Intégration des collections topologiques et des transformations dans un langage fonctionnel

Julien Cohen

N° d'identification : 2004EVRY0024

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

Julien Cohen

Sujet : **Intégration des collections topologiques et des transformations dans un langage fonctionnel**

Soutenue le 16 Décembre 2004 devant le jury composé de :

MM.	Bernard Lorho	*Président du jury*
	Christian Queinnec	*Rapporteur*
	Jean-Pierre Banâtre	*Rapporteur*
	Pierre-Étienne Moreau	*Examinateur*
	Bernard Paul Serpette	*Examinateur*
	Olivier Michel	*Encadrant*
	Jean-Louis Giavitto	*Directeur de Thèse*

Remerciements

À Évry...

Mes premiers remerciements vont à **Olivier Michel** et **Jean-Louis Giavitto**. J'ai pu grâce à eux travailler trois années sur un sujet excitant et dans un cadre amical. **Olivier** a également été un enseignant à part pour moi, qui a su me communiquer son goût de l'informatique.

Le cadre amical de ma thèse est aussi dû aux gens du LaMI. Je remercie **Pascale Le Gall** et **Catherine Dubois** qui m'ont permis de mettre un pied au LaMI dès la fin de ma Maîtrise. Je remercie également **Mathieu Jaume** et **Sophie Coudert** avec qui j'ai travaillé jusqu'au commencement de cette thèse.

Mes collègues de bureau ont largement participé à la bonne humeur quotidienne dans mon travail : **Christian Destré**, **Karim Berkani** et **Fabrice Barbier**. Je leur souhaite à tous les trois une belle carrière d'enseignant-chercheur.

Enfin, je remercie **Antoine Spicher** pour ses échanges de points de vue sur mon travail et **Christophe Kodrnja** pour sa participation au développement du visualisateur de simulations 2D.

Ailleurs...

Je remercie particulièrement les gens qui ont trouvé le temps de s'intéresser à mon travail et qui m'ont orienté dans mes choix : **Jacques Garrigue**, **Pascal Fradet**, **Alain Frisch**, **Giuseppe Castagna**, **François Pottier** et **Olivier Danvy**, par ordre chronologique.

Je remercie les gens de l'équipe Alchemy à Orsay de l'accueil qu'ils m'ont fait alors que je terminais la rédaction de ce manuscrit. Je pense en particulier à **Olivier Temam**, **Frédéric Gruau**, **Christine Eisenbeis** et **Albert Cohen**.

Je suis très reconnaissant envers la communauté des logiciels libres. Je remercie particulièrement toutes les personnes ayant participé au développement des logiciels suivants, tous intensément utilisés pour élaborer cette thèse : Linux, Latex, OCaml, Mozilla, Active DVI, Tuareg et TOM.

Enfin, je remercie les membres de mon jury : **Jean-Pierre Banâtre** et **Christian Queinnec** qui ont accepté l'importante mission d'être rapporteurs ; **Bernard Paul Serpette** qui a accepté d'être examinateur bien qu'il soit à l'autre bout de la France ; **Bernard Lorho** qui a accepté de trouver un moment dans son emploi du temps ministériel pour être examinateur de ma thèse et enfin **Pierre-Étienne Moreau** pour avoir accepté d'être examinateur, mais également pour les échanges stimulants que nous avons eus à propos de filtrage.

Entourage...

Bien évidemment, je remercie mes proches qui m'ont soutenu durant ces trois années.

Introduction

Les travaux de recherche effectués dans cette thèse s'inscrivent dans le cadre du projet MGS. Ce projet poursuit deux objectifs complémentaires : l'étude et le développement de l'apport de notions de nature topologique dans les langages de programmation, d'une part, et l'application de ces notions à la conception de nouvelles structures de données et de contrôle à la fois expressives et efficaces pour la simulation de systèmes dynamiques à structure dynamique, d'autre part. Ces études se concrétisent par le développement d'un langage de programmation expérimental et son application à la modélisation et à la simulation de systèmes dynamiques, en particulier dans le domaine de la biologie et de la morphogénèse. Ce langage est lui aussi nommé MGS.

Dans ce cadre, notre travail a consisté à étudier et développer les notions de *collection topologique* 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 sont initialement motivées par les problèmes particuliers posés par la simulation des systèmes dynamiques à structure dynamiques en biologie. Mais ces notions, motivées par un besoin particulier, peuvent s'intégrer dans un langage fonctionnel classique (comme ML) où elles ouvrent d'intéressantes perspectives qui justifient à elles seules leur étude : elles offrent un *cadre uniforme* pour spécifier et manipuler des structures de données qui étendent la notion de type algébrique ; elles permettent d'étendre la *définition par cas* de fonctions à tous les types de données ; elles offrent un cadre alternatif à la notion de *polytypisme* qui n'est pas restreint aux types de données algébriques.

Nos travaux se sont développés autour de trois axes :

1. Le premier axe de recherche concerne le développement de l'*interprète* du langage et a pour objectif à la fois le prototypage rapide de celui-ci et l'évaluation efficace des programmes MGS. Ces travaux ont mené à la définition d'un algorithme générique de filtrage fonctionnant sur toutes les collections topologiques et à un schéma d'évaluation fondé sur la traduction au vol des expressions du langage (en particulier des fonctions) en expressions du langage hôte (dans lequel est développé l'interprète).
2. Le second axe de recherche concerne nos travaux permettant à terme l'élaboration d'un *compilateur* pour MGS. Il s'agit principalement de la définition d'un système de typage (avec sous-typage) permettant d'inférer le type des transformations vues comme des fonctions *polytypiques* dont les arguments sont des collections hétérogènes. Les informations de types ainsi disponibles doivent permettre de guider le processus de compilation. Un générateur de code, encore préliminaire, nous permet de tester différentes stratégies de compilation.
3. Enfin, nous avons validé par de nombreux exemples significatifs la pertinence des choix effectués par rapport aux domaines d'application. Ces exemples mettent en évidence le

gain en expressivité apporté par l'approche topologique. Les notions de collection topologique et de transformation permettent la spécification de modèles exécutables de systèmes dynamiques de manière très concise et dans un cadre déclaratif.

Ce document se structure en quatre parties :

Première partie. La première partie expose les motivations de nos travaux et décrit le langage MGS. Les notions de collection topologique et de transformation et leur intégration dans un langage fonctionnel sont présentées de manière informelle. Nous détaillons particulièrement la notion de GBF qui permet de généraliser la notion de tableau.

De nombreux exemples de programmes MGS sont donnés et mettent en évidence l'adéquation du langage à manipuler de façon uniforme des structures de données aussi différentes que les tableaux, les multi-ensembles ou les graphes de Delaunay.

Seconde partie. La seconde partie de ce document définit et étudie les notions nécessaires à la mise en œuvre d'un interprète efficace pour le langage MGS. Nous y étudions dans un premier temps le filtrage uniforme de MGS : après avoir défini les notions essentielles de chemin et de motif, nous proposons un algorithme générique de filtrage fonctionnant sur toutes les collections topologiques.

Dans un second temps, nous proposons un mécanisme original de traduction au vol des expressions fonctionnelles du langage en terme de combinateurs implémentés dans le langage hôte, suivant une approche de type *syntaxe d'ordre supérieur*. À notre connaissance, représenter une fonction du langage source par une fonction du langage hôte pour l'évaluation des expressions dans un interprète est nouveau, d'autant plus que nous prenons en compte des traits impératifs. Nous proposons également dans cette partie des optimisations possibles à l'approche que nous proposons.

Troisième partie. La troisième partie de ce document est consacrée au typage. Nous proposons un système de types original, fondé sur une notion de typage ensembliste et doté d'une inférence automatique des types des collections et des transformations. Ce système nous permet de prendre en compte des fonctions polytypiques en étendant naturellement le polymorphisme paramétrique disponible dans les langages fonctionnels. Ce système permet également de dénoter des collections hétérogènes, c'est-à-dire des collections dont les éléments sont de types différents.

Nous proposons également deux spécialisations de notre système de types. La première est une simplification du système pouvant être utilisée dans un contexte où les collections sont homogènes et qui permet une inférence automatique simple (*à la* Damas/Milner). La seconde spécialisation est une tentative d'ajouter des informations concernant la taille des collections dans les types. Ce système permet de détecter certaines erreurs qui échappent au système initial.

Quatrième partie. La quatrième partie conclue le document. Nous évoquons les études préliminaires que nous avons entreprises afin de développer un compilateur. Notre idée est d'utiliser les informations issues de l'inférence de types pour produire un code spécialisé plus efficace. Afin de tester cette approche, nous avons débuté le développement d'un générateur de code mais les éléments rapportés ici sont encore préliminaires. Le dernier chapitre résume nos travaux et discute des nombreuses perspectives ouvertes par cette recherche.

Table des matières

Liste des exemples

Table des figures

Première partie

Le langage MGS

Chapitre I

Motivations et contributions

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.

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

Dans ce cadre, notre travail a consisté à étudier et développer les notions de *collection topologique* 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. Nos travaux se sont développés autour de trois axes :

1. Le premier axe de recherche concerne le développement de l'*interprète* du langage et a pour objectif à la fois le prototypage rapide de celui-ci et l'évaluation efficace des programmes MGS. Ces travaux ont mené à la définition d'un algorithme générique de filtrage fonctionnant sur toutes les collections topologiques et à un schéma d'évaluation fondé sur la traduction au vol des expressions du langages (en particulier des fonctions) en expressions du langage hôte (dans lequel est développé l'interprète).
2. Le second axe de recherche concerne nos travaux permettant à terme l'élaboration d'un *compilateur* pour MGS. Il s'agit principalement de la définition d'un système de typage (avec sous-typage) permettant d'inférer le type des transformations vues comme des fonctions *polytypiques* dont les arguments sont des collections hétérogènes. Les informations de types ainsi disponibles doivent permettre de guider le processus de compilation. Un générateur de code, encore préliminaire, nous permet de tester différentes stratégies de compilation.
3. Enfin, nous avons validé par de nombreux exemples significatifs la pertinence des choix effectués par rapport aux domaines d'application. Ces exemples mettent en évidence le gain en expressivité apporté par l'approche topologique. Les notions de collection topologique et de transformation permettent la spécification de modèles exécutables de systèmes dynamiques de manière très concise et dans un cadre déclaratif.

Les notions de collection topologique et de transformation sont initialement motivées par les problèmes particuliers posés par la simulation des systèmes dynamiques à structure dynamiques en biologie. Mais ces notions, motivées par un besoin particulier, peuvent s'intégrer dans un langage fonctionnel classique (comme ML) où elles ouvrent d'intéressantes perspectives qui justifient à elles seules leur étude :

- ces notions offrent un *cadre uniforme* pour spécifier et manipuler des structures de données qui étendent la notion de type algébrique ;
- elles permettent d'étendre la *définition par cas*, un mécanisme puissant de spécification de fonction, à tous les types de données (par exemple aux tableaux) ;
- elles offrent un cadre alternatif à la notion de *polytypisme* qui n'est pas restreint aux types de données algébriques ;
- elles apportent une *nouvelle notion de réécriture* qui ne se réduit pas aux approches existantes.

Plan du chapitre

Dans la suite de ce chapitre, nous présentons succinctement les collections topologiques et les transformations permettant de les manipuler, puis nous détaillons les points évoquées ci-dessus (section I.1).

Nous renvoyons à la fin du chapitre (section I.4) l'analyse des problèmes posés par la simulation des systèmes dynamiques à structure dynamiques en biologie, analyse qui a initialement motivé le développement des notions de collection topologique et de transformation. En effet, bien que ces considérations soient à l'origine de ces notions, elles sont orthogonales au reste de notre travail.

Mais avant de clore le chapitre par cette analyse, nous détaillerons l'organisation de ce document (section I.2) puis nos contributions (section I.3).

I.1 Notion de collection topologique et de transformation

Collections topologiques

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

Nous appelons *collection topologique* une collection dont les éléments sont organisés par une relation dite de voisinage. Nous notons « $,$ » la relation de voisinage : x,y signifie que l'élément y est un voisin de l'élément x (dans la collection considérée). 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} pour tout $i \leq n-1$.

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 ou un multi-ensemble peuvent aussi être vus comme une collection topologique dont tout élément est voisin de tous les autres éléments (cette relation de voisinage peut sembler arbitraire mais elle est justifiée par les opérations habituelles sur ces structures de données).

Dans tout ce travail, il est commode de visualiser une collection topologique par un graphe : 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 simplicial* développée en topologie algébrique. Les graphes sont un cas particulier de complexe simplicial. Ce cadre théorique abstrait est esquissé à la section I.4.3 mais ne sera pas utile pour le travail mené ici.

Transformations

Une *transformation* est un mécanisme de calcul qui permet d'associer une collection topologique à une autre. L'application d'une transformation à une collection C consiste à :

1. sélectionner un ensemble $\{A_i\}$ de sous-collections de C,
2. pour chaque sous-collection A_i sélectionnée, calculer une nouvelle sous-collection A'_i à partir de A_i (et éventuellement de son voisinage),
3. substituer les sous-collections calculées aux sous-collections sélectionnées.

Pour spécifier une transformation il faut préciser comment sélectionner les sous-collections A_i, comment sont calculées les nouvelles sous-collections A'_i et enfin les contraintes de la substitution des A_i par les A'_i. Le premier et le troisième point dépendent de la relation de voisinage définie sur C.

Application à un langage de programmation fonctionnel : le langage MGS

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 A' correspond à la fonction h. L'expression a peut être vue comme un motif de filtrage.

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 et nous développons un compilateur pour une version typée. Les travaux de cette thèse visent à développer l'interprète et le compilateur.

Insistons sur le fait qu'une transformation est une fonction ordinaire qu'on applique et qu'on utilise exactement comme les fonctions définies par abstraction. Dans un langage fonctionnel permettant l'ordre supérieur, comme MGS, on peut passer une transformation en argument ou bien la retourner comme résultat d'une application.

