partitionner l'espace dans lequel on cherche à résoudre les équations. Pour nous, le système se réduit à la partie de la membrane lamellipodale en contact avec la matrice extra-cellulaire. Pour réaliser la discrétisation de cet espace, on utilise un *diagramme de Voronoï* associé à son concept dual la *triangulation de Delaunay* (voir chapitre 2, page 31). La figure 2 présente ce partitionnement : chaque sommet de Delaunay représente une partie de la membrane. Cette partie correspond au polygone de Voronoï qui l'entoure, dual du sommet. Ces polygones sont représentés en pointillés sur la figure 2. Les sommets sont liés par des arcs de Delaunay dessinés en trait plein grâce à la triangulation de Delaunay.

En règle générale, la méthode des éléments finis a besoin d'une certaine homogénéité des éléments pour fournir une bonne approximation. Or les éléments issus du diagramme de Voronoï sont des polygones irréguliers dont le nombre de sommets varie. Son utilisation ne semble pas convenir au cadre usuel des éléments finis. Cependant dans notre problème, le modèle tire parti du fait que les équations ne dépendent pas de la forme exacte de l'élément fini, mais uniquement de critères plus globaux comme sa surface ou la longueur d'un côté. Cette constatation permet d'utiliser un diagramme de Voronoï de la manière suivante :

- chaque élément de surface est associé à un sommet du graphe de Delaunay, et
- chaque élément de surface correspond à la cellule de Voronoï associée au sommet,
- un arc dans le graphe correspond aux interactions entre deux éléments de surface visités.

Cette approche permet d'implanter très facilement un maillage *adaptatif* : pour *raffiner* un élément fini, il suffit de rajouter des points (et inversement), le maillage est automatiquement recalculé. La structure issue de la triangulation de Delaunay est particulièrement adaptée à la modélisation de systèmes à structure dynamique.

L'étape suivante consiste à associer à chaque partie de la discrétisation de l'espace les informations qui vont être utilisées lors de la résolution des équations. Les sommets, représentant la membrane, sont de trois types selon leur place sur la membrane :

1. Les NRnodes (pour *nuclear region nodes*) sont les nœuds situés à la frontière entre la région lamellipodale et la région péri-nucléaire de la cellule où a lieu la rétraction. Ils figurent en gris sur la figure 2.
2. Les Bnodes (pour *boundary nodes*) sont les nœuds délimitant l'endroit où la membrane n'est plus en contact avec la matrice extérieure. Par rapport au système que nous cherchons à décrire, ils correspondent à la bordure où la protusion se produit. Ils figurent en gris foncé sur la figure 2.
3. Les Inodes (pour *inner nodes*) regroupent les nœuds restants représentant le corps du système. Ils figurent en gris clair sur la figure 2.

Afin de simuler l'évolution du système, les équations décrites précédemment doivent être résolues sur chaque élément de la partition, puis les solutions coordonnées pour chaque élément de façon à obtenir une solution globale du mouvement du spermatozoïde. Pour décrire l'action d'un nœud sur un autre, nous allons utiliser les arcs issus de la triangulation de Delaunay. Aussi, les deux contraintes mécaniques relevées précédemment et appliquées en tout point de la membrane, vont être représentées par la mise en parallèle d'un élément élastique et d'un élément de tension entre deux sommets liés par un arc de Delaunay. D'autre part, nous ajoutons sur les sommets un élément de frottement pour représenter la viscosité. En ce qui concerne le pH, nous supposons une répartition homogène de la concentration en protons sur un élément (autrement dit, un point du polygone de Voronoï a comme concentration celle du nœud qui lui est associé), et sur les arcs, on calcule le pH comme la moyenne des pH des deux sommets liés.

Finalement, on discrétise les équations continues pour ne conserver que les interactions entre les polygones de Voronoï.

Discrétisation des équations continues

Les équations précédentes sont traduites de la façon suivante ; pour tout sommet X_i :

$$\mu_i \frac{\partial u_i}{\partial t} = \sum_j C_2^{ij}(\kappa|X_i - X_j| - \tau_{ij})\frac{X_j - X_i}{|X_j - X_i|} \qquad \text{(forces mécaniques)}$$

$$\mathrm{D}\sum_j C_1^{ij}([\mathrm{H}^+]_i - [\mathrm{H}^+]_j) = \mathrm{P}([\mathrm{H}^+]_i - [\mathrm{H}^+]_{\mathrm{ext}}) \quad \text{(répartition du pH)}$$

Dans ces deux équations, l'opérateur ∇ de l'équation continue est remplacé par une itération finie sur les voisins X_j du sommet X_i ; u_i est le vecteur position du nœud X_i et l'expression $\frac{X_j - X_i}{|X_j - X_i|}$ représente le vecteur unitaire dont la direction est celle de l'arc de Delaunay reliant X_i à X_j. Dans la première équation les contraintes σ utilisées dans l'équation continue équivalente, sont précisées ; le terme $\kappa|X_i - X_j|$ correspond à la force élastique entre X_i et X_j, et la valeur de l'élément de tension est donnée par la fonction τ_{ij} qui est détaillée dans [BMR$^+$02].

Deux nouveaux paramètres sont apparus lors de cette discrétisation ; les coefficients C_k^{ij} insèrent dans le calcul les propriétés géométriques des polygones de Voronoï et de la triangulation de Delaunay (telles que la surface des polygones de Voronoï, la longueur des arcs de Delaunay, ...). Leur présence est due au caractère non microscopique des équations discrètes, à l'instar des équations continues.

Polymérisation et dépolymérisation

La polymérisation et la dépolymérisation du gel ont lieu lors des étapes d'adhésion et de rétraction de la membrane. En fait ils correspondent au changement de structure auquel nous

devons faire face lors de cette modélisation. En effet, lors de l'adhésion, la membrane se *colle* au substrat ; elle compose alors le système qui, rappelons-le, est *la partie de la membrane de la région lamellipodale en contact avec la matrice*. Il nous faut alors prendre en compte la croissance de cette structure en modifiant l'espace représentant le système, espace servant notamment de support aux calculs des modèles mécanico-chimiques présentés ci-dessus. En revanche, lors de la rétraction, la membrane se désolidarise de la matrice et sort du domaine traité par le modèle. Le système que nous considérons est hautement dynamique, ce qui rend la résolution des équations complexe.

Pour prendre en compte ces deux phénomènes, nous allons simplement considérer la création et la destruction d'Inodes. Deux seuils encadrent les longueurs autorisées des arcs de Delaunay. Soient X_i et X_j les deux sommets et l_{ij} la longueur de l'arc de Delaunay reliant X_i et X_j. Si $l_{ij} > l_{\max}$, un sommet est créé au milieu de l'arc avec un pH égal à la moyenne des pH de X_i et X_j. Au contraire, si $l_{ij} < l_{\min}$, on supprime l'un des deux nœuds. Bien que ces contraintes soient appliquées partout, on peut constater dans la simulation que les sommets sont créés près des Bnodes, et détruits près de la frontière péri-nucléaire.

2.3 Implantation en MGS

Le modèle proposé dans [BMR+02] se prête bien à une implantation MGS. En effet, le voisinage émergeant de la triangulation de Delaunay fournit les bases pour la définition d'une collection topologique, et la localité des équations discrètes correspond parfaitement à un cadre de réécriture, et par conséquent à une implantation par transformation.

Dans cette dernière section, nous montrons comment ce modèle est directement traduit en MGS. Dans un premier temps, nous nous intéressons à la structure du système, puis à sa dynamique.

Structures de données

L'espace suit la structure issue de la triangulation de Delaunay. Pour définir une collection de Delaunay, il est nécessaire de fournir une fonction extrayant les coordonnées du sommet à partir de la valeur qui le décore. Nous commençons donc par décrire cette valeur.

Nous utilisons un enregistrement composé de 8 champs dont la définition est la suivante :

```
record Node = {
  px    :  float, py     :  float,
  vx    :  float, vy     :  float,
  H     :  float, pH     :  float,
  Bflag :  bool,  NRflag :  bool
};
```

Nous rappelons que cette écriture définit également un prédicat `Node` qui permet de vérifier que son argument est un enregistrement de type `Node`. Ici, les champs `px` et `py` doivent être des réels ; ils représentent la position du nœud. Les champs `vx` et `vy` spécifient le vecteur vitesse, et `H` et `pH` la concentration en protons et le pH. Les deux derniers champs vont permettre de déterminer le type de nœud auquel on a affaire. Il est en effet possible de définir trois autres prédicats `Inode`, `Bnode` et `NRnode` de la façon suivante :

```
    record Inode  = Node + { Bflag  = false, NRflag = false }
and record Bnode  = Node + { Bflag  = true }
and record NRnode = Node + { NRflag = true } ;;
```

Nous pouvons désormais spécifier la collection de Delaunay que nous souhaitons manipuler :

```
delaunay(2) D2 = \elt.(
  if   Node(elt)
  then (elt.px, elt.py)
  else error("élément de type incorrect") fi
) ;;
```

Nous définissons ainsi le prédicat `D2` pour une collection Delaunay de dimension 2 et dont les éléments sont de type `Node`. La fonction donnée lors de cette définition permet d'extraire le couple `(elt.px, elt.py)`, c'est-à-dire la position d'un élément de type `Node`.

À partir de cette définition, MGS est capable de calculer automatiquement le voisinage résultant de la triangulation de Delaunay d'une séquence d'éléments. La séquence de code suivante présente un exemple de génération d'état initial :

```
{ px = -3.374538, py = 3.031106,
  vx = 0.000000,  vy = 0.000000
  Bflag = true,   NRflag = true,
  H = 0.000001,   pH = 6.019161 }, ...
..., { px = -2.132081, py = 8.084903,
       vx = 0.000000,  vy = 0.000000
       Bflag = true,   NRflag = false,
       H = 0.000001,   pH = 6.144690 }, D2:()
```

Ici, seuls deux nœuds ont été présentés ; il s'agit d'enregistrements vérifiant la spécification `Node` donnée précédemment. La liste des nœuds se termine par la collection topologique vide `D2:()` correspondant à une triangulation de Delaunay sans nœud. La virgule sert d'opérateur d'insertion.

Lois d'évolution

Maintenant que nous disposons d'une représentation pour les données, nous devons spécifier les lois d'évolution à partir des équations discrètes. L'algorithme principal est le suivant :

1. calcul de l'équilibre chimique pour déterminer le gradient de pH,
2. calcul des forces agissant sur chaque nœud et intégration,
3. ajout/suppression de nœuds suivant la longueur des arcs,
4. retour à l'étape 1.

Équilibre chimique

L'équation discrète de la distribution de pH est un équilibre chimique et ne peut donc pas être résolue directement. En effet, rien n'affirme que la répartition du pH sur les nœuds de la structure vérifie cet équilibre. Cependant, nous savons comment évolue la diffusion et la fuite de protons ; il est donc possible de calculer, dans un pas de temps plus petit, l'évolution chimique du système. L'itération jusqu'à un point fixe de ce calcul nous permet alors de calculer l'équilibre.

Pour cela, nous transformons l'équation de distribution de pH en une suite dont nous cherchons la limite ; pour chaque nœud X_i, on a

$$\left([\mathrm{H}^+]_i\right)_n = \frac{\mathrm{D}\sum_j C_1^{ij}([\mathrm{H}^+]_j)_{n-1} - \mathrm{P}[\mathrm{H}^+]_{\mathrm{ext}}}{\mathrm{D}\sum_j C_1^{ij} - \mathrm{P}}$$

L'équation calcule la valeur de la concentration en protons pour un sommet comme une fonction de la concentration en protons de ses voisins ; la traduction en MGS est immédiate :

```
trans update_pH = {
  Xi:NRnode => ...;
  Xi:Bnode => ...;
  Xi => let num = (P/D) * H_ext
                + neighborfold(
                    (fun Xj acc -> C1(Xi,Xj) * Xj.H + acc),
                    0, Xi)
        and den = (P/D)
                + neighborfold(
                    (fun Xj acc -> C1(Xi,Xj) + acc),
                    0, Xi)
        in Xi + {H = num/den, pH = -log10(num/den) }
} ;;
```

La transformation `update_pH` est composée de 3 règles. Les deux premières traitent les conditions aux frontières de l'équation dont nous ne parlerons pas. La dernière calcule un pas d'itération de la suite définie précédemment. La primitive `neighborfold` itère les voisins de `Xi` pour en récupérer la valeur. L'expression `C1(Xi,Xj)` correspond au coefficient C_1^{ij}. Enfin, `Xi` est remplacé par `Xi + { H = num/den, pH = -log10(num/den) }` qui dénote la nouvelle valeur de `Xi` où les champs `H` et `pH` sont mis à jour.

L'itération jusqu'au point fixe est spécifiée au moment de l'application de la transformation `update_pH` sur une collection `c` :

```
update_pH[fixpoint](c) ;;
```

Cette méthode de résolution est différente de celle fournie dans [Bot00] qui consiste à inverser un système matriciel. Ici nous nous rapprochons d'une résolution par relaxation.

Mécanique et intégration

L'équation de calcul des forces agissant sur chaque nœud est directement traduisible en MGS. Elle calcule alors la nouvelle vitesse de chaque nœud :

```
trans update_velocities = {
  Xi:NRnode => ...;
  Xi:Bnode => ...;
  Xi => (
    let zero = { fx = 0.0, fy = 0.0 } in
    let sum = fun Xj acc -> (
      let Xij = { x = Xj.px - Xi.px, y = ...} in
      let Lij = norm(Xij) and Fij = force(Xij) in
        { fx = acc.fx + Fij * Xij.x / Lij, fy = ...} )
    in
    let fi = neighborfold(sum, zero, Xi) in
      Xi + { vx = fi.fx / mu(Xi), vy = ...}
```

```
    )
} ;;
```

On remarquera l'utilisation de la primitive `neighborfold` pour calculer la somme des forces. L'ensemble des coefficients apparaissant dans le calcul des forces dépend uniquement des valeurs portées par les sommets et de la longueur des arcs, à l'exception de la longueur au repos des ressorts. Celle-ci est maintenue dans une structure à part (c'est-à-dire indépendamment de la collection) puisqu'il est impossible d'associer des valeurs aux arcs dans une collection de Delaunay, ceux-ci représentant uniquement la relation de voisinage.

L'intégration consiste à déterminer la nouvelle position de chaque nœud en fonction de la vitesse qui vient d'être calculée. Nous utilisons le schéma de résolution d'Euler pour simplifier la présentation de l'exemple :

```
trans euler_integration[dt] = {
  Xi => Xi + { px = Xi.px + dt * Xi.vx, py = ...}
} ;;
```

où un paramètre optionnel `dt` donne la valeur d'un pas de temps dans la simulation.

Mise à jour de la structure

La dernière étape de l'algorithme ajoute des nœuds aux endroits où les arcs sont trop grands et en retire où ils sont trop petits. La transformation suivante réalise cette modification de la structure :

```
trans update_structure = {
  polymerization :
    Xi, Xj / (length(Xi,Xj) > lmax) =>
      Xi, { pH = (Xi.pH+Xj.pH) / 2,...}, Xj

  depolymerization :
    Xi:Inode, Xj / (length(Xi,Xj) < lmin) => Xj
} ;;
```

où la fonction `length` retourne la longueur de l'arc entre deux sommets. Aussitôt que ces règles sont appliquées, le voisinage de Delaunay est automatiquement mis à jour.

Résultat

Le modèle ainsi programmé a été comparé au programme Matlab équivalent fourni avec [Bot00]. Au total, 750 lignes de codes auront suffi pour coder le modèle (dont 350 dédiées aux transformations présentées plus haut). Elles prennent en compte l'ensemble des fonctions nécessaires à la simulation, depuis l'initialisation des données jusqu'à la gestion de la sortie dans un format convenant à notre éditeur graphique Imoview. L'état initial et la valeur des paramètres respectent ceux de [Bot00] afin de respecter le modèle initial.

Notre sentiment est que le code développé en MGS est plus concis et plus lisible que celui de l'implantation Matlab équivalente. De plus, le caractère déclaratif du langage et les techniques de réécriture qu'il fournit, nous permettent d'écrire un code proche des équations mathématiques régissant les mécanismes chimiques et mécaniques en jeu dans le modèle ; on remarquera d'ailleurs qu'il n'a suffi de programmer qu'une seule transformation pour chaque équation, et que les différents modèles ont été combinés par simple composition de ces transformations.

En ce qui concerne l'efficacité, 6 minutes sur PC sous linux possédant un processeur Intel Xeon 1.2Ghz sont suffisantes pour générer une simulation de 800 pas de temps (considérant qu'à chaque pas, un point fixe doit être calculé pour atteindre l'équilibre chimique), ce qui est comparable aux informations fournies dans [BMR+02], et ce malgré le fait que l'interprète utilisé ne soit qu'une version prototype du langage ; il ne présente en effet aucune optimisation *ad-hoc* en ce qui concerne le filtrage ou les mécanismes de réécriture des collections Delaunay.

La figure 3 montre les images générées par cette simulation où l'on peut constater que la forme du spermatozoïde est conservée malgré un maillage qui a fortement évolué. Deux films correspondants sont disponibles en ligne[3].

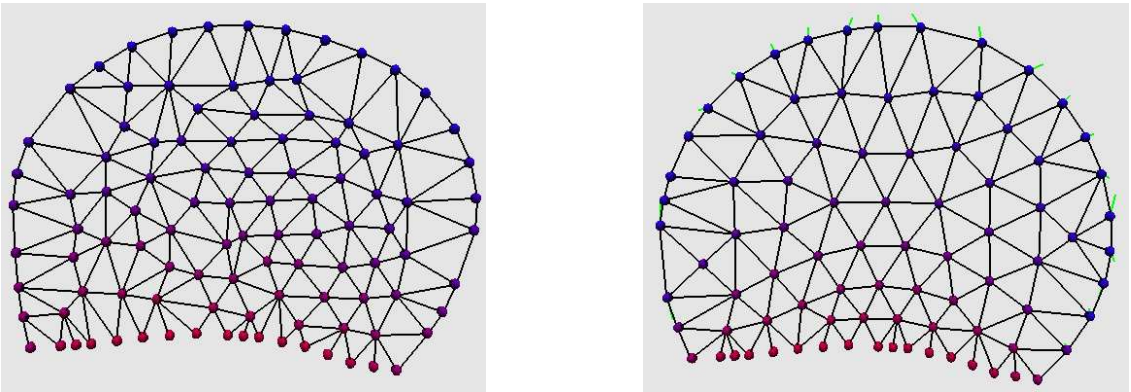


FIG. 3 – Images générées lors de la simulation du modèle par l'interprète MGS correspondant : à gauche l'état initial, et à droite l'état final après 800 itérations.

Avec cet exemple, nous souhaitons mettre en avant que l'utilisation d'un paradigme de calcul fondé sur les interactions locales (par l'utilisation de la réécriture) produit bien évidemment les mêmes résultats, en terme de simulation, qu'une modélisation globale (s'appuyant pour notre exemple sur du calcul matriciel). L'expressivité se voit alors améliorée considérablement pour se rapprocher du modèle mathématique initial. D'autre part, la notion de réécriture a dans cet exemple parfaitement joué son rôle, nous ramenant naturellement vers deux propriétés essentielles des systèmes dynamiques à structure dynamique :

- elle a permis dans un premier temps de ne spécifier que les interactions *locales* par le biais des motifs de filtre et par un accès facile au voisinage des éléments (grâce notamment à la primitive `neighborfold`) ;
- la dynamique de la structure a été parfaitement retranscrite, chaque nœud évoluant indépendamment tout en respectant les contraintes imposées par son voisinage, et le calcul du voisinage étant fourni « gratuitement » grâce à la collection topologique de Delaunay. La triangulation est faite de façon implicite (ce qui n'est pas le cas pour l'implantation MatLab) alors que son importance était primordiale dans le modèle (par exemple pour le calcul du gradient de pH).

[3] `http://mgs.lami.univ-evry.fr/ImageGallery/EXEMPLES/Spermcrawling/`, fichiers `ascaris.avi` et `ascaris2.avi`

3 MGS et la physique discrète

L'un des prolongements naturels de la modélisation du solide consiste à la généraliser pour modéliser des propriétés spatialement et temporellement distribuées sur des solides.

E. Tonti a été un précurseur dans ce domaine. Il a analysé la structure géométrique, algébrique et analytique partagée par différents modèles physiques afin de considérer les analogies qui existent entre eux. Cette étude a mis en évidence l'association des quantités physiques à des éléments géométriques de l'espace (points, lignes, surfaces et volumes) et du temps (instants et intervalles). Il élabora à partir de ce constat une classification des quantités et des équations dans chaque domaine de la physique [Ton74].

L'association des quantités physiques à des éléments d'espace-temps permet une formulation discrète des modèles physiques classiques. Cette formulation passe par les concepts développés en topologie algébrique [Sha01] : incidences, chaînes, cochaînes, bord, cobord, etc. Cette spécification directement discrète permet notamment d'éviter de passer par le schéma classique où une formulation à partir d'équations différentielles est discrétisée afin d'être résolue numériquement. On pense en particulier aux méthodes des différences finies ou des éléments finis. E. Tonti a développé une méthode numérique pour la résolution d'équations de champ reposant sur la formulation discrète des lois du champ, sans passer par une spécification différentielle. Il montre en particulier que sa méthode coïncide avec, ou améliore (suivant le type d'interpolation utilisé), celle des éléments finis [Ton01].

Ce type de méthode a été étendue pour mettre en place un véritable calcul différentiel discret dont nous avons déjà discuté au début du chapitre 6. La principale motivation est de retrouver l'expression de propriétés données par le calcul différentiel standard, comme le théorème de Stokes, mais dans une version discrète. Nous notons en cela le considérable apport de [Hir03] qui a posé les fondements du *calcul extérieur discret* où l'on trouve l'expression des équivalents discrets d'une grande partie des opérateurs du calcul extérieur standard, dans un cadre général. Ces travaux sont également développés dans [Leo04]. Une bonne introduction à ces travaux est donnée dans [DKT06].

Du point de vue langage dans lequel nous nous plaçons, nous nous sommes intéressés aux formalismes issus de ces considérations et qui permettent l'expression de modèles physiques. En particulier, le langage de programmation CHAINS développé par R. Palmer et V. Shapiro [PS93, Pal94] met en œuvre ces concepts ; on y trouve une structure de données fondée sur le concept de chaîne topologique avec la prise en charge d'une orientation relative entre les cellules, ainsi qu'une multitude d'opérateurs pour construire les chaînes et manipuler les valeurs associées aux cellules topologiques (déplacement de valeurs vers des cellules incidentes, avec ou sans prise en compte de l'orientation, etc.). On trouve dans [ES04] une extension de ce langage sous la forme d'une API où la principale nouveauté permet de construire des chaînes de chaînes.

Dans [ES04], les auteurs développent plusieurs exemples d'utilisation de l'API pour la simulation et l'implantation d'opérations topologiques. Nous nous sommes inspirés d'un de leurs exemples, la simulation de diffusion de particules dans un fluide, pour montrer que MGS dispose de mécanismes équivalents. Pour cela, nous avons traduit littéralement leurs programmes en $\langle n, p\rangle$-transformations. La figure 4 montre l'évolution de la densité de particules sur une surface triangulée. Ils proposent également la programmation de l'algorithme de subdivision fondé sur le masque de Cattmul-Clark (voir chapitre 8), mais son expression ne semble pas être simple, contrairement à notre proposition.

Les considérations précédentes et cette expérience de traduction de l'API de [ES04], même si elle est limitée, nous font penser que MGS dispose de toutes les notions nécessaires au développement des approches de type physique discrète dans le domaine de la simulation. La

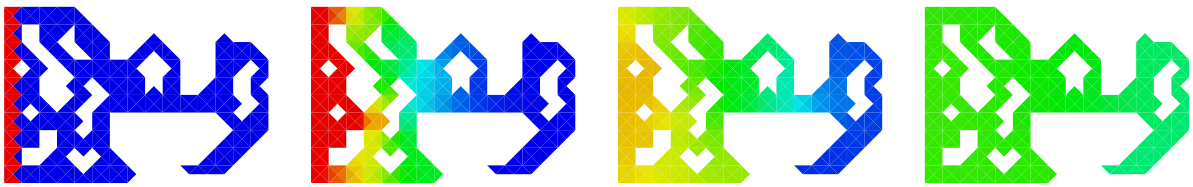


FIG. 4 – Évolution de la densité de particules sur une surface non-homogène. La densité est figurée de la façon suivante : en gris foncé une très forte densité de particules, en noir l'absence de particule et en gris clair la densité à l'état d'équilibre. Les images montrent l'évolution au cours du temps de gauche (où les particules sont concentrées à gauche de la surface) à droite (où l'état d'équilibre est atteint).

notion de transformation englobe en effet une très grande variété d'opérateurs, aussi bien dans les opérations de modifications topologiques (voir chapitre 8) que dans les opérations de manipulation de champs physiques (comme nous l'avons constaté ci-dessus et dans le chapitre 6).

Chapitre 10

La stratégie ‘gillespie

En préambule à ce chapitre, nous souhaitons préciser que les travaux décrits dans ce chapitre ont été initiés durant ma visite au Professeur Przemyslaw Prusinkiewicz à l'Université de Calgary (Canada) en Avril 2006. Ces travaux ont donné lieu à la publication [SMC+06] dont nous nous sommes très largement inspirés pour ce chapitre.

Dans ce chapitre, nous nous intéressons à une stratégie d'application des règles spécialisée issue de l'algorithme de simulation exacte de réaction chimique proposé par D.T. Gillespie [Gil77]. Son utilisation est illustrée à travers deux exemples de modélisation biologique.

D.T. Gillespie a développé une méthode stochastique [Gil77] pour la simulation de systèmes chimiques homogènes. Cette méthode et ses différentes évolutions [Gil00, GB00, Gil01] ont récemment trouvé nombre d'applications dans le domaine de la simulation de systèmes biologiques. La force de la simulation stochastique est qu'elle permet de manipuler des systèmes chimiques où le nombre de molécules est faible, ne pouvant donc pas être bien caractérisés par une approche classique fondée sur les équations cinétiques des réactions chimiques et des concentrations des réactants. En effet, cette dernière se révèle trop faible et perd tout son sens pour un nombre de molécules peu élevé.

Ce chapitre s'intéresse à l'utilisation de cet algorithme pour paramétrer l'application des règles d'un système de réécriture MGS. La métaphore chimique ou biochimique est à l'origine de nombreux formalismes. Ces formalismes sont souvent fondés sur la réécriture de multi-ensemble : les règles correspondent aux différents types de réactions, les éléments des multi-ensembles aux molécules pouvant interagir, et la structure de multi-ensemble représente une solution chimique homogène où toutes les molécules d'une espèce ont la même probabilité de rencontrer les autres molécules. Ces formalismes sont, pour ne nommer qu'eux, la CHAM[1] [BB90], le langage Gamma [BL86, BL90, BFL01], et le calcul par membrane provenant d'une extension de la réécriture de multi-ensemble à la réécriture de multi-ensembles imbriqués [Pău01, Car04]. Ils proposent pour la plupart de dériver[2] un multi-ensemble en appliquant de façon maximale parallèle les règles de réécriture. Cependant, une telle application des règles rend difficile la prise en compte des cinétiques chimiques lorsque les formalismes sont utilisés pour simuler une équation chimique réelle. L'algorithme de D.T. Gillespie proposant une telle forme d'application, nous avons « détourné » son utilisation de la simulation de systèmes chimiques vers la dérivation des systèmes de réécriture.