En première approche, cette extension ne semble pas augmenter de manière significative les mécanismes de programmation déjà présent dans un langage comme ML. Cependant, l'utilisation de la relation de voisinage dans la définition de nouvelles structures de données et

dans la spécification du motif a ouvrent des perspectives réellement nouvelles en généralisant le mécanisme de définition de fonctions par cas, en étendant le polytypisme à des structures de données non-algébriques, en permettant le développement d'une notion nouvelle de réécriture et en unifiant plusieurs styles de programmation.

Extension du mécanisme de définition de fonction par cas. La définition des fonctions par cas en utilisant le filtrage est l'un des atouts des langages fonctionnels actuels. C'est un mécanisme de programmation puissant qui permet d'exprimer simplement et de manière concise des traitements complexes. Toutefois le filtrage n'est bien maîtrisé que sur des structures de données algébriques (définies généralement par un type somme).

Les tableaux (à plusieurs dimensions) constituent une très bonne illustration du problème. Les tableaux ne se manipulent pas aussi naturellement que les listes dans les langages fonctionnels en raison de leur nature algébrique différente. Il est notamment impossible de définir des fonctions par cas sur les tableaux dans un langage comme OCaml et ceci, bien que efforts pour permettre le filtrage sur les tableaux existent depuis [Bir77, Bak78] et plus récemment dans [WA85] et [Jeu92]. Mais la notion naturelle de sous-tableaux utilisée dans ces travaux ne permet pas de déconstruire de manière unique un tableau. Le mécanisme de filtrage que nous étudions dans le chapitre IV permet la sélection de motifs plus généraux que dans les travaux cités et s'intègre naturellement dans un langage fonctionnel classique.

La notion de transformation permet d'étendre la définition de fonction par cas sur un type algébrique à toutes les collections topologiques. Nous donnerons de nombreux exemples de transformation agissant sur des structures de données comme les tableaux et les graphes dans le chapitre III.

Polytypisme. Dans un langage fonctionnel typé, le polymorphisme paramétrique permet de spécifier des fonctions génériques qui peuvent s'appliquer à une collection de données indépendamment de la nature de ces données. Par exemple, on peut spécifier une fonction qui s'applique à une liste d'éléments indépendamment de la nature de ces éléments.

Un programmeur peut spécifier une fonction polymorphe f soit parce f s'exprime comme combinaison de fonctions polymorphes préexistantes (en particuliers de fonctions polymorphes prédéfinies dans le noyau du langage) soit parce que la définition de f tire parti du filtrage (f « n'inspecte pas » les valeurs filtrées et se contente de les recombiner).

Le *polytypisme* étend cette possibilité par la paramétrisation de la structure de la collection. Par exemple, le polytypisme permet de définir une fonction map_g qui remplace chaque élément x d'une collection par $g(x)$. Cette opération peut s'appliquer à toute collection quelque soit sa structure : liste, arbre, tableau.

L'approche classique du polytypisme, décrite à la section IV.1, consiste soit à générer automatiquement pour chaque définition de type algébrique un ensemble d'itérateurs comme *map*, *fold*, etc. et à gérer automatiquement la surcharge de ces itérateurs, soit à étendre le mécanisme de filtrage en permettant de paramétrer les constructeurs qui interviennent dans les motifs. Dans les deux cas, cette approche est restreinte aux types de données algébriques.

La notion de transformation permet de définir des fonctions polytypiques qui s'appliquent à toute collection topologique, principalement grâce à un langage de motifs où la nature de la relation de voisinage reste implicite. Ce polytypisme ne se fait pas au détriment du typage : la troisième partie de cette thèse développe une système d'inférence de type permettant de typer

statiquement des fonctions polytypiques sur des collections homogènes et, en admettant un peu de vérification dynamique de types à l'exécution, sur des collections hétérogènes.

Une nouvelle notion de réécriture. La définition d'une fonction par cas sur un type de données algébrique correspond à une fonction définie par des règles de réécriture simples sur un terme : les règles ne sont pas ambigües (même si les motifs ne sont pas exclusifs, l'ordre des cas détermine de manière unique la règle à appliquer) et une règle ne s'applique qu'à la racine du terme (cette limitation ne concerne pas des systèmes comme ELAN, plus orientés vers la réécriture que vers la programmation généraliste).

La notion de transformation se rapproche plus de la notion de réécriture, ou plus exactement de la notion de *système de réduction*. Un système de réduction abstrait est une paire $\langle \mathcal{T}, \rightarrow \rangle$ 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 $C \rightarrow T(C)$. La relation induite par une transformation est donc une relation fonctionnelle.

Un système de réduction abstrait permet d'abstraire plusieurs notions de réécriture. Nous avons indiqué plus haut qu'une collection topologique pouvait se représenter par un graphe. 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. Un exemple concernant les collections de Delaunay est donné à la section IV.5.1. Les graphes de Delaunay constituent un exemple de collection topologique où les arcs du graphe associé sont calculés en fonction des valeurs aux sommets et ne sont pas prédéfinis.

Les transformations permettent donc de définir de nouveaux mécanismes de réécriture. Ces mécanismes sont utiles pour la programmation et trouvent de nombreuses applications, comme le montre les exemples du chapitre III. L'étude de ces mécanismes du point de vue de la réécriture dépasse cependant le cadre de cette thèse.

Unification de plusieurs modèles de programmation. Les concepts présentés reposent sur des notions de topologie et unifient plusieurs modèles de calcul comme :

- la réécriture de multi-ensembles (Gamma [BM86], la machine chimique [BB92], les P-systèmes [PĎ00]),
- la réécriture des chaînes (les L-systèmes [RS92], Snobol [GH91], ELAN [MK98] pour les termes modulo associativité)
- et les automates cellulaires [VN66] (vus comme de la réécriture de tableaux).

Les multi-ensembles et les chaînes sont des structures de données dites monoïdales [Man01, GM01a] et les techniques de réécriture associées sont maintenant bien maîtrisées : il s'agit de considérer des constructeur modulo associativité (pour les chaîne) et associativité-commutativité (pour les multi-ensembles). Il n'est pas usuel de considérer les automates cellulaires comme de la réécriture de tableaux, parce que la réécriture ne concerne qu'une seule cellule (même si la partie droite d'une règle d'évolution dépend des voisins de cette cellule). Mais certaines variations de ce modèle, comme les gaz sur réseaux [TN87], spécifient bien la réécriture d'un ensemble de cellules voisines.

Il est encore trop tôt pour affirmer que le cadre théorique qui permet le développement des notions de collections topologiques et de transformations 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, une cadre uniforme qui permet d'intégrer

simplement ces différents paradigmes de programmation dans le même langage, et ce, par une extension conservative du modèle fonctionnel.

I.2 Organisation de cette thèse

Ce document se structure en quatre parties.

Première partie

La première partie décrit le langage MGS. Les notions de collection topologique et de transformation et leur intégration dans un langage fonctionnel sont présentées de manière informelle à travers des exemples dans le chapitre II.

La notion de GBF et les opérations sur les GBF sont présentées de manière plus détaillée dans ce même chapitre. En effet, les GBF permettent de définir de nouveaux types de collections topologiques. Ils illustrent l'approche topologique qui a été évoquée ci-dessus dans le cadre de la définition de nouvelles structures de données.

La première partie s'achève avec le chapitre III sur de nombreux exemples de programmes MGS. Ces exemples mettent en évidence l'adéquation du langage à manipuler de façon uniforme des structures de données aussi différentes que les tableaux, les multi-ensembles ou les graphes de Delaunay. 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 collections topologiques et de transformations sont intéressantes et utiles pour la programmation en général.

Seconde partie

La seconde partie de ce document définit et étudie les notions nécessaires à la mise en œuvre d'un interprète efficace pour le langage MGS.

Le chapitre IV débute cette partie par l'étude du filtrage dans MGS. Après avoir défini les notions essentielles de chemin et de motif, nous proposons un algorithme générique de filtrage fonctionnant sur toutes les collections topologiques. Un exemple d'application de l'algorithme sur les collections de type GBF est donné.

Le chapitre V poursuit par la proposition d'un mécanisme original de traduction au vol des expressions fonctionnelles du langage en terme de combinateurs implémentés dans le langage hôte, suivant une approche de type *syntaxe d'ordre supérieur*. L'idée de représenter une fonction dans un langage source par une fonction d'un langage hôte n'est pas nouvelle (elles remonte aux années soixante). Mais cette idée a surtout été utilisée dans des travaux théoriques (par exemple pour établir la sémantique d'un langage) et dans une moindre mesure, dans certaines approches de la compilation. À notre connaissance, la mise en œuvre de cette idée pour l'évaluation des expressions dans un interprète est nouvelle, d'autant plus que nous prenons en compte des traits impératifs.

L'introduction de ce chapitre détaille les avantages de ce schéma original d'évaluation en terme d'architecture logicielle extensible. Cette architecture nous a permis de développer très rapidement le prototype de l'interprète.

Le chapitre VI suivant propose des optimisations possibles au cadre général défini au chapitre V. Il est important de noter que tous les exemples présentés dans ce document ont été exécutés grâce à l'interprète décrit dans cette partie.

Troisième partie

La troisième partie de ce document est consacrée au typage. Si l'on souhaite aller au delà de la simple interprétation des programmes vers la compilation de MGS, il est nécessaire de définir un système de types permettant d'identifier les programmes corrects et de prédire le type de chaque valeur manipulée à l'exécution.

Nous proposons au chapitre VII un système de types original, fondé sur une notion de typage ensembliste et doté d'une inférence automatique des types des collections et des transformations. Le développement de ce système nous permet de prendre en compte des fonctions polytypiques en étendant naturellement le polymorphisme paramétrique disponible dans les langages fonctionnels. Ce système permet également de dénoter des collections hétérogènes, c'est-à-dire des collections dont les éléments sont de types différents à l'aide de types union et d'une relation de sous-typage ensembliste. Le jugement de la relation de sous-typage est réalisé par un algorithme de résolution de contraintes ensemblistes.

Le chapitre VIII propose deux spécialisations de notre système de types. La première est une simplification du système pouvant être utilisée dans un contexte où les collections sont homogènes et qui permet une inférence automatique simple (*à la* Damas/Milner). Ce système peut également être vu comme une simplification du système présenté au chapitre VII. La seconde spécialisation est une tentative d'ajouter des informations concernant la taille des collections dans les types. Ce système permet de détecter certaines erreurs qui échappent au système initial. Les contraintes sur les tailles sont vues comme des contraintes ensemblistes, ce qui permet de réutiliser l'algorithme d'inférence mis en place dans le système initial sans le complexifier.

Quatrième partie

La quatrième partie conclue le document. Le chapitre IX évoque les études préliminaires que nous avons entreprises afin de développer un compilateur. Notre idée est d'utiliser les informations issues de l'inférence de types pour produire un code spécialisé plus efficace. Afin de tester cette approche, nous avons débuté le développement d'un générateur de code mais les éléments rapportés ici sont encore préliminaires.

Enfin, le dernier chapitre résume nos travaux et discute des nombreuses perspectives ouvertes par cette recherche.

I.3 Contributions

Le travail réalisé au cours de cette thèse prend place dans le projet MGS de l'équipe SPÉCIF du LaMI. Dans ce cadre,

- j'ai participé à la formalisation et à l'implémentation des collections topologiques ;
- j'ai participé à l'implémentation et j'ai formalisé le schéma d'évaluation par combinatorisation au vol ;
- j'ai participé à la formalisation et à l'implémentation de l'algorithme générique de filtrage de chemins ;
- j'ai conçu et j'ai développé le système de typage des collections topologiques et des transformations ;
- j'ai conçu et j'ai développé un générateur de code permettant de compiler des expression MGS en code OCaml ;

- j'ai collaboré avec l'équipe de développement de TOM dirigée par Pierre-Étienne Moreau du Loria (projet PROTHEO) afin d'étendre cet outil au langage OCaml et de pouvoir l'utiliser dans mon générateur de code ;
- j'ai imaginé et j'ai développé de nombreux exemples pour tester et valider les constructions du langage et les développements logiciels effectués ;
- j'ai développé un outil de visualisation graphique permettant d'afficher les valeurs de GBF correspondant aux grilles NEWS et aux aux treillis hexagonaux ;
- enfin, j'ai réalisé de nombreuses comparaisons afin de sélectionner une technique d'implémentation parmi plusieurs variantes (choix de représentation des structures de données, comparaison de différents schéma d'évaluation, stratégies de résolution des contraintes de typage, etc.).

Ces travaux ont donné lieux à plusieurs publications dans un journal [GMC02a], des conférences internationales [Coh03b, Coh03c, GMCS04a], des conférences nationales [MGC02, CMG03, Coh04], et des rapports techniques [Coh03a, GMC02b].

I.4 Motivations initiales : la simulation des systèmes dynamiques à structure dynamique en biologie

Dans cette section nous revenons sur les motivations et les objectifs du projet MGS. Comme nous l'avons évoqué au début de ce chapitre, ces motivations 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.).

Si la prise en compte de ces relations topologiques dans un langage de programmation permet de simplifier la spécification des SD, le pari du projet MGS est que les mécanismes spécifiques correspondants seront aussi *utiles et intéressants dans un langage de programmation généraliste*. L'histoire de l'informatique recèle de tels exemples où les constructions motivées par un objectif particulier ont débouché sur des notions universelles intégrées aux langages généralistes. Le langage ML en constitue un très bon exemple : initialement dédié à l'expression des tactiques dans un prouveur de théorèmes, les langages de la famille ML et les mécanismes développés (par exemple l'inférence de type *à la* Hindley/Milner et la définition de fonction par cas) se sont révélés formidablement utiles dans tous les domaines d'application.

Dans la suite de cette section, nous allons décrire succinctement le domaine d'application privilégié du projet MGS et nous expliciterons les mécanismes nécessaires à ces applications. L'analyse conduite ici synthétise les éléments présentés dans les articles [GM01a, GM01b, GGMP02, MGC02, GM02, Gia03, GMCS04a, SMG04] et a motivé initialement le développement et l'étude des notions de collection topologique et de transformation.

Nous espérons que le travail présenté dans la suite de cette thèse convaincra le lecteur que ces notions, motivées par un domaine d'application particulier, présentent un intérêt et sont utiles dans un langage de programmation généraliste.

I.4.1 Un langage de simulation spécifique pour la biologie

Modélisation et simulation informatique pour la post-génomique. La biologie constitue à la fois un domaine d'application croissant (par exemple avec les outils pour le séquençage du génome) et une source d'inspiration pour l'informatique [Pat94] : réseaux de neurones, automate cellulaire, algorithmes génétiques, algorithmes évolutionnistes, vie artificielle, calcul par ADN, calcul aqueux, calcul chimique, calcul par membranes... On se reportera au volume [GBFM04] 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 ce domaine, la *post-génomique* pose de nouveaux problèmes [GGMP02]. La post-génomique vise, après l'achèvement du séquençage complet du génome de plusieurs organismes vivants, à découvrir le rôle des produits des gènes et leurs interactions dans les cellules.

Dans ce challenge, 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...) et de découvrir de nouveaux modes de régulation.

Mais les problèmes informatiques qui sont posés par ces nouvelles applications sont très difficiles et nécessitent de nouveaux concepts pour la représentation d'entités biologiques, leur construction modulaire, la construction incrémentale des programmes de simulation, leur validation et l'exploitation des résultats.

L'approche des langages dédiés. Ces difficultés conduisent au développement de langages de programmation dédiés et de nouvelles techniques d'analyse automatique. Le développement d'un langage spécifique à un domaine d'application, 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 meilleurs 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 post-génomique doit permettre l'exploitation de l'énorme masse de données produite par 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.) afin de rendre compte, de comprendre et de maîtriser les nombreuses fonctions biologiques et physiologiques.

On peut rapidement lister les problèmes qui se posent de manière particulièrement aigüe dans le domaine de la simulation en post-génomique :

- 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 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 distribuée.* Nous qualifions ce type de processus de *système dynamique à structure dynamique* ou, en abrégé, $(\text{SD})^2$, suivant la terminologie introduite dans [GM01b].

Ce type de système dynamique est extrêmement complexe à modéliser et à simuler et nécessite la conception et le développement de nouveaux modèles de programmation et les outils associés.

I.4.2 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, itération de fonctions, automates cellulaires, etc. 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'états, sont listés à la table 1.

	EDO	Itérations de fonctions	Automate à états finis
Temps	continu	discret	discret
État	continu	continu	discret

TAB. 1 – Quelques formalismes utilisé pour spécifier les SD suivant le caractère discret ou continu du temps et des variables d'état (repris de [GGMP02]).

De nombreux SD sont structurés, c'est-à-dire 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 vue comme le résultat des changements des états de chaque parties. 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'états du système complet. Dans ce contexte, nous disons que le SD exhibe une *structure statique* si :

1. l'état du système est décrit 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 (car 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).

Suivant l'analyse développée dans [GGMP02], 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, change au cours du temps. Ces systèmes sont des systèmes dynamiques à structure dynamique.

Un exemple immédiat de $(SD)^2$ est donné par le développement d'un embryon. Initialement, l'état du système est décrit par le seul état chimique o_0 de la cellule œuf (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[1]. Le nombre des cellules, leur organisation spatiale et leurs interactions évoluent constamment au cours du développement et ne peut être décrite par 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 doit être calculé *en même temps* que l'état courant du système. Autrement dit, l'espace des phases doit être *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 [ES79], *autopoïese* dans des travaux sur les systèmes autonomes [Var79], *variable structure system* dans la théorie du contrôle [Itk76, HGH93], *developmental grammar* dans [MSR91] ou encore *organisation* dans des travaux sur l'émergence de structures stables dans les systèmes ouverts [FB94a, FB94b, FB96].

Cependant, ce type de systèmes ne se rencontre pas uniquement en biologie : la modélisation de réseaux dynamiques (internet, réseaux mobiles), les phénomènes de morphogénèse en physique (croissance dans un médium dynamique), la mécanique des milieux élastiques ou des systèmes déformables regorgent d'exemples de $(SD)^2$.

I.4.3 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 notre analyse est la décomposition d'un système *en fonction de son évolution*. A un instant donné, nous découpons un système S en plusieurs sous-systèmes $S_1, S_2, \ldots, S_n$ disjoints tel que le prochain état $s_i(t+1)$ du sous-système S_i ne dépende que du précédent état $s_i(t)$. Autrement dit, chaque sous-système S_i évolue indépendamment entre un instant et l'instant 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 boite a besoin d'être connu.

[1] 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 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.

La partition de S en sous-systèmes 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 contrepied d'une « approche objet » qui commence par décrire les constituants structurels d'un système avant de définir leurs interactions ; ici ce sont les activités du système qui constituent notre point de départ 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 autonome 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, au 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 entre la fonction d'évolution « globale » h et les fonctions d'évolutions locales :

$$s(t+1) = h(s(t)) = \varphi^{t+1}(h_1^t(s_1(t)), \ldots, h_{n_t}^t(s_{n_t}(t)))$$

Si l'on suit cette analyse, la spécification d'un $(\text{SD})^2$ 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 $S', S'' \subseteq S_j^t$. Deux sous-systèmes S' et S'' sont *séparables* s'il existe un S_j^t tel que $S' \subseteq S_j^t$ et $S'' \cap S_j^t = \emptyset$ ou inversement. Cela nous amène à considérer l'ensemble $\mathcal{S}$ défini comme le plus petit ensemble clos par intersection contenant les S_j^t (voir la 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 n'inclue aucun autre élément de $\mathcal{S}$ correspondent aux parties élémentaires recherchées.

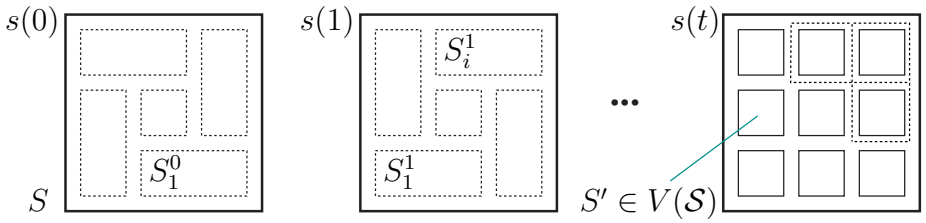


FIG. 1 – La structure d'interaction d'un système S résultant des sous-systèmes des éléments en interaction à un pas de temps donné.

L'ensemble $\mathcal{S}$ possède une *structure topologique* naturelle : $\mathcal{S}$ correspond à un *complexe simplicial abstrait*. Cette notion est développée en topologie algébrique et correspond à la description combinatoire d'un espace construit en « recollant » des espaces très simples (des boules dans

le langage de la topologie). 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 dimension de ses simplexes de plus grande dimension. 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 ensemble à un élément de $\mathcal{C}$.

La notion de complexe simplicial abstrait généralise l'idée de *graphe* : un simplexe de dimension 0 est un sommet du graphe, un simplexe de dimension 1 est un arc qui relie deux sommets, un simplexe f de dimension 2 peut être vu comme une surface dont les frontières sont les simplexes de dimensions 1 inclus dans f, etc. Deux simplexes sont *voisins* s'il partagent une face ou bien s'ils sont faces d'un simplexe de dimension plus grande.

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 face d'un autre élément, appelé simplexes maximaux, correspondent aux S_i^t. La dimension d'un simplexe maximal correspond au nombre (diminué de un) 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 connexion 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 $\mathcal{S}$. Cependant, très souvent, les lois de la physique n'expriment pas d'action instantanée à distance. Autrement dit, seules des entités « spatialement proches » interagissent. Il n'est donc pas étonnant que le voisinage logique décrit par $\mathcal{S}$ recoupe le voisinage spatial des parties physiques du système, comme on le verra dans de nombreux exemples du chapitre III.

I.4.4 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.

On « retrouve » les notions de collection topologique et de transformation que nous avons introduites arbitrairement : une collection topologique correspond à une structure $\mathcal{S}$ et une transformation 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. Cependant, les parties filtrées S_i^t elles 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.

I.4.5 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 des entités. Une sous-collection représente un sous-ensemble d'entités en interactions ainsi que des messages du système.

L'évolution du système est spécifiée par une transformation où, typiquement, la partie gauche d'une règle filtre des entités et des messages qui leur sont destinés, 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 nombreux exemples dans le chapitre III).

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.

Chapitre II

Présentation de MGS

La présentation du langage que nous faisons dans ce chapitre a été publiée partiellement dans [MGC02] et [GMC02b].

Nous présentons dans ce chapitre le langage MGS. MGS est un langage fonctionnel généraliste dont l'originalité est de proposer un point de vue topologique unifié sur les structures de données et ainsi de permettre de programmer par filtrage sur toutes les structures de données disponibles dans le langage.

En plus des collections topologiques et des transformations qui font, comme nous l'avons dit, l'originalité de ce langage, MGS est équipé de fonctionnalités caractérisant les langages fonctionnels actuels comme les traits impératifs, les arguments optionnels, la surcharge et le polymorphisme (dont une forme de polytypisme).

Plusieurs interprètes MGS ont été produits à ce jour (deux écrits en OCaml et un écrit en C++). Les développements du projet MGS sont actuellement concentrés sur l'un d'eux, nommé 1-MGS. Ce développement contribue au logiciel libre : les sources, les exécutables et la documentation sont distribués gratuitement à partir du site internet du projet[1]. L'interprète 1-MGS constitue une implémentation dynamiquement typée du langage. Ceci signifie que les types sont vérifiés lors de l'exécution du programme. Les types sont accessibles au programmeur : des tests de types permettent un contrôle de l'exécution, notamment dans le filtrage.

Notons qu'en tant que langage applicatif (ou fonctionnel), MGS opère par calcul de nouvelles valeurs et non pas par effet de bord, même si une notion de variable impérative est disponible dans le langage.

Ce chapitre est organisé comme suit. La première section présente les valeurs scalaires dans MGS et la notion de collection topologique. La section II.2 décrit le fonctionnement des transformations et la manière de les définir. La section II.3 présente les différents types de collections disponibles dans MGS. Enfin, la section II.4 décrit en détails l'un d'eux : les GBF, permettant de représenter des tableaux et plus généralement, des structures indexées.

[1] Accessible depuis `http://mgs.lami.univ-evry.fr`

II.1 Valeurs : scalaires et collections topologiques

Les scalaires

Hormis les types de valeurs scalaires usuels (entiers, flottants, booléens, caractères et chaînes de caractères), plusieurs autres types de données sont disponibles dans MGS et sont traités de manière atomique.

Symboles. L'utilisateur a accès à des valeurs symboliques de la forme `'Nom` où `Nom` est un identificateur quelconque. Le fonctionnement des valeurs symboliques est similaire à celui des constructeur de *variants* polymorphes d'arité nulle de OCaml).

Valeur indéfinie. La valeur indéfinie notée `<undef>` est utilisée dans plusieurs cas : accès à une variable impérative non-initialisée, accès à un élément non existant d'une structure de données, résultat d'un calcul non défini.

Fonctions. Les fonctions peuvent être arguments d'autres fonctions ou résultat d'une fonction (ordre supérieur). Les fonctions ne pouvant pas être décomposées au niveau du langage (un programme ne peut inspecter la définition d'une fonction), nous les classons dans les valeurs scalaires.

Autres. D'autres types de valeurs atomiques sont en cours d'intégration dans le langage. Il s'agit des *combinateurs*, qui sont des fonctions particulières utilisées dans l'implémentation de l'interprète, des *brins*, utilisés dans l'implémentation des G-cartes (il s'agit d'un type de structures de données) et enfin des *simplexes*, utilisés dans l'implémentation des complexes simpliciaux (un autre type de structures de données).

Collections topologiques

Nous venons de voir que les scalaires sont des valeurs qui ne peuvent pas être décomposées. La notion de collection s'oppose à celle de scalaire puisqu'il s'agit d'un agrégat de valeurs. Les collections correspondent à la notion de structure de données mais nous les considérons ici d'un point de vue topologique. En effet, les collections que nous considérons sont munies d'une relation de voisinage. Par exemple, dans une séquence, chaque élément a (au plus) un élément à sa droite et un élément à sa gauche. Dans un arbre binaire, chaque nœud a (au plus) un fils gauche, un fils droit et un père. Dans un ensemble (ou un multi-ensemble) on pourra considérer simplement que tout élément est voisin de tous les autres éléments. Pour souligner l'importance de la relation de voisinage dans les collections, nous les appelons des *collections topologiques*.

On peut voir les collections comme des fonctions d'un ensemble de positions vers un ensemble de valeurs. Toutefois ce point de vue n'est pas celui que nous adopterons car elles ne sont pas manipulables comme des fonctions dans le langage et elles sont implémentées extensionnellement (tous les éléments sont tabulés en mémoire et le nombre de valeurs est fini). Il est commode de visualiser une collection topologique par un graphe : les sommets du graphe correspondent aux positions de la collection et sont décorés par les valeurs, les arcs correspondent à la relation de voisinage.

La relation de voisinage permet de définir la notion de chemin : un *chemin* dans une collection c est une suite de positions $p_1, p_2, \ldots, p_n$ de c telles que pour tout i entre 1 et $n-1$, p_{i+1} est voisin de p_i (dans c).

Nous définissons également la notion de sous-collection : un sous ensemble P de positions d'une collection c dénote une *sous-collection* de c lorsque pour tout couple d'éléments $\langle p_1, p_2 \rangle$

de P, p_1 est en relation avec p_2 par la clôture réflexive, transitive et symétrique de la relation de voisinage de c (on dira que P est *connexe*).

Enregistrements

Les enregistrements extensibles sont souvent utilisés dans les programmes MGS pour décrire un élément d'une collection. Ils permettent de décrire des structures *a priori* sans organisation tout en permettant un accès intuitifs aux différentes composantes. Ainsi, un enregistrement est parfois vu comme une collection (dont les positions sont les noms des champs). Toutefois, pour des raisons pratiques, nous les considérerons souvent comme des scalaires, bien que n'étant pas atomiques par nature.