Nous proposons l'utilisation de cette stratégie en MGS pour deux raisons. Tout d'abord, par son point de vue générique des structures de données, MGS permet la manipulation de multi-ensembles en tant que collections topologiques illustrant donc l'utilisation de l'algorithme de D.T. Gillespie dans ce cadre. La seconde raison concerne l'application de cette stratégie sur des structures de données non-réduites à un multi-ensemble. En effet, l'un des enjeux actuels de la simulation de systèmes chimiques vise à étendre l'algorithme de D.T. Gillespie à la modélisation de solutions non-homogènes [TAT05, LDVS05, Cie06] (où les concentrations en espèces chimiques diffèrent d'un point à l'autre de l'espace, l'espace lui-même évoluant), celui-ci rendant compte des lois de probabilité pour des simulations stochastiques *exactes* de réactions dans un milieu *homogène*. Nous proposons notamment dans ce chapitre une extension de l'algorithme de D.T. Gillespie à une structure de multi-ensembles imbriqués (réécriture proche du calcul par membrane) permettant d'isoler un sous-ensemble d'individus évoluant indépendamment du reste du système.

Nous commençons par présenter l'algorithme de D.T. Gillespie et l'utilisation que nous en faisons en MGS pour implanter une nouvelle stratégie d'application de règles pour les transformations. Nous présentons notamment notre extension de l'algorithme à des multi-ensembles imbriqués. Pour illustrer ces propos, nous développons deux exemples, le premier concerne la simulation du switch génétique du phage λ, une utilisation directe de l'algorithme de D.T. Gillespie, et le second, la simulation d'un modèle d'infection virale où des multi-ensembles imbriqués sont mis en jeu.

1 L'algorithme de D.T. Gillespie

1.1 Réactions chimiques de simulation stochastique

D'un point de vue informatique, la méthode de D.T. Gillespie repose sur une simulation à évènements discrets [Kre86] des réactions entre des molécules individuelles. Une réaction R_μ, par exemple $A + B \rightarrow C$, peut se produire quand les molécules réactantes (A et B) se heurtent avec suffisamment d'énergie pour créer le produit (molécule C). La probabilité $P(\mu, d\tau)$ qu'une réaction R_μ se produise durant un intervalle de temps infinitésimal $d\tau$ est proportionnelle à

[1] pour CHemical Abstract Machine.

[2] Une dérivation correspond à une étape de réduction d'un système de réécriture.

- la *constante stochastique* c_μ de la réaction, dont la valeur dépend de la nature de la réaction et de la température,
- le nombre h_μ de combinaisons moléculaires distinctes du système pouvant être à l'origine de la réaction R_μ (si $[X]$ dénote le nombre total de molécules d'espèce X, le nombre de combinaisons moléculaires h_μ de notre exemple vaut $[A][B]$),
- la longueur de l'intervalle de temps $d\tau$,

de telle sorte que :

$$P(\mu, d\tau) = h_\mu c_\mu d\tau = a_\mu d\tau, \tag{1}$$

où le produit $a_\mu = h_\mu c_\mu$ est appelée la *propension* de la réaction R_μ.

Soit $X(t)$ l'état du système considéré à l'instant t. Cet état est caractérisé par le nombre de molécules de chaque espèce X_i pour $i = 1, 2, \ldots, N$. D.T. Gillespie a montré dans [Gil77] que la probabilité $\tilde{p}(\tau, \mu)d\tau$ avec laquelle la prochaine réaction aura lieu dans l'intervalle de temps infinitésimal $[t + \tau, t + \tau + d\tau]$, est donnée par :

$$\tilde{p}(\tau, \mu)d\tau = a_\mu e^{-a_\mu \tau} d\tau, \tag{2}$$

L'équation 2 diffère de l'équation 1 par le terme $e^{-a_\mu \tau}$, fournissant la probabilité qu'aucune réaction R_μ ne se produise avant, c'est-à-dire durant l'intervalle de temps $[t, t + \tau]$. Si le nombre de réactions chimiques différentes mises en jeu par le système est noté M, la probabilité que la prochaine réaction produite soit R_μ et qu'elle ait lieu pendant l'intervalle de temps $[t + \tau, t + \tau + d\tau]$, est donnée par :

$$p(\tau, \mu)d\tau = a_\mu e^{-a_0 \tau} d\tau, \tag{3}$$

où $a_0 = \Sigma_{\nu=1}^{M} a_\nu$ calcule la propension combinée de l'ensemble des M réactions [Gil77]. En sommant les probabilités exprimées par l'équation 3 pour les M types de réaction, nous obtenons la probabilité $p_1(\tau)d\tau$ que la première réaction d'un type arbitraire se produise pendant l'intervalle de temps $[t + \tau, t + \tau + d\tau]$:

$$p_1(\tau)d\tau = \sum_{\mu=1}^{M} p(\tau, \mu)d\tau = \sum_{\mu=1}^{M} a_\mu e^{-a_0 \tau} d\tau = a_0 e^{-a_0 \tau} d\tau. \tag{4}$$

L'évolution de l'état du système au cours du temps est simulée en itérant les deux étapes suivantes :

- connaissant l'état du système $X(t)$, on détermine le type μ de la prochaine réaction et le temps d'*inter-réaction* τ à attendre avant que cette réaction ait lieu,
- on modifie l'état du système $X(t)$ en prenant en considération le nombre de réactants détruits et le nombre de molécules produites par la réaction R_μ,
- on fait progresser le temps de simulation t de τ.

D.T. Gillespie propose deux méthodes pour déterminer le type de réaction μ et le temps d'inter-réaction τ en accord avec les distributions de probabilité données par l'équation 3 :

méthode *directe* : le temps d'inter-réaction est donné par l'équation 4, considérant tous les types de réaction en même temps. Une réaction particulière est alors choisie selon leurs propensions.

méthode de la *première réaction* : le temps d'inter-réaction de chaque équation μ est choisi à l'aide de l'équation 2. La réaction présentant le temps d'inter-réaction la plus faible est appliquée pour mettre à jour le système.

La méthode directe est fondée sur la probabilité conditionnelle [Gil77, page 418]

$$p(\tau, \mu)d\tau = p_1(\tau)d\tau \cdot P_2(\mu|\tau), \tag{5}$$

où $p_1(\tau)d\tau$ est la probabilité que la prochaine réaction se produise durant l'intervalle de temps $(t + \tau, t + \tau + d\tau)$ (voir équation 4), et $P_2(\mu|\tau)$ est la probabilité que la prochaine réaction soit R_μ, sachant que la date de la réaction est $t + \tau$. La probabilité conditionnelle est obtenue en divisant l'équation 3 par l'équation 4 :

$$P_2(\mu|\tau) = p(\tau, \mu)/p_1(\tau) = a_\mu/a_0. \tag{6}$$

Le temps d'inter-réaction τ et la réaction R_μ sont choisis en accord avec les probabilités données par les équations 4 et 6 en utilisant la méthode d'inversion [Ros89, page 564]. Concrètement, le temps d'inter-réaction est donné à partir de deux nombres r_1 et r_2 de $[0, 1]$ produits indépendamment par un générateur de nombres aléatoires :

$$\tau = \frac{1}{a_0} \ln \frac{1}{r_1}, \tag{7}$$

et le type de réaction μ est déterminé en résolvant l'équation

$$\sum_{\nu=1}^{\mu-1} a_\nu < r_2 a_0 \leq \sum_{\nu=1}^{\mu} a_\nu. \tag{8}$$

Dans la méthode de la première réaction, le temps τ_ν de première réaction du type de réaction ν est choisi indépendamment des autres réactions pour chaque $\nu = 1, 2, \ldots, M$ avec la méthode d'inversion appliquée à l'équation 2. Pour cela, M nombres r_ν sont générés aléatoirement et indépendamment dans $[0, 1]$, et les temps τ_ν sont calculés à partir de l'équation 9, similaire à l'équation 7 :

$$\tau_\nu = \frac{1}{a_\nu} \ln \frac{1}{r_\nu} \quad \text{pour} \quad \nu = (1, 2, \ldots, M). \tag{9}$$

La plus petite valeur τ_ν est choisie comme temps d'inter-réaction, et l'état du système est mis à jour en appliquant la réaction R_ν.

1.2 Gillespie en MGS

L'algorithme de D.T. Gillespie permet la simulation de réactions dans des solutions chimiques homogènes. Si une telle solution est représentée par un multi-ensemble d'objets élémentaires représentant les molécules [BL86, DZB01], les réactions chimiques peuvent être exprimées par des règles de réécriture. L'algorithme de D.T. Gillespie conduit donc au développement d'une nouvelle stratégie d'application de ces règles : une et une seule règle est appliquée par dérivation (ou par pas de simulation) en accord avec les lois de probabilité précédemment décrites.

Algorithme de D.T. Gillespie en MGS. Les règles de réécriture sont paramétrées de la façon suivante :

1. `=`{`C=`c_μ`}=>` pour spécifier explicitement la constante stochastique c_μ de la règle ;
2. `=`{`A=\self.f(self)`}`=>` pour fournir directement la propension de la règle.

Dans le second cas, `f(self)` est une fonction s'appliquant sur la collection passée en argument à la transformation. Par exemple, sur la règle suivante

```
‘X, ‘Y ={A=\self.count(self,‘X)*count(self,‘Y)}=> ‘Z
```

la propension de la règle est calculée par l'évaluation de la fonction fournie en paramètre de la règle qui retourne le nombre de symboles `‘X` multiplié par le nombre de symboles `‘Y` contenus dans la collection argument `self`.

La stratégie stochastique fondée sur l'algorithme de D.T. Gillespie est utilisée par

```
T[strategy = ‘gillespie](c)
```

pour appliquer une transformation `T` sur une collection `c`. Cette application suppose que chaque règle présente une fonction de propension ou une constante stochastique, et échoue dans le cas contraire. Une variable globale prédéfinie `tau` contient le temps d'inter-réaction et peut être utilisée dans la définition du prélude, du postlude et de l'interlude. La stratégie implante la méthode de la première réaction.

1.3 Gillespie dans l'espace

Hormis les collections de type ensemble ou multi-ensemble, les collections topologiques violent la condition d'homogénéité requise dans le contexte défini par D.T. Gillespie. Bien que la stratégie puisse être utilisée sur n'importe quel type de collection, laissant au programmeur le soin de donner un sens à cette application, nous présentons une extension de l'utilisation de l'algorithme de D.T. Gillespie dans le cadre de multi-ensembles imbriqués :

- les réactions ayant lieu dans les compartiments (nom donné aux sous-ensembles imbriqués) sont simulées en considérant chaque compartiment individuellement (nous supposons que que les molécules sont réparties de façon homogène dans chaque compartiment),
- des règles de réécriture impliquant des transports de molécules, des créations et des dissolutions de compartiments, présentent leurs propres propensions et sont traitées comme des réactions impliquant des compartiments considérés alors comme des individus atomiques. Deux compartiments peuvent ainsi réagir ensemble (pour fusionner par exemple).

Nous définissons un pas de dérivation d'un tel système par analogie avec les méthodes directe et de première réaction. Dans la méthode directe, une réaction d'un certain type, mais associée à différents compartiments, est appliquée indépendamment dans chaque compartiment avec la propension correspondant à ce compartiment. Formellement, la règle est renommée pour chaque compartiment pour créer autant de nouveaux types de réaction et se retrouver dans le cadre général de l'algorithme de D.T. Gillespie.

Dans la méthode de première réaction, l'algorithme de D.T. Gillespie est appliqué dans chaque compartiment m séparément, sélectionnant alors une réaction R_m et un temps de réaction τ_m pour chaque compartiment. Le compartiment avec le temps de réaction le plus faible est le seul à évoluer. L'algorithme ci-dessous décrit cette méthode.

Soit `split` une fonction divisant un multi-ensemble m en deux parties : les molécules contenues dans le multi-ensemble d'une part, et les compartiments d'autre part. Soit R_m, l'ensemble

des règles applicables sur le multi-ensemble m, et $\langle \tau; p \rangle = SSA(m)$ le résultat de l'application d'une de ces règles en accord avec l'algorithme original de D.T. Gillespie. Dans le couple $\langle \tau; p \rangle$, τ est le temps d'inter-réaction relatif à l'évolution de m en p. L'algorithme est le suivant :

function *nestedSSA* (m **: nested multiset**)
  $\langle m'; m'' \rangle$:= split(m)
  $\langle \tau_0; n_0 \rangle$:= $SSA(m')$
  let N = **size**(m'')
  for $i = 1$ **to** N **do**
    $\langle \tau_i; n_i \rangle$:= $nestedSSA(m''_i)$
  let j **such that** $\tau_j = \min_{(0 \leq i \leq N)} \tau_i$
  if $j = 0$ **then return** $\langle \tau_0; (n_0, m'') \rangle$
  else return $\langle \tau_j;\ m' :: m''_1 :: \ldots :: m''_{j-1} :: n_j :: m''_{j+1} :: \ldots :: m''_N \rangle$

2 Switch génétique du phage λ

La simulation stochastique de systèmes biochimiques est intéressante quand le nombre de molécules et/ou l'intervalle de temps sont faibles. Nous décrivons dans cette section l'usage de MGS pour la description et la simulation d'un processus biologique très connu, la *régulation du phage* λ.

2.1 Contexte biologique

Le bactériophage λ [Pta92] est un virus qui infecte les cellules de la bactérie *Escherichia coli*[3]. Ce phage dispose de deux voies de développement :

1. réplication et *lyse* (dissolution et destruction) de la cellule hôte, libérant ainsi environ 100 virions,
2. intégration de son ADN dans l'ADN de la bactérie, et entrée en phase de *lysogénie*.

En phase lysogénique, le virus va se dupliquer de façon silencieuse à chaque fois que la bactérie se divise. De plus, une bactérie en phase de lysogénie dispose d'une immunité face à de futures infections du phage, protégeant la bactérie d'une destruction possible due à une nouvelle infection lytique. Sous certaines conditions (exposition aux U.V. par exemple) une lyse peut être *induite* à partir de la voie lysogénique : l'ADN viral s'extrait de l'ADN bactérien et entame une réplication normale et une lyse.

En résumé, le virus qui infecte la bactérie a deux devenirs possibles : la *lyse* ou la *lysogénie*. Suivant le milieu, l'une ou l'autre de ces voies est privilégiée, la décision étant sous le contrôle d'une petite région du génome du phage (une centaine de paires de bases, comparativement aux 48502 paires de bases de l'ADN), de deux gènes (cI et cro) et deux promoteurs[4]. On appelle cette région de régulation le *switch génétique*.

Cette région de l'ADN (voir figure 1) du phage λ est composée de deux gènes cI et cro codant respectivement pour les protéines CI et CRO. Lors de la transcription, l'ARN polymérase[5] se fixe sur les promoteurs de ces gènes (respectivement Prm et Pr) pour synthétiser l'ARNm ensuite traduit en protéines monomères CI et CRO. Ces monomères se dimérisent en CI_2 et CRO_2. Ces

[3]Cet organisme unicellulaire, découvert par T. Escherich en 1885, est un hôte normal de la flore intestine chez l'homme et dont certaines souches sont toxiques.

[4]Un *promoteur* est une séquence d'ADN précédant un gène sur laquelle l'ARN polymérase se fixe pour commencer la transcription du gène en ARNm.

[5]L'*ARN polymérase* est l'enzyme responsable de la transcription.

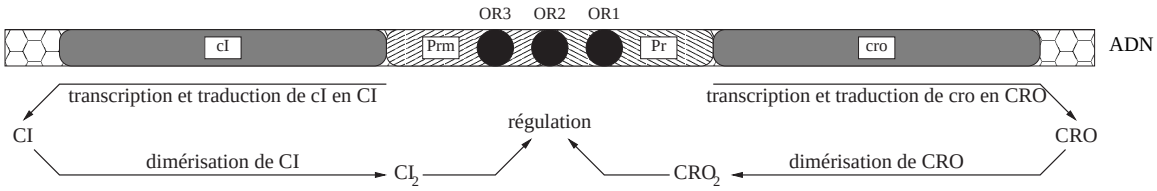


FIG. 1 – Description de la région du switch génétique (provenant de [Mes06b]) : l'ADN présente deux gènes, cI et cro, codant respectivement pour les protéines CI et CRO. Celles-ci sont capables de dimériser puis de se fixer sur les opérateurs OR1, OR2 et OR3 : OR1 chevauche Pr, OR3 chevauche Prm et OR2 chevauche à la fois Prm et Pr. La fixation de l'ARN-polymérase sur ces promoteurs (entraînant la transcription des deux gènes) est activée ou inhibée par les dimères, et est donc régulée.

derniers peuvent se fixer sur les opérateurs[6] OR1, OR2 et OR3 chevauchant les promoteurs (voir figure 1). L'absence ou la présence de dimères liés aux promoteurs, gène ou facilite suivant le cas la fixation de l'ARN polymérase, ayant pour conséquence la régulation de l'expression de cI et cro. La liaison des dimères aux promoteurs répond à certaines règles d'affinité :

- CI_2 se lie de préférence à OR1, puis OR2 et enfin OR3 ;
- CRO_2 se lie de préférence à OR3, puis OR2 et enfin OR1 ;
- si CI_2 est lié à OR1, il facilite la liaison d'un autre dimère CI_2 à OR2 et par conséquent la fixation de l'ARN polymérase sur le promoteur Prm. Cette propriété n'est pas vérifiée par CRO_2 ;
- l'ARN polymérase peut se fixer sur Pr mais CI_2 doit être lié à OR2 pour qu'elle puisse se fixer à Prm.

Combinées, ces règles créent un comportement complexe entraînant la régulation génétique ; pour simplifier, chacune des deux protéines CI et CRO va inhiber la synthèse de l'autre et augmenter sa propre synthèse (elles s'auto-inhibent tout de même lorsqu'elles sont trop exprimées). Selon les conditions de culture, une des deux protéines va prendre le dessus sur l'autre (en terme de concentration) et on entrera alors en phase lytique (CRO l'emporte) ou lysogénique (CI l'emporte).

2.2 Les équations biochimiques

La figure 2 présente le modèle utilisé pour notre implantation. Il respecte exactement les quatre règles de comportement des dimères CI_2 et CRO_2 présentées ci-dessus. Il est tout de même simplifié dans le sens où l'ensemble des liaisons dimères-opérateurs n'est pas considéré. En fait, le sous-ensemble auquel nous nous sommes restreints correspond aux liaisons les plus probables. Une liaison CI_2 sur OR2, avec OR1 libre, aurait pu être envisagée ; cette conformation reste rare [Mes06b]. Nous considérons également que cI est exprimé quand OR2 est occupé par CI_2 et OR3 est libre. Le gène cro est exprimé quand OR1 et OR2 sont libres.

L'ensemble des interactions (dimérisation, fixation des protéines sur l'ADN et expression des gènes) sont vues comme des équations biochimiques :

$$\text{CI} \qquad \longrightarrow_{C_0} \qquad . \tag{10}$$

[6]Les *opérateurs* sont des petites régions de l'ADN reconnues par des complexes moléculaires spécifiques induisant une *régulation* transcriptionnelle.

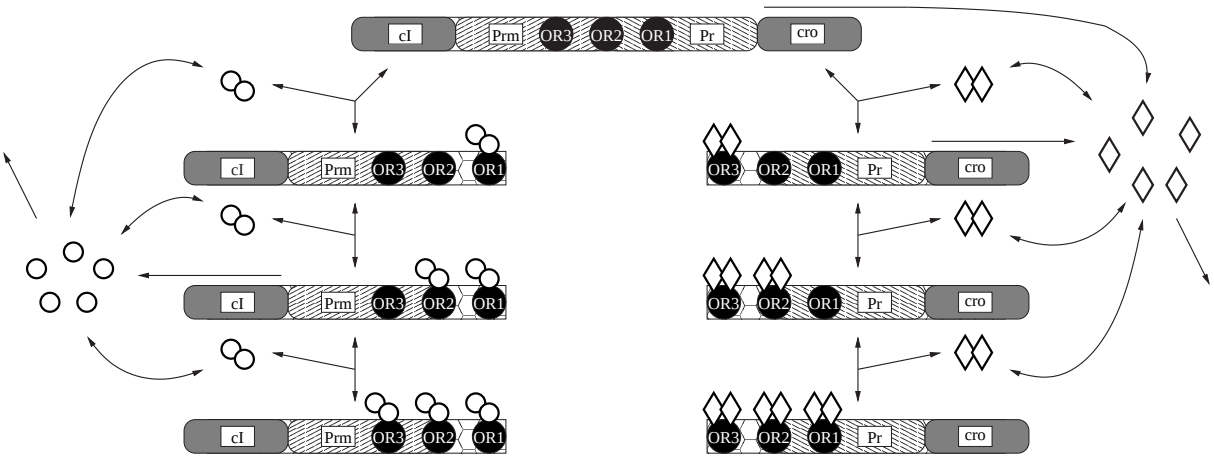


FIG. 2 – Réseau de régulation du switch génétique du phage λ : en ce fixant sur les opérateurs, CI_2 et CRO_2 régulent l'expression des gènes cI et cro. Les monomères CI et CRO sont respectivement représentés par des cercles et des losanges. Les dimères sont construits à partir des monomères. Le schéma respecte la description de l'ADN donnée figure 1.

$$
\begin{array}{lll}
2\,\mathrm{CI} & \longleftrightarrow_{C_{12}\,C_{21}} & \mathrm{CI}_2 \quad (11)\\
\mathrm{D} + \mathrm{CI}_2 & \longleftrightarrow_{C_1\,C_{-1}} & \mathrm{D}_1 \quad (12)\\
\mathrm{D}_1 + \mathrm{CI}_2 & \longleftrightarrow_{C_2\,C_{-2}} & \mathrm{D}_2 \quad (13)\\
\mathrm{D}_2 + \mathrm{CI}_2 & \longleftrightarrow_{C_3\,C_{-3}} & \mathrm{D}_3 \quad (14)\\
\mathrm{D}_2 + \mathrm{P} & \longrightarrow_{C_t} & \mathrm{D}_2 + \mathrm{P} + CI \quad (15)\\
\mathrm{CRO} & \longrightarrow_{C'_0} & . \quad (16)\\
2\,\mathrm{CRO} & \longleftrightarrow_{C'_{12}\,C'_{21}} & \mathrm{CRO}_2 \quad (17)\\
\mathrm{D} + \mathrm{CRO}_2 & \longleftrightarrow_{C'_3\,C'_{-3}} & \mathrm{D}'_3 \quad (18)\\
\mathrm{D}'_3 + \mathrm{CRO}_2 & \longleftrightarrow_{C'_2\,C'_{-2}} & \mathrm{D}'_2 \quad (19)\\
\mathrm{D}'_2 + \mathrm{CRO}_2 & \longleftrightarrow_{C'_1\,C'_{-1}} & \mathrm{D}'_1 \quad (20)\\
\mathrm{D} + \mathrm{P} & \longrightarrow_{C'_t} & \mathrm{D}_2 + \mathrm{P} + CRO \quad (21)\\
\mathrm{D}'_3 + \mathrm{P} & \longrightarrow_{C'_t} & \mathrm{D}_2 + \mathrm{P} + CRO \quad (22)
\end{array}
$$

Les équations 10 et 16 décrivent la dégradation naturelle des monomères CI et CRO. Les réactions 11 et 17 correspondent à la dimérisation de CI et CRO en CI_2 et CRO_2. Les équations 12–14 et 18–20 expriment les liaisons des dimères sur les opérateurs ; les différents états de l'ADN sont représentés : la constante D correspond à l'ADN sans liaison, D_1, D_2 et D_3 à l'ADN avec 1, 2 ou 3 dimères CI_2 fixés, et D'_3, D'_2 et D'_1 à l'ADN avec 1, 2 ou 3 dimères CRO_2 fixés (voir figure 2). Finalement, l'expression des gènes est donnée par les réactions 15, 21 et 22, où P est l'ARN polymérase. Chaque réaction est paramétrée par une constante stochastique, C_i pour les réactions concernant CI et C'_i pour CRO.

Du point de vue biochimique où nous nous plaçons, la bactérie est vue comme une solution chimique homogène où les réactions décrites ci-dessus se produisent.

2.3 Implantation en MGS

La traduction en programme MGS est immédiate. Chaque équation est traduite en une règle de réécriture (ou deux s'il s'agit d'une réaction réversible) et la valeur des constantes `C` correspondant aux valeurs des coefficients C_i et C'_i est déterminée par des expériences biologiques [KN05].