II.2 Transformations

II.2.1 Motifs, règles et transformations

Les collections topologiques peuvent être manipulées fonctionnellement par des fonctions définies par filtrage appelées des *transformations*. Une transformation est un ensemble de règles noté :

$$\texttt{trans}\ T = \{\ ...\quad m \Rightarrow f(m);\quad ...\ \}$$

où la partie gauche d'une règle est un motif et la partie droite est une fonction des éléments introduits dans le motif. L'application d'une transformation à une collection topologique opère de la façon suivante : étant donné une collection C à traiter,

1. des sous-collections de C sont sélectionnées,
2. pour chaque sous-collection A sélectionnée, une nouvelle sous-collection A' est calculée à partir de A (et éventuellement de son voisinage),
3. les sous-collections calculées viennent se substituer aux sous-collections sélectionnées.

(Voir Fig. 1). Pour spécifier une transformation il faut préciser : comment sélectionner A (à l'aide des motifs), comment est calculé A' (à l'aide de la partie droite des règles) et parfois les contraintes de la substitution de A par A' dans C (en général la manière de substituer les parties dépend de la nature de la topologie de C).

Motifs

Les motifs MGS ont un sens générique. Ceci signifie qu'ils peuvent filtrer n'importe quelle sorte de collection. La grammaire des expressions de motif est :

$$\begin{array}{lll} \text{Motif} & m ::= & x \mid c \mid (p) \mid \_ \mid <\texttt{undef}> \mid \{...\} \\ & & \mid m, m' \mid m\delta m' \mid m+ \mid m* \mid m\delta+ \mid m\delta* \\ & & \mid m : P \mid m/exp \mid m\ \texttt{as}\ x \mid (m|m') \end{array}$$

où m et m' sont des motifs, x décrit des variables de motifs, P est un prédicat, exp est une expression s'évaluant en un booléen et δ est une direction (dénotant un sous-ensemble particulier de la relation de voisinage). Les explications qui suivent donnent une sémantique informelle de ces motifs :

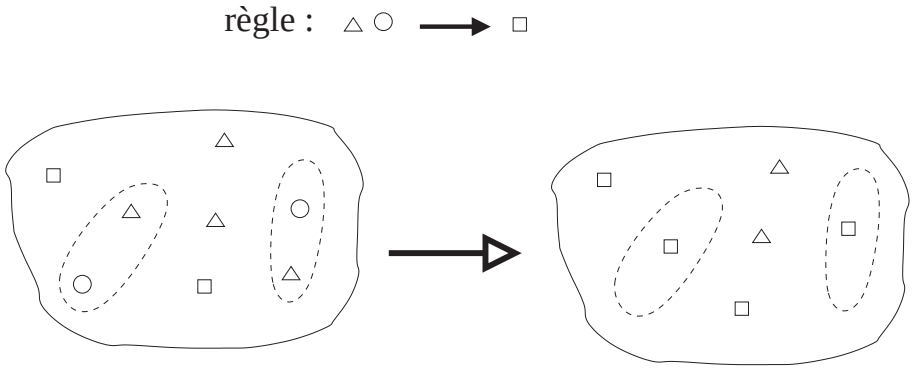


FIG. 1 – Transformation d'une collection. Une règle spécifie qu'une sous-collection A doit être remplacée par la sous-collection A' calculée à partir de A. La collection calculée dans la partie droite de la règle dépend de la sous-collection filtrée dans sa partie gauche et éventuellement de son voisinage.

variable : Une variable de motif x filtre une position de la collection possédant une valeur v. L'identificateur x peut alors être utilisé dans une garde ou en dehors du motif pour dénoter la valeur filtrée v.

Si la variable n'apparaît pas ailleurs, on peut utiliser le motif anonyme dénoté par le symbole « _ » afin de ne pas nommer inutilement une position filtrée.

constante : Un motif c, où c est une constante, filtre une position dont la valeur est égale à la constante c.

vide : Le motif `<undef>` filtre une position qui ne possède pas de valeur. En pratique, on n'utilisera `<undef>` dans un motif que comme voisin d'un élément bien défini (*i.e.* différent de `<undef>`).

voisinage : Le motif m, m' filtre un chemin $p_0, \ldots, p_k, p_{k+1}, \ldots, p_l$ où $p_0, \ldots, p_k$ (resp. $p_{k+1}, \ldots, p_l$) est filtré par m (resp. m') et tel que p_k et p_{k+1} soient voisins. Idem pour le motif $m\delta m'$ mais en plus les positions notées p_k et p_{k+1} ci-dessus doivent être voisines *selon la direction* δ.

garde : Le motif x/e filtre une position vérifiant l'expression e. Par exemple `x,y / x>y` filtre deux positions p_0 et p_1 telles que la valeur en p_0 est plus grande que la valeur en p_1. Le motif $e : P$ est équivalent au motif $e/P(e)$.

nommage : La construction m `as` x sert à lier x aux éléments du chemin filtré par m.

répétition : Le motif $m*$ peut filtrer un chemin vide, un chemin filtré par m ou une suite de positions composée de chemins filtrés par m et dont la jonction est constituée de deux positions voisines. Le symbole « * » (étoile) est utilisé en référence aux expressions régulières où elle exprime une répétition éventuellement nulle. Le motif $m+$ filtre les mêmes chemins que $m*$ sauf le chemin vide.

Le motif $m\delta*$ est similaire à $m*$ 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 les chemins filtrés par m_1 et les chemins filtrés par m_2.

Par exemple, le motif

```
(x:int / x<3)+ as S / (card(S) < 5) && (fold[+](S) > 10)
```

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

Certaines constructions de motifs n'existent que pour une sorte de collection. Les opérateurs `left` et `right` sont un autre exemple de constructions particulières. Elles peuvent êtres utilisées comme gardes dans un motif (ou dans la partie droite d'une règle) pour faire référence à l'élément à la droite, ou à la gauche d'une sous-séquence filtrée. Un motif utilisant ces opérateurs n'a pas de sens quand la transformation est appliquée par exemple à un ensemble. Une erreur signale alors l'inconsistance du filtrage.

Règles

Une transformation est un ensemble de règles. Lorsqu'une transformation est appliquée à une collection, la stratégie est d'appliquer en parallèle le plus de règles possible (on dit qu'une règle peut être appliquée lorsque son motif « filtre » une partie de la collection considérée). On peut affiner cette stratégie.

Priorité. Par défaut, les règles sont munies d'une priorité : la première règle de la transformation est la plus prioritaire. Ceci signifie que la première règle sera appliquée le plus possible et que les règles suivantes pourrons être appliquées aux parties n'ayant pas été déjà filtrées avec succès.

Mode stochastique. Une autre stratégie consiste à appliquer les règles sans considérer de priorités entre elles. On peut exprimer des degrés entre le pleinement aléatoire et le strictement prioritaire en donnant un coefficient à chaque règle dénotant une probabilité d'application. Ainsi si une règle R_1 a un coefficient 1 et une règle R_2 a un coefficient 2, la règle R_2 sera appliquée deux fois plus (si l'on considère que leurs motifs ont tous les deux le même nombre d'instances dans la collection).

Mode asynchrone. Dans le mode asynchrone, les règles ne sont pas appliquées en parallèle mais séquentiellement.

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

```
trans T [a=0] = { ...; R =   x / a<5 => (a := a+1; 2*x) ; ...  }
```

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

II.2.2 Gérer l'application des transformations

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

L'expression `T(c)` dénote l'application d'une étape de transformation à la collection `c`. La transformations peut être itérées de diverses manières :

`T[n](c)`	dénote l'application de `n` étapes de transformation à `c`.
`T[fixpoint](c)`	dénote l'application de `T` jusqu'à atteindre un point fixe.
`T[fixrule](c)`	dénote l'application de `T` jusqu'à ce qu'aucune règle ne s'applique.

II.3 Les collections topologiques disponibles dans MGS

Dans cette section, nous détaillons plusieurs caractéristiques des collections topologiques, dont la manière générique avec laquelle elles peuvent être manipulées. Ensuite nous décrivons les types de collections disponibles dans MGS (ou, plus précisément, dans l'interprète 1-MGS).

Notons que dans MGS, les éléments d'une collection peuvent être des valeurs atomiques, des enregistrements ou des collections quelconques et ne sont pas forcément du même type (collections *hétérogènes*).

II.3.1 Types de collections

Deux collections de nature différente (un ensemble et une séquence par exemple) ont un type différent et ce type rend compte de cette nature. On peut établir une hiérarchie entre les types de collections. La figure 2 montre cette hiérarchie sur plusieurs sortes de collections.

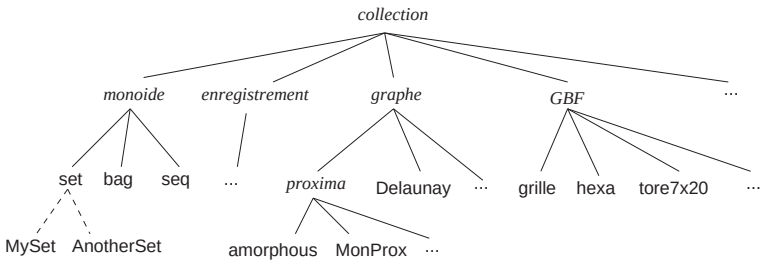


FIG. 2 – Hiérarchie sur les types de collections. MySet et AnotherSet sont des types de collections définis par l'utilisateur. Les types *collection* et *monoïde* ne correspondent pas à des structures de données concrètes. Formellement, un enregistrement est un ensemble de paires ⟨*nom-du-champ, valeur-du-champ*⟩ mais son utilisation nécessite des opérateurs particuliers.

Types définis par l'utilisateur. Il est souvent nécessaire de distinguer plusieurs usages pour des collections de même nature. MGS permet de faire ceci en déclarant de nouveaux types qui seront de la même nature qu'un type déjà existant. À ce niveau, le type d'une collection doit

être vu comme une étiquette qui ne change pas la structure de la collection (cette notion sera affinée dans le chapitre VII).

Voici un exemple de déclarations de types collection :

```
collection MySet = set ;;
collection AnotherSet = set ;;
```

Ces déclarations définissent deux nouveaux types différents de `set` mais implémentés de la même manière. De nouvelles constantes sont également rendues disponibles : des prédicats permettant de tester le type d'une valeur (ici, `MySet` et `AnotherSet`), des collections vides (`MySet:()` et `AnotherSet:()`) et des opérateurs de construction (à travers les constructeurs surchargés sur les collections).

Il existe d'autres manières de créer de nouveaux types : en déclarant de nouvelles topologies appartenant à la famille des GBF (voir section II.4) ou de graphes proximaux (voir section II.3.5). Par exemple :

```
gbf grille = < nord, est > ;;
proximal amorphous = a_portee ;;
```

Ici, `< nord, est >` est une *présentation de groupe* représentant des directions élémentaires dans une relation de voisinage et `a_portee` est un prédicat à deux arguments induisant une relation de voisinage, nous reviendrons sur ces notions plus loin dans ce chapitre. Comme pour la création des types `MySet` et `AnotherSet`, ces deux définitions génèrent des prédicats de types, des collections vides et des constructeurs via les constructeurs surchargés existants.

II.3.2 Opérations génériques

Il existe un grand nombre d'opérations génériques pour tous les types collections, fondées sur l'algèbre fonctionnelle développée par exemple dans [BNTW95]. La table ci-dessous décrit les principales opérations utilisées sur les ensembles, multi-ensembles et séquences. La ligne (*)

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

Tab. 1 – Constructeurs sur les monoïdes.

décrit une syntaxe surchargée. La virgule dénote un opérateur surchargé appelé *join* dénotant selon les cas l'ajout d'un élément à une collection ou la combinaison de deux collections. Notons que le *join* est utilisable sur toutes les collections, y compris les enregistrements. Toutefois, son comportement peut être peu intuitif sur certains types de collections (en particulier, la combinaison de GBF notamment n'est pas spécifiée).

De même qu'il existe des constructeurs génériques, il existe des *déconstructeurs* génériques. Il s'agit des opérateurs `one_of` et `rest`. Le premier renvoie un élément d'une collection et le second renvoie la collection privée d'un éléments. Ces deux opérateurs sont implémentés de manière à ce

que `(one_of(c),rest(c))` contienne les mêmes éléments que `c`. Autrement dit, l'élément retiré par `rest` est le même que celui renvoyé par `one_of`.

L'opérateur `size` renvoie le nombre d'éléments d'une collection. L'opérateur `map` applique une fonction à tous les éléments d'une collection. L'itérateur `fold` permet de parcourir la collection en réduisant ses valeurs en une seule valeur suivant une fonction donnée. Par exemple :

```
fold['fct=\x,y.x+y, 'zero=0](c)
```

renvoie la somme de tous les éléments de la collection `c` (les éléments entre crochets sont des arguments étiquetés de `fold`).

On peut également construire des fonctions génériques en combinant des opérations génériques. Par exemple, la fonction `iota` ci dessous produit une collection comportant les entiers de 1 à n et dont le type dépend d'une collection (vide) en argument :

```
fun iota (n,c) =
 if n = 1 then c
 else ( n , iota(n-1,c) )
 fi
```

On produira la séquence des entiers entre 1 et 10 par `iota(10,seq:())`. On produira l'ensemble de ces 10 entiers par `iota(10,set:())`.

II.3.3 Collections monoïdales

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

II.3.4 Enregistrements

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

On peut fusionner des enregistrements avec l'opérateur +. L'expression `r1+r2` donne un nouvel enregistrement `r3` possédant les champs de `r1` et de `r2`. Si un champ `a` est présent dans `r2` alors `r3.a` vaut `r2.a`, sinon `r3.a` vaut `r1.a`. Cette fusion est asymétrique [Rém93a].

Aucune déclaration de type n'est nécessaire à la manipulation d'enregistrements. Toutefois on peut déclarer des types d'enregistrements afin de disposer de prédicats (des tests de types) sur les enregistrements.