Ce jeu de constantes est donné par :

$k_0 = k'_0 = 0.005$
$k_{12} = k'_{12} = 0.01$
$k_{21} = k'_{21} = 0.25$
$k_1 = k_2 = k_3 = k'_1 = k'_2 = k'_3 = 9.768$
$k_{-1} = k_{-2} = 15.0$
$k_{-3} = 2022.38$
$k'_{-1} = k'_{-2} = 245.371$
$k'_{-3} = 29.77$
$k_t = k'_t = 0.5$

Le programme 6 est la traduction des réactions biochimiques en MGS, où les différents complexes moléculaires mis en jeu sont représentés de façon abstraite par des symboles. La bactérie, vue

```
trans Phage = {
//
// Règles pour CI
//
  ‘CI          ={ C = 0.005   }=> <undef>        ;
  #2 ‘CI       ={ C = 0.01    }=> ‘CI2           ;
  ‘CI2         ={ C = 0.25    }=> #2 ‘CI         ;
  ‘D0, ‘CI2    ={ C = 9.768   }=> ‘D1            ;
  ‘D1          ={ C = 15.0    }=> ‘D0, ‘CI2      ;
  ‘D1, ‘CI2    ={ C = 9.768   }=> ‘D2            ;
  ‘D2          ={ C = 15.0    }=> ‘D1, ‘CI2      ;
  ‘D2, ‘CI2    ={ C = 9.768   }=> ‘D3            ;
  ‘D3          ={ C = 2022.38 }=> ‘D2, ‘CI2      ;
  ‘D2, ‘P      ={ C = 0.5     }=> ‘D2, ‘P, ‘CI   ;
//
// Règles pour CRO
//
  ‘CRO         ={ C = 0.005   }=> <undef>        ;
  #2 ‘CRO      ={ C = 0.01    }=> ‘CRO2          ;
  ‘CRO2        ={ C = 0.25    }=> #2 ‘CRO        ;
  ‘D0, ‘CRO2   ={ C = 9.768   }=> ‘D’3           ;
  ‘D’3         ={ C = 29.77   }=> ‘D0, ‘CRO2     ;
  ‘D’3, ‘CRO2 ={ C = 9.768   }=> ‘D’2           ;
  ‘D’2         ={ C = 245.371 }=> ‘D’3, ‘CRO2    ;
  ‘D’2, ‘CRO2 ={ C = 9.768   }=> ‘D’1           ;
  ‘D’1         ={ C = 245.371 }=> ‘D’2, ‘CRO2    ;
  ‘D0, ‘P      ={ C = 0.5     }=> ‘D0, ‘P, ‘CRO  ;
  ‘D’3, ‘P     ={ C = 0.5     }=> ‘D’3, ‘P, ‘CRO ;
};;
```

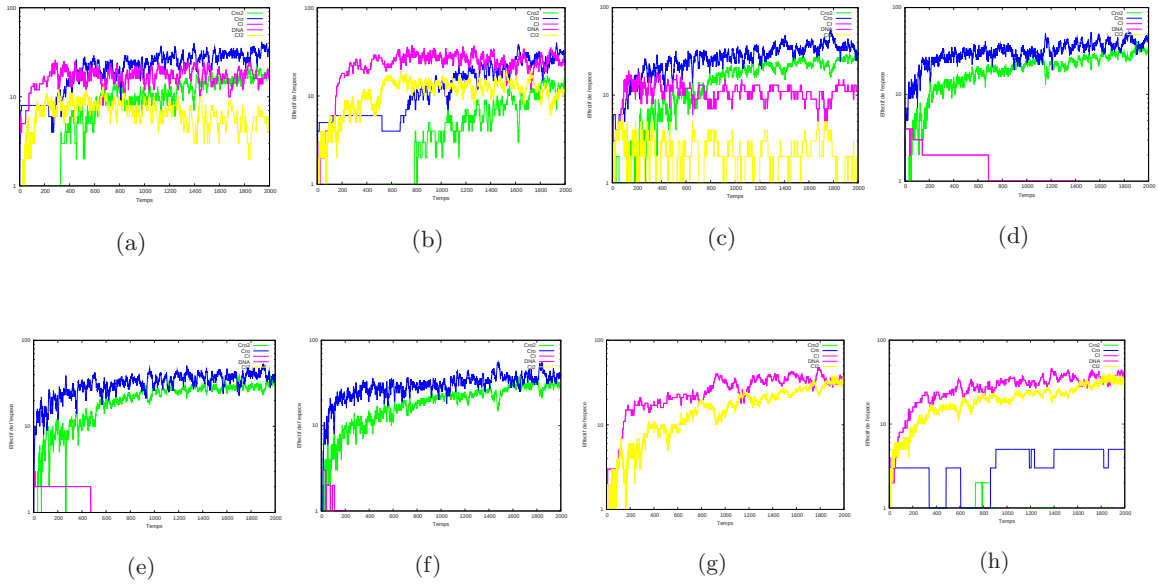
Prog. 6: Modèle du switch du phage λ en MGS.

comme une solution chimique où les molécules uniformément réparties sont animées par un

mouvement brownien, est représentée par un multi-ensemble MGS. À l'état initial, une molécule d'ADN est placée dans ce multi-ensemble, ainsi que quelques molécules d'ARN polymérase. Comme dans la réalité biologique, la probabilité que CI l'emporte sur CRO est faible ; pour que la simulation tourne à l'avantage de CI de façon significative, nous rajoutons quelques protéines CI à l'état initial :

```
Bact := ‘D :: #10 ‘P :: #15 ‘CI :: bag:() ;;
```

La figure 3 montre le résultat de huit applications de la transformation `Phage` avec la stratégie `‘gillespie` proposée par l'interprète ; ces graphiques ont été générés par GNUPLOT (les instructions nécessaires pour cette sortie ne sont pas reproduites ici). Les trois premières figures montrent le système dans un état qui n'a pas encore évolué vers une phase de lyse ou de lysogénie ; les trois figures suivantes montrent le système ayant effectué le switch pour la lyse (seuls CRO et CRO_2 persistent) ; enfin, les deux dernières figures montrent le système ayant évolué vers la voie lysogénique (seuls CI et CI_2 sont présents).



(a) (b) (c) (d)

(e) (f) (g) (h)

FIG. 3 – Résultats de la simulation sous GNUPLOT du switch du phage λ. Les 3 premières figures montrent un état de la bactérie où le switch vers la lyse ou la lysogénie n'est pas encore déterminé ; les trois figures suivantes montrent le système en état de lyse, les deux dernières figures le montrent dans un état lysogénique. Toutes ces simulations ont consisté en 2000 itérations de la transformation `Phage` (voir programme 6) sur l'état initial `Bact` défini dans le texte, en utilisant la stratégie `‘gillespie` pour l'application des règles. La variété des résultats montre parfaitement la nature stochastique du phénomène de switch.

L'utilisation de l'algorithme de D.T. Gillespie pour simuler l'évolution de la solution apporte une représentation de l'espace implicite : en effet, on suppose que dans la solution, un mouvement brownien anime les molécules qui peuvent alors se rencontrer. L'intérêt de l'algorithme de D.T. Gillespie est la capture de ce mouvement à travers des probabilités, nous permettant de ne pas avoir à modéliser le déplacement spatial des molécules. On trouve dans [Mes06a] une description du même modèle par un système multi-agents ; les molécules sont représentées par des agents et se déplacent aléatoirement dans un espace déterminé.

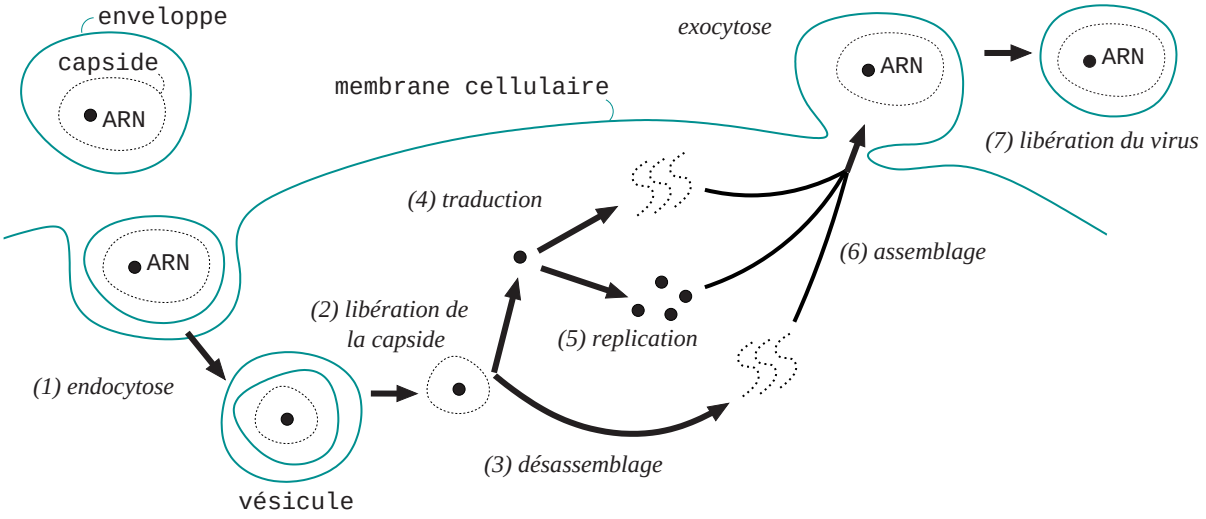


FIG. 4 – Croquis approximatif en 7 étapes de l'infection d'une cellule par le virus de la forêt de Semliki. Cette image est inspirée de [Car04] et reprise de [SMC+06].

3 Modèle d'une infection virale

Nous présentons dans cette section un modèle de haut niveau d'une infection virale qui suit le processus mis en avant dans [ABL+94, pages 273-280]. Cet exemple requiert en particulier la formation et la dissolution de membranes ainsi que le transport de molécules seules et de compartiments entiers. Ce processus a déjà été modélisé en *brane calculus* [Car04], dans lequel les compartiments imbriqués sont traités de façon similaire aux P système [Pău01]. Néanmoins, le brane calculus ne fournit pas les moyens suffisants pour embarquer le caractère stochastique des réactions moléculaires.

3.1 Contexte biologique

Les virus sont du matériel génétique encapsulé dans une protéine, rendant possible sa transmission d'une cellule à une autre. La structure et le cycle de vie du virus de la forêt de *Semliki* sont décrits figure 4. Le virus consiste en un seul brin d'*ARN viral* entouré d'une coquille appelée *capside*. La capside est composée de nombreuses copies de la même protéine C. En dehors d'une cellule, le virus présente une seconde couche externe de protéines appelée l'*enveloppe*. Une infection commence lorsque le virus se lie à un récepteur de la membrane de la cellule hôte (phase 1 de la figure 4). Le virus pénètre alors une cellule saine par une endocytose[7] classique. De par l'endocytose, le virus est doté d'une coquille supplémentaire, appelée *vésicule*, dont l'origine est la membrane cellulaire. Un processus de dissolution résorbe alors à la fois la vésicule et l'enveloppe, laissant le virus encapsulé dans sa capside (phase 2). La capside se désassemble en le brin d'ARN et les protéines C (phase 3). L'ARN viral est alors traduit en protéines structurelles du virus (phase 4), et dupliqué (phase 5). Les anciennes protéines C et les protéines nouvellement synthétisées se lient aux nouveaux brins d'ARN formant ainsi de nouvelles capsides (phase 6). Lorsqu'une capside s'approche de la membrane cellulaire, un mécanisme d'*exocytose*[8] recons-

[7] L'endocytose est le mécanisme de transport des molécules vers l'intérieur de la cellule : la membrane cellulaire s'invagine autour des molécules devant pénétrer dans la cellule pour finalement l'entourer complètement. Le compartiment se détache alors de la membrane se retrouvant à l'intérieur de la cellule

[8] Mécanisme inverse de l'endocytose.

truit l'enveloppe virale et laisse échapper dans le milieu extra-cellulaire un nouveau virus prêt à infecter une autre cellule saine (phase 7).

3.2 Modèle stochastique avec compartiments imbriqués

Nous modélisons l'attaque du virus de la forêt de Semliki de la façon suivante : un compartiment de type *Univ* représente le système complet, comprenant une population de virus et de cellules saines. Une cellule saine est un compartiment vide de type *Cell* (nous omettons la représentation de la structure interne des cellules saines qui n'entre pas en jeu dans notre modèle). En revanche, une cellule infectée contient les virus et leurs composants. Un virus du milieu extra-cellulaire est un compartiment de type *Env* dans lequel se trouve un compartiment de type *Cap*. Finalement, ce dernier compartiment contient une molécule d'ARN. À l'intérieur d'une cellule, le compartiment de type *Env* est contenu dans un compartiment de type *Ves*.

En dénotant un compartiment de type T par une paire d'accolades $[\![\dots]\!]_T$, l'endocytose (phase 1) correspond à la « réaction chimique » suivante :

$$\Big[\!\Big[[\![RNA]\!]_{Cap} \Big]\!\Big]_{Env} + [\![]\!]_{Cell} \xrightarrow{Univ} C_1 \; \Big[\!\Big[\Big[\!\Big[[\![\mathrm{RNA}]\!]_{Cap} \Big]\!\Big]_{Env} \Big]\!\Big]_{Ves} \Big]\!\Big]_{Cell} \tag{23}$$

Cette règle est associée à un compartiment de type *Univ* où elle doit s'appliquer. La constante stochastique C_1 de cette réaction est proportionnelle à la probabilité qu'une cellule rencontre un virus. Cette probabilité est de la forme :

$$c_1 \; [Env] \; [Cell]$$

où c_1 est le coefficient de proportionnalité, $[T]$ est le nombre d'entités de type T pouvant réagir selon l'équation. L'opérateur + utilisé ici est celui des réactions chimiques.

L'ensemble complet des réactions chimiques est le suivant :

$$\Big[\!\Big[\Big[\!\Big[[\![\mathrm{RNA}]\!]_{Cap} \Big]\!\Big]_{Env} \Big]\!\Big]_{Ves} \xrightarrow{Cell} C_2 \quad [\![\mathrm{RNA}]\!]_{Cap} \tag{24}$$

$$[\![\mathrm{RNA}]\!]_{Cap} \xrightarrow{Cell} C_3 \quad \mathrm{RNA} + n\,\mathrm{C} \tag{25}$$

$$\mathrm{RNA} \xrightarrow{Cell} C_4 \quad \mathrm{RNA} + \mathrm{C} \tag{26}$$

$$\mathrm{RNA} \xrightarrow{Cell} C_5 \quad \mathrm{RNA} + \mathrm{RNA} \tag{27}$$

$$\mathrm{RNA} + n\,\mathrm{C} \xrightarrow{Cell} C_6 \quad [\![\mathrm{RNA}]\!]_{Cap} \tag{28}$$

$$\Big[\!\Big[[\![\mathrm{RNA}]\!]_{Cap} + \dots \Big]\!\Big]_{Cell} \xrightarrow{Univ} C_7 \quad \Big[\!\Big[[\![RNA]\!]_{Cap} \Big]\!\Big]_{Env} + [\![\dots]\!]_{Cell} \tag{29}$$

Toutes ces réactions sont associées à un compartiment de type *Cell* (c'est en effet à cet endroit que la réplication du virus a lieu), exceptée la dernière qui se situe au niveau englobant, associée à un compartiment de type *Univ*. Le coefficient n apparaissant dans les réactions 25 et 28 correspond au nombre de protéines C nécessaires à la construction d'un virus. Nous fixons ce coefficient à 5 pour l'implantation.

3.3 Implantation en MGS

Les types de compartiments définis ci-dessus sont représentés par des types de multi-ensembles MGS :

```
collection Univ = bag ;;
collection Cap  = bag ;;
collection Env  = bag ;;
collection Cell = bag ;;
collection Ves  = bag ;;
```

Un virus présent dans le milieu extra-cellulaire est par conséquent défini par :

```
(‘RNA :: Cap:()) :: Env:() ;;
```

Les processus décrivant l'infection se produisant dans les compartiments de types `Univ` et `Cell`, deux transformations sont utilisées pour décrire ces réactions. La première transformation

```
trans T_Univ = {
  P1 =
    e:Env, ce:Cell
      ={ A = \x.(c1 * count(Env,x) * count(Cell,x)) }=>
        (e :: Ves:()) :: ce ;
  P7 =
    (ce:Cell) \/ (ca:Cap)
      ={ A = \x.(c7 * countAll(Cell,Capsid,x)) }=>
        (ca :: Env:()), ce
} ;;


trans T_Cell = {
  P2 = (v:Ves) \/ ((e:Env) \/ (ca:Cap))
         ={ A = \x.(c2 * count(Ves,x)) }=>
           ca ;
  P3 = (ca:Cap) \/ ‘RNA
         ={ A = \x.(c3 * count(Cap,x)) }=>
           ‘RNA, #5‘C ;
  P4 = ‘RNA ={ C = c4 }=> ‘C,‘RNA ;
  P5 = ‘RNA ={ C = c5 }=> ‘RNA,‘RNA ;
  P6 = #5‘C, ‘RNA ={ C = c6 }=> ‘RNA :: Cap:() ;
} ;;
```

Prog. 7: Modèle de l'infection du virus de la forêt de Semliki en MGS

`T_Univ` (voir programme 7) décrit les réactions se produisant dans un compartiment de type *Univ*. La règle `P1` décrit la phase 1 de la figure 4, et la règle `P7` la phase 7. Les propensions sont données explicitement (paramètre `A` de chaque flèche). En effet, les motifs qui ne sont pas composés uniquement de constantes ne sont pas gérés directement par l'interprète (dans le cas contraire, on utilise le paramètre `C` donnant la constante de réaction, les propensions étant calculées de façon interne). La fonction donnée calcule donc le nombre d'occurrences de la règle. Pour cela, `countAll(Cell,Cap,u)` compte les capsides présentes dans l'ensemble des cellules

dans l'univers `u`. La seconde transformation est `T_Cell` (voir programme 7) et relate les phases $2-6$ de la figure 4. Dans la règle `P2`, l'opérateur `\/` est utilisé deux fois pour filtrer une vésicule encapsulant une enveloppe, contenant elle-même une capside. Les propensions sont données de façon explicite pour les règles `P2` et `P3`, et de façon implicite (paramètre `C` des flèches) pour les autres, l'interprète calculant lui-même les propensions.

3.4 Un exemple de simulation

La figure 5 montre le résultat d'une simulation ; à l'état initial, l'univers contient un virus pour vingt cellules saines :

```
initial_state := ((‘RNA :: Cap:()) :: Env:() ::
                  #20 Cell:() ::
                  Univ:() ;;
```

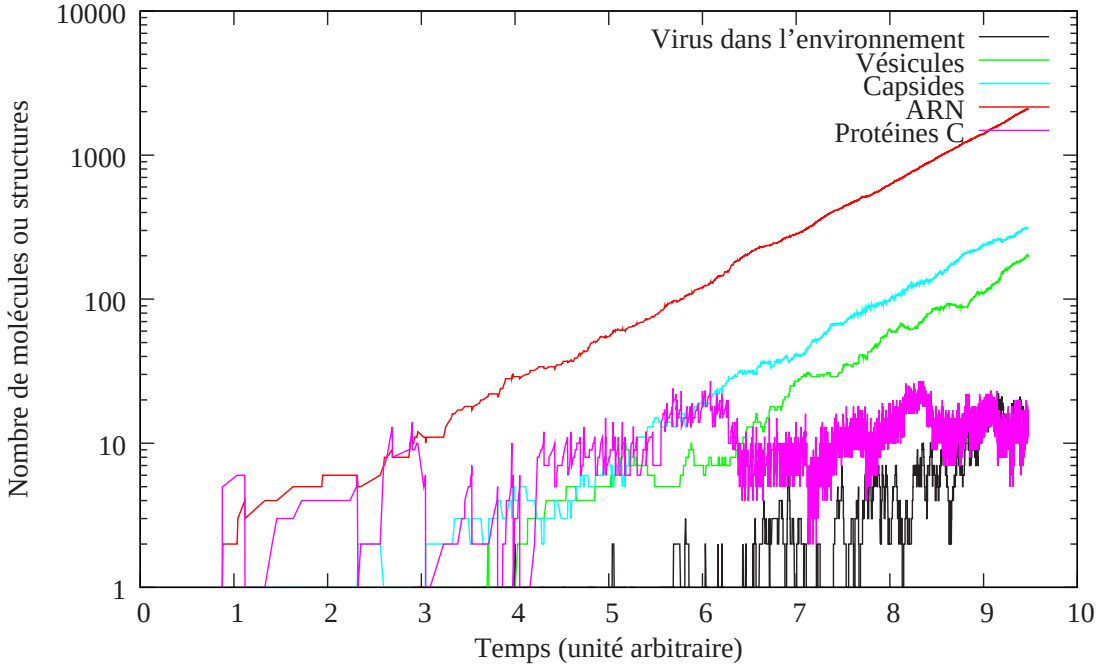


Fig. 5 – Résultats pour une simulation dont l'univers est composé d'un virus et de 20 cellules saines à l'état initial. Chaque courbe montre le nombre total de molécules seules ou de compartiments dans l'univers entier.

Toutes les constantes stochastiques sont les mêmes et arbitrairement fixées à 1.0. Le nombre total de molécules d'ARN, de vésicules, de capsides et de virus augmente de façon exponentielle en temps, comme attendu. La simulation révèle une variation significative en nombre de virus en accord avec leur petit nombre. Le nombre de protéines C libres dans les cellules augmente relativement doucement ; cela est dû à leur presqu'immédiate réutilisation pour former de nouvelles capsides. Une analyse plus poussée de ce modèle d'un point de vue biologique sera possible une fois que les constantes de réactions seront rendues disponibles par des données expérimentales.

Chapitre 11

Modification topologique et simulation biologique

1 Comprendre le développement pour expliquer le vivant

Cette section est un résumé de l'introduction de [GS06a].

1.1 De la biomécanique à l'auto-reproduction et au développement

R. Descartes (1596 – 1650) a développé une philosophie biomécaniste qui réduit les organes du corps humain aux pièces d'une machine « agencée par Dieu ». R. Descartes pense en effet comprendre la vie en comparant le corps à une machine : on peut expliquer les principales fonctions corporelles – la digestion, la locomotion, la respiration, mais aussi la mémoire et l'imagination – comme si elles résultaient d'un automate, à l'image d'une horloge destinée à montrer les heures par la seule disposition de ses roues et contrepoids. Mais la Reine Christine de Suède, que R. Descartes essaya de convaincre, lui rétorqua qu'elle y croirait quand il lui aurait montré comment une horloge peut se reproduire...

La démonstration prit trois siècles : il fallu attendre un article de John von Neumann en 1951, *The General and Logical Theory of Automata*, pour se convaincre qu'une machine peut effectivement construire une copie d'elle-même [vN66]. Pour J. von Neumann, qui a sur ce point une approche très fonctionnaliste, la mécanique est *in fine* ce qui peut être réduit à un

programme d'ordinateur et la reproduction correspond à dupliquer ce programme. L'objectif de J. von Neumann était clairement de montrer que les processus du vivant sont réductibles à des processus mécaniques, décrits par des opérations pouvant être exécutées de manière autonome, sans l'aide d'un « cornac caché » : une machine. Et pour J. von Neumann, comme pour la Reine Christine, la reproduction et le développement sont une caractéristique spécifique du vivant. Mais pour J. von Neumann cette caractéristique est juste une propriété particulière possédée par certaines machines, et non pas une qualité transcendant les processus physiques pour donner un statut particulier aux processus biologiques. L'existence d'une machine, d'un automate, capable de reproduction, est donc un élément de réponse crucial dans un débat fort ancien opposant le statut de la biologie à celui de la physique.

Le développement. Mais si les biologistes contemporains se posent les mêmes questions que les philosophes et les reines des siècles passés, la compréhension des mécanismes de la reproduction passe aujourd'hui par l'élucidation des *processus* qui conduisent de la cellule[1] germinale à l'organisme complet : il s'agit de comprendre étape par étape la *construction au cours du temps* d'un organisme à partir des très nombreuses interactions locales des éléments constituant l'organisme. On parle alors de *développement*.

Les éléments de réponse apportés par J. von Neumann sont très abstraits : ils reposent sur la description d'un automate cellulaire qui reproduit au cours du temps la configuration d'un sous-domaine spatial dans une région voisine. On voit que l'on est très loin des mécanismes moléculaires auxquels les biologistes contemporains veulent réduire les phénomènes biologiques. L'existence d'un automate auto-reproducteur est un argument qui indique qu'il n'y a pas de problème de principe à l'existence d'une telle machine, mais qui ne nous éclaire pas sur le « comment » mis en œuvre dans les processus biologiques.

1.2 Le développement comme un système dynamique

La notion de *système dynamique*[2] permet de formaliser la notion de processus de développement. Un système dynamique (SD) est caractérisé par des observations qui évoluent dans le temps. Ces observations sont les *variables* du système et sont reliées par certaines relations. Ces variables rendent compte des propriétés pertinentes du système (qu'elles soient biologiques, physiques, chimiques, sociologiques, ...). À un instant donné, elles prennent une valeur et l'ensemble de ces valeurs constitue l'*état du système*. L'ensemble de tous les états possibles d'un système constitue son *espace d'état* (ou espace des configurations).

La séquence temporelle des états du système est appelée une *trajectoire*. Un SD est un moyen formel pour spécifier comment on passe d'un point dans l'espace des configurations (un état) à un autre (l'état suivant). Ceci peut se faire directement par une fonction (la fonction d'évolution du système) ou indirectement en donnant des contraintes (équations) sur l'état futur possible (qui n'est pas nécessairement unique si le système n'est pas déterministe).

Dans les cas simples, la trajectoire d'un système dynamique peut s'exprimer explicitement par une fonction analytique du temps t. Par exemple, dans le cas de la pierre qui tombe, les équations différentielles $dx/dt = v$ et $dv/dt = \mathbf{g}$ peuvent s'intégrer explicitement pour donner la distance parcourue par la pierre en fonction du temps : $x = \mathbf{g}t^2/2$.

Dans les cas un peu plus complexes, une formule analytique donnant la trajectoire n'existe pas et la simulation par ordinateur est alors une approche privilégiée pour étudier les trajectoires du système. Par ailleurs, on peut s'intéresser non pas à une trajectoire particulière mais aux

[1] Dans toute cette section, le terme de cellule réfère à une cellule biologique.

[2] Voir aussi l'introduction de cette thèse et le chapitre 7.

propriétés qualitatives vérifiées par toutes les trajectoires possibles, comme par exemple : « si on attend assez longtemps, le système finira par prendre un état bien déterminé qu'il ne quittera plus » ou encore « si on passe par ces états, on n'y repassera jamais ». On parle de *propriétés émergeantes* quand il n'existe aucun moyen plus rapide pour les prédire que de les observer ou de les simuler. Notons que des SD dont la spécification est très simple peuvent donner des trajectoires extrêmes complexes (on parle parfois de *comportement chaotique*) ; d'autre part, le calcul de la trajectoire du système peut être coûteux en temps d'ordinateur et exiger beaucoup de mémoire.

Le développement comme trajectoire d'un système dynamique. Les étapes successives du développement peuvent se voir comme les états successifs de la trajectoire d'un système dynamique. Mais le développement implique le mouvement et la réorganisation de matière. En termes de système dynamique, cela veut dire que l'espace des états et sa topologie peuvent évoluer avec le temps. On retrouve la une notion que nous avons déjà introduite : un *système dynamique à structure dynamique*.

Illustrons cette notion en l'opposant à l'exemple de la pierre qui tombe : la vitesse et la position de la pierre changent à chaque instant mais ce système dynamique est toujours adéquatement décrit par un couple de vecteurs. Nous dirons dans ce cas que *la structure du SD est statique*. L'espace des états peut être décrit adéquatement à l'origine des temps, avant la simulation ; il correspond à l'espace des mesures du système. La valeur de ces mesures change avec le temps, mais la donnée de l'espace des états et sa topologie ne font pas partie des variables du système et elles ne peuvent pas évoluer au cours du temps.

Le cas est tout autre pour les processus de développement : les processus biologiques constituent des systèmes dynamiques hautement structurés et hiérarchiquement organisés, dont la structure spatiale varie au cours du temps et doit être calculée conjointement avec l'état du système.

Pour illustrer la notion de $(\text{SD})^2$ dans le cadre du développement, prenons pour exemple le développement d'un embryon. L'état de l'embryon est initialement décrit par l'état $s_0 \in \mathcal{S}$ de la cellule germinale (aussi compliquée cette description puisse-t-elle être). Après la première division, il faut décrire l'état de 2 cellules, c'est-à-dire un nouvel état $s_1 \in \mathcal{S} \times \mathcal{S}$. Mais dès que le nombre n de cellules de l'embryon est assez grand, l'état du système n'est pas décrit adéquatement par un élément de $\mathcal{S}^n$. En effet, cet ensemble décrit uniquement l'état de chaque cellule mais il ne contient pas l'information spatiale permettant de décrire le réseau des cellules (leur positionnement les unes par rapport aux autres). Or ce réseau est de la première importance car il conditionne la diffusion des signaux (chimiques, mécaniques, électriques) entre cellules, et donc *in fine* leur mode de fonctionnement. À chaque mouvement, division ou mort cellulaire, la topologie de ce réseau se transforme. Par exemple, lors de la phase de *gastrulation*, des cellules initialement éloignées deviennent voisines, permettant leur interaction et la modification de leur destin (différentiation des cellules).

1.3 Quel formalisme pour les systèmes dynamiques à structure dynamique ?

Nous avons indiqué que la position de chaque cellule changeait au cours du temps ce qui rend par exemple difficile la spécification des processus de diffusion entre les cellules. Une solution qui vient immédiatement à l'esprit est donc de compléter l'état d'une cellule par une information de position et de considérer $\mathcal{T} = \mathcal{S} \times \mathbb{R}^3$ comme une brique de base[3] permettant de construire

[3]Pour simplifier, on ne prend en compte que la position dans $\mathbb{R}^3$ de chaque cellule, mais il faudrait en plus spécifier sa forme qui conditionne son voisinage et les échanges avec les autres cellules (penser à l'aire de la surface

l'ensemble :

$$\begin{aligned}\mathcal{T}^+ &= \mathcal{T} \cup \mathcal{T}^2 \cup \cdots \cup \mathcal{T}^n \cup \ldots \\ &= \mathcal{T} \cup \mathcal{T} \times \mathcal{T}^+\end{aligned}$$

Il est sans doute possible de caractériser un embryon comme un point dans cet espace des phases, mais cela n'apporte pas grand chose : $\mathcal{T}^+$ a en effet très peu de structure intrinsèque et ne nous apprend que peu de choses sur les trajectoires possibles du système. Par exemple, la fonction d'évolution sera très difficile à définir et elle a peu de chances d'être continue.

Le problème de la localité. Pour comprendre pourquoi la fonction d'évolution sera difficile à définir, il faut voir que la spécification de la position de chaque cellule par ses coordonnées dans $\mathbb{R}^3$ présuppose la définition d'un repère global. Lors de l'évolution de l'embryon, la croissance d'une cellule va repousser les cellules voisines et de proche en proche, affecter la position de chaque cellule. Entre deux états successifs, il faut donc exprimer le changement de position de chaque cellule par une *transformation globale* des coordonnées. Parce qu'elle doit exprimer globalement les évolutions de chaque position, et que ces évolutions sont dues à de multiples transformations locales concurrentes, l'expression de cette transformation peut être arbitrairement complexe.

L'origine de ce problème est dans l'expression *extrinsèque et globale* de la forme du système[4] et une solution est donc de spécifier de manière intrinsèque la position de chaque cellule, par exemple en complétant l'état $s \in \mathcal{S}$ de chaque cellule par sa distance aux cellules voisines. Dans ce cas, la spécification des changements de position d'une cellule est locale mais, le voisinage de chaque cellule se modifiant, on retrouve le problème d'un espace d'état qui évolue au cours du temps.

Le problème de la continuité Si on reprend l'exemple de la pierre qui tombe, la position de la pierre varie continûment et il en va de même de sa vitesse. L'état du système varie donc continûment au cours du temps et la trajectoire du système est une *fonction continue* du temps dans l'espace des états. Cette continuité permet de raisonner sur les évolutions infinitésimales du système et de poser une équation différentielle qui caractérise la trajectoire. Dans les cas plus compliqués, on obtient une équation aux dérivées partielles (quand l'état possède une structure spatiale) ou un ensemble de telles équations quand il faut prendre en compte plusieurs modes de fonctionnement (en nombre fini et généralement petit).

Dans le cas du développement de l'embryon, ce n'est plus possible : tant qu'il n'y a pas de mouvement[5], de division ou de mort cellulaire, l'état s appartient à un certain $\mathcal{T}^n$ et cette évolution est continue (en supposant que les potentiels électriques, les concentrations chimiques, les contraintes mécaniques, ..., évoluent continûment). Mais les événements morphogénétiques essentiels (par exemple une division cellulaire qui fait passer de $\mathcal{T}^n$ à $\mathcal{T}^{n+1}$) sont discontinus par nature[6].

---

de contact entre deux cellules voisines qui conditionne les flux inter-membranaires).

[4] Dans l'approche évoquée, la spécification de la position des cellules se fait dans un repère global indépendant de l'embryon en croissance. Ce repère correspond au repérage des points de l'espace dans lequel la forme est plongée, et non à un processus intrinsèque à la forme en croissance : les lois gouvernant le mouvement, la division et la mort cellulaire seraient les mêmes si l'embryon se développait dans un volume torique (mais le résultat pourrait être différent car les voisinages des cellules seraient alors différents).

[5] Les mouvements cellulaires suffisent à changer la topologie et donc l'interaction des cellules.

[6] Dans l'exemple qui nous sert de fil conducteur, les événements morphogénétiques sont discontinus parce que la modélisation est faite au niveau cellulaire. On aurait pu modéliser la concentration des différentes molécules en chaque point de l'espace, ce qui éviterait peut-être ce problème de discontinuité (le mouvement de chaque molécule étant *a priori* continu). Mais un autre problème surgit alors : comment retrouver dans ces concentrations les entités biologiques qui nous intéressent : cellules, tissus, organes, etc ?

Vers d'autres solutions. La modélisation et la simulation du développement sont donc particulièrement ardues car il est difficile de définir à la fois la structure et la dynamique du système, l'une dépendant de l'autre. L'exemple précédent souligne l'inadéquation des formalismes globaux et des formalismes continus (on veut exprimer une évolution comme une succession d'événements morphogénétiques discrets qui correspondent à des ruptures et des changements qualitatifs). Cependant, la description de ces systèmes reste possible et les lois d'évolution sont alors souvent informellement décrites comme un ensemble de transformations *locales* agissant sur un ensemble organisé d'entités discrètes.

Face à ces difficultés, plusieurs chercheurs ont proposé d'utiliser les *systèmes de réécriture* pour formaliser ce type de description [GGMP02a, GMM04]. Un bel exemple de cette approche est donné par le formalisme des *systèmes de Lindenmayer* [PLH+90, LJ92, Pru99]. Cependant, ce formalisme est limité au développement de structures filamentaires et arborescentes. MGS est une solution possible pour la spécification du développement de structure de plus grande dimension. Le reste de ce chapitre développe un exemple paradigmatique.

2 Modélisation de l'évolution topologique d'un feuillet épithélial

Dans cette section, nous utilisons les concepts de collection topologique et de transformation pour spécifier un modèle de changement de topologie d'un feuillet épithélial. Cette modélisation représente un premier pas vers la modélisation déclarative du phénomène de neurulation.

2.1 Description du modèle

Avec la formation du *tube neural*, la *neurulation* marque, chez les vertébrés, le début de l'organogénèse. Le processus de neurulation consiste en une modification topologique de la

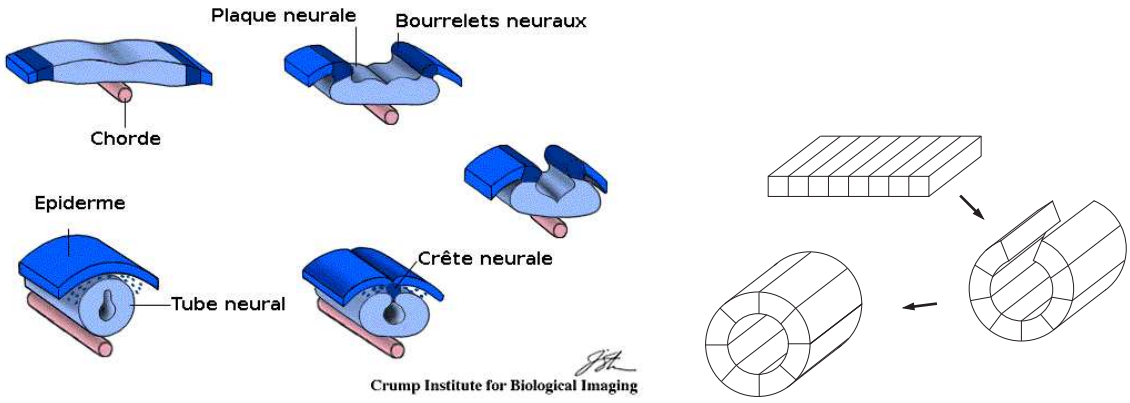


FIG. 1 – La figure de gauche donne une description diagrammatique du processus de neurulation (traduction d'un dessin de Patricia Phelps). La figure à droite détaille les trois étapes de notre modèle. Initialement, le système se compose d'un feuillet de cellules représentant la plaque neurale. Puis, il est incurvé sous l'action de la déformation des cellules. Enfin, lors de la dernière étape, le feuillet est *clos* pour devenir un véritable cylindre topologique continu.

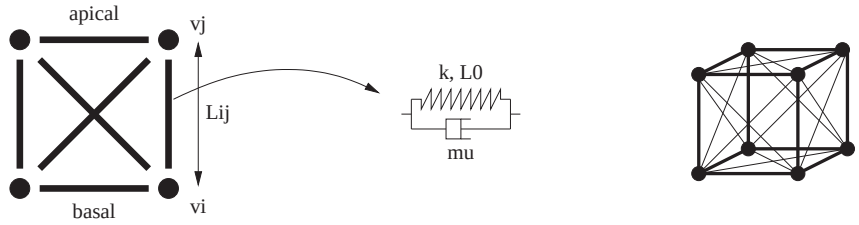


FIG. 2 – Sur la figure de gauche, le modèle d'Odell pour la simulation des cellules épithéliales est représenté. Sur la droite, nous présentons l'extension du modèle d'Odell en trois dimensions.

région antérieure de l'embryon (voir les images de gauche de la figure[7] 1) : la plaque neurale se creuse pour former les *bourrelets neuraux*. Puis, dans une seconde étape, les bourrelets neuraux se soulèvent et se rapprochent pour finalement se souder. Enfin, lors de la dernière étape, l'achèvement de la soudure des bourrelets neuraux conduit à la formation du tube neural. La déformation globale de la plaque neurale provient de changements locaux de la forme, de la division et de la migration des cellules qui composent la plaque neurale. Le changement de la géométrie est suivi d'une modification topologique qui correspond à la transformation de la plaque en tube. Cette modification est particulièrement difficile à modéliser.

Pour notre exemple, nous avons simplifié le modèle en considérant que le feuillet de cellules épithéliales ne se repliait que sous la seule action de la déformation locale des cellules (voir la figure de droite de la figure 1). Après la migration, les cellules qui étaient sur des cotés opposés au début du processus, se retrouvent très proches les unes des autres. Une fois qu'elles sont suffisamment proches, elles *fusionnent* pour transformer le feuillet initial en un cylindre.

Notre modèle mécanique est fondé sur le modèle présenté dans [OOAB81] pour la modélisation des cellules épithéliales. Dans ce modèle, une cellule épithéliale est décrite en deux dimensions par un système masses/ressorts. La figure 2 décrit cette représentation. Une cellule est un carré composé de 4 sommets et 6 arcs de *fibres* (4 pour les bords et 2 autres pour les diagonales). Les bords du carré correspondent à la membrane de la cellule et les diagonales sont utilisées pour modéliser les fibres internes, et prévenir une implosion de la cellule. De plus, la cellule est polarisée : les fibres au dessus sont *apicales* et celles en dessous sont *basales*. Par la contraction des ressorts de la partie basale et/ou apicale, la cellule change sa forme.

Chaque arc correspond à un ressort d'une constante de raideur k et d'une longueur au repos égale à L_0 en parallèle à une friction d'un facteur d'amortissement μ. Soit L la longueur du ressort, les forces mécaniques sont données par la seconde loi du mouvement de Newton :

$$\frac{\mathrm{d}^2 L}{\mathrm{d}t} = k(L_0 - L) - \mu \frac{\mathrm{d}L}{\mathrm{d}t} \tag{1}$$

Chaque sommet est relié à plusieurs ressorts et le vecteur de son accélération peut être calculé en sommant les forces appliquées par les ressorts. Soit p_i le vecteur de position du sommet i, on a :

$$m\frac{\mathrm{d}^2 p_i}{\mathrm{d}t} = \sum_{j \in \text{voisins}(i)} \frac{\mathrm{d}^2 L_{ij}}{\mathrm{d}t} \cdot \frac{p_j - p_i}{||p_j - p_i||} \tag{2}$$

où L_{ij} est la longueur du ressort entre i et j, donné par l'équation 1. La figure 3 présente l'application de ce modèle sur un anneau de cellules en MGS.

[7]Dessin pris, avec autorisation de l'auteur, P. Phelps, de `http://www.physci.ucla.edu/research/phelps/index.php` et `http://biology.kenyon.edu/courses/biol114/Chap14/Chapter_14.html`

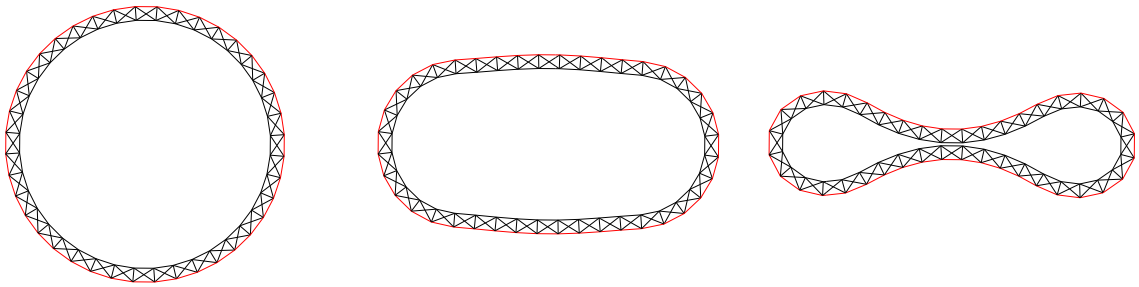


FIG. 3 – Application du modèle d'Odell sur un anneau de cellules en MGS. Les figures de gauche à droite représentent des copies d'écran des différentes étapes de la simulation.

Ce modèle à été étendu à la troisième dimension par R. Nagpal [Nag01]. Nous utilisons cette extension dans notre modèle. Plutôt que de représenter une cellule par un carré, nous utilisons des cubes, comme nous le voyons sur la figure de droite de la figure 2 : une cellule est composée de 8 sommets, 24 arcs, 6 faces (les faces du cube) et 1 volume. Remarquons que les fibres internes ne sont pas représentées, et plutôt que de correspondre aux fibres internes du modèle en 2 dimensions, les diagonales représentent les fibres dans la membrane en 3 dimensions.

2.2 Implantation en MGS

2.2.1 Représentation du système

On commence par définir le type des valeurs associées à chaque cellule.

```
record Vertex = { px:float, py:float, pz:float,
                  vx:float, vy:float, vz:float,
                  ax:float, ay:float, az:float,
                  m:float } ;;

record Edge = { k:float, L0:float, mu:float, vL:float,
                vi:cell, vj:cell } ;;
```

On définit deux types d'enregistrements MGS pour les cellules de dimension 0 et 1. Le type `Vertex` est composé de 10 champs pour décrire la mécanique newtonienne : `px`, `py` et `pz` représentent le vecteur position, `vx`, `vy` et `vz` le vecteur vitesse, `ax`, `ay` et `az` le vecteur accélération, et `m` la masse. Le type `Edge` encapsule les différentes constantes associées au modèle des ressorts `k`, `mu` et `L0`. La vitesse de l'élongation est enregistrée dans les champs `vL`. De plus, deux champs, `vi` et `vj` réfèrent aux bords des arcs ; ils sont utilisés pour générer une orientation arbitraire des arcs.

Les faces et les volumes ne sont pas utilisés dans cet exemple. Il est cependant facile d'imaginer que les volumes renferment un certain type de « script génétique » et que les faces sont les lieux où se passent les échanges de substances chimiques entre les cellules (les canaux ioniques par exemple). Ainsi, ces cellules permettraient de mettre en place des mécanismes additionnels comme des processus de réaction-diffusion en plus de la régulation de réseaux génétiques.

Dans un deuxième temps, nous devons initialiser la chaîne représentant le filament de cellules. Il est possible de charger des complexes cellulaires en MGS à partir de fichiers externes. On considère par conséquent que la structure du filament a été construit à l'aide d'un logiciel de CAO puis importé en MGS. Soit `c` une chaîne où les positions { `px=..., py=..., pz=...` }

sont associées à chaque sommet, et où les autres cellules n'ont aucune valeur. L'initialisation est faite par 4 transformations pour chaque dimension. À titre d'exemple, nous décrivons ici l'initialisation des sommets :

```
trans <0> init_0 =
  { v => v + { m=1.0, vx=0.0, vy=0.0, vz=0.0, ax=0.0, ay=0.0, az=0.0 } };;
```

Pour chaque élément filtré, la variable `v` contient la position du sommet. L'initialisation consiste en l'ajout à l'enregistrement `v` des informations de vitesse et d'accélération (nulles au début).

2.2.2 Le modèle mécanique

Le modèle mécanique est décrit par deux équations. Nous écrivons donc deux transformations pour calculer la force appliquée sur chaque arc et par conséquent le déplacement de chaque sommet.

```
trans <1> update_spring = {
  e:Edge =>
    let vi = self.(e.vi) and vj = self.(e.vj) in
    let L  = dist(vi,vj) in
    let f  = (e.k*(e.L0-L) + e.mu*e.vL) in
    let eij_x = (vj.px - vi.px) / L and eij_y = ... and eij_z = ... in
      e + { vL = vL, fx = f*eij_x, fy = f*eij_y, fz = f*eij_z }
};;
```

Cette transformation (dédiée aux cellules de type `Edge`) est une traduction directe de l'équation 1. L'amplitude de la force élastique est associée à la variable `f`. L'orientation des arcs est prise en compte : le vecteur $\vec{e}_{ij} = \frac{p_j - p_i}{||p_j - p_i||}$ est calculé et la force est distribuée selon les 3 directions. La variable `self` dénotant la collection argument sur laquelle `update_spring` est appliquée, l'expression `self.(x)` retourne la valeur associée à la cellule `x` de la collection `self`. La fonction `dist` calcule la distance entre ses deux arguments de type `Vertex`.

```
trans <0> integration [delta_t = 0.01] = {
  vi:Vertex =>
    let forces = cofacesfold (
      (fun e acc ->
         let ve = self.(e) in
         if (vi == ve.vi) then
           { fx = acc.fx+ve.fx, fy = ..., fz = ...}
         else
           { fx = acc.fx-ve.fx, fy = ..., fz = ...}
         fi),
      { fx=0.0, fy=0.0, fz=0.0 },
      vi
    ) in
      vi + { ax = forces.fx / vi.m, ay = ..., az = ...,
             vx = vi.vx + delta_t * vi.ax, vy = ..., vz = ...,
             px = vi.px + delta_t * vi.vx, py = ..., pz = ...}
} ;;
```

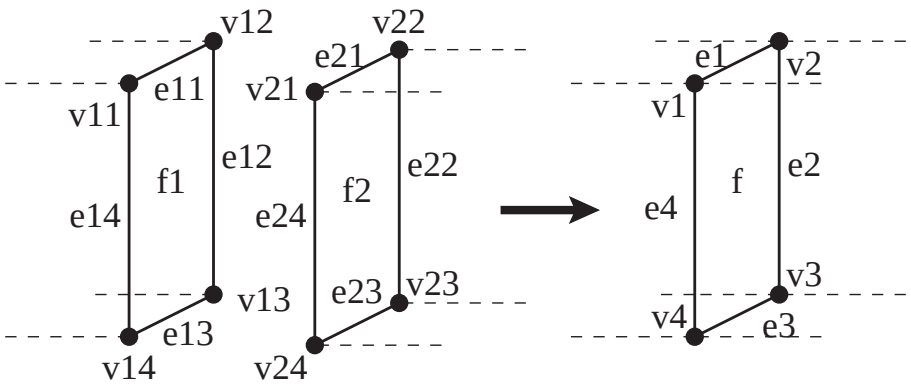


FIG. 4 – Schéma pour une règle de réécriture pour la transformation topologique d'un feuillet de cellules vers un cylindre. Pour simplifier la figure, les diagonales ne sont pas dessinées. Les arcs en pointillés représentent le reste du complexe cellulaire.

La transformation `integration` calcule les nouvelles positions des sommets en utilisant l'équation 2 (pour obtenir la mise à jour des accélérations) et une approximation par intégration d'Euler classique. La variable `delta_t` correspond à la discrétisation du temps, et est donnée comme argument optionnel de la transformation, dont la valeur par défaut est `0.01`. Dans la partie droite de la règle, la fonction `cofacesfold` correspond au *fold* standard sur la séquence des cofaces de `vi`. Ces cofaces sont les arcs dont le `vi` est un des bord. La fonction passée en argument fait la somme des forces appliquées sur chaque arc `vi`. Notons le test conditionnel utilisé pour prendre en compte l'orientation de l'arc, en conséquence de quoi la force est ajoutée ou soustraite.

2.2.3 Chirurgie topologique

La transformation que nous venons de voir permet d'incurver le feuillet de cellules. Cependant, aucune intersection entre les cellules n'est vérifiée. Le patch suivant fusionne deux faces suffisamment proches l'une de l'autre. Cette transformation est *ad-hoc* et la longueur initiale au repos a été choisie de façon à obtenir une courbure réaliste.

La figure 4 détaille une vue diagrammatique des règles de réécriture que l'on souhaite spécifier en MGS. Les cellules sont regroupées deux à deux et fusionnées. De plus, le reste du complexe cellulaire (représenté par les arcs en pointillés) doit être pris en compte pour la reconstruction. À titre d'exemple, les cofaces non filtrées des sommets v_{11} et v_{21} doivent êtres des cofaces de v_1. Le patch est :

```
patch surgery = {
  f1:[dim=2, (e11,e12,e13,e14) in faces]
  v11 < e11 > v12 < e12 > v13 < e13 > v14 < e14 > v11

  f2:[dim=2, (e21,e22,e23,e24) in faces]
  v21 < e21 > v22 < e22 > v23 < e23 > v24 < e24 > v21

  [P(v11,v12,v3,v14,v21,v22,v23,v24)]
  =>
    `v1:[dim=0,faces=(),
```

```
            cofaces=(‘e1,‘e4)@unmatched_cofaces(v11,v12),
            average_0(v11,v21)]
        ...
        ‘e1:[dim=1,faces=(‘v1,‘v2),
            cofaces=(‘f)@unmatched_cofaces(f1,f2),
            average_1(e11,e21)]
        ...
        ‘f:[dim=2,faces=(‘e1,‘e2,‘e3,‘e4),
            cofaces=cofaces(f1,f2),
            average_2(f1,f2)]
    } ;;
```

Pour simplifier le programme, les diagonales ne sont pas spécifiées sur le patch `surgery`. Un prédicat `P` est utilisé pour vérifier certaines propriétés géométriques additionnelles des sommets (les deux carrés doivent être très proches).

Sans entrer dans les détails du patch, remarquons par exemple que l'on crée une nouvelle cellule `‘v1` de dimension 0 qui a deux nouveaux éléments (`‘e1` et `‘e4`), et les anciennes cofaces de `v11` et `v12` qui n'ont pas été filtrées (elles sont données par la fonction `unmatched_cofaces`) comme cofaces. La valeur associée avec `‘v1` est calculée par la fonction `average_0(v11,v21)`. La figure 5 montre quatre étapes de l'animation générée par MGS sur un feuillet de 20x2 cellules.

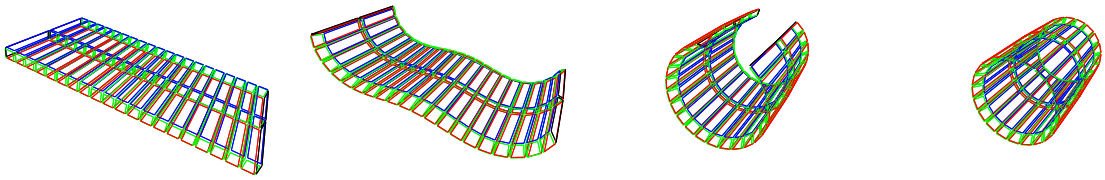


FIG. 5 – Simulation du processus neurulation en MGS : de la gauche à la droite, un feuillet de cellules épithéliales s'incurve jusqu'à ce que les bords soient joints pour former un tube.

2.2.4 Déplacement d'un jeton sur un anneau de cellules

Pour montrer que le cylindre est réellement clos après l'étape de chirurgie topologique, nous avons placé un jeton dans les volumes. Nous souhaitons déplacer le jeton d'un bord à l'autre du feuillet. Après la formation du cylindre, le jeton doit être capable de se déplacer sur l'anneau sans être bloqué au niveau de l'ancienne frontière[8]. La transformation suivante peut aussi être vue comme un transport de substance entre cellules biologiques.

Nous associons des valeurs aux volumes (3-cellules) afin de simuler le mouvement du jeton. Au début, chaque cube porte une valeur `‘cell`. Tous les volumes possèdent cette décoration, excepté le volume qui contiendra la valeur `‘token` qui dénote le jeton ainsi qu'une de ses cellules voisines qui contiendra le jeton `‘back`. Ce dernier symbole est utilisé pour ne permettre le déplacement du jeton que dans une seule direction.

```
trans <3,2> token = {
  ‘back, ‘token, ‘cell => ‘cell, ‘back, ‘token ;
```

[8]Le film de cette simulation est disponible à l'url suivante `http://www.ibisc.univ-evry.fr/~mgs/ImageGallery/EXEMPLES/Neurulation/neurulation.avi`

```
    ‘back, ‘token        => ‘token, ‘back
  } ;;
```

Cette transformation est appliquée sur les éléments de dimension 3 et l'on considère une relation de 2-voisinage entre elles. Ainsi, le motif `‘back, ‘token, ‘cell` signifie que l'on cherche un chemin composé de trois cellules dont les valeurs correspondent à `‘back`, `‘token` et `‘cell` dans cet ordre. On réécrit `‘cell, ‘back, ‘token` qui permet au jeton de se déplacer en avant. La seconde règle correspond au cas où le jeton est à une extrémité. Ne pouvant aller plus loin, il se déplace dans la direction opposée.

Quatrième partie

Conclusions et annexes

Chapitre 12

Conclusions et perspectives

1 Conclusions

Nous avons conçu et développé les notions de collection topologique de dimension arbitraire et de transformation, dans le contexte du développement d'un langage de programmation dédié à la modélisation et à la simulation de systèmes dynamiques à structure dynamique. Ces constructions nouvelles ont été formalisées, mise en relation avec les notions de chaîne, de cochaîne et de forme différentielle, implantées dans un interprète expérimental et validées à travers de nombreuses applications.

1.1 Formalisation

Collections topologiques. Nous avons commencé par poser les besoins exprimés par la représentation de l'état d'un système à structure dynamique. Inspiré par des notions de topologie algébrique déjà utilisées dans des domaines tels que la modélisation géométrique et l'informatique graphique, nous avons élaboré la notion de *collection topologique*. Une collection topologique est une structure de données adaptée à la représentation de champs sur des organisations spatiales discrètes et arbitrairement complexes, suffisamment souple pour autoriser les modifications locales de ces organisations et offrant la notion de dimension.