La déclaration d'un type enregistrement se fait ainsi :

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

La première déclaration spécifie le type d'enregistrement R défini par la présence d'un champ a. Les types peuvent être utilisés comme prédicats, ainsi

`R({a=2, x = 3})` ou `{a=2, x = 3}:R` (notation équivalente)

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

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

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

II.3.5 Collections proximales

Les collections proximales (ou *proxima*) sont des collection dont la relation de voisinage est définie par une relation donnée sur les éléments.

Par exemple, si `r` est une relation sur les entiers implémentée par un prédicat à deux arguments (par exemple `fun r (x,y) = abs(y-x) < 10`), on peut définir un type de collections proximales comme suit :

```
proximal MonProx = r ;;
```

Deux entiers n et m seront voisins dans une collection de type `MonProx` si et seulement si leur « distance » est inférieure à 10. On dira que `r` est la *relation caractéristique* des collections proximales de type `MonProx`.

Des exemples d'utilisation de proxima en MGS sont donnés en section III.5.

II.3.6 Graphes de Delaunay

Soit un ensemble E de points de $\mathbb{R}^n$. On appelle *diagramme de Voronoï* associé à E le partitionnement de $\mathbb{R}^n$ dont chaque partie (appelée *région*) contient un unique élément de E et tel que la région associée à un point p de E soit l'ensemble de points de $\mathbb{R}^n$ qui sont plus proches de p que de n'importe quel autre point de E.

On appelle graphe de Delaunay le graphe dont les sommets sont les éléments de E et où deux éléments sont reliées si leurs régions respectives dans le diagramme de Voronoï ont une frontière commune.

Dans MGS, le graphe de Delaunay associé à un ensemble de points est considéré comme une collection topologique dont la relation de voisinage correspond aux arêtes du graphe. Des graphes de Delaunay (appelés plus simplement Delaunays) peuvent être construits de plusieurs manières dans MGS : en manipulant un Delaunay existant avec par exemple l'opérateur d'ajout

(le *join*), en transformant un ensemble dont les éléments dénotent des points de $\mathbb{R}^n$ (fonction `delaunayfy`), ou en appliquant une transformation à un Delaunay existant.

À chaque fonction de distance `d` (fonction prenant deux valeurs MGS et renvoyant un flottant) on peut associer un type de graphes de Delaunay. Cette fonction de distance permet de calculer un graphe de Delaunay dont les sommets ne sont pas restreints à des points de $\mathbb{R}^n$. La déclaration d'un tel type de collections se fait de la manière suivante :

```
delaunay MyDelaunay = d ;;
```

Des exemples d'utilisation des Delaunays sont proposés en section III.6.

II.3.7 Généralisation des tableaux

Les GBF sont une généralisation des tableaux introduite dans [GMS96]. Les GBF constituent une famille de collections topologiques dans MGS et permettent de représenter des structures indexées régulières. Nous détaillons le fonctionnement des GBF dans MGS dans la section suivante.

II.3.8 Travaux en cours

D'autres types de collections topologiques sont en cours d'intégration dans MGS : les graphes quelconques, les G-cartes [SMG04, SM04] et les complexes cellulaires.

II.4 Les GBF en détails

Cette section décrit les GBF qui correspondent aux collections représentant un espace *homogène* dans MGS. De telles structures de données se rencontrent lorsqu'on discrétise de manière régulière un domaine de l'espace physique, par exemple pour la résolution numérique d'une équation aux dérivées partielles.

Notre idée est de se concentrer sur la notion de voisinage d'un point et sur les déplacements permis dans un espace homogène. Un voisinage uniforme peut alors se coder par la *présentation* d'un groupe. Ceci constitue le fondement de la notion de *GBF*, abréviation de l'anglais « Group Based Field » ou *champ de données basé sur une structure de groupe* en français. La notion de GBF a été élaborée dans [Mic96].

II.4.1 Tableaux, champs de données et représentation des espaces homogènes

Le tableau est habituellement utilisé pour la représentation informatique des espaces homogènes. Par exemple un treillis rectangulaire, où chaque point a 4 voisins (les voisins *nord*, *est*, *sud* et *ouest*), peut se représenter par un tableau ; les treillis rectangulaires à 8 voisins aussi (voir figure 3). Chaque élément du tableau correspond à un point de l'espace. L'indice qui indexe un point du tableau peut être vu comme la coordonnée ou bien la position de ce point dans l'espace. Les manipulations d'indices correspondent alors à un déplacement dans l'espace : par exemple, si on ajoute 1 au premier indice, on se délace du point courant à son voisin sud (l'association d'une direction à une dimension est une convention arbitraire).

Avec quelques précautions et manipulations supplémentaires d'indices, on peut représenter des treillis *rebouclés*, par exemple pour discrétiser des formes circulaires. La représentation d'une

discrétisation hexagonale, où chaque point a 6 voisins, est plus compliquée, et plus généralement, l'utilisation de tableaux pour représenter des espaces homogènes arbitraires présente de nombreux défauts [Mic96] : ils ne permettent pas la représentation de formes arbitraires, la topologie n'est pas explicite et elle reste trop simple, il faut la coder dans les manipulations d'indices, etc.

Notion de champ de données. Une première généralisation de la notion de tableau est le *champ de données* (en anglais : *data field*). Un champ de données est une fonction partielle de $\mathbb{Z}^d$ dans un ensemble de valeurs. Les champs de données généralisent la notion de tableau puisqu'on peut voir un tableau de dimension d comme une *fonction totale*[2] de $\{1,\ldots,n_1\}\times\cdots\times\{1,\ldots,n_d\}$: d est la dimension du tableau et n_i le nombre d'éléments dans la i^{eme} dimension. Un champ de données est plus général puisqu'il correspond à des *fonctions partielles* sur $\mathbb{Z}^d$. L'intérêt des champs de données par rapport aux tableaux est de permettre la représentation de formes arbitraires : la valeur associée à un indice n'est pas nécessairement définie, ce qui permet de représenter des tableaux avec des trous, des zones triangulaires, des domaines arbitraires (voir figure 4).

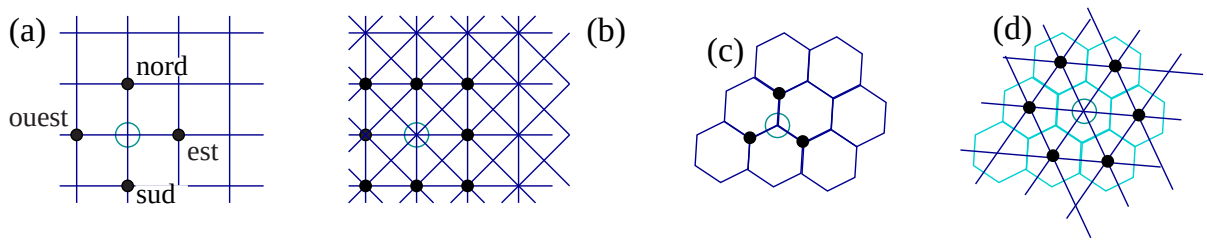


FIG. 3 – Exemples de treillis discrétisant de manière homogène un espace plan. **(a)** : treillis rectangulaire à 4 voisins appelé *voisinage de Von Neuman*. **(b)** : treillis rectangulaire à 8 voisins appelé *Voisinage de Moore*. **(c)** : treillis triangulaire : chaque point possède trois voisins. **(d)** : treillis hexagonal : chaque point possède 6 voisins. Si on représente chaque point de l'espace par une cellule hexagonale, et que la connexion entre deux points est représentée par le partage d'une face de la cellule, on obtient un pavage hexagonal.

Représentation de topologies arbitraires. Si la notion de champ de données permet la représentation de *régions de géométrie arbitraire*, le choix de $\mathbb{Z}^d$ comme ensemble d'indices ne permet pas de représenter des *topologies arbitraires*. La topologie naturelle associée à un champ de données est celle de $\mathbb{Z}^d$: une grille d-dimensionnelle avec un voisinage de Von Neuman ou bien de Moore. Par exemple dans $\mathbb{Z}$, chaque indice i représente un point qui possède deux voisins, un à gauche d'indice $i-1$ et un à droite d'indice $i+1$. On ne peut donc pas directement représenter directement un anneau dans lequel, en partant de i, on peut retourner à i en ne se déplaçant que vers la droite.

Bien sûr, on peut encoder un anneau sur un champ de données, en prenant garde aux manipulations d'indices : si par exemple l'anneau compte $n+1$ points de discrétisation, il faut éviter de calculer $(n+1)$ pour atteindre le voisin de droite du point n, et passer directement à 0 (voir figure 5). Mais comme il a déjà été mentionné, le codage d'une topologie triangulaire comme celle de la figure 3 n'est de loin pas aussi simple.

[2]Une fonction est totale si chaque élément de l'ensemble de départ possède une image bien définie (ce qui s'oppose à fonction *partielle*). Ici, le caractère total correspond au fait que chaque élément du tableau possède une valeur bien définie. Si f est une fonction totale d'un ensemble A dans un ensemble B, on notera $f : A \mapsto B$.

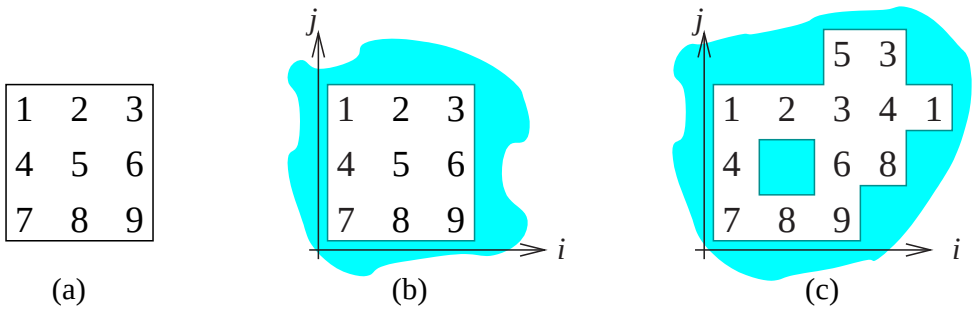


FIG. 4 – Notion de tableau et de champ de données. **(a)** : un tableau. **(b)** : le même tableau vu comme un champ de donnée : c'est une fonction qui associe à chaque indice (i, j) une valeur. Cette fonction est partielle : la plupart des indices (il y en a une infinité) ne sont associés à aucune valeur. Le domaine de définition du champ de données est symboliquement représenté par une région blanche sur le fond grisé. **(c)** : un champ de données permet de représenter simplement des régions arbitraires qui généralisent les régions rectangulaires des tableaux.

Notion de GBF. Afin de pouvoir représenter des topologies arbitraires, en sus de géométries arbitraires, il faut étendre la notion de champ de données. Nous allons nous restreindre ici à des espaces *homogènes*. Un espace est dit homogène quand chaque point a le *même* voisinage : par exemple, tous les points ont « un voisin à gauche » et « un voisin à droite ». Dans le reste de ce paragraphe, nous décrivons succinctement le cadre théorique qui sous-tend la démarche suivie dans MGS pour définir des collections topologiques permettant la représentation d'espaces homogènes et la généralisation de la structure de données de tableau *via* les GBF. La suite de la section présente de manière plus concrète comment définir des GBF et comment calculer avec.

Nommons a, b, c... les directions permettant de se déplacer vers les voisins d'un point et $\mathcal{V}oisin(a, P)$ le voisin suivant la direction a d'un point P. Une direction a peut s'identifier avec l'opération de déplacement élémentaire $\mathcal{V}oisin(a, \cdot)$. Ces déplacements peuvent être composés et cette composition possède une *structure de groupe mathématique* : elle est associative, pour chaque déplacement a on considère un déplacement inverse noté $-a$ et il existe un déplacement nul qui consiste à « ne pas se déplacer ». L'application des déplacements à un point est l'*action* du groupe sur l'espace des points. On appelle *espace homogène* un ensemble de points muni d'un groupe de déplacements[3].

Si nous voulons définir un espace homogène arbitraire, il faut donc spécifier deux choses :

1. définir le groupe des déplacements à partir des déplacements élémentaires $a, b, \ldots$;
2. définir l'ensemble des points sur lequel ce groupe va agir.

Pour résoudre le point 2, le plus simple est de faire agir le groupe des déplacements 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 : $\mathcal{V}oisin(a, P) = P + a$.

Pour spécifier le groupe, nous allons utiliser une *présentation* : une présentation donne une liste de générateurs et une liste d'équations entre générateurs. Tout élément du groupe s'obtient comme somme de générateurs. Dans notre cas, les générateurs correspondent aux déplacements

[3] Le groupe de déplacement doit de plus agir transitivement, c'est-à-dire qu'il existe un déplacement qui va de n'importe quel point à n'importe quel autre point (connexité). Voir par exemple [col95] pour un rappel des définitions mathématiques concernant la structure de groupe. Mais notre utilisation des notions sur les groupes reste à un niveau élémentaire.

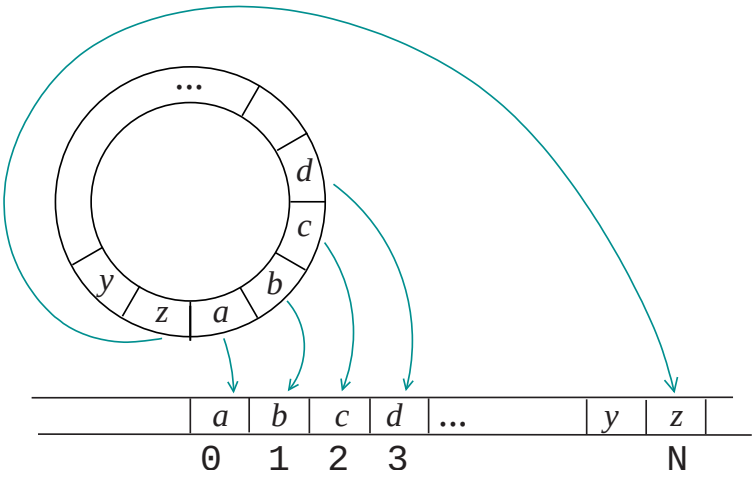


FIG. 5 – Codage d'une topologie d'anneau sur un champ de données de dimension 1. Chaque point d'indice i pour $0 \leq i \leq n$ dans le champ de données correspond à un élément de l'anneau. L'opération qui permet de calculer l'indice j du voisin de droite d'un élément d'indice i est alors définie par : $j = i + 1 \mod n$.