Nous avons utilisé les concepts de complexe cellulaire abstrait et de complexe de chaînes pour respectivement représenter des espaces complexes comme l'agencement de briques élémentaires,

et associer des valeurs à la structure obtenue. Le cadre mathématique peut sembler sophistiqué pour l'usage que nous en faisons (nous nous appuyons principalement sur des définitions et nous utilisons peu de résultats profonds), mais nous utilisons les possibilités qu'il offre pour établir un cadre général permettant d'unifier plusieurs modèles de calcul (du point de vue de leur programmation) et pour établir une « algèbre » des collections topologiques fondée sur quatre opérations : la création de position, la décoration, la concaténation de deux collections et le changement de domaine.

Si l'utilisation des chaînes et des cochaînes avait déjà été suggérée dans le projet MGS, la formulation précise, en prenant en compte à la fois la notion d'orientation et de valeur décorant une cellule, ainsi que la définition des quatre opérations, est un travail original. De même, la mise en relation de la notion de forme différentielle et de transformation est nouvelle.

Transformations. Nous avons ensuite décrit, dans une version simplifiée, le langage que nous avons développé : un langage fonctionnel étendu avec les collections topologiques et les *transformations*, une forme de réécriture locale des collections topologiques.

Nous avons défini la sémantique de ce langage dans le style de la *sémantique naturelle* afin de rendre compte des trois mécanismes fondamentaux mis en jeu dans une transformation :

- *le filtrage d'une sous-collection :* avec deux langages de motifs (motifs de chemin et motifs de patch) ;
- *les stratégies d'applications des règles :* synchrone/asynchrone avec ou sans priorité d'une part, et stochastique d'autre part (en particulier les stratégies StS et SED) ;
- *la reconstruction* : fondée sur l'algèbre d'opérations sur les collections topologiques.

Notre sémantique est la première sémantique qui rende compte de ces trois étapes simultanément. Notre objectif était de « résumer », de manière concise et formelle, les nombreux développements logiciels. Un des enjeux était de rendre compte des stratégies d'application qui sont difficiles à décrire de manière informelle, en particulier les stratégies stochastiques. Ce but a été atteint, même si ce n'est le cas que pour un langage limité à des valuations finies.

Analogie avec les formes différentielles. Outre la possibilité d'exprimer facilement l'étape de reconstruction des transformations, la définition des collections topologiques par des concepts issus de topologie algébrique rapproche la notion de transformation de celles de forme différentielle et d'intégration à partir de la notion de cochaîne.

Ce rapprochement est fertile : il suggère de paramétrer les transformations par l'opération de reconstruction, il permet de simplifier l'expression des modèles dans le style de la physique discrète et du calcul différentiel discret, et il ouvre la voie à une notion d'homomorphisme avec l'étude d'une algèbre de ces opérateurs, étude qui pourrait se faire dans la lignée des algèbres que R.S. Bird et B.M. Merteens ont développées pour les listes.

1.2 Applications et exemples

Une part importante de notre travail a consisté à éprouver et valider les concepts que nous avons développés par des applications typiques. Nous avons souhaité mettre en avant les possibilités de MGS pour répondre à différents types de problématiques gravitant autour de la simulation des $(\text{SD})^2$:

- *Modifications topologiques :* comment calculer déclarativement un espace complexe de dimension arbitraire à partir de règles de construction locales ? Cette question a été clairement posée dans [SPS04] et nous y apportons une réponse avec la notion de patch.

Les patches permettent la spécification locale et déclarative d'opérations transformant la structure d'une collection topologique.

- *Calcul numérique :* comment exprimer simplement la résolution numérique d'équations différentielles sur des organisations spatiales arbitraires ? Les équations différentielles décrivent de façon locale (en temps et en espace) des conservations ou des transferts de quantités à l'aide d'opérateurs tels que le calcul de la différentielle d'une forme. Les $\langle n, p \rangle$-transformations fournissent les moyens de programmer localement les équivalents discrets de ces opérateurs. A l'aide de schémas de résolution classiques (intégration d'Euler, de Runge Kutta, méthodes des éléments finis, etc.) ou en utilisant des méthodes développées dans le cadre de la physique discrète (la *cell method* de E. Tonti), les $\langle n, p \rangle$-transformations peuvent être utilisées pour résoudre numériquement ces systèmes d'équations. L'expression est implicite dans le sens où les coordonnées des objets spatiaux n'apparaissent pas, pas plus que les schéma d'indexation qui obscurcissent classiquement les programmes numériques en FORTRAN.
- *Modèles et simulations stochastiques :* comment spécifier de manière déclarative l'évolution d'un phénomène connaissant les lois de probabilités qui le caractérisent ? Les stratégies d'application des règles de transformations MGS autorisent une forme de modélisation du temps. La stratégie fondée sur l'algorithme de D.T. Gillespie correspond par exemple à des modèles à évènements discrets probabilisés. Le cadre offert par MGS autorise l'utilisation d'une telle méthode dans un cadre différent de celui pour lequel elle a été créée : nous avons développé dans ce document son utilisation dans le cadre des imbrications de multi-ensembles.

Ces trois problématiques se rencontrent simultanément dans la modélisation de la morphogénèse biologique, un exemple de système dont la structure croît au cours du temps et faisant intervenir des modèles mécanico-chimiques. Le modèle très simplifié du processus de neurulation que nous avons simulé, présente les deux premières problématiques. Une extension afin de prendre en compte un modèle de réaction-diffusion de morphogènes est en cours de développement.

1.3 Implantation

Enfin, l'ensemble de ce travail n'aurait pas pu être réalisé sans un important effort d'implantation. Celui-ci se situe à deux niveaux :

1. Implantation **de** MGS : nous avons participé au développement d'un interprète complet du langage MGS correspondant à la formalisation donnée dans ce document, et qui a permis de valider l'approche par collection topologique et transformation.
2. Implantation **en** MGS : nous avons développé et simulé un grand nombre de modèles exprimés dans le langage MGS. Outre l'illustration des concepts, nous avons profité de l'expérience gagnée pour déterminer les limites de notre formalisme et pour apporter les nouveautés et les améliorations nécessaires.

2 Prolongement du travail en cours

Le langage MGS est en constante évolution, répondant à la fois aux besoins émergeants de son utilisation et aux différents développements théoriques.

2.1 L'interprète MGS

Si au niveau théorique les différents types de données en MGS sont décrits de manière unifiée par la structure de collection topologique, au niveau de l'implantation, chaque type de collection topologique dispose de sa propre implantation *ad-hoc*. Cela est nécessaire pour des raisons d'efficacité. Par exemple, il n'est pas raisonnable de représenter des multi-ensembles de plus de 10^5 éléments, ce que nous avons fait dans le cas de simulations chimiques, par un graphe complet.

Cette diversité de représentation n'empêche pas de développer des codes génériques (par exemple pour le filtrage). Nous sommes persuadés que les techniques permettant d'implanter des algorithmes génériques peuvent être développées afin de factoriser plus de code dans l'interprète.

Par ailleurs, l'amélioration des performances à l'exécution passe très certainement par la compilation et des phases d'optimisation. Le problème de la compilation a été abordé dans [Coh04] sous l'angle du typage. Nous avons abordé le problème de l'optimisation des filtres dans notre stage de DEA en proposant de réécrire certains filtres en des filtres équivalents mais dont les occurrences sont plus efficaces à rechercher. Nous avons aussi implanté des stratégies paramétrables de parcours des cellules d'un complexe, afin d'optimiser la recherche des occurrences d'un patch. La stratégie par défaut peut être modifié explicitement par le programmeur mais les paramètres devraient idéalement être déterminées automatiquement par une analyse statique des motifs et par la connaissance du type de collections sur lequel on travaille.

2.2 Les transformations

La distinction entre les patches et les $\langle n, p\rangle$-transformations ne semble pas justifiée. La différence entre ces deux types de transformation ne réside que dans les deux langages de motifs. Quel que soit le langage utilisé, l'application d'une règle retourne un couple (collection filtrée, collection calculée). Cela conduit à considérer les transformations comme des ensembles ordonnés (ou non) de règles de réécriture, de façon indépendante des langages de motifs. La composition de ces ensembles de règles est une voie pour modulariser les simulations. Un des enjeux de la simulation est en effet de pouvoir définir des composants réutilisables, capitalisant un savoir-faire et permettant de construire aisément des simulations par assemblage. On retrouve là des problématiques développées dans le domaine des langages à objets et des aspects.

L'analogie entre transformation et forme différentielle nécessite d'être approfondie. Les sémantiques des chapitres 4 et 5 décrivent le fonctionnement de l'application des transformations mais ne fournissent pas d'objet mathématique correspondant à une transformation. La notion de cochaîne semble être un point de départ pour définir cet objet formel. Néanmoins, comme nous l'avons vu, les cochaînes expriment des transformations dont les règles sont de la forme `x => f(x)`, partitionnant les collections en des sous-collections constituées d'un seul élément. Des filtres plus complexes ne partitionnent pas les collections aussi simplement. On pense par exemple à un motif `x, y` constituant des sous-collections comprenant deux éléments. Nous envisageons d'étendre les cochaînes en utilisant par exemple le treillis des sous-collections d'une collection ; ce treillis définit un complexe simplicial sur lequel on peut ramener les motifs les plus complexes à des motifs élémentaires. Cette extension des cochaînes permettrait alors de donner une identité mathématique aux transformations et de définir une algèbre des transformations par analogie à l'algèbre des formes différentielles. Les identités remarquables de cette algèbre correspondraient alors à des transformations de programmes (par exemple pour les simplifier où les rendre plus efficaces). Un équivalent du théorème de Stokes permettrait par exemple de simplifier le domaine de la collection sur laquelle s'applique une transformation (au prix de la complexification de la fonction à appliquer).

Mentionnons une dernière direction de recherche concernant les transformations : les relations

avec la réécriture (et en particulier avec la réécriture de graphe) doivent être étudiées. Des concepts développés dans ce domaine comme la notion de forme normale ou de confluence ont des interprétations utiles en modélisation. Une forme normale peut par exemple correspondre à un attracteur de la trajectoire du système. La confluence peut s'interpréter comme une forme de stabilité ou de robustesse. L'étude des propriétés des systèmes de réécriture peuvent donc apporter beaucoup à MGS et à la modélisation des (SD)^2.

3 Perspectives : la programmation spatiale

Nous avons mentionné dans l'introduction de cette thèse que le projet MGS s'inscrivait aussi dans la problématique des langages de programmation non-conventionnels. Plus précisément, MGS fait partie des langages conçus pour le *calcul spatial* (*spatial computing* en anglais). Ce nouveau domaine de recherche vise à développer des formalismes, des langages et des architectures matérielles qui permettent de prendre en compte la notion d'espace[1]. Le calcul spatial répond à trois motivations :

- dépasser l'architecture séquentielle de von Neumann pour développer des modèles parallèles dans lequel l'espace est une ressource prise en compte pour optimiser les performances (coût de communication, organisation des circuits VLSI, accès aux données, reconfiguration de l'architecture, etc.) ;
- fournir l'environnement adapté à des applications qui doivent construire des espaces (réalité virtuelle, simulation et expérience numérique, géométrie computationnelle, etc.) ;
- fournir un environnement adapté à des applications qui doivent interagir avec l'espace réel (réseau de capteurs, *claytronics*, robots mobiles, etc.).

Le modèle classique séquentiel de von Neumann atteint des limites et il est nécessaire d'étudier des alternatives. Les progrès de la biologie moléculaire[2] et les avancées en nanosciences offrent de nouveaux supports du calcul : on peut disposer à présent d'un très grand nombre d'entités élémentaires (supérieur à 10^6) qui interagissent et coopèrent de manière aléatoire, irrégulière, défaillante et dynamique dans le temps. Les questions posées par ces nouveaux supports sont

- *Comment obtenir un comportement global cohérent ?*
- *Comment programmer une grande quantité d'entités pour parvenir à un résultat particulier ?*

Ces problèmes sont aussi abordés par le *calcul amorphe* [AAC$^+$00] (*amorphous computing* en anglais). La programmation d'un support de calcul amorphe demande de passer de la spécification d'un objectif global à la définition d'un programme local à chaque entité. Plusieurs langages ont déjà été développés pour des applications dédiées (GPL [Coo91], OSL [Nag01], AML [Bea05], ...). Toutes les entités possèdent initialement le même programme. Celui-ci consiste en l'envoi de signaux formant ainsi des gradients virtuels sur le support de calcul. Les entités perçoivent ces signaux et sont aptes à modifier en conséquence leur comportement.

La programmation de ce type de médium est un problème difficile qui nécessite le développement de langages spécialisés. Par ailleurs, la biologie du développement est une source d'inspiration pour concevoir de nouvelles approches dans ce domaine. Par exemple, l'utilisation de gradients n'est pas sans rappeler les patrons de développement observés par H. Meinhardt [Mei82].

[1] Voir à ce sujet [BFGM05] et le récent séminaire qui s'est tenu à Dagstuhl en septembre 2006 sur « Computing Media and Languages for Space-Oriented Computation ».

[2] Nous pensons en particulier aux outils développés par la *biologie synthétique*.

La détermination d'une position dans un réseau amorphe peut s'inspirer des mécanismes étudiés par L. Wolpert pour l'établissement d'une information de position. E. Cohen et P. Prusinkiewicz ont étudié comment on pouvait contrôler la géométrie (globale) d'un tissu en croissance à partir d'un petit ensemble de paramètres qui peuvent se relier à l'action (locale) des gènes [CRLM+04]. On pourrait multiplier les exemples.

On voit que dans ce domaine, les *rétroactions entre un niveau global et un niveau local* sont déterminantes et s'inscrivent donc complètement dans la problématique des (SD)^2. C'est pourquoi nous pensons que les structures de données et de contrôle développées dans MGS offrent des outils intéressants pour la modélisation et la simulation de ce type de systèmes. Le développement d'une algèbre de transformations, l'approfondissement des opérateurs différentiels, le raffinement des stratégies stochastiques sont des voies qui permettraient d'établir des ponts entre le comportement local des entités et les propriétés globales assignées au système. La perspective ouverte est de développer une « ingénierie des populations » permettant de donner une forme à un médium amorphe.

Annexe A

Détails d'implantation et logiciels compagnons

1 L'interprète MGS

Notre implantation du langage MGS consiste en un interprète qui a largement été présenté dans le chapitre 2. Nous proposons de détailler dans cette section quelques éléments d'implantation mettant en avant différents problèmes rencontrés. Nous illustrons nos propos par la présentation de l'introduction des collections `QMF` dans l'interprète. Nous commençons par décrire l'organisation des sources de l'interprète ; nous nous intéressons en particulier à la mise en place du polytypisme *ad-hoc* de MGS et d'un mécanisme permettant l'ajout de nouveaux types de valeurs sans modifier les fichiers existants. Nous voyons ensuite la mise en place du type de collections `QMF`.

1.1 Structuration du code et mécanisme d'indirection

L'interprète MGS est en constante évolution. Ces évolutions consistent la plupart du temps en l'ajout d'un nouveau type de valeurs et des fonctions qui lui sont associées.

L'interprète est programmé dans un environnement fonctionnel dont le squelette est une fonction d'évaluation qui prend en argument une expression du langage (sous la forme d'un arbre de syntaxe abstraite) pour calculer une valeur. Cette valeur est décrite à l'aide d'un type somme :

```
type value = VAL_int of int
           | VAL_float of float
           | ...
           | VAL_funct value -> value
;;
```

On remarque la présence du constructeur `VAL_funct` qui a pour argument une fonction OCaml pour la représentation des fonctions MGS.

Un point important du développement de l'interprète consiste en l'enrichissement de l'environnement par l'introduction de fonctions prédéfinies. Une fonction prédéfinie du langage correspond à un filtrage sur les constructeurs du type `value`. À titre d'exemple, l'addition s'écrit :

```
let __add = function
  | VAL_int i1, VAL_int i2     -> VAL_int (i1+i2)
  | VAL_int i1, VAL_float f2   -> VAL_float ((float_of_int i1)+.f2)
  | VAL_float f1, VAL_int i2   -> VAL_float (f1+.(float_of_int i2))
  | VAL_float f1, VAL_float f2 -> VAL_float (f1+.f2)
  ...
;;
```

On remarque que le filtrage permet ici de programmer la surcharge de l'opérateur d'addition (extrêmement surchargé dans l'interprète). Plus généralement, ce type d'implantation facilite la programmation du polymorphisme et du polytypisme *ad-hoc*. La valeur MGS correspondant à ce calcul est alors

```
VAL_funct(function v1 -> VAL_funct (v2 -> __add (v1,v2)))
```

Elle est associée à l'identificateur `add` de l'interprète dans l'environnement initial, donnant ainsi l'accès au programmeur à cette fonction prédéfinie.

L'ajout d'un nouveau type de donné, dénoté `xxx`, correspond simplement à la définition d'un nouveau constructeur pour le type `value`, `VAL_xxx`. Cependant la spécification de ce nouveau constructeur amène deux difficultés concernant les fonctions prédéfinies :

- toute fonction OCaml sur le type `value` doit être modifiée pour compléter le filtrage sur les constructeurs de `value`, devenu non exhaustif par l'introduction du constructeur `VAL_xxx` ;
- toutes les fonctions prédéfinies sur le type de données `xxx` doivent être soit programmées par filtrage sur les constructeurs de `value` (les cas d'erreur sont pris en compte), soit intégrées à une définition existante (ce cas correspond à la surcharge d'un opérateur).

MGS compte à l'heure actuelle 24 types de données différents dont 12 types scalaires et 12 types de collections topologiques, ainsi que 225 fonctions prédéfinies. L'implantation de l'interprète (analyseurs syntaxique et lexical, évaluation et fonctions prédéfinies) nécessite environ 50 000 lignes de code OCaml, C et C++, réparties en 75 fichiers.

Une organisation du code de l'interprète ainsi qu'un mécanisme de redirection ont été mis en place pour faciliter le développement incrémental de l'interprète.

Structuration du code. L'organisation des sources de l'interprète est présentée figure 1. Elle consiste en une structuration par couches. Les couches les plus foncées dépendent des couches les plus claires. Elles sont, de la plus foncée à la plus claire :

La machine virtuelle : il s'agit du cœur de l'interprète. On y trouve l'analyseur lexical, l'analyseur syntaxique, le moteur pour la combinatorisation SK[1] et l'évaluateur.

Les fonctions prédéfinies : il s'agit des modules associant un identificateur à une définition de fonction prédéfinie, comme nous l'avons vu pour la fonction d'addition.

[1] L'interprète repose sur la SK-traduction au vol des expressions ; la combinatorisation SK permet de transformer les fonctions définies par l'utilisateur en composition de fonctions prédéfinies du langage hôte (ici OCaml) et par suite d'utiliser une valeur fonctionnelle OCaml pour représenter une fonction MGS définie par le programmeur. Cette traduction est particularisée par la prise en compte de traits impératifs autorisés par MGS.

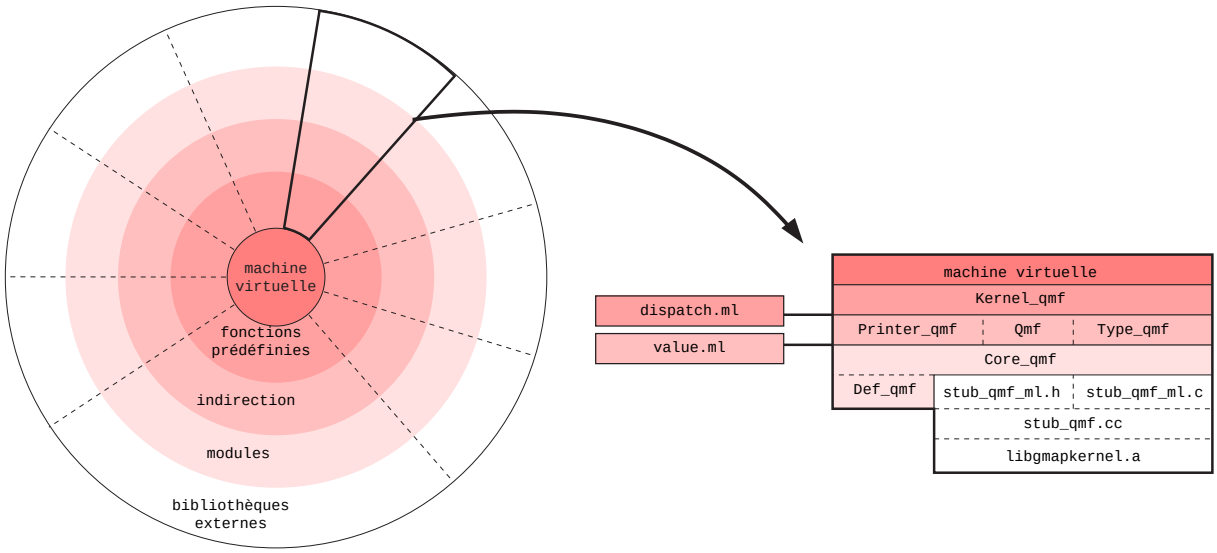


FIG. 1 – Organisation par couches des sources de l'interprète MGS : à gauche, la figure décrit les dépendance entre les couches sous la forme d'une tour de Hanoï vue de dessus : chaque couche est représentée par un étage de la tour ; les étages les moins larges (au dessus dans la tour) représentent les couches dépendantes de celles correspondant aux étages les plus larges. À droite les couches sont représentées de façon verticale pour le cas particulier des collections `QMF`.

Les indirections : elles font le lien entre les valeurs MGS, de type `value`, et les objets réellement manipulés. Intuitivement, en reprenant la fonction d'addition décrite ci-dessus, l'indirection correspond au lien qui est fait entre « l'addition de deux valeurs MGS » d'une part (`__add(v1,v2)`) et, par exemple, « l'addition de deux entiers » (`i1+i2`) dans le cas du constructeur `VAL_int`.

Les modules : il s'agit de l'implantation concrète des structures de données et des fonctions utilisées pour manipuler les valeurs MGS. On trouve également dans cette couche la description réelle des types de données utilisés pour encoder une valeur MGS.

Les bibliothèques externes : certains types de valeurs nécessitent l'utilisation de procédures spécifiées dans des bibliothèques spécialisées. Cette dernière couche contient à la fois ces bibliothèques et l'interface avec les modules de la section précédente.

Mécanisme d'indirection. Le point important de l'organisation des sources de MGS réside dans une découpe particulière des couches présentées ci-dessus. En effet, les secteurs en pointillés observés sur la figure 1 correspondent chacun à un type de données MGS particulier, et chaque secteur est indépendant des autres. L'exemple de secteur à droite sur la figure 1 correspond à l'introduction des collections `QMF` dans l'interprète. Nous détaillons les différents fichiers et modules apparaissant dans cet exemple :

- Les collections `QMF` sont fondées sur la notion de G-carte. La bibliothèque `libgmapkernel`, développée dans le cadre du projet MOKA, est dédiée à cette structure de données. Il s'agit d'une bibliothèque écrite en C++ que nous devons interfacer avec l'interprète MGS :

  `stub_qmf.cc` : OCaml permet d'interfacer des programmes C [LDG+04, chapitre 18]. Ce fichier correspond à un *wrapper* entre les classes et les objets C++ de la bibliothèque

`libgmapkernel` et les fonctions C correspondantes. Ces fonctions sont soit des procédures de la bibliothèque qu'on souhaite rendre accessibles dans le langage cible (MGS), soit des nouvelles fonctions implantées à partir des primitives fournies par `libgmapkernel`.

`stub_qmf_ml.[hc]` : Ces deux fichiers implantent l'interface entre les fonctions définies dans `stub_qmf.cc` et le langage OCaml. Cela consiste à encapsuler les structures de données C/C++ en valeurs OCaml utilisables dans les modules de l'interprète à travers des types abstraits. Ceci est détaillé plus loin dans l'annexe.

- Les fichiers de la couche *modules* spécifient en OCaml les types et fonctions utilisés pour l'implantation des collections topologiques `QMF`.

  `Def_qmf` : On trouve dans ce fichier la déclaration des types abstraits correspondant aux structures de données et aux valeurs OCaml décrites dans `stub_qmf_ml` pour la manipulation des G-cartes, ainsi que la déclaration du type polymorphe `'a qmf` émulant les collections topologiques `QMF` dont les positions sont décorées par des valeurs de type `'a`.

  `Code_qmf` : Ce module contient les fonctions OCaml de manipulation des valeurs de type `'a qmf`. En particulier, on y trouve les itérateurs `fold`, `map`, ... implantés **uniquement** pour les collections `QMF`. Chaque secteur implante ces fonctions pour chaque type de collection topologique. Ces fonctions ne sont pas polytypiques mais simplement polymorphes. Par exemple, la fonction `map` définie dans ce module a pour signature

  ```
  map : ('a -> 'b) -> 'a qmf -> 'b qmf
  ```

- Les couches les plus internes consistent à utiliser les types de valeurs décrits dans les fichiers de la couche *modules* pour spécifier les fonctions MGS s'appliquant sur des collections topologiques `QMF`.

  `value.ml` : On trouve ce fichier à la frontière des couches *modules* et *indirections*. Il contient la déclaration du type somme `value` que nous avons évoqué plus haut. Ce fichier est le **seul** commun à tous les secteurs et il est nécessaire de le modifier pour ajouter un nouveau type de données. En particulier, pour les `QMF`, il est mis à jour par :

  ```
  open Def_qmf ;;
  type value = ...| VAL_qmf of qmf ;;
  ```

  `Qmf` : Ce module permet l'instanciation des objets de la couche *modules* en éléments agissant sur des éléments de type `value`. Dans le cas de `map`, l'instanciation donne :

  ```
  _map_funct_qmf : (value -> value) -> value qmf -> value qmf
  ```

  `Printer_qmf` : Il regroupe les fonctions pour l'affichage des collections `QMF` dans l'interprète.

  `Type_qmf` : Il construit et met à jour la hiérarchie des types des valeurs MGS présentée au chapitre 2.

- Dans la couche *indirections*, les fonctions prédéfinies sont associées aux identificateurs auxquels elles réfèrent dans l'interprète. C'est ici que les indirections sont faites suivant les types arguments des fonctions MGS. Par exemple, pour la fonction `map`, il est nécessaire de définir la fonction suivante :

```
let __map v1 v2 = match (v1,v2) with
  | VAL_funct v1, VAL_seq v2 -> Seq._map_funct_seq v1 v2
  | VAL_funct v1, VAL_qmf v2 -> Qmf._map_funct_qmf v1 v2
  ...
;;
```

La fonction `__map` est polytypique car s'appliquant sur tous les types de collections topologiques et quel que soit le type des éléments contenus. Dans notre exemple, elle redirige le calcul suivant le type du second argument vers les fonctions `_map_xxx_yyy` adéquates.

`dispatch.ml` : Ce fichier contient l'ensemble des déclarations des fonctions nécessitant une redirection vers une fonction spécifique par filtrage sur les constructeurs du type `value`. On y trouve par exemple la définition de la fonction `__map` ci-dessus.

La spécification de fonctions telles que `__map` est au cœur des difficultés évoquées plus haut à propos de l'introduction d'un nouveau type de données. En ajoutant un constructeur au type somme `value`, tous les filtrages de `dispatch.ml` sont à mettre à jour. Afin d'éviter ce travail fastidieux, nous avons conçu un mécanisme générant automatiquement le fichier `dispatch.ml` à partir des fonctions d'indirection spécifiées dans les modules de la couche *indirections*. Ainsi, si une fonction *_fct_arg* (où *fct* est le nom de la fonction et *arg* est le type de son argument) est spécifiée dans l'un de ces fichiers, le mécanisme crée automatiquement dans `dispatch.ml` la fonction :

```
let __fct = function
  | VAL_arg v -> _fct_arg v
  | _         ->
    failwith "Dispatch: fct: error in argument type"
;;
```

si celle-ci n'existe pas ou la complète si elle existe déjà.

La construction automatique du fichier `dispatch.ml` permet une indépendance entre les secteurs de la figure 1.

`Kernel_qmf` : Ce dernier module, associe la fonction `__map` à l'identificateur MGS `map` si cela n'a pas été fait par ailleurs :

```
addenv "map" __map ;;
```

où `addenv` est la fonction de mise à jour de l'environnement de l'interprète MGS.

1.2 Collection topologique

Nous avons vu d'un point de vue pratique comment les sources sont organisées pour rendre accessibles dans l'interprète de nouvelles structures de données ainsi que leurs fonctions. Dans cette partie, nous allons nous intéresser plus au fond de l'implantation qu'à la forme, en décrivant le codage des collections `QMF`.

G-cartes et complexes cellulaires. Rappelons tout d'abord qu'un complexe cellulaire de dimension N est un triplet $\mathcal{K} = (S, \prec, dim)$ où $(S, \prec)$ est un ensemble de cellules topologiques ordonnées par la relation d'incidence $\prec$; la fonction dim associe une dimension à chaque cellule de S ; N est la plus grande dimension associée à une cellule de S. Une G-carte de dimension N est un couple $\mathcal{G} = (\mathcal{B}, (\alpha_0, \ldots, \alpha_N))$ où $\mathcal{B}$ est un ensemble de brins et les α_i sont des involutions de $\mathcal{B}$ dans $\mathcal{B}$; les involutions vérifient :

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

La G-carte $\mathcal{G}$ décrit le complexe cellulaire $\mathcal{K}$ s'il existe une bijection entre les brins de $\mathcal{G}$ et les N-tuples[2] de $\mathcal{K}$.

Les collections topologiques de type `QMF` sont restreintes à des espaces de positions représentés par des complexes cellulaires pouvant être décrits par des G-cartes (au même titre que les collections de type `seq` décrivent des espaces de positions correspondant à des graphes linéaires).

Nous proposons donc de programmer les collections `QMF` en utilisant la structure de données de G-carte pour encoder l'espace de positions. Il est important de conserver à l'esprit que la structure de données de G-carte n'est pas accessible depuis l'interprète où seul le graphe d'incidence et les cellules topologiques peuvent être manipulés. Tout l'enjeu de cette implantation est de rendre opaque l'utilisation des G-cartes.

Afin de réaliser la traduction d'une G-carte $\mathcal{G}$ en le complexe cellulaire $\mathcal{K}$ qu'elle représente, il est nécessaire de retrouver les cellules de $\mathcal{K}$ à partir de $\mathcal{G}$. Intuitivement, une cellule σ de $\mathcal{K}$ est représentée implicitement par l'ensemble des brins dont le N-tuple associé contient σ. En supposant $b \in \mathcal{B}$ un tel brin, l'ensemble des brins contenant σ est accessible au moyen d'une *orbite*. Une orbite est définie par une séquence d'involutions $<\alpha_{i_1}, \ldots, \alpha_{i_k}>$; soit b un brin de $\mathcal{B}$, on définit de façon inductive l'ensemble $<\alpha_{i_1}, \ldots, \alpha_{i_k}>(b)$ des brins accessibles à partir de l'orbite $<\alpha_{i_1}, \ldots, \alpha_{i_k}>$ par :

$$\left\{ \begin{array}{l} b \in <\alpha_{i_1}, \ldots, \alpha_{i_k}>(b) \\ b' \in <\alpha_{i_1}, \ldots, \alpha_{i_k}>(b) \end{array} \right. \quad \Rightarrow \quad \forall\, 1 \leq j \leq k, \quad \alpha_{i_j}(b') \in <\alpha_{i_1}, \ldots, \alpha_{i_k}>(b)$$

Ainsi, l'ensemble des brins représentant une cellule σ de dimension i est donné par l'orbite $<\alpha_0, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_N>$ et un des brins b dont le N-tuple associé contient σ. Cette orbite permet d'emprunter toutes les involutions exceptée α_i pour parcourir les N-tuples accessibles à partir de b. En interdisant α_i, tous les N-tuples considérés possèdent la même cellule de dimension i, à savoir σ.

Finalement, on déduit de ce parcours des brins qu'un couple $(b, i) \in \mathcal{B} \times \mathbb{N}$ permet d'identifier la cellule σ de dimension i telle que le N-tuple associé à b contient σ. Tous les brins de $<\alpha_0, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_N>(b)$ conviennent pour représenter σ. Afin de normaliser le couple (b, i), nous considérons que b est choisi tel que son adresse mémoire soit la plus petite.

Types OCaml. À partir de la bibliothèque `libgmapkernel.a`, nous mettons en place deux types OCaml abstraits (voir les *custom blocks* dans [LDG+04, page 254]) : `gmap` pour les G-cartes et `dart` pour les brins. Ces types sont mis en place côté C dans les fichiers `stub_qmf_ml.h` et `stub_qmf_ml.c`, et déclarés côté OCaml dans le module `Def_qmf` :

```
type gmap
and  dart ;;
```

Une valeur de type `gmap` est un pointeur vers un objet de la bibliothèque `libgmapkernel.a` encodant une G-carte ; une valeur de type `dart` est un pointeur vers un brin d'une G-carte. Les fonctions remontées en OCaml permettent entre autre de parcourir les brins d'une G-carte ou les brins d'une orbite, de récupérer la G-carte à laquelle appartient un brin, de redéfinir les involutions, etc.

À partir de ces deux types abstraits, nous définissons dans `Def_qmf` deux types supplémentaires respectivement pour les cellules des G-cartes (c'est-à-dire l'espace de positions) et pour les collections `QMF` :

[2] Un N-tuple de $\mathcal{K}$ est une séquence ordonnée de cellules $(\sigma_N, \ldots, \sigma_0)$ telle que $\forall 0 \leq i \leq N$, $dim(\sigma_i) = i$ et $\sigma_i \prec \sigma_{i+1}$ pour $i \neq N$.

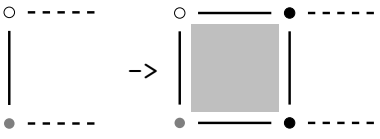
```
type ncell  = dart * int
and  'a qmf = gmap * (ncell, 'a) Hash.t ;;
```

Comme nous l'avons décrit plus haut, une cellule topologique est un couple (brin, dimension). En ayant assuré l'unicité de la représentation des cellules, on représente une collection `QMF` comme une table de hachage[3] associant une valeur de type `'a` aux cellules de type `ncell`. La donnée d'une G-carte dans la déclaration du type `qmf` permet d'assurer que seules les cellules de cette G-carte sont considérées. Ainsi, le type d'une collection `QMF` est paramétré par une G-carte (voir chapitre 2).

Construction déclarative de G-cartes. La construction des structures dans le langage est similaire à la paramétrisation d'objets géométriques par le λ-calcul proposée dans [DL02]. Il y est présentée 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.

À l'instar de ces travaux, nous nous plaçons dans un contexte où le point de vue « brins et involutions » est caché pour ne laisser apparaître que les cellules topologiques. Les primitives proposées dans l'interprète sont, dans ce contexte, de plus haut niveau que celles de [DL02]. Il est alors possible d'utiliser la puissance d'expression d'un langage fonctionnel pour construire de nouveaux objets (par notamment la composition de fonctions et la récursivité). Dans l'exemple suivant, on propose de construire une ligne de n carrés (vus comme des 2-cellules à 4 côtés). Le programme suivant schématise comment faire :

```
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 (encodé dans une G-carte) référencé par un arc (les cercles gris et vide dénotent le bord de cet arc, les lignes pointillées représentent la partie restante du complexe cellulaire). Elle construit un nouveau complexe identique au précédent (la partie du complexe figurant en pointillés n'est pas modifiée) auquel est ajouté un nouveau carré construit à partir de l'arc passé en paramètre. Elle retourne alors un des côtés de ce nouveau complexe à partir duquel un nouvel appel à la fonction construira un autre carré. La fonction `line` utilise ce principe en appelant de façon récursive `add_square` pour construire une ligne de n carrés à partir d'un arc distingué originel (pour $n = 0$).

Les fonctions prédéfinies pour construire les G-cartes sont deux types :

1. la création de nouveaux objets indépendants dans une G-carte (`add_vertex`, `add_edge`, `add_facet`, `add_polygon`, `add_sphere`, `add_polyline`, etc.) ;
2. des transformations simples d'objets :

   `insert_vertex :` divise en deux un arc donné en argument ;

[3] Le module `Hash` fournit par OCaml implante les tables de hachage.

`insert_edge` : divise en deux une face en créant un nouvel arc à partir de deux des sommets de la face ;

`insert_face` : divise un volume en deux à partir de la liste des arcs bordant la nouvelle face ;

`remove_ncell` : opération inverse de `insert_[vertex|edge|face]` ;

`create_edge` : crée un nouvel arc à partir de deux sommets si cela est possible ;

`create_face` : crée une nouvelle face à partir de la liste ordonnée des arcs qui la borde ;

`delete_ncell` : opération inverse de `create_[edge|face]`.

Ces fonctions ont été programmées lors d'une semaine de collaboration avec l'équipe du projet MOKA. La difficulté de l'implantation de telles primitives provient de l'opacité que nous imposons vis-à-vis des G-cartes.

2 Les logiciels compagnons

2.1 Le visualisateur Imoview

Le langage MGS permet la définition de structures de données évoluant dans le temps pour la simulation de processus complexes en biologie. L'exploitation des résultats de la simulation nécessite la visualisation sous plusieurs formes des résultats de l'exécution d'un programme MGS. Pour cela, il est possible d'associer une représentation graphique abstraite à chaque structure et sous-structure.

Ces représentations graphiques sont transmises sous forme abstraite au serveur graphique en utilisant une architecture de type client/serveur unidirectionnelle. La communication entre le programme MGS et le serveur graphique passe par une couche intermédiaire : MGS génère un fichier reflétant la représentation des structures et transformations de la simulation effectuée et respectant une syntaxe bien définie ; le serveur quant à lui interprète le fichier et en produit l'affichage à l'écran sous forme d'objets tridimensionnels permettant l'exploitation de ces résultats.

La communication entre le client et le serveur repose sur un langage intermédiaire, Teom. Ce langage offre des primitives de haut-niveau permettant de construire des scènes graphiques qui sersont affichées par le serveur graphique, Imoview. Une caractéristique de Teom est de permettre la description symbolique du placement des objets ainsi que de leur positionnement à l'aide de coordonnées explicites.

La connexion entre le client et le serveur est faite par l'intermédiaire d'un fichier[4]. L'ajout d'une couche intermédiaire, matérialisée par un fichier respectant un langage bien précis, facilite l'interaction entre les application et le logiciel d'affichage Imoview.

Nous commençons par présenter le langage Teom puis nous décrivons le logiciel Imoview.

Le langage Teom. La syntaxe du langage Teom a été définie de telle sorte que les fichiers soient facilement générés par un programme MGS. Une méthode classique pour représenter une scène 3D est d'utiliser une structure hiérarchique d'objets, c'est-à-dire un arbre, les nœuds et les feuilles contenant les informations sur les éléments qui constituent la scène :

[4]Une communication par *socket* est envisagé et fait l'objet d'un développement actuel.

- Les nœuds sont appelés des *groupes*. Ils connectent différents sous-arbres afin de former et de placer des objets complexes. Ils correspondent aux opérations géométriques ou aux groupements de sous-objets.
- Les feuilles correspondent aux objets primitifs, contenant leur propre description.

Détaillons les trois catégories d'objets que nous venons d'introduire :

Les objets primitifs : Ils représentent les formes géométriques fondamentales de la scène. Ils ne peuvent pas être directement en relation avec d'autres objets de l'arbre et possèdent leurs propres paramètres.

Le langage Teom dispose d'un large inventaire d'objets simples listés dans le tableau 1.

Nom	Objet géométrique
`Box`	Parallélépipède rectangle
`Cone`	Cône
`Cylinder`	Cylindre
`Disc`	Disque
`Empty`	Case vide
`Frustum`	Tronc de cône
`Polyline`	Ligne brisée
`Polygone`	Polygone convexe
`Sphere`	Sphère

Tab. 1 – Liste de primitives du langage Teom.

Les opérations géométriques : Les opérations géométriques représentent des nœuds dont les fils subissent une transformation géométrique particulière. Le tableau 2 présente les différentes opérations offertes par le langage Teom ainsi que les mots clés qui leur sont associés.

Nom	Opération
`AxisRotated`	Rotation
`Scaled`	Homothétie
`Translated`	Translation

Tab. 2 – Liste des opérations géométriques du langage Teom.

Les groupes : Les groupes permettent de considérer un ensemble d'objets (leurs fils dans l'arbre) comme une entité monolithique. Ils permettent également de positionner dans l'espace et de façon implicite ces objets les uns par rapport aux autres, suivant des patrons d'organisations simples (droites, grilles carrées et grilles hexagonales). Les groupes conviennent à la représentation de GBF par exemple. On remarquera notamment que l'objet vide (`empty`) trouve ici toute son utilité pour représenter les positions non définies des collections newtoniennes.

Le logiciel Imoview. Ce logiciel permet l'affichage de scènes décrites dans le langage Teom évoqué ci-dessus. L'implantation est fondée sur l'utilisation de la bibliothèque graphique OpenGL ainsi que des suites LablGL, LablGTK et CamlImages développées autour d'OCaml. La figure 2 est une capture d'écran de la version actuelle d'Imoview.

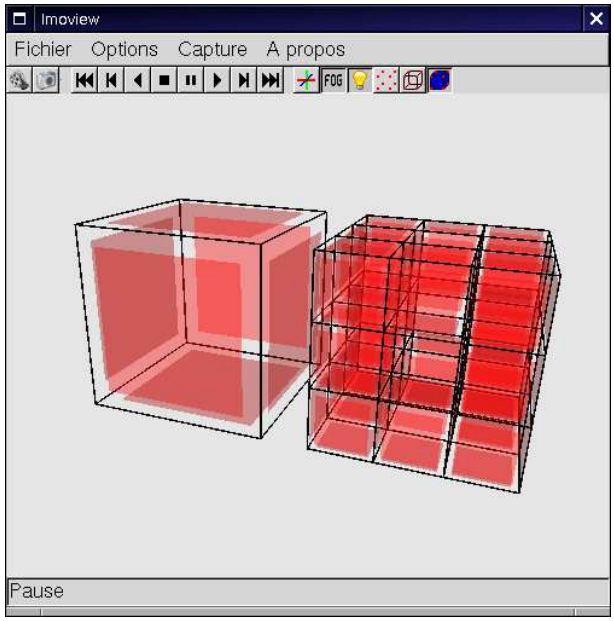


FIG. 2 – Capture d'écran du logiciel Imoview : construction de l'éponge de Menger

Imoview permet :

- de charger le contenu d'un fichier décrivant des scènes dans le langage Teom,
- d'animer l'affichage de ces scènes pour rendre compte de l'évolution du système représenté,
- de naviguer dans la scène (zoom, translation et rotation de la caméra à l'aide de la souris) pour l'observer depuis n'importe quel point de vue,
- de faire des captures d'écran et de film.

Pour plus d'informations sur le logiciel Imoview, le lecteur intéressé se référera au rapport de stage de F. Thonnérieux [Tho03]. Je suis intervenu sur le logiciel Imoview pour :

- corriger les bugs,
- le mettre à jour,
- rajouter quelques fonctionnalités.

2.2 L'éditeur de règles PatchGen

La motivation principale du développement du logiciel PatchGen est née d'un constat fait à l'issue du développement de la construction de l'éponge de Menger (voir chapitre 8). L'éponge de Menger est une fractale générée à partir d'un cube plein qui est « troué » suivant chacune des trois directions. Cette construction demande trois étapes :

1. la subdivision des arcs en trois arcs,
2. la subdivision des faces en huit carrés (le carré central n'est pas créé), et
3. la subdivision du volume en vingt cubes.

Ce processus de construction est ensuite itéré sur les cubes créés pour obtenir à la limite la figure fractale.

La troisième étape consiste en une seule règle de patch filtrant 32 sommets, 108 arcs, 48 faces et un volume pour 8 sommets, 36 arcs, 48 faces et 20 volumes créés. Bien que particulièrement symétrique et extrêmement régulière, la règle correspondante compte environ 200 lignes sur lesquelles 301 variables doivent être prises en compte (voir pages 284–283). La programmation de ce patch est très fastidieuse et sujette à de nombreuses erreurs. C'est pourquoi, nous avons proposé un éditeur graphique de règles de patch.

La figure 3 présente une capture d'écran du logiciel PatchGen où la règle correspondant à la troisième étape de la construction de l'éponge de Menger est spécifiée graphiquement.

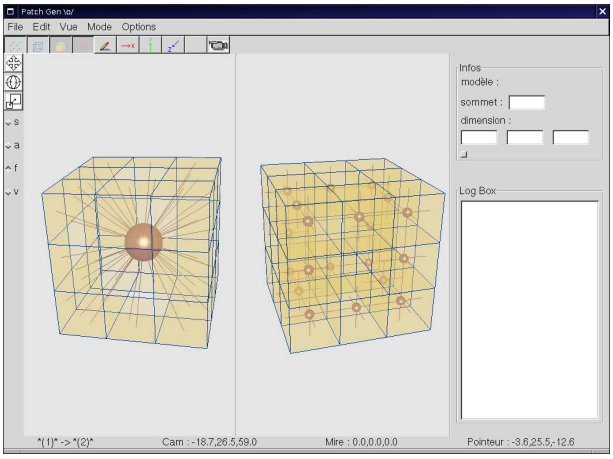


FIG. 3 – Capture d'écran du logiciel PatchGen : troisième étape de la construction de l'éponge de Menger

L'environnement graphique de PatchGen propose deux fenêtres OpenGL conjointes représentant respectivement la partie gauche et la partie droite de la règle éditée. Ces deux fenêtres sont indépendantes en termes de scène 3D, et il est possible d'y prendre deux points de vues différents comme le montre la figure 3. En revanche, les objets représentés de part et d'autre sont liés : les éléments composant l'objet de gauche représente les cellules filtrées ; à droite, celles-ci apparaissent toujours si elles sont réécrites. Afin de manipuler les deux objets, trois modes d'édition sont fournis :

1. toute action dans l'une des scènes est également appliquées dans l'autre ;
2. toute action dans la cellule de gauche (resp. de droite) est également appliquée dans la cellule de droite (resp. de gauche), mais pas l'inverse ;
3. les deux parties sont indépendantes.

Afin de réaliser ce couplage entre les deux scènes, les deux structures de données sous-jacentes, fondées sur la notion de graphe d'incidence pour respecter les besoins de MGS, sont liées l'une à l'autre.

PatchGen dispose également d'un format pour importer et exporter les règles graphiques. La figure 3 a été réalisée de la façon suivante :

- un maillage tridimensionnel 3x3x3 a été créé à l'aide MOKA ;
- ce maillage a ensuite été importé dans l'interprète MGS pour générer un fichier de données pour PatchGen contenant une règle filtrant ce maillage et le retournant inchangé[5] ;

[5] PatchGen ne permet pas pour l'instant d'importer des objets dans des formats autre que le sien. Nous tra-

- cette règle est ensuite éditée dans PatchGen :
  - dans le mode d'édition s'appliquant en même temps sur les parties gauches et droites de la règle, le maillage est perforé pour obtenir une itération de la construction de l'éponge de Menger, spécifiant ainsi la partie droite de la règle,
  - dans le mode d'édition laissant les deux parties indépendantes, la partie gauche (correspondant au filtre) est travaillée pour détruire l'intérieur de l'éponge et créer un unique volume ;
- la règle est finalement générée par PatchGen et utilisée dans un programme MGS.

Pour le cas de l'éponge de Menger, la programmation directe de la règle a nécessité 2 jours de travail contre 10 minutes avec l'utilisation de PatchGen. Pour plus d'informations sur le logiciel PatchGen, le lecteur intéressé se référera au rapport de stage de Y. Jullian [Jul05]. J'ai co-encadré avec O. Michel ce stage, conçu l'architecture du projet et participé au développement du logiciel.

vaillons à l'heure actuelle sur l'importation des G-cartes.

```
patch subdivide_volumes[gen] = {
vvol:[dim=3,faces in (vf11,vf12,vf13,vf14,vf15,vf16,vf17,vf18,
                      vf21,vf22,vf23,vf24,vf25,vf26,vf27,vf28,
                      vf31,vf32,vf33,vf34,vf35,vf36,vf37,vf38,
                      vf41,vf42,vf43,vf44,vf45,vf46,vf47,vf48,
                      vf51,vf52,vf53,vf54,vf55,vf56,vf57,vf58,
                      vf61,vf62,vf63,vf64,vf65,vf66,vf67,vf68
                     )]

~vf11 > ~ve1_12:[(vv11) in faces] < ~vf12 > ~ve1_23:[(vv12) in faces] <
~vf13 > ~ve1_34:[(vv12) in faces] < ~vf14 > ~ve1_45:[(vv13) in faces] <
~vf15 > ~ve1_56:[(vv13) in faces] < ~vf16 > ~ve1_67:[(vv14) in faces] <
~vf17 > ~ve1_78:[(vv14) in faces] < ~vf18 > ~ve1_81:[(vv11) in faces] < ~vf11

~vf21 > ~ve2_12:[(vv21) in faces] < ~vf22 > ~ve2_23:[(vv22) in faces] <
~vf23 > ~ve2_34:[(vv22) in faces] < ~vf24 > ~ve2_45:[(vv23) in faces] <
~vf25 > ~ve2_56:[(vv23) in faces] < ~vf26 > ~ve2_67:[(vv24) in faces] <
~vf27 > ~ve2_78:[(vv24) in faces] < ~vf28 > ~ve2_81:[(vv21) in faces] < ~vf21

~vf31 > ~ve3_12:[(vv31) in faces] < ~vf32 > ~ve3_23:[(vv32) in faces] <
~vf33 > ~ve3_34:[(vv32) in faces] < ~vf34 > ~ve3_45:[(vv33) in faces] <
~vf35 > ~ve3_56:[(vv33) in faces] < ~vf36 > ~ve3_67:[(vv34) in faces] <
~vf37 > ~ve3_78:[(vv34) in faces] < ~vf38 > ~ve3_81:[(vv31) in faces] < ~vf31

~vf41 > ~ve4_12:[(vv41) in faces] < ~vf42 > ~ve4_23:[(vv42) in faces] <
~vf43 > ~ve4_34:[(vv42) in faces] < ~vf44 > ~ve4_45:[(vv43) in faces] <
~vf45 > ~ve4_56:[(vv43) in faces] < ~vf46 > ~ve4_67:[(vv44) in faces] <
~vf47 > ~ve4_78:[(vv44) in faces] < ~vf48 > ~ve4_81:[(vv41) in faces] < ~vf41

~vf51 > ~ve5_12:[(vv51) in faces] < ~vf52 > ~ve5_23:[(vv52) in faces] <
~vf53 > ~ve5_34:[(vv52) in faces] < ~vf54 > ~ve5_45:[(vv53) in faces] <
~vf55 > ~ve5_56:[(vv53) in faces] < ~vf56 > ~ve5_67:[(vv54) in faces] <
~vf57 > ~ve5_78:[(vv54) in faces] < ~vf58 > ~ve5_81:[(vv51) in faces] < ~vf51

~vf61 > ~ve6_12:[(vv61) in faces] < ~vf62 > ~ve6_23:[(vv62) in faces] <
~vf63 > ~ve6_34:[(vv62) in faces] < ~vf64 > ~ve6_45:[(vv63) in faces] <
~vf65 > ~ve6_56:[(vv63) in faces] < ~vf66 > ~ve6_67:[(vv64) in faces] <
~vf67 > ~ve6_78:[(vv64) in faces] < ~vf68 > ~ve6_81:[(vv61) in faces] < ~vf61

~ve1_12 > ~vv12_1 < ~ve2_12
~ve1_23 > ~vv12_2 < ~ve2_23 ~ve2_34 > ~vv25_1 < ~ve5_12
~ve1_34 > ~vv15_1 < ~ve5_81 ~ve2_45 > ~vv25_2 < ~ve5_23
~ve1_45 > ~vv15_2 < ~ve5_78 ~ve5_56 > ~vv54_1 < ~ve4_34
~ve1_56 > ~vv14_1 < ~ve4_56 ~ve5_67 > ~vv54_2 < ~ve4_45
~ve1_67 > ~vv14_2 < ~ve4_67 ~ve4_78 > ~vv46_1 < ~ve6_67
~ve1_78 > ~vv16_1 < ~ve6_78 ~ve4_81 > ~vv46_2 < ~ve6_56
~ve1_81 > ~vv16_2 < ~ve6_81 ~ve6_12 > ~vv62_1 < ~ve2_81
~ve6_23 > ~vv62_2 < ~ve2_78
~ve3_12 > ~vv34_1 < ~ve4_12
~ve3_23 > ~vv34_2 < ~ve4_23
~ve3_34 > ~vv35_1 < ~ve5_45
~ve3_45 > ~vv35_2 < ~ve5_34
~ve3_56 > ~vv32_1 < ~ve2_56
~ve3_67 > ~vv32_2 < ~ve2_67
~ve3_78 > ~vv36_1 < ~ve6_34
~ve3_81 > ~vv36_2 < ~ve6_45

=>
let fct = \p1,p2.((common_cofaces(p1,p2)).(0)) in

(* Création des huit points centraux *)
‘v126:[dim=0,val={gen=0}+bary(c2_3,vv11,c1_3,vv34)]
‘v326:[dim=0,val={gen=0}+bary(c1_3,vv11,c2_3,vv34)]
‘v125:[dim=0,val={gen=0}+bary(c2_3,vv12,c1_3,vv33)]
‘v325:[dim=0,val={gen=0}+bary(c1_3,vv12,c2_3,vv33)]
‘v145:[dim=0,val={gen=0}+bary(c2_3,vv13,c1_3,vv32)]
‘v345:[dim=0,val={gen=0}+bary(c1_3,vv13,c2_3,vv32)]
‘v146:[dim=0,val={gen=0}+bary(c2_3,vv14,c1_3,vv31)]
‘v346:[dim=0,val={gen=0}+bary(c1_3,vv14,c2_3,vv31)]