élémentaires qui définissent le voisinage homogène.

Le cadre que nous venons de décrire constitue le fondement de la notion de *GBF* abréviation de l'anglais « Group Based Field » ou *champ de données basé sur une structure de groupe* en français. Si on veut comparer l'approche des GBF aux tableaux et aux champs de données, on a le dictionnaire suivant :

tableau

objet mathématique	fct. totale : $\{0,\ldots,n_1\} \times \ldots \times \{0,...,n_d\} \longmapsto$ *Valeur*
point	index $(i,\ldots,k) \in \{0,\ldots,n_1\} \times \cdots \times \{0,...,n_d\}$
déplacement	$(i,\ldots,k) \rightarrow (i \pm 1,\ldots,k),\ \ldots,\ (i,\ldots,k \pm 1)$

champ de données

objet mathématique	fct. partielle : $\mathbb{Z}^d \longrightarrow$ *Valeur*
point	élément $p \in \mathbb{Z}^d$
déplacement	$p \rightarrow p \pm e_i$ (e_i vecteur de la base canonique)

GBF

objet mathématique	fct. partielle : *groupe* $\mathcal{G} \longrightarrow$ *Valeur*
point	élément $p \in \mathcal{G}$
déplacement	$p \rightarrow p \pm a$ (a générateur)

II.4.2 Un exemple de GBF

La déclaration suivante :

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

introduit un nouveau type de collection topologique, faisant partie de la famille des GBF, et nommé `G`. La spécification de `G` apparaît en partie droite du signe égal, entre les délimiteurs `<` et `>`. Cette spécification, appelée aussi *présentation* du groupe, introduit ici deux nouveaux noms : `nord` et `est`, qui sont appelés des *générateurs*. Les générateurs correspondent à des directions de déplacement élémentaires et définissent le voisinage de chaque élément dans un GBF : chaque élément possède quatre voisins suivant les directions `nord`, `est` et les directions inverses.

Il est possible de donner un nom aux directions inverses, en introduisant ces directions et en complétant la liste des générateurs par des *équations* reliant une direction et sa direction inverse :

```
gbf G = < nord, est, sud, ouest ; nord + sud = 0, est + ouest = 0 > ;;
```

Les équations apparaissent derrière la liste des générateurs et sont séparées par une virgule. Le point-virgule est utilisé pour délimiter la liste des générateurs. Avec cette nouvelle déclaration, on peut dire que les voisins d'un élément dans un GBF de type `G` sont atteints en suivant les quatre directions `nord`, `est`, `ouest` et `sud`. En effet, l'équation

`nord + sud = 0` peut se réécrire en `sud = - nord`

ce qui montre que `sud` est bien la direction inverse de `nord`.

Les équations dans une présentation peuvent servir à autre chose qu'à donner des noms explicites aux inverses d'un générateur. Par exemple la présentation suivante :

```
gbf A5 = < a ; 5*a = 0 > ;;
```

définit un GBF où chaque point possède deux voisins (suivant la direction `a` et son inverse `-a`). L'équation `5*a = 0` indique que si on suit cinq fois la direction `a` tout se passe comme si on n'avait pas progressé : on se retrouve au même point. Le GBF `A5` définit donc une topologie de cinq points disposés en anneau.

La définition d'un GBF correspond à la *présentation d'un groupe*. Les éléments du groupe, qui sont aussi les positions du GBF, correspondent à des sommes formelles qu'on peut écrire à l'aide des générateurs. Comme dans tout groupe, on peut additionner et soustraire des positions, inverser une position, etc. Nous dirons que deux positions P et Q sont voisines s'il existe un générateur `g` tel que $P + \mathtt{g} = Q$ ou bien $Q + \mathtt{g} = P$.

Si la déclaration d'un GBF `G` introduit un nouveau type `G` associé à la présentation d'un groupe, une valeur de type `G` correspond à l'association de valeurs quelconques à certaines positions du GBF, de la même manière qu'un tableau associe des valeurs à des indices.

II.4.3 Visualisation de la topologie d'un GBF par un graphe de Cayley

Il est simple de visualiser les positions d'un GBF `G` et leurs relations de voisinages par le *graphe de Cayley* du groupe associé à `G`. Dans ce graphe :

- chaque sommet représente un élément du groupe,
- chaque arc est étiqueté par un générateur,
- un arc étiqueté g se trouve entre les sommets P et Q si $P + g = Q$.

En termes de collection topologique, le graphe de Cayley d'un GBF visualise la relation de voisinage : chaque sommet est un élément de la collection et il est relié à ses voisins. La figure 6 illustre la correspondance entre groupes et graphes de Cayley.

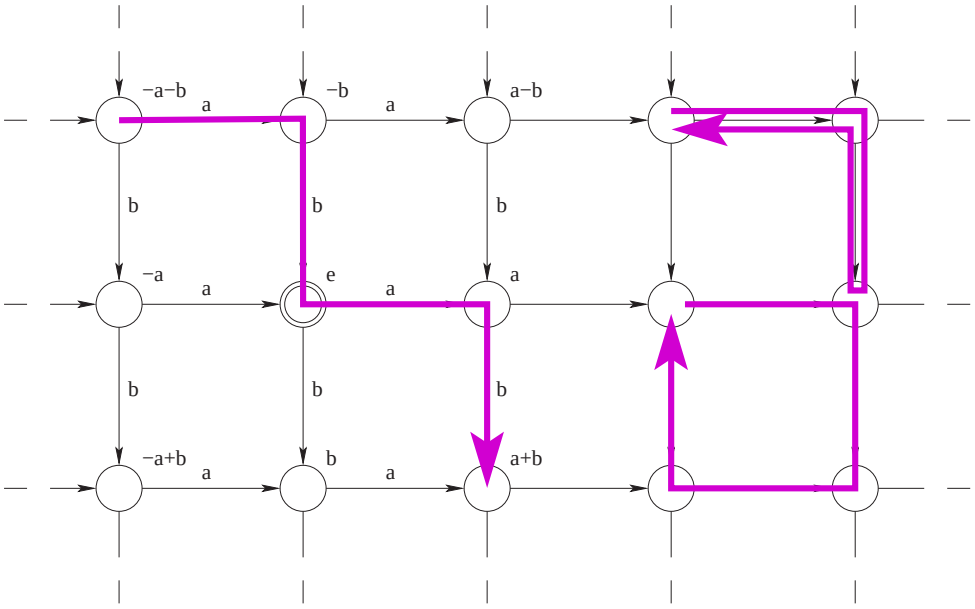


FIG. 6 – Les graphes de Cayley permettent de faire un pont entre la théorie des graphes et les concepts de la théorie des groupes. Étant donné une présentation d'un groupe, on peut lui associer une représentation par un graphe appelé graphe de Cayley. Le sommet correspondant à l'élément neutre 0 du groupe est distingué : c'est l'origine. Le graphe de Cayley que nous avons représenté correspond au GBF : `gbf H = < a, b ; a+b = b+a >`. En termes de collection topologique, le graphe de Cayley d'un GBF permet de visualiser la relation de voisinage : chaque sommet est un élément de la collection et il est *relié* à ses voisins immédiats. On peut visualiser une valeur `v` de type `G` de manière analogue, par le sous-graphe du graphe de Cayley de `G`, restreint aux sommets appartenant au domaine de définition de `v`, et où chaque sommet est décoré par la valeur correspondante.

II.4.4 GBF abéliens

Le graphe de Cayley que nous avons représenté à la figure 6 correspond au GBF :

```
gbf H = < a, b ; a+b = b+a > ;;
```

La seule équation de la présentation correspond au cycle : `a+b-a-b = 0` et spécifie la commutativité de l'opération de groupe.

Implicitement, tous les groupes définis dans ce chapitre correspondent à des *groupes commutatifs*. On dit aussi que le groupe ou bien le GBF est *abélien*. La propriété de commutativité indique qu'on peut réorganiser la somme décrivant un chemin, afin de regrouper des directions égales. Par exemple, le chemin `a+b+a+b` peut se réécrire `a+a+b+b` où plus simplement encore `2*a + 2*b`.

Une présentation de groupe débutant par `<` et se terminant par `>` est une présentation de groupe abélien, et les équations de commutativité entre générateurs n'ont pas besoin d'être explicitées. On aurait donc pu définir le GBF `H` plus simplement par

```
gbf H = < a, b > ;;
```

On voit que le GBF `H` est semblable au GBF `G` de la section II.4.2, au renommage des générateurs près : `nord` devient[4] `a` et `est` devient `b`. On dit alors que `G` et `H` sont des *groupes isomorphes*, et il est usuel de ne pas les distinguer en mathématique (on travaille « à isomorphisme près »). En revanche, en MGS, les deux déclarations correspondent à des types différents, dont les valeurs sont distinguées et qu'on ne peut pas combiner arbitrairement : on dit que l'égalité des types est « syntaxique » et n'est pas « structurale ».

Dans une grille, il est vrai que se déplacer vers le nord, puis vers l'est, conduit au même endroit que se déplacer tout d'abord vers l'est puis le nord. Mais ce n'est pas vrai de tous les espaces homogènes. Considérons par exemple un arbre binaire : de chaque sommet, je peux descendre dans le fils gauche ou bien le fils droit. On peut montrer qu'un tel graphe correspond à un GBF à deux générateurs. Cependant, à partir d'un sommet donné, descendre dans le fils gauche puis dans le fils droit n'aboutit pas au sommet atteint en descendant tout d'abord dans le fils droit et ensuite dans le fils gauche. Autrement dit, un arbre binaire correspond à un *GBF non-abélien* [Lar02].

II.4.5 Déplacements

Reprenons notre premier exemple :

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

cette définition introduit un nouveau type `G` de collection. Une fois le GBF défini, on peut former des expressions correspondant à des mots. Les mots de la présentation `G` sont appelés les *déplacements* de `G` et correspondent à des sommes formelles de générateurs. Dans MGS, les déplacements associés à un GBF sont des valeurs comme les autres : on peut les définir et les passer en paramètre d'une fonction, les combiner par certaines opérations primitives, et les retourner comme valeur d'une fonction.

Les constantes. Les déplacements les plus simples correspondent aux générateurs. En dehors de la présentation, les générateurs doivent débuter par le symbole `|` et le symbole `>` qui sont appelés « indication de direction ». Par exemple :

```
|nord>
```

Pour noter l'inverse d'un générateur, il suffit de renverser les indications de directions :

`<nord|` ou bien d'utiliser le signe "`-`" `- |nord>`

ce qui correspond à la direction sud.

Combinaison linéaires de générateurs. Un déplacement est une somme de générateurs. Les opérations d'addition et de soustractions sont surchargées pour permettre de calculer simplement des déplacements. Par exemple :

```
|nord> + |est>
```

[4] On aurait tout aussi bien pu choisir de renommer `nord` en `b` et `est` en `a`.

est une expression dont l'évaluation rend une valeur notée `|nord> + |est>`. En fait, comme les GBF considérés ici sont tous des GBF abéliens, l'ordre dans lequel on effectue chaque mouvement élémentaire importe peu. Chaque déplacement dans un GBF de générateurs $g_1, \ldots, g_d$ peut donc s'écrire plus simplement :

$$a_1.g_1 + a_2.g_2 + \cdots + a_n.g_n$$

C'est cette forme qui est utilisée pour afficher le résultat d'un calcul de déplacements. Par exemple :

```
|nord> + |est> - <nord| + <est|
```

est une expression dont le résultat est :

```
2*|nord>
```

En effet, `|nord> = - <nord|` et `|est> + <est|` est le déplacement vide, correspondant à l'élément neutre du groupe et indiquant qu'aucun déplacement n'a été effectué.

Nous avons vu que l'on pouvait additionner ou soustraire deux déplacements. On peut aussi multiplier un déplacement par un entier, ce qui correspond à le répéter :

$$n * \texttt{|g>} \quad = \quad \underbrace{\texttt{|g> + ... + |g>}}_{n \text{ fois}}$$

II.4.6 Opérations sur les GBF

De la même manière qu'il existe des fonctions permettant de manipuler des séquences, des multi-ensembles ou des ensembles, il existe des opérateurs qui permettent de manipuler des GBF. Ces opérateurs s'appliquent à tous les GBF abéliens (il s'agit donc d'une forme de polytypisme).

Construire des GBF en extension

GBF vide. Le GBF le plus simple que l'on peut construire est le GBF vide. Supposons que l'on ait déclaré :

```
gbf G = < ... > ;;
```

alors le GBF vide de type `G` se note :

`G:()` ou bien `():G`

cette notation est consistante avec les autres types collections. La fonction `size` renvoie bien sur 0 sur un GBF vide (son domaine de définition, *i.e.* l'ensemble des positions portant une valeur est l'ensemble vide).

Le nom du type, comme le nom de n'importe quel autre type, sert de prédicat permettant de tester si une valeur donnée est de ce type : `(0 == size(():G)) && G(():G)` retourne la valeur `true`.

Définition en extension d'un GBF. On peut construire un GBF en listant ses éléments dans une séquence et en indiquant les directions suivant lesquelles il faut placer ces éléments. C'est la construction `following`. Par exemple, en reprenant le GBF `A5` :

```
gbf A5 = < a; 5a > ;;

G1 := (1,2,3) following |a> ;;
G2 := (1,2,3) following <a| ;;
G3 := (1,2,3,4,5,6,7) following |a> ;;
```

les trois GBF `G1`, `G2` et `G3` sont représentés dans la figure 7. On voit que les éléments s_1, s_2, etc. de la séquence sont placés en partant du sommet origine[5] et en se déplaçant suivant la direction indiquée par le second argument de `following`. Le i^{eme} élément v_i de la séquence correspond ainsi à la visite d'une position p_i et la valeur v_i est associée à la position p_i. Dans le cas de `G1`, la position p_i est atteinte par le déplacement $(i-1)$`*|a>` à partir de l'origine et on va successivement passer par 0, `|a>` et `2*|a>`. Pour `G3`, on va repasser par des positions déjà valuées : dans ce cas, c'est le dernier passage qui impose sa valeur.