(* Création des arcs internes *)
‘e13_14_1:[dim=1,faces=(~vv11,‘v126),val=‘edge]
‘e13_14_2:[dim=1,faces=(‘v126,‘v326),val=‘edge]
‘e13_14_3:[dim=1,faces=(‘v326,~vv34),val=‘edge]
‘e13_23_1:[dim=1,faces=(~vv12,‘v125),val=‘edge]
‘e13_23_2:[dim=1,faces=(‘v125,‘v325),val=‘edge]
‘e13_23_3:[dim=1,faces=(‘v325,~vv33),val=‘edge]
‘e13_32_1:[dim=1,faces=(~vv13,‘v145),val=‘edge]
‘e13_32_2:[dim=1,faces=(‘v145,‘v345),val=‘edge]
‘e13_32_3:[dim=1,faces=(‘v345,~vv32),val=‘edge]
‘e13_41_1:[dim=1,faces=(~vv14,‘v146),val=‘edge]
‘e13_41_2:[dim=1,faces=(‘v146,‘v346),val=‘edge]
‘e13_41_3:[dim=1,faces=(‘v346,~vv31),val=‘edge]

‘e24_14_1:[dim=1,faces=(~vv21,‘v126),val=‘edge]
‘e24_14_2:[dim=1,faces=(‘v126,‘v146),val=‘edge]
‘e24_14_3:[dim=1,faces=(‘v146,~vv44),val=‘edge]
‘e24_23_1:[dim=1,faces=(~vv22,‘v125),val=‘edge]
‘e24_23_2:[dim=1,faces=(‘v125,‘v145),val=‘edge]
‘e24_23_3:[dim=1,faces=(‘v145,~vv43),val=‘edge]
‘e24_32_1:[dim=1,faces=(~vv23,‘v325),val=‘edge]
‘e24_32_2:[dim=1,faces=(‘v325,‘v345),val=‘edge]
‘e24_32_3:[dim=1,faces=(‘v345,~vv42),val=‘edge]
‘e24_41_1:[dim=1,faces=(~vv24,‘v326),val=‘edge]
‘e24_41_2:[dim=1,faces=(‘v326,‘v346),val=‘edge]
‘e24_41_3:[dim=1,faces=(‘v346,~vv41),val=‘edge]
```

```
‘e56_11_1:[dim=1,faces=(^vv51,‘v125),val=‘edge]
‘e56_11_2:[dim=1,faces=(‘v125,‘v126),val=‘edge]
‘e56_11_3:[dim=1,faces=(‘v126,^vv61),val=‘edge]
‘e56_22_1:[dim=1,faces=(^vv52,‘v325),val=‘edge]
‘e56_22_2:[dim=1,faces=(‘v325,‘v326),val=‘edge]
‘e56_22_3:[dim=1,faces=(‘v326,^vv62),val=‘edge]
‘e56_33_1:[dim=1,faces=(^vv53,‘v345),val=‘edge]
‘e56_33_2:[dim=1,faces=(‘v345,‘v346),val=‘edge]
‘e56_33_3:[dim=1,faces=(‘v346,^vv63),val=‘edge]
‘e56_44_1:[dim=1,faces=(^vv54,‘v145),val=‘edge]
‘e56_44_2:[dim=1,faces=(‘v145,‘v146),val=‘edge]
‘e56_44_3:[dim=1,faces=(‘v146,^vv64),val=‘edge]

(* Création des faces internes communes à 2 bords...  *)
‘f12_1:[dim=2,faces=(^ve1_12,^ve2_12,‘e13_14_1,‘e24_14_1),val=‘face]
‘f12_2:[dim=2,faces=(^ve1_23,^ve2_23,‘e13_23_1,‘e24_23_1),val=‘face]
‘f15_1:[dim=2,faces=(^ve1_34,^ve5_81,‘e13_23_1,‘e56_11_1),val=‘face]
‘f15_2:[dim=2,faces=(^ve1_45,^ve5_78,‘e13_32_1,‘e56_44_1),val=‘face]
‘f14_1:[dim=2,faces=(^ve1_56,^ve4_56,‘e13_32_1,‘e24_23_3),val=‘face]
‘f14_2:[dim=2,faces=(^ve1_67,^ve4_67,‘e13_41_1,‘e24_14_3),val=‘face]
‘f16_1:[dim=2,faces=(^ve1_78,^ve6_78,‘e13_41_1,‘e56_44_3),val=‘face]
‘f16_2:[dim=2,faces=(^ve1_81,^ve6_81,‘e13_14_1,‘e56_11_3),val=‘face]

‘f34_1:[dim=2,faces=(^ve3_12,^ve4_12,‘e13_41_3,‘e24_41_3),val=‘face]
‘f34_2:[dim=2,faces=(^ve3_23,^ve4_23,‘e13_32_3,‘e24_32_3),val=‘face]
‘f35_1:[dim=2,faces=(^ve3_34,^ve5_45,‘e13_32_3,‘e56_33_1),val=‘face]
‘f35_2:[dim=2,faces=(^ve3_45,^ve5_34,‘e13_23_3,‘e56_22_1),val=‘face]
‘f32_1:[dim=2,faces=(^ve3_56,^ve2_56,‘e13_23_3,‘e24_32_1),val=‘face]
‘f32_2:[dim=2,faces=(^ve3_67,^ve2_67,‘e13_14_3,‘e24_41_1),val=‘face]
‘f36_1:[dim=2,faces=(^ve3_78,^ve6_34,‘e13_14_3,‘e56_22_3),val=‘face]
‘f36_2:[dim=2,faces=(^ve3_81,^ve6_45,‘e13_41_3,‘e56_33_3),val=‘face]

‘f25_1:[dim=2,faces=(^ve2_34,^ve5_12,‘e24_23_1,‘e56_11_1),val=‘face]
‘f25_2:[dim=2,faces=(^ve2_45,^ve5_23,‘e24_32_1,‘e56_22_1),val=‘face]
‘f26_1:[dim=2,faces=(^ve2_78,^ve6_23,‘e24_41_1,‘e56_22_3),val=‘face]
‘f26_2:[dim=2,faces=(^ve2_81,^ve6_12,‘e24_14_1,‘e56_11_3),val=‘face]

‘f45_1:[dim=2,faces=(^ve4_34,^ve5_56,‘e24_32_3,‘e56_33_1),val=‘face]
‘f45_2:[dim=2,faces=(^ve4_45,^ve5_67,‘e24_23_3,‘e56_44_1),val=‘face]
‘f46_1:[dim=2,faces=(^ve4_78,^ve6_67,‘e24_14_3,‘e56_44_3),val=‘face]
‘f46_2:[dim=2,faces=(^ve4_81,^ve6_56,‘e24_41_3,‘e56_33_3),val=‘face]

(* ...  et des volumes qui vont avec *)
‘v126:[dim=3,faces=(^vf11,^vf21,^vf61,‘f12_1,‘f26_2,‘f16_2),val=‘volume]
‘v125:[dim=3,faces=(^vf13,^vf23,^vf51,‘f12_2,‘f25_1,‘f15_1),val=‘volume]
‘v145:[dim=3,faces=(^vf15,^vf45,^vf57,‘f14_1,‘f45_2,‘f15_2),val=‘volume]
‘v146:[dim=3,faces=(^vf17,^vf47,^vf67,‘f14_2,‘f46_1,‘f16_1),val=‘volume]
‘v326:[dim=3,faces=(^vf37,^vf27,^vf63,‘f32_2,‘f26_1,‘f36_1),val=‘volume]
‘v325:[dim=3,faces=(^vf35,^vf25,^vf53,‘f32_1,‘f25_2,‘f35_2),val=‘volume]
‘v345:[dim=3,faces=(^vf33,^vf43,^vf55,‘f34_2,‘f45_1,‘f35_1),val=‘volume]
‘v346:[dim=3,faces=(^vf31,^vf41,^vf65,‘f34_1,‘f46_2,‘f36_2),val=‘volume]
```

```
(* Création des faces internes formant les trous...  *)

‘f12:[dim=2,faces=(‘e13_14_1,‘e13_23_1,‘e56_11_2,fct(^vv11,^vv12)),val=‘face]
‘f14:[dim=2,faces=(‘e13_23_1,‘e13_32_1,‘e24_23_2,fct(^vv12,^vv13)),val=‘face]
‘f16:[dim=2,faces=(‘e13_32_1,‘e13_41_1,‘e56_44_2,fct(^vv13,^vv14)),val=‘face]
‘f18:[dim=2,faces=(‘e13_41_1,‘e13_14_1,‘e24_14_2,fct(^vv14,^vv11)),val=‘face]

‘f32:[dim=2,faces=(‘e13_41_3,‘e13_32_3,‘e56_33_2,fct(^vv31,^vv32)),val=‘face]
‘f34:[dim=2,faces=(‘e13_32_3,‘e13_23_3,‘e24_32_2,fct(^vv32,^vv33)),val=‘face]
‘f36:[dim=2,faces=(‘e13_23_3,‘e13_14_3,‘e56_22_2,fct(^vv33,^vv34)),val=‘face]
‘f38:[dim=2,faces=(‘e13_14_3,‘e13_41_3,‘e24_41_2,fct(^vv34,^vv31)),val=‘face]

‘f22:[dim=2,faces=(‘e24_14_1,‘e24_23_1,‘e56_11_2,fct(^vv21,^vv22)),val=‘face]
‘f24:[dim=2,faces=(‘e24_23_1,‘e24_32_1,‘e13_23_2,fct(^vv22,^vv23)),val=‘face]
‘f26:[dim=2,faces=(‘e24_32_1,‘e24_41_1,‘e56_22_2,fct(^vv23,^vv24)),val=‘face]
‘f28:[dim=2,faces=(‘e24_41_1,‘e24_14_1,‘e13_14_2,fct(^vv24,^vv21)),val=‘face]

‘f42:[dim=2,faces=(‘e24_41_3,‘e24_32_3,‘e56_33_2,fct(^vv41,^vv42)),val=‘face]
‘f44:[dim=2,faces=(‘e24_32_3,‘e24_23_3,‘e13_32_2,fct(^vv42,^vv43)),val=‘face]
‘f46:[dim=2,faces=(‘e24_23_3,‘e24_14_3,‘e56_44_2,fct(^vv43,^vv44)),val=‘face]
‘f48:[dim=2,faces=(‘e24_14_3,‘e24_41_3,‘e13_41_2,fct(^vv44,^vv41)),val=‘face]

‘f52:[dim=2,faces=(‘e56_11_1,‘e56_22_1,‘e13_23_2,fct(^vv51,^vv52)),val=‘face]
‘f54:[dim=2,faces=(‘e56_22_1,‘e56_33_1,‘e24_32_2,fct(^vv52,^vv53)),val=‘face]
‘f56:[dim=2,faces=(‘e56_33_1,‘e56_44_1,‘e13_32_2,fct(^vv53,^vv54)),val=‘face]
‘f58:[dim=2,faces=(‘e56_44_1,‘e56_11_1,‘e24_23_2,fct(^vv54,^vv51)),val=‘face]

‘f62:[dim=2,faces=(‘e56_11_3,‘e56_22_3,‘e13_14_2,fct(^vv61,^vv62)),val=‘face]
‘f64:[dim=2,faces=(‘e56_22_3,‘e56_33_3,‘e24_41_2,fct(^vv62,^vv63)),val=‘face]
‘f66:[dim=2,faces=(‘e56_33_3,‘e56_44_3,‘e13_41_2,fct(^vv63,^vv64)),val=‘face]
‘f68:[dim=2,faces=(‘e56_44_3,‘e56_11_3,‘e24_14_2,fct(^vv64,^vv61)),val=‘face]

(* ...  et les volumes qui vont avec *)
‘v12:[dim=3,faces=(‘f12,‘f22,^vf12,^vf22,‘f12_1,‘f12_2),val=‘volume]
‘v15:[dim=3,faces=(‘f14,‘f58,^vf14,^vf58,‘f15_1,‘f15_2),val=‘volume]
‘v14:[dim=3,faces=(‘f16,‘f46,^vf16,^vf46,‘f14_1,‘f14_2),val=‘volume]
‘v16:[dim=3,faces=(‘f18,‘f68,^vf18,^vf68,‘f16_1,‘f16_2),val=‘volume]
‘v32:[dim=3,faces=(‘f36,‘f26,^vf36,^vf26,‘f32_1,‘f32_2),val=‘volume]
‘v35:[dim=3,faces=(‘f34,‘f54,^vf34,^vf54,‘f35_1,‘f35_2),val=‘volume]
‘v34:[dim=3,faces=(‘f32,‘f42,^vf32,^vf42,‘f34_1,‘f34_2),val=‘volume]
‘v36:[dim=3,faces=(‘f38,‘f64,^vf38,^vf64,‘f36_1,‘f36_2),val=‘volume]
‘v25:[dim=3,faces=(‘f24,‘f52,^vf24,^vf52,‘f25_1,‘f25_2),val=‘volume]
‘v26:[dim=3,faces=(‘f28,‘f62,^vf28,^vf62,‘f26_1,‘f26_2),val=‘volume]
‘v45:[dim=3,faces=(‘f48,‘f56,^vf48,^vf56,‘f45_1,‘f45_2),val=‘volume]
‘v46:[dim=3,faces=(‘f44,‘f66,^vf44,^vf66,‘f46_1,‘f46_2),val=‘volume]

} ;;
```

Bibliographie en relation avec ce travail

Chapitres de livres.

[GS06a] Jean-Louis Giavitto and Antoine Spicher. *Morphogénèse*, chapter Morphogénèse informatique, pages 178–198. Belin, 2006.

[GS06b] Jean-Louis Giavitto and Antoine Spicher. *Systems Self-Assembly : multidisciplinary snapshots*, chapter Simulation of self-assembly processes using abstract reduction systems. Elsevier, 2006.

Revues internationales.

[SM05a] Antoine Spicher and Olivier Michel. Declarative modeling of a neurulation-like process. *BioSystems*, 2005.

[SMC+06] Antoine Spicher, Olivier Michel, Mikolaj Cieslak, Jean-Louis Giavitto, and Przemyslaw Prusinkiewicz. Stochastic P Systems and the simulation of biochemical processes with dynamic compartments. **Accepted for *BioSystems* special issue : Membrane Computing - Applications in Systems Biology, 2006**.

Conférences et workshops internationaux.

[SMG04a] Antoine Spicher, Olivier Michel, and Jean-Louis Giavitto. A topological framework for the specification and the simulation of discrete dynamical systems. In *Sixth International conference on Cellular Automata for Research and Industry (ACRI'04)*, volume 3305 of *LNCS*, Amsterdam, October 2004. Springer.

[SM04a] Antoine Spicher and Olivier Michel. Integration and pattern-matching of topological structures in a functional language. In *International Workshop on Implementation and Application of Functional Languages (IFL04)*, Lübeck, September 2004. draft proceedings published as a technical report of the Institute of Computer Science of the University of Kiel.

[GMCS05] Jean-Louis Giavitto, Olivier Michel, Julien Cohen, and Antoine Spicher. Computation in space and space in computation. In *Unconventional Programming Paradigms (UPP'04)*, volume 3566 of *LNCS*, pages 137–152, Le Mont Saint-Michel, September 2005. Springer.

[SMG05] Antoine Spicher, Olivier Michel, and Jean-Louis Giavitto. Algorithmic self-assembly by accretion and by carving in MGS. In *Proc. of the 7th International Conference on Artificial Evolution (EA'05)*, LNCS, University of Lille - France, October 2005. Springer-Verlag. (to be published).

[SM05b] Antoine Spicher and Olivier Michel. Using rewriting techniques in the simulation of dynamical systems : Application to the modeling of sperm crawling. In *Fifth International Conference on Computational Science (ICCS'05), part I*, volume 3514 of *LNCS*, pages 820–827, Atlanta, GA, USA, May 2005. Springer.

Conférences nationales.

[SM04b] Antoine Spicher and Olivier Michel. Manipulations de structures topologiques dans un langage déclaratif pour la simulation. In *11ème Journées du GT "Animation et Simulation" (GTAS'2004)*, Reims, juin 2004. AFIG et LERI, Université de Reims.

[Spi05] Antoine Spicher. Représentation et manipulation de structures topologiques dans un langage fonctionnel. In O. Michel, editor, *Journées Francophones des Langages Applicatifs (JFLA'2005)*, pages 113–128. INRIA, 2005.

[SM06] Antoine Spicher and Olivier Michel. Stratégie d'application stochastique de règles de réécriture dans le langage MGS. In *Journées Francophones des Langages Applicatifs (JFLA'2006)*, pages 147–163. INRIA, 2006.

Rapports techniques.

[Spi03] Antoine Spicher. Typage et compilation de filtrage de chemins dans des collections topologiques. Master's thesis, Université d'Évry, 2003.

[SMG04b] Antoine Spicher, Olivier Michel, and Jean-Louis Giavitto. A topological framework for the specification and the simulation of discrete dynamical systems. Technical Report LaMI-99-2004, LaMI, May 2004.

[GMCS04] Jean-Louis Giavitto, Olivier Michel, Julien Cohen, and Antoine Spicher. Computation in space and space in computation. Technical Report 103-2004, LaMI - University of Évry, May 2004.

Bibliographie

[AAC+00] Harold Abelson, Don Allen, Daniel Coore, Chris Hanson, George Homsy, Thomas F. Knight, Radhika Nagpal, Erik Rauch, Gerald Jay Sussman, and Ron Weiss. Amorphous computing. *CACM : Communications of the ACM*, 43, 2000.

[ABL+94] Bruce Alberts, Dennis Bray, Julian Lewis, Martin Raff, Keith Roberts, and James Watson. *Molecular Biology of the Cell.* Garland, New-York, third edition, 1994.

[AC03] Ioan Ardelean and Matteo Cavaliere. Modelling biological processes by using a probabilistic P system software. *Natural Computing*, 2(2) :173–197, 2003.

[AIRRH03] Charles Auffray, Sandrine Imbeaud, Magali Roux-Rouquié, and Leroy Hood. From functional genomics to systems biology : concepts and practices. *CR biologies*, 326(10–11) :879–892, 2003.

[Ale82] Paul Alexandroff. *Elementary concepts of topology.* Dover publications, New-York, 1982.

[Aur91] Franz Aurenhammer. Voronoi diagrams–a survey of a fundamental geometric data structure. *ACM Comput. Surv.*, 23(3) :345–405, 1991.

[Axe98] Ulrike Axen. *Topological Analysis using Morse theory and auditory display.* PhD thesis, University of Illinois at Urbana Champaign, 1998.

[Bau74] Bruce Guenther Baumgart. *Geometric modeling for computer vision.* PhD thesis, 1974.

[BB90] Gérard Berry and Gérard 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.

[BdR05] Pierre Barbier de Reuille. *Vers un modèle dynamique du méristème apical caulinaire d'Arabidopsis thaliana.* PhD thesis, Université Montpellier II, 2005.

[BdRBCL+06] Pierre Barbier de Reuille, Isabelle Bohn-Courseau, Karin Ljung, Halima Morin, Nicola Carraro, Christophe Godin, and Jan Traas. Computer simulations reveal properties of the cell-cell signaling network at the shoot apex in Arabidopsis. *PNAS*, 103(5) :1627–1632, 2006.

[Bea05] Jacob Beal. Amorphous medium language. In *Large-Scale Multi-Agent Systems Workshop - AAMAS 2005*, 2005.

[Ber01] Guntram Berti. *Generic software components for scientific computing.* PhD thesis, Technical University of Cottbus, june 2001.

[BFGM05] Jean-Pierre Banâtre, Pascal Fradet, Jean-Louis Giavitto, and Olivier Michel, editors. *Unconventional Programming Paradigms*, Revised Selected and Invited Papers of the International Workshop UPP 2004, Le Mont-Saint-Michel, France, 2005. Springer-Verlag, LNCS, Vol. 3566.

[BFL01] 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.

[BH03] Olivier Bournez and Mathieu Hoyrup. Rewriting logic and probabilities. In Robert Nieuwenhuis, editor, *14th Int. Conf. on Rewriting Techniques and Applications (RTA'03)*, volume 2706 of *Lecture Notes in Computer Science*, pages 61–75, Berlin, 2003. Springer.

[Bir87] Richard S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design, NATO ASI Series, vol. F36*, pages 217–245. sv, 1987.

[BK02] Olivier Bournez and Claude Kirchner. Probabilistic rewrite strategies. applications to ELAN. In Sophie Tison, editor, *Rewriting Techniques and Applications, 13th International Conference, RTA 2002, Copenhagen, Denmark, July 22-24, 2002, Proceedings*, volume 2378 of *Lecture Notes in Computer Science*, pages 252–266, Berlin, 2002. Springer.

[BK03] Jean-Daniel Boissonnat and Menelaos I. Karavelas. On the combinatorial complexity of euclidean voronoi cells and convex hulls of d-dimensional spheres. In *SODA '03 : Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 305–312, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[BL86] Jean-Pierre Banâtre and Daniel Le Métayer. A new computational model and its discipline of programming. Technical Report RR-0566, INRIA, 1986.

[BL90] Jean-Pierre Banâtre and Daniel Le Métayer. The GAMMA model and its discipline of programming. *Science of Computer Programming*, 15(1) :55–77, 1990.

[BMR+02] Dean Bottino, Alex Mogilner, Thomas Roberts, Murray Stewart, and George Oster. How nematode sperm crawl. *Journal of Cell Science*, 115 :367–384, 2002.

[BNTW95] Peter Buneman, Shamim Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1) :3–48, 1995.

[Bot00] Dean Bottino. *Ascaris suum* sperm model documentation, 2000.

[BPSM+06] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (xml) 1.0 (fourth edition). Technical report, W3C Recommendation, August 2006.

[Bri89] Erik Brisson. Representing geometric structures in d dimensions : topology and order. In *SCG '89 : Proceedings of the fifth annual symposium on Computational geometry*, pages 218–227, New York, NY, USA, 1989. ACM Press.

[Car04] Luca Cardelli. Brane calculi. In Vincent Danos and Vincent Schächter, editors, *Computational Methods in Systems Biology*, volume 3082 of *Lecture Notes in Computer Science*, pages 257–278, Berlin, 2004. Springer.

[CC78] Edwin E. Catmull and Jim Clark. Recursively generated b-spline surfaces on arbitrary topological surfaces. *Computer-Aided Design*, 10(6) :350–355, 1978.

[Cha74] G. Chaikin. An algorithm for high speed curve generation. *Computer Graphics and Image Processing*, 3 :346–349, 1974.

[Cie06] Mikolaj Cieslak. Stochastic simulation of pattern formation : An application of L-systems. Master's thesis, University of Calgary, 2006.

[Coh04] Julien Cohen. *Intégration des collections topologiques et des transformations dans un langage de programmation fonctionnel*. PhD thesis, Université d'Évry, December 2004. `http://www.ibisc.univ-evry.fr/~jcohen/THESE/these.officiel.pdf`.

[Coh05] Julien Cohen. Interprétation par SK-traduction et syntaxe abstraite d'ordre supéieur. In O. Michel, editor, *Journées Francophones des Langages Applicatifs (JFLA 2005)*, pages 17–34. INRIA, 2005.

[Coo91] Daniel Coore. *Botanical Computing : A Developmental Approach to Generating Interconnect Topologies on an Amorphous Computer*. PhD thesis, Massachusetts Institute of Technology, 1991.

[Cou93] Bruno Courcelle. Réécriture de graphes : orientation bibliographique. Spring school on rewriting, Odeillo, May 1993, unpublished, available on author's web page, 1993.

[CRLM+04] Enrico Coen, Anne-Gaelle Rolland-Lagan, Mark Matthews, J. Andrew Bangham, and Przemyslaw Prusinkiewicz. The genetics of geometry. *PNAS*, 101(14) :4728–4735, 2004.

[CS00] Jeffrey A. Chard and Vadim Shapiro. A multivector data structure for differential forms and equations. *Math. Comput. Simul.*, 54(1-3) :33–64, 2000.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4) :471–523, 1985.

[Del02] Emmanuel Delsinne. Structure de données indexées par un groupe : isomorphsime de GBF abéliens et extensions aux structures automatiques. Master's thesis, Université d'Évry, 2002.

[Der93] Nachum Dershowitz. *A Taste of Rewrite Systems*, volume 693 of *Lecture Notes in Computer Science*, pages 199–228. Springer Verlag, 1993.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewriting systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 243–320. Elsevier Publishers, Amsterdam, 1990.

[DKT06] Mathieu Desbrun, Eva Kanso, and Yiying Tong. Discrete ifferential forms for computational modeling. In *Discrete differential geometry : an applied introduction*, pages 39–54. Schröder, P, 2006. SIGGRAPH'06 course notes.

[DL02] Jean-François Dufourd and Sven Luther. Parametrizing geometric objects using λ-calculus. In *SCCG '02 : Proceedings of the 18th spring conference on Computer graphics*, pages 185–194, New York, NY, USA, 2002. ACM Press.

[DLG90] Nira Dyn, David Levine, and John A. Gregory. A butterfly subdivision scheme for surface interpolation with tension control. *ACM Trans. Graph.*, 9(2) :160–169, 1990.

[DR04] Romain De Rasse. Implantation de la structure de graphe dans le langage MGS. Master's thesis, Université d'Évry, 2004.

[DS78] Daniel Doo and Malcolm Sabin. Analysis of the behaviour of recursive division surfaces near extraordinary points. *Computer-Aided Design*, 10(6) :356–360, 1978.

[DZB01] Petter Dittrich, Jens Ziegler, and Wolfgang Banzhaf. Artificial chemistries - a review. *Artificial Life*, 7(3) :225–275, 2001.

[EKL+02] Steven Eker, Merrill Knapp, Keith Laderoute, Patrick Lincoln, and Carolyn L. Talcott. Pathway logic : Executable models of biological networks. *Electr. Notes Theor. Comput. Sci*, 71, 2002.

[EPS73] Hartmut Ehrig, Michael Pfender, and Hans Jürgen Schneider. Graph grammars : An algebraic approach. In *FOCS : IEEE Symposium on Foundations of Computer Science (FOCS)*, 1973.

[ES79] Manfred Eigen and Peter Schuster. *The Hypercycle : A Principle of Natural Self-Organization.* Springer, 1979.

[ES80] Peter Eichhorst and Walter J. Savitch. Growth functions of stochastic Lindenmayer systems. *Information and Control*, 45(3) :217–228, 1980.

[ES04] Richard Egli and Neil F. Stewart. Chain models in computer simulation. *Math. Comput. Simul.*, 66(6) :449–468, 2004.

[FB94] Walter Fontana and Leo W. Buss. "the arrival of the fittest" : Toward a theory of biological organization. *Bulletin of Mathematical Biology*, 1994.

[FG95] Jun P. Furuse and Jacques Garrigue. A label-selective lambda-calculus with optional arguments and its compilation method. RIMS Preprint 1041, Kyoto University, October 1995.

[Fla63] Harley Flanders. *Differential forms with applications to the physical sciences.* Mathematics in Science and Engineering, New York : Academic Press, 1963.

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

[Fre90] Edward Fredkin. Digital mechanics : An informational process based on reversible universal CA. *Physica D*, 45 :254, 1990.

[FW04] David C. Fallside and Priscilla Walmsley. Xml schema part 0 : Primer second edition. Technical report, W3C Recommendation, 2004.

[GB00] Michael A. Gibson and Jehoshua Bruck. Efficient exact stochastic simulation of chemical systems with many species and many channels. *J. Chem. Physics*, 104 :1876–1889, 2000.

[GDT+06] Mark Galassi, Jim Davies, James Theiler, Brian Gough, Gerard Jungman, Michael Booth, and Fabrice Rossi. *GNU Scientific Library – Reference Manual*, release 1.8 edition, 2006. `http://www.gnu.org/software/gsl/manual/html_node/`.

[GGMP02a] Jean-Louis Giavitto, Christope Godin, Olivier Michel, and Przemyslaw Prusinkiewicz. Computational models for integrative and developmental biology. Technical Report 72-2002, LaMI – Université d'Évry Val d'Essonne, March 2002. draft version of [GGMP02b].

[GGMP02b] Jean-Louis Giavitto, Christophe Godin, Olivier Michel, and Przemyslaw Prusinkiewicz. *Modelling and Simulation of biological processes in the context of genomics*, chapter "Computational Models for Integrative and Developmental Biology". Hermes, July 2002. Also republished as an high-level course in the proceedings of the Dieppe spring school on "Modelling and simulation of biological processes in the context of genomics", 12-17 may 2003, Dieppes, France.

[Gia03] Jean-Louis 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.

[Gia04] Jean-Louis Giavitto. *Modelling and simulation of biological processes in the context of genomics*, chapter On "The biochemical abstract machine BIOCHAM", pages 175–176. Genopole & Platypus Press, ISBN 2-84704-0374, 2004. Comments on a presentation of François Fages to the thematic school held in Évry (France) in April 2004.

[Gil77] Daniel T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *J. Phys. Chem.*, 81(25) :2340–2361, 1977.

[Gil00] Daniel T. Gillespie. Chemical Langevin equation. *J. Chem. Physics*, 113 :297–306, 2000.

[Gil01] Daniel T. Gillespie. Approximate accelerated stochastic simulation of chemically reacting systems. *J. Chem. Physics*, 115 :1716–1733, 2001.

[GM01a] Jean-Louis Giavitto and Olivier Michel. MGS : a rule-based programming language for complex objects and collections. In Mark van den Brand and Rakesh Verma, editors, *Electronic Notes in Theoretical Computer Science*, volume 59. Elsevier Science Publishers, 2001.

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

[GM02a] Jean-Louis Giavitto and Olivier 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] Jean-Louis Giavitto and Olivier Michel. The topological structures of membrane computing. *Fundamenta Informaticae*, 49 :107–129, 2002.

[GM03] Jean-Louis Giavitto and Olivier Michel. Modeling the topological organization of cellular processes. *BioSystems*, 70(2) :149–163, 2003.

[GMCS04] Jean-Louis Giavitto, Olivier Michel, Julien Cohen, and Antoine Spicher. Computation in space and space in computation. Technical Report 103-2004, LaMI - University of Évry, May 2004.

[GMCS05] Jean-Louis Giavitto, Olivier Michel, Julien Cohen, and Antoine Spicher. Computation in space and space in computation. In *Unconventional Programming Paradigms (UPP'04)*, volume 3566 of *LNCS*, pages 137–152, Le Mont Saint-Michel, September 2005. Springer.

[GMM04] Jean-Louis Giavitto, Grant Malcolm, and Olivier Michel. Rewriting systems and the modelling of biological systems. *Comparative and Functional Genomics*, 5 :95–99, February 2004.

[GMS96] Jean-Louis Giavitto, Olivier Michel, and Jean-Paul Sansonnet. Group-based fields. In *Proceedings of the International Workshop on Parallel Symbolic Languages and Systems (PSLS'95)*, volume 1068 of *Lecture Notes in Computer Science*, pages 209–215. Springer, 1996.

[Gol91] David Goldberg. What every computer scientist should know about floating-point arithmetics. *ACM Comput. Surv.*, 23(1) :5–48, 1991.

[GS06a] Jean-Louis Giavitto and Antoine Spicher. *Morphogénèse*, chapter Morphogénèse informatique, pages 178–198. Belin, 2006.

[GS06b] Jean-Louis Giavitto and Antoine Spicher. *Systems Self-Assembly : multidisciplinary snapshots*, chapter Simulation of self-assembly processes using abstract reduction systems. Elsevier, 2006.

[Har99] John C. Hart. Using the CW-complex to represent the topological structure of implicit surfaces and solids. In *Proc. Implicit Surfaces'99*, pages 107–112, 1999. Eurographics/SIGGRAPH.

[Hat02] Allen Hatcher. *Algebraic Topology*. Cambridge University Press, 2002.

[Hen94] Michael Henle. *A combinatorial introduction to topology*. Dover publications, New-York, 1994.

[HF92] Paul Hudak and Joseph H. Fasel. A gentle introduction to Haskell. *ACM SIGPLAN Notices*, 27(5) :T–1–T–53, May 1992.

[HGH93] John Y. Hung, Weibing Gao, and James C. Hung. Variable structure control : A survey. *IEEE Transactions on Industrial Electronics*, 40(1) :2–22, 1993.

[Hil85] Daniel W. Hillis. *The Connection Machine*. MIT Press, Cambridge, Mass., 1 edition, 1985.

[Hir03] Anil N. Hirani. *Discrete exterior calculus*. PhD thesis, California Institute of Technology, 2003.

[Hor01] Paul Horn. Autonomic computing : IBM's perspective on the state of information technology. Technical report, IBM Research, October 2001. `http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf%`.

[Ibă04] Liliana Ibănescu. *Programmation par règle et stratégies pour la génération automatique de mécanisme de combustion d'hydrocarbures polycycliques*. PhD thesis, Institut National Polytechnique de Lorraine, 2004.

[Itk76] Yevgeny Itkis. *Control Systems of Variable Structure*. Wiley, 1976.

[Jon90] Claire Jones. *Probabilistic Non-Determinism*. PhD thesis, University of Edinburgh, 1990.

[Jul05] Yann Jullian. Conception et développement d'un éditeur graphique de règle. Rapport de stage 3^{e} année - IIE, 2005.

[Jür76] Helmut Jürgensen. Probabilistic L-systems. In Aristid Lindenmayer and Grzegorz Rozenberg, editors, *Automata, Languages, Development*, pages 211–225, Amsterdam, 1976. North-Holland.

[Kaa92] Jaap A. Kaandorp. *Modelling growth forms of biological objects using fractals*. PhD thesis, University of Amsterdam, 1992.

[Kah87] Gilles Kahn. Natural semantics. Technical Report 601, INRIA, February 1987.

[KKMA03] Sen Koushik, Nirman Kumar, José Meseguer, and Gul Agha. Probabilistic rewrite theories. Technical Report 2343, University of Illinois at Urbana Champaign, 2003.

[Kle00] Reinhard Klette. Cell complexes through time. In L. J. Latecki, D. M. Mount, and A. Y. Wu, editors, *Proc. SPIE Vol. 4117, p. 134-145, Vision Geometry IX, Longin J. Latecki; David M. Mount; Angela Y. Wu; Eds.*, pages 134–145, October 2000.

[KN05] Céline Kuttler and Joachim Niehren. Gene regulation in the pi calculus : simulation coperativity at the *lambda* switch. *Transactions on Computational Systems Biology*, 2005.

[Kob96] Leif Kobbelt. Interpolatory subdivision on open quadrilateral nets with arbitrary topology. In *Proceedings of Eurographics'96*, pages 409–420, 1996. Computer Graphics Forum.

[Kob00] Leif Kobbelt. $\sqrt{3}$-subdivision. In *SIGGRAPH '00 : Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 103–112, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[Kor97] Jacob Kornerup. Parlists – a generalization of powerlists. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Euro-Par'97 Parallel Processing, Third International Euro-Par Conference*, number 1300 in LNCS, pages 614–618, Passau, Germany, August 1997. Springer Verlag.

[Kov01] Vladimir Kovalevsky. Algorithms and data structures for computer topology. pages 38–58, 2001.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1978.

[Kre86] Wolfgang Kreutzer. *System Simulation Programming Styles and Languages*. Addison-Wesley Publishing Co., Reading, Mass., 1986.

[KS00] Kiriakos N. Kutulakos and Steven M. Seitz. A theory of shape by space carving. *International Journal of Computer Vision*, 38(3) :199–218, July 2000.

[LAB00] Franck Ledoux, Agnès Arnould, Pascale Le Gall, and Yves Bertrand. A high-level operation in 3D modeling : A CASL case study. Technical Report 52, Lami, Université d'Évry-Val d'Essonne, Évry, 2000.

[LAGB01] Franck Ledoux, Agnès Arnould, Pascale Le Gall, and Yves Bertrand. Geometric modeling with CASL. In M. Cerioli and G. Reggio, editors, *Recent Trends in Algebraic Development Techniques, 15th International Workshop, WADT 2001, Joint with the CoFI WG Meeting, Genova, Italy, 2001, Selected Papers*, LNCS Vol. 2267, pages 176–200. Springer, 2001.

[LC94] Björn Lisper and Jean-François Collard. Extent analysis of data fields. *Lecture Notes in Computer Science*, 864 :208+, 1994.

[LDG+04] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system, Documentation and user's manual*, release 3.09 edition, 2004. `http://caml.inria.fr/pub/docs/manual-ocaml/index.html`.

[LDVS05] Caroline Lemerle, Barbara Di Ventura, and Luis Serrano. Space as the final frontier in stochastic simulations of biological systems. *FEBS Letters*, 579 :1789–1794, 2005.

[Led02] Franck Ledoux. *Étude et spécifications formelles de l'arrondi d'objets géométriques*. PhD thesis, Université d'Évry, 2002.

[Leo04] Melvin Leok. *Foundations of computational geometric mechanics*. PhD thesis, California Institute of Technology, 2004.

[Lév99] Bruno Lévy. *Topologie algorithmique : combinatoire et plongement*. Thèse d'université, INPL, Oct 1999. Prix SPECIF 2000.

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

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

[Lis93] Björn 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.

[LJ92] Aristid Lindenmayer and Hans 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.

[LM99] Bruno Levy and Jean-Laurent Mallet. Cellular modelling in arbitrary dimension using generalized maps. Technical report, ISA-GOCAD (Inria-Lorraine/CNRS, 1999.

[LMA+00] Franck Ledoux, Jean-Marc Mota, Agnès Arnould, Catherine Dubois, Pascale Le Gall, and Yves Bertrand. Formal specification for a mathematics-based application domain : Geometric modeling. Technical Report 51, LaMI, Université d'Évry-Val d'Essonne, Évry, 2000.

[Loo87] Charles Loop. Smooth subdivision surfaces based on triangles. Master's thesis, University of Utah, 1987.

[Mad03] Mutyam Madhu. Probabilistic rewriting P Systems. *International Journal of Foundations of Computer Science*, 14(1) :157–166, February 2003.

[Mal90] Grant Malcolm. Data structure and program transformation. *Science of computer programming*, 14 :255–279, August 1990.

[Man01] Vincenzo Manca. Logical string rewriting. *Theoretical Computer Science*, 264(1) :25–51, 2001.

[Mar98] Norman Margolus. Crystalline computing. In Kathleen P. Hirons, Manuel Vigil, and Ralph Carlson, editors, *Proc. Conf. on High Speed Computing LANL * LLNL*, pages 249–255, Gleneden Beach, OR, April 1998. LANL. LA-13474-C Conference, UC-705.

[MBFG04] Olivier Michel, Jean-Pierre Banâtre, Pascal Fradet, and Jean-Louis Giavitto. The Unconventional Programming Paradigms home page (UPP04). `http://upp.lami.univ-evry.fr`, 2004. International workshop for "Challenges, Visions and Research Issues for New Programming Paradigms", 15 - 17 September 2004, Mont Saint-Michel, France.

[Mei82] H. Meinhardt. *Models of biological pattern formation*. Academic Press, New York, 1982.

[Mes06a] Denis Mestivier. Agent-based approaches for the modeling and simulation of biological systems : application to the λ phage. Notes to the spring school "Modélisation de systèmes biologiques complexes dans le contexte de la génomique" on various modelization of the λ phage, Bordeaux, april 2006.

[Mes06b] Denis Mestivier. Modélisation déterministe de réseaux de gènes : le répresseur λ du bactériophage λ. Notes to the spring school "Modélisation de systèmes biologiques complexes dans le contexte de la génomique" on various modelization of the λ phage, Bordeaux, april 2006.

[MFP91] Erik Meijer, Marteen Fokkinga, and Ross Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *5th ACM Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144, Cambridge, MA, August 26–30, 1991. Springer, Berlin.

[MGC02] Olivier Michel, Jean-Louis Giavitto, and Julien Cohen. MGS : transformer des collections complexes pour la simulation en biologie. In L. Rideau, editor, *Journées Francophones des Langages Applicatifs (JFLA02)*, Anglet (France), January 2002. INRIA.

[Mic96] Olivier Michel. *Représentations dynamiques de l'espace dans un langage déclaratif de simulation.* PhD thesis, Université de Paris-Sud, centre d'Orsay, 1996.

[Mis94] Jayadev Misra. Powerlist : a structure for parallel recursion. *ACM Trans. on Prog. Languages and Systems*, 16(6) :1737–1767, November 1994.

[MJ05] Olivier Michel and Florent Jacquemard. *An Analysis of a Public-Key Protocol with Membranes*, pages 281–300. Natural Computing Series. Springer Verlag, 2005.

[MJG04] Olivier Michel, Florent Jacquemard, and Jean-Louis Giavitto. Three variations on the analysis of the needham-schroeder public-key protocol with MGS. Technical Report LaMI-98-2004, LaMI – Université d'Évry - CNRS, May 2004. 25 p.

[MLB71] Saunders Mac Lane and Garrett Birkhoff. *Algebre – Tome II.* Gauthier-Villard, 1971.

[MMVP99] Vincenzo Manca, Carlos Martin-Vide, and Gheorge Păun. New computing paradigms suggested by dna computing : computing by carving. *Biosystems*, 52(1-3) :47–54, October 1999.

[Mon00] David Monniaux. Abstract interpretation of probabilistic semantics. In *Seventh International Static Analysis Symposium (SAS'00)*, number 1824 in Lecture Notes in Computer Science, pages 322–339. Springer Verlag, 2000. Extended version on the author's web site.

[Mor04] Claude Morlet. *Topologie Algébrique*, chapter Mathématiques. Editions Universalis, 2004.

[MSD04] Tan Chee Meng, Sandeep Somani, and Pawan Dhar. Modeling and simulation of biological systems with stochasticity. *In Silico Biology*, 4, 2004.

[MSR91] Eric Mjolsness, David H. Sharp, and John Reinitz. A connectionist model of development. *Journal of Theoretical Biology*, 152(4) :429–454, 1991.

[Mun84] James Munkres. *Elements of Algebraic Topology.* Addison-Wesley, 1984.

[Nag01] Radhika Nagpal. *Programmable Self-Assembly : Constructing Global Shape using Biologically-inspired Local Interactions and Origami Mathematics.* PhD thesis, Massachusetts Institute of Technology, 2001.

[Nis80] Taishin Nishida. K0L-Systems simulating almost but not exactly the same development — the case of Japanese cypress. *Memoirs Fac. Sci., Kyoto University, Ser. Bio*, 8 :97–122, 1980.

[OBSC00] Atsuyuki Okabe, Barry Boots, Kokichi Sugihara, and Sung Nok Chiu. *Spatial Tessellations — Concepts and Applications of Voronoi Diagrams.* Wiley, Chichester, second edition, 2000.

[Obt03] Adam Obtułowicz. Probabilistic P Systems. In Gheorghe Păun, Grzegorz Rozenberg, Arto Salomaa, and Claudio Zandron, editors, *Membrane Computing : International Workshop, WMC' 02*, volume 2597 of *Lecture Notes in Computer Science*, pages 377–387. Springer, Berlin, July 2003.

[OOAB81] G.-M. Odell, George Oster, P. Alberch, and B. Burnside. The mechanical basis of morphogenesis. i. epithelial folding and invagination. *Developemental Biology*, 85(2) :446–462, 1981.

[Pal94] Richard Palmer. The Chains algebraic topological programming languages. 26 transparents disponibles à http ://www.cs.cornell.edu/Simlab/slides/chains/chains.html, February 1994. Department of Computer Science, Cornell University, located in the SymLab project web pages.

[Pat94] Ray Paton, editor. *Computing With Biological Metaphors.* Chapman & Hall, 1994.

[Pău00] Gheorghe Păun. Computing with membranes. *Journal of Computer and System Sciences*, 1(61) :108–143, 2000.

[Pău01] Gheorghe Păun. From cells to computers : computing with membranes (P systems). *Biosystems*, 59(3) :139–158, March 2001.

[PBMZ06] Dario Pescini, Daniela Besozzi, Giancarlo Mauri, and Claudio Zandron. Dynamical probabilistic P Systems. *International Journal of Foundations of Computer Science*, 17(1) :183–204, February 2006.

[PH89] Przemyslaw Prusinkiewicz and Jim Hanan. *Lindenmayer Systems, Fractals, and Plants.* Springer, Berlin, 1989.

[PH92] Przemyslaw Prusinkiewicz and Jim Hanan. L systems : from formalism to programming languages. In G. Ronzenberg and A. Salomaa, editors, *Lindenmayer Systems, Impacts on Theoretical Computer Science, Computer Graphics and Developmental Biology*, pages 193–211. Springer Verlag, February 1992.

[PLH+90] Przemyslaw Prusinkiewicz, Aristid Lindenmayer, Jim Hanan, et al. *The Algorithmic Beauty of Plants.* Springer-Verlag, 1990.

[PPT05] Park, Pfenning, and Thrun. A probabilistic language based upon sampling functions. *SPNOTICES : ACM SIGPLAN Notices*, 40, 2005.

[Pru87] Przemyslaw Prusinkiewicz. Applications of L-systems to computer imagery. In H. Ehrig, M. Nagl, A. Rosenfeld, and G. Rozenberg, editors, *Graph grammars and their application to computer science ; Third International Workshop*, pages 534–548. Springer, Berlin, 1987. Lecture Notes in Computer Science 291.

[Pru99] Przemyslaw Prusinkiewicz. A look at the visual modeling of plants using L-Systems. *Agronomie*, 19 :211–224, 1999.

[PS93] Richard S. Palmer and Vadim Shapiro. Chain models of physical behavior for engineering analysis and design. *Research in Engineering Design*, 5 :161–184, 1993. Springer International.

[PSSK03] Przemyslaw Prusinkiewicz, Faramarz Samavati, Colin Smith, and Radoslaw Karwowski. L-system description of subdivision curves. *International Journal of Shape Modeling*, 9(1) :41–59, 2003.

[Pta92] Mark Ptashne. *A genetic switch : phage lambda and higher organisms.* 1992.

[Rau03] Erik Rauch. Discrete, amorphous physical models. *International Journal of Theoretical Physics*, 42(2) :329–348, Februray 2003.

[Rei05] Cliff Reiter. A local cellular model for snow crystal growth. *Chaos, Solitons & Fractals*, 2005.

[Ric06] Adrien Richard. *Modèle formel pour les réseaux de régulation génétique & Influence des circuits de rétroaction.* PhD thesis, Université d'Évry, 2006.

[Ros89] Sheldon M. Ross. *Introduction to Probability Models.* Academic Press, fourth edition, 1989.

[RPW04] Paul W. K. Rothemund, Nick Papadakis, and Erik Winfree. Algorithmic self-assembly of dna sierpinski triangles. *PLoS Biol*, 2(12) :e424, 2004. www.plosbiology.org.

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

[RS97] Fritz Reinhardt and Heinrich Soeder. *Atlas des Mathématiques*, chapter Topologie algébrique. La pochotèque, 1997.

[Sha01] Vadim Shapiro. Solid modeling. Technical Report SAL 2001-2, University of Wisconsin, October 2001.

[SM04a] Antoine Spicher and Olivier Michel. Integration and pattern-matching of topological structures in a functional language. In *International Workshop on Implementation and Application of Functional Languages (IFL04)*, Lübeck, September 2004. draft proceedings published as a technical report of the Institute of Computer Science of the University of Kiel.

[SM04b] Antoine Spicher and Olivier Michel. Manipulations de structures topologiques dans un langage déclaratif pour la simulation. In *11ème Journées du GT "Animation et Simulation" (GTAS'2004)*, Reims, juin 2004. AFIG et LERI, Université de Reims.

[SM05a] Antoine Spicher and Olivier Michel. Declarative modeling of a neurulation-like process. *BioSystems*, 2005.

[SM05b] Antoine Spicher and Olivier Michel. Using rewriting techniques in the simulation of dynamical systems : Application to the modeling of sperm crawling. In *Fifth International Conference on Computational Science (ICCS'05), part I*, volume 3514 of *LNCS*, pages 820–827, Atlanta, GA, USA, May 2005. Springer.

[SM06] Antoine Spicher and Olivier Michel. Stratégie d'application stochastique de règles de réécriture dans le langage MGS. In *Journées Francophones des Langages Applicatifs (JFLA'2006)*, pages 147–163. INRIA, 2006.

[SMC+06] Antoine Spicher, Olivier Michel, Mikolaj Cieslak, Jean-Louis Giavitto, and Przemyslaw Prusinkiewicz. Stochastic P Systems and the simulation of biochemical processes with dynamic compartments. Submitted for *BioSystems* special issue : Membrane Computing - Applications in Systems Biology, 2006.

[SMG04a] Antoine Spicher, Olivier Michel, and Jean-Louis Giavitto. A topological framework for the specification and the simulation of discrete dynamical systems. In *Sixth International conference on Cellular Automata for Research and Industry (ACRI'04)*, volume 3305 of *LNCS*, Amsterdam, October 2004. Springer.

[SMG04b] Antoine Spicher, Olivier Michel, and Jean-Louis Giavitto. A topological framework for the specification and the simulation of discrete dynamical systems. Technical Report LaMI-99-2004, LaMI, May 2004.

[SMG05] Antoine Spicher, Olivier Michel, and Jean-Louis Giavitto. Algorithmic self-assembly by accretion and by carving in MGS. In *Proc. of the 7th International Conference on Artificial Evolution (EA'05)*, LNCS, University of Lille - France, October 2005. Springer-Verlag. (to be published).

[Spi03] Antoine Spicher. Typage et compilation de filtrage de chemins dans des collections topologiques. Master's thesis, Université d'Évry, 2003.

[Spi05] Antoine Spicher. Représentation et manipulation de structures topologiques dans un langage fonctionnel. In O. Michel, editor, *Journées Francophones des Langages Applicatifs (JFLA'2005)*, pages 113–128. INRIA, 2005.

[SPS04] Colin Smith, Przemyslaw Prusinkiewicz, and Faramarz Samavati. Local specification of surface subdivision algorithms. In Boris Böhlen John L. Pfaltz, Manfred Nagl, editor, *Applications of Graph Transformations with Industrial Relevance (Second International Workshop, AGTIVE 2003, Charlottesville, VA, USA, September 27 - October 1, 2003 Revised Selected and Invited Papers)*, volume 3062 of *LNCS*, pages 313–327. Springer-Verlag, Heidelberg, 2004.

[Ste08] Ernst Steinitz. Beitraege zur analysis situs. *Sitzungsbericht Berliner Mathematischer Gesellschaft*, 7 :29–49, 1908.

[Ste95] Ian Stewart. Four encounters with sierpinski's gasket. *Mathematical Intelligencer*, 17 :52–64, 1995.

[Str86] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[TAT05] Kouichi Takahashi, Satya Nanda Vel Arjunan, and Masaru Tomita. Space in systems biology of signaling pathways–towards intracellular molecular crowding in silico. *FEBS Letters*, 579 :1783–1788, 2005.

[Ter94] Olivier Terraz. *Programmation de métamorphoses d'objets surfaciques et volumiques*. PhD thesis, Université de Strasbourg 1, 1994.

[Tho03] Fabien Thonnérieux. Réalisation d'une interface graphique pour le traitement des sorties du progamme MGS. Rapport de stage 2^{e} année - IIE, 2003.

[TK01] René Thomas and Morten Kaufman. Multistationarity, the basis of cell differentiation and memory. i. structural conditions of multistationarity and other nontrivial behavior. *Chaos*, 11(1) :170–179, 2001.

[TN87] Tommaso Toffoli and Margolus Norman. *Cellular Automata Machine*. MIT Press, Cambridge MA, 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.

[Ton75] Enzo Tonti. On the formal structure of physical theories. Technical report, CNR – Instituto di Matematica del Politecnico di Milano, Piazza Leonardo da Vinci, 20133 Milano, 1975. 252 pp.

[Ton76] Enzo Tonti. The reason for analogies between physical theories. *Appl. Math. Modelling*, 1 :37–50, June 1976.

[Ton01] Enzo Tonti. A direct discrete formulation of field laws : The cell method. *Computer Modeling in Engineering & Sciences*, 2(2) :237–258, 2001.

[TTNT97] Atsuko Takamatsu, Kengo Takahashi, Makoto Nagao, and Yoshimi Tsuchiya. Frequency coupling model for dynamics of responses to stimuli in plasmodium of physarum polycephalum. *J. Phys. Soc. Jpn.*, 66 :1638–1646, 1997.

[Var79] Fransisco J. Varela. *Principle of Biological Autonomy*. McGraw-Hill/Appleton & Lange, 1979.

[VG98] Erika Valencia and Jean-Louis Giavitto. Algebraic topology for knowledge representation in analogy solving. In *European Conference on Artificial Intelligence (ECAI'98)*, pages 23–28, Brighton, August 1998.

[vN66] John von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, 1966.

[Wei85] Kevin Weiler. Edge-based data structures for solid modeling in curved-surface environments. *IEEE Computer Graphics and Applications*, 5(1) :21–40, 1985.

[Whi49] John H. C. Whitehead. Combinatorial homotopy ii. *Bull. Am. Math. Soc.*, (55) :453–496, 1949.

[WK04] Thomas Williams and Colin Kelley. *Gnuplot, an Interactive Plotting Program*, release 4.0 edition, 2004. `http://www.gnuplot.info/documentation.html`.

[Yok80] Takashi Yokomori. Stochastic characterizations of EOL languages. *Information and Control*, 45(1) :26–33, April 1980.

[Zor00] Denis Zorin. Subdivision zoo. In *Subdivision for modeling and animation*, pages 65–104. Schröder, Peter and Zorin, Denis, 2000.

[ZT00] Olgierd C. Zienkiewicz and Robert L. Taylor. *The Finite Element Method : the Basis*, volume 1. Fifth Edition, 2000.

Table des matières du document