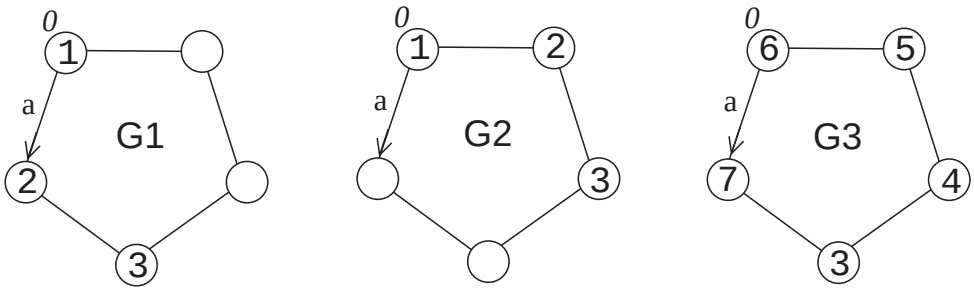
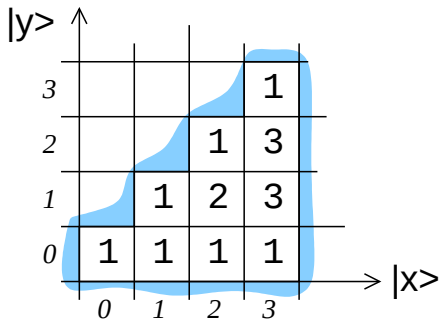


FIG. 7 – Trois instances de GBF de type `A5` construit avec l'opérateur `following`.

Dans la forme précédente, le remplissage du GBF se fait suivant une seule direction. Il est possible de changer de direction en utilisant une séquence imbriquée, comportant autant d'imbrications que de directions :

```
gbf grid2 = < x, y > ;;

Pascal := ( (1, ():seq)  ::
            (1, 1)       ::
            (1, 2, 1)    ::
            (1, 3, 3, 1) ::
            ():seq )
            following |x> |y> ;;
```



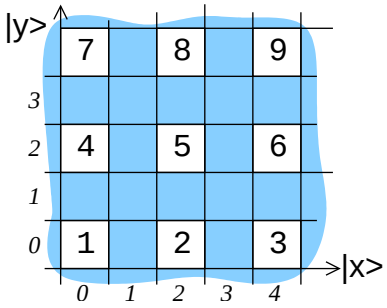
Dans cet exemple, le premier argument est une séquence de séquences s_i. La direction `|x>` est utilisée pour se déplacer entre les s_i et la direction `|y>` est utilisée pour se déplacer entre les éléments $s_{i,j}$ de s_i.

La construction `following` est une syntaxe infixe de la fonction primitive `build_gbf(`v`,` d`)` qui prend une (imbrication de) séquence(s) de valeurs v et une séquence d de directions. Les directions arguments du `following` doivent être des générateurs. En revanche, la fonction `build_gbf`

[5] Le sommet origine est étiqueté par l'élément neutre `0` du groupe.

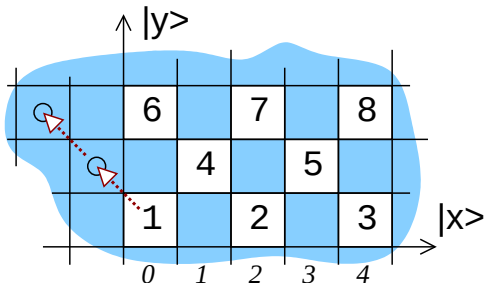
admet des déplacements quelconques. L'expression ci-dessous permet de construire un GBF avec une valeur toutes les deux cases suivant les directions `|x>` et `|y>` :

```
mod2 := build_gbf( ((1, 2, 3) ::
                    (4, 5, 6) ::
                    (7, 8, 9) ::
                    ():seq)
                   ,
                   (2*|x>, 2*|y>)) ;;
```



La valeur `<undef>` joue un rôle particulier dans l'énumération des valeurs d'un GBF : elle spécifie que la valeur de la position correspondante est indéfinie[6]. Ceci permet de construire des GBF « avec des trous ». Par exemple, un GBF « en échiquier » peut se construire par l'expression :

```
chess := build_gbf(
    ((1, 2, 3)          ::
     (<undef>, 4, 5)    ::
     (<undef>, 6, 7, 8) ::
     ():seq)
    ,
    (2*|x>,
     |y> - |x>) ) ;;
```



En effet, chaque fois que l'on se déplace suivant la direction `|y> - |x>`, on se déplace suivant la direction diagonale indiquée sur le schéma. La valeur de la position `|y> - |x>` est la première valeur de la seconde sous-séquence, et c'est donc `<undef>`. La position correspondante (indiquée par un cercle dans le schéma) ne possède donc pas de valeur. Pour la troisième « ligne », on se déplace deux fois suivant `|y> - |x>` et donc on « démarre » la description deux positions plus à gauche que pour la première ligne ; mais comme on se déplace horizontalement de deux en deux, il suffit de spécifier une seule valeur indéfinie afin que la valeur `6` ait la même coordonnée en `|x>` que la valeur `1`.

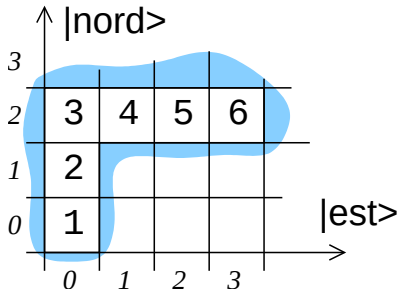
Ajout d'un élément à un GBF. Nous avons vu que la virgule sert à noter le *join* de deux collections monoïdales et indique la relation de voisinage entre deux éléments d'un motif. Ces deux usages sont consistants, car l'opération de concaténation servant à construire la collection monoïdale sert aussi à définir la relation de voisinage (cf. section II.3.3). Cette double notation ne peut pas être maintenue dans le cas des GBF car, pour un GBF donné, la virgule ne permet pas de distinguer entre toutes les relations de voisinages possibles. Par exemple, pour `gbf G = < nord, est >` il y a quatre voisins possibles à un point donné, suivant les deux directions `|nord>` et `|est>` et leurs inverses. L'idée est donc d'utiliser le nom du générateur à la place de la virgule. Par analogie avec l'expression x,ℓ valable pour les collections monoïdales ℓ, une expression comme :

x `|nord>` g

[6] Ceci contraste avec les collections monoïdales dans lesquelles un élément ne peut avoir comme valeur `<undef>`. Ces deux points de vue sont différents comme nous le verrons au chapitre IV.

où un générateur remplace la virgule, ajoute l'élément x au GBF g de la manière suivante : tous les éléments de g sont translatés vers le `|nord>` et l'élément x est inséré à la position `|0>` (même s'il y avait déjà une valeur). Si g n'est pas un GBF, on construit un nouveau GBF, avec une seule valeur g en `|0>` et c'est ce GBF auquel on rajoute x. Par exemple :

```
1 |nord> 2 |nord> 3 |est> 4 |est> 5 |est> 6 ;;
```



construit un GBF avec 6 valeurs. Cette construction peut s'interpréter de la manière suivante : mettre `1` à l'origine puis se déplacer selon la direction `|nord>`, mettre la valeur `2` et se déplacer selon `|nord>`, mettre la valeur `3` puis se déplacer selon `|est>`, mettre la valeur `|4>` puis se déplacer, etc. On voit que cette notation est assez analogue à la construction `following` mais que les directions à suivre sont indiquées après chaque valeur. Comme la virgule, cette construction associe à droite :

`x |g> y |g'> z` s'interprète comme `x |g> (y |g'> z)`

Toujours comme la virgule, cette construction admet des arguments mixtes : un élément et un GBF sont associés pour donner un nouveau GBF (ajout d'un élément à un GBF) et deux éléments sont associés pour faire un nouveau GBF les contenant.

Map et *fold* d'un GBF

Comme pour toutes les collections, les fonctions `map` et `fold` permettent de parcourir tous les éléments d'un GBF. Le résultat de

```
map[f](g)
```

est un GBF de même type que `g`, et dont la valeur en une position `p` est `f(`v`)`, avec v la valeur en la position `p` dans `g`. Le résultat comporte donc des éléments exactement aux mêmes positions que `g`.

La fonction `fold` garde sa sémantique usuelle : une fonction binaire `fct` est utilisée pour réduire les valeurs du GBF en une valeur unique et une valeur `zero` est utilisé pour donner une valeur initiale (qui sera la valeur du GBF vide). Par exemple :

```
fold['fct=(\x,y.x+y), 'zero=0](g)
```

permet de faire la somme de tous les éléments du GBF `g` (la notation entre crochet est spécifique aux arguments étiquetés). L'ordre dans lequel les éléments sont parcourus n'est pas précisé. La fonction `fct` doit donc être associative et commutative pour que le résultat soit bien défini.

II.4.7 Éléments d'implémentation : représentation normalisée d'une position dans un GBF abélien

Pour conclure cette description des GBF, nous donnons quelques indications sur la représentation normalisée d'une position dans un groupe abélien. Les considérations ci-dessous permettent de calculer un déplacement par rapport à l'origine identifiant une position de manière unique. Cette représentation s'appuie sur le fait que tout groupe abélien possède une structure canonique de $\mathbb{Z}$-modules.

Notion de $\mathbb{Z}$-module. Un $\mathbb{Z}$-module de dimension 1 est un groupe abélien, noté $(\mathbb{Z}/n, +)$ avec $n \in \mathbb{N}$. Formellement, ce groupe est construit comme le quotient de $\mathbb{Z}$ par le sous-groupe $n\mathbb{Z}$ des multiples de n. Intuitivement, $\mathbb{Z}/n$ représente les entiers modulo n avec l'addition modulo. Le module $\mathbb{Z}/0$ est isomorphe à $\mathbb{Z}$ et est appelé un *module libre*. Les autres modules $\mathbb{Z}/n$, $n \neq 0$, sont appelés généralement *modules de torsion*. Un élément d'un module de torsion $\mathbb{Z}/n$ peut se représenter de manière unique par un entier positif compris entre 0 inclus et n exclus.

Les $\mathbb{Z}$-modules précédents sont des modules de dimension 1 : ils sont engendrés par un seul générateur (qui est l'entier 1). Les $\mathbb{Z}$-modules de dimension supérieure à 1 sont simplement des produits cartésiens de $\mathbb{Z}$-modules de dimension 1.

Autrement dit, tout élément d'un $\mathbb{Z}$-module peut se représenter de manière unique par un vecteur d'entiers. Certaines composantes de ce vecteur prennent leurs valeurs dans tout $\mathbb{Z}$ (il s'agit des composantes correspondant aux $\mathbb{Z}$-modules libres) et les autres composantes prennent une valeur positive entre 0 et n (où n est le coefficient de torsion de la composante).

Théorème fondamental des groupes abéliens finiment engendrés. Le théorème fondamental des groupes abéliens finiment engendrés indique que tout groupe G engendré par un nombre fini de générateurs et de relations (tel que ceux définis par une présentation) est isomorphe à :

$$G \simeq \mathbb{Z}^n \times \mathbb{Z}/t_1 \times \mathbb{Z}/t_2 \times \ldots \times \mathbb{Z}/t_q$$

où t_i divise t_{i+1}. Avec cette condition, les coefficients t_i sont uniques et peuvent être effectivement calculés à partir de la présentation de G : c'est la *forme normale de Smith* du groupe.

Cet isomorphisme fournit une *représentation canonique* des éléments de G. À toute position p identifié par un déplacement c dans G, on peut associer l'uplet d'entiers représentant de manière unique l'élément correspondant à p dans le $\mathbb{Z}$-module isomorphe à G.

Cas des groupes abéliens libres. Un groupe abélien libre est défini uniquement par la donnée de ses générateurs et des équations correspondant à la commutativité. Dans ce cas, les modules de torsion du $\mathbb{Z}$-module associé sont réduits au groupe trivial et un GBF libre à n générateurs est isomorphe au module $\mathbb{Z}^n$. En d'autres mots, la position d'un tel GBF est simplement représentée par un n-uplet d'entiers et on retrouve le cas classique des tableaux à n dimensions ou plus précisément, des champs de données.

Chapitre III

Exemples de programmes

III.1 Multi-ensembles

La programmation par récriture sur les multi-ensembles a été étudiée par les travaux autour du langage Gamma [BM86, BCM88, BLM92, BFM01, BFR04]. Les exemples de cette section sur les multi-ensembles ont tous été conçus dans le projet Gamma.

III.1.1 La fonction *factorielle*

La fonction factorielle est usuellement définie par une fonction récursive aussi bien dans les langages impératifs que fonctionnels : `fact n = if n<=1 then 1 else n*fact(n-1)`. Ce type de définition introduit une dépendance artificielle entre `fact(n)` et `fact(n-1)` et induit donc un ordre dans dans lequel les multiplications doivent être effectuées.

Or la définition intrinsèque de la fonction factorielle ne nécessite pas cet ordre. Par exemple $4! = 1 * 2 * 3 * 4$ et rien n'impose d'effectuer une multiplication avant une autre (associativité de la multiplication sur les entiers). Si l'on dispose de ressources parallèles, on peut même en réaliser plusieurs à la fois.

Plutôt que de considérer la définition habituelle pour la fonction `fact`, considérons la suivante : soit S le multi-ensemble contenant les nombres entiers entre 1 et n. Tant qu'il y a plus d'un nombre dans cet ensemble, y substituer n'importe quelle paire de nombres par leur produit.

En MGS, choisir deux nombres et les remplacer par leur produit s'exprime par la règle de transformation `x,y => x*y`. Dans un multi-ensemble, cette règle sélectionne des instances du motif `x,y`, c'est-à-dire des paires et les remplace par leur produit.

La fonction factorielle consiste simplement à produire le multi-ensemble initial contenant les n premiers entiers puis à appliquer la transformation constituée de cette règle jusqu'à atteindre un point fixe :

```
fun iota (n,c) = if n<=0 then c else n::(iota(n-1,c)) fi ;;

trans fac = { x,y => x*y } ;;

fun factorielle n = oneof ( fac['fixpoint](iota(n,bag:())) ) ;;
```

Remarque : le multi-ensemble initial est produit séquentiellement mais on peut également le produire par une transformation :

```
trans generate = { {val=x} => x, {val=x+1} } ;;
trans succeed = { {val=x} => x } ;;
fun iota n = succeed(generate[n]({val=1},set:())) ;;
```

III.1.2 Calcul des nombres premiers

Le calcul des nombres premiers est un autre exemple de programme où aucune séquencialité n'est nécessaire. Pour calculer les nombres premiers inférieurs à n il suffit de considérer l'ensemble des entiers entre 2 et n et d'appliquer la règle suivante : choisir deux nombres x et y tel que x divise y et les remplacer par x. Lorsque cette règle ne peut plus être appliquée, l'ensemble contient exactement les nombres premiers inférieurs à n.

```
fun iota2 (n,c) = if n<=1 then c else n::(iota(n-1,c)) fi ;;

trans prem = { x,y/(y mod x = 0) => x } ;;

fun premiers n = prem['fixpoint](iota2(n,set:())) ;;
```

III.1.3 Enveloppe convexe d'un ensemble de points du plan

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

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

```
state  Point={x,y} ;;
```

On définit ensuite une fonction `inside` prenant quatre points en arguments et indiquant si le premier est dans le triangle formé par les trois autres. Cette fonction est détaillée dans [GM01a] et utilise essentiellement des calculs simples sur les flottants.

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

```
trans Convex = { X,U,V,W / inside(X,U,V,W) => U,V,W } ;;
```

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

```
Convex[fixrule]( (
   {x=0,   y=0}, {x=0.2,   y=0.1}, {x=0.5,   y=0.7},
   {x=1,   y=0}, {x=0.1,   y=0.2}, {x=1,   y=1},
   {x=0.2,   y=0.4}, {x=0.4,   y=0.6}, {x=0,   y=1}, set:()
) ) ;;
```

donne le résultat attendu :

```
{x=0, y=0}, {x=0, y=1}, {x=1, y=0}, {x=1, y=1}, ():set
```

III.1.4 Propagation de la chaleur sur une barre

Usuellement, la propagation de la chaleur sur une barre est simulée en utilisant une discrétisation de l'équation différentielle la modélisant. La structure de donnée correspondante est alors un tableau.

On peut adopter un point de vue complètement différent et modéliser le même phénomène avec des multi-ensembles. La barre est discrétisées en n segments consécutifs que l'on adresse de 1 à n. Chaque segment porte des *quanta* de chaleur. Nous allons représenter la barre par un multi-ensemble de quanta, chaque quantum étant représenté par un entier dénotant le segment sur lequel il se trouve. L'état de la barre est donc un multi-ensemble d'entiers. Le cardinal de cet entier correspond directement à la quantité de chaleur dans la barre.

La règle d'évolution est très simple : un quantum peut soit passer sur un segment voisin, soit rester sur le segment où il se trouve. La transformation MGS correspondante est la suivante :

```
trans diffuse = {     n => n + random(-1, 0, 1)     } ;;
```

Deux autres règles de transformation doivent être ajoutées pour tenir compte du comportement aux bornes de la barre. La fonction `random` choisit un de ses trois arguments. La probabilité pour qu'un quantum se déplace est déterminée par l'équation différentielle régissant le modèle continu.

La figure 1 montre les résultats d'une exécution du programme MGS sur une barre de 30 segments contenant un total de 2500 quanta. Les bornes sont maintenues à 0 quanta.

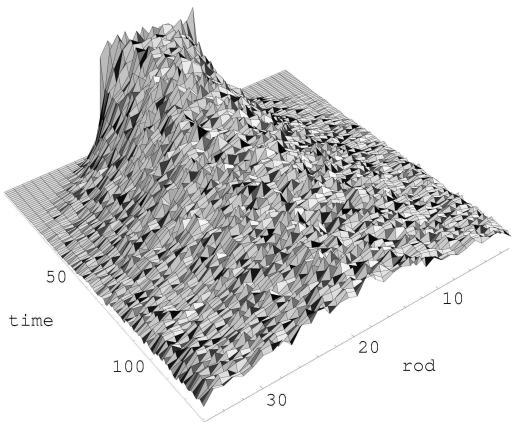


FIG. 1 – Diffusion de la chaleur sur une barre simulée par un multi-ensemble de *quanta* de chaleur. Le temps croît sur l'axe *nord-ouest→sud-est*. La barre est représentée sur l'autre axe.

III.2 Ensembles

III.2.1 Manipulation de formules logiques

En partant de l'observation que la conjonction et la disjonction logiques sont associatives, commutatives et idempotentes, tout comme le constructeur des ensembles, on peut élaborer très simplement des programmes manipulant des formules logiques. En effet, inutile de programmer ces trois propriétés (ACI) puisqu'elles existent déjà dans le langage. Par exemple on va représenter la formule $f_1 \wedge f_2 \wedge f_3$ par l'ensemble `f1,f2,f3,set:()`. Afin de différencier une conjonction d'une disjonction nous allons définir deux sortes d'ensembles :

```
collection Et = set ;;
collection Ou = set ;;
```

Le code MGS ci-dessus définit deux nouvelles sortes de collections, ayant les mêmes propriétés que les ensembles. De plus, les prédicats `Et` et `Ou` peuvent être utilisés pour déterminer si une valeur est une collection `Et` ou une collection `Ou`.

Une implication $f_1 \Rightarrow f_2$ est la donnée de deux formules f_1 et f_2 dont l'ordre est important : l'implication n'est pas commutative. Une séquence convient pour représenter une implication puisque le constructeur des séquences n'est pas commutatif. Enfin, la négation est nilpotente ($\neg\neg f = f$) mais aucun constructeur n'ayant cette propriété dans MGS, nous devrons coder la nilpotence nous mêmes. Nous définirons simplement la négation par une séquence particulière :

```
collection Imp = seq ;;
fun impl(f1,f2) = f1::f2::Imp:() ;;
fun pre(x)=hd(x) ;;
fun post(x)=hd(tl(x)) ;;

collection Non = seq ;;
fun neg f = f ::  Non:() ;;
fun get_non(x)=hd(x) ;;
```

On peut calculer la forme normale disjonctive d'une formule à l'aide de la transformation ci-dessous définie par des règles simples :

```
trans FND = {
   x:Imp                   => post(x) :: (neg(pre(x)) :: Ou:() ;;
   x:Non / Non(get_non(x)) => get_non(get_non(x)) ;;
   x:Non / Ou (get_non(x)) => fold [ (::) , neg , Et:() ] (get_non(x)) ;;
   x:Non / Et (get_non(x)) => fold [ (::) , neg , Ou:() ] (get_non(x)) ;;
   x:Et  / contient_Ou(x)  =>
    fold [ (::) ,( fun (l)=hd(extract_Ou(x))::l),Ou:()] (tl(extract_Ou(x)));;
   x:Ou  / contient_Ou(x)  => flatten_Ou (x);;
   x:Et  / contient_Et(x)  => flatten_Et (x);;
   x:Ou  / size(x)=1    => hd(x);;
   x:Et  / size(x)=1    => hd(x);;
     } ;;
```

Afin de calculer la forme normale disjonctive d'une formule `f`, cette transformation doit être

– appliquée récursivement aux sous-formules de la formule considérée et
– itérée jusqu'à atteindre un point fixe.

III.3 Séquences

III.3.1 Tri à bulles

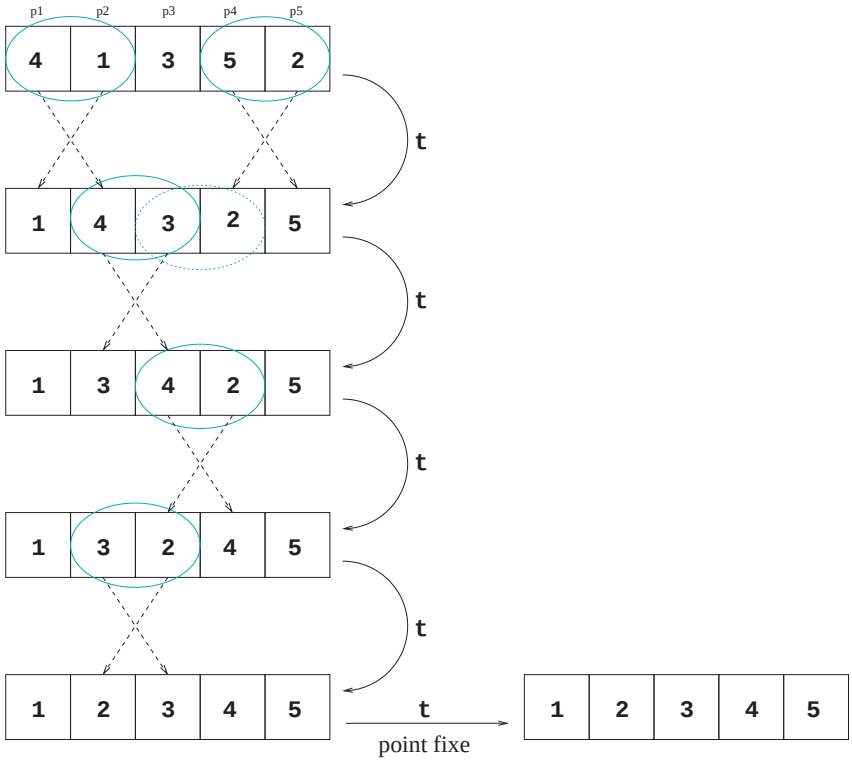


FIG. 2 – Une trace possible du tri d'une séquence.

La transformation suivante permet d'implémenter un tri sur les séquences :

```
trans t = { x,y / y<x => y,x } ;;
fun tri s = t['fixpoint] (s);;
```

Le motif filtre des éléments voisins rangés dans le mauvais ordre. De tels couples d'éléments seront replacés dans le bon ordre (partie droite de la règle). Cette transformation doit être appliquée sur une séquence jusqu'à ce qu'un point fixe soit atteint pour obtenir la séquence triée.

Détaillons le fonctionnement de ce tri. Partons de la séquence `[4,1,3,5,2]` (FIG. 2). L'algorithme de filtrage trouve les deux chemins correspondant aux couples d'éléments voisins mal rangés : $\{[p_1, p_2]; [p_4, p_5]\}$. Ces deux chemins ne s'intersectant pas, ils sont tout deux séléctionnés pour être « transformés ». Ainsi les valeurs en positions p_1 et p_2 sont remplacées respectivement par 1 et 4 et les valeurs en position p_4 et p_5 sont remplacées par 2 et 5. La transformation est

ensuite appliquée sur la nouvelle collection renvoyée : `[1,4,3,2,5]`. Deux chemins sont filtrés avec succès : $\{[p_2, p_3]; [p_3, p_4]\}$. Leur intersection n'est pas vide. On doit donc en sélectionner un seul, par exemple $[p_2, p_3]$ (ce choix est arbitraire). Donc 4 et 3 seront replacés dans le bon ordre mais 2 reste inchangé. La nouvelle collection est `[1,3,4,2,5]`. L'application suivante ne filtre qu'un chemin, ainsi que la suivante. Après ces quatre applications, la séquence est `[1,2,3,4,5]`. La séquence étant triée, la transformation ne trouve aucune instance du motif et renvoit donc cette même séquence. Le point fixe est atteint et la séquence triée est renvoyée.

III.3.2 Le crible d'Eratosthene

Le principe est d'affiner l'algorithme de la section III.1.2 en utilisant une collection topologique de séquence. Chaque élément i dans la séquence correspond au i^{e} nombre premier P_i calculé et est représenté par un enregistrement $\{$`prime =` $P_i\}$. Cet élément peut recevoir comme candidat un nombre n, qui est représenté par un enregistrement $\{$`prime =` P_i`, candidate =` $n\}$. Si le nombre candidat vérifie le test, alors l'élément se transforme en un enregistrement `r =` $\{$`prime =` P_i`, ok =` $n\}$. Si le voisin droit de `r` est de la forme $\{$`prime =` $P_{i+1}\}$, alors le nombre candidat n se déplace de `r` vers le voisin droit. Quand il n'existe plus de voisin à la droite de `r`, alors le nombre n est un nombre premier et un nouvel élément est ajouté à la fin de la séquence. Le premier élément de la séquence est distingué des autres éléments et produit les entiers candidats.

```
trans Erato = {
 n:integer / ~right  n => n, {prime =n} ;                    (* Genere1 *)

 n:integer, {prime as x, ~candidate , ~ok }
    => n+1, { prime = x, candidate = n };                     (* Genere2 *)

 {prime as x, candidate as y, ~ok } / y  mod  x = 0
    => {prime = x} ;                                          (* Test1 *)

 {prime as x, candidate as y, ~ok } / y  mod x <> 0
    => {prime = x, ok = y} ;                                  (* Test2 *)

 {prime as x1, ok as y }, { prime as x2, ~ok , ~candidate }
    => {prime = x1}, {prime = x2, candidate = y} ;              (* Next *)

 {prime as x, ok as y} as s / ~right s
    => { prime =x}, {prime = y} ;                           (* NextCreate *)
} ;;
```

La figure 3 illustre le fonctionnement du programme. L'expression `Eratos[N]((2, seq:()))` exécute N pas du crible d'Eratosthene. Par exemple, l'expression `Eratos[100]((2, seq:()))` calcule la séquence : `42, {candidate =42, prime =2}, {ok =41, prime =3}, {prime =5}, {prime =7}, {prime =11}, {prime =13}, {ok =37, prime =17}, {prime =19}, {prime =23}, {prime =29}, {prime =31}, seq:()`.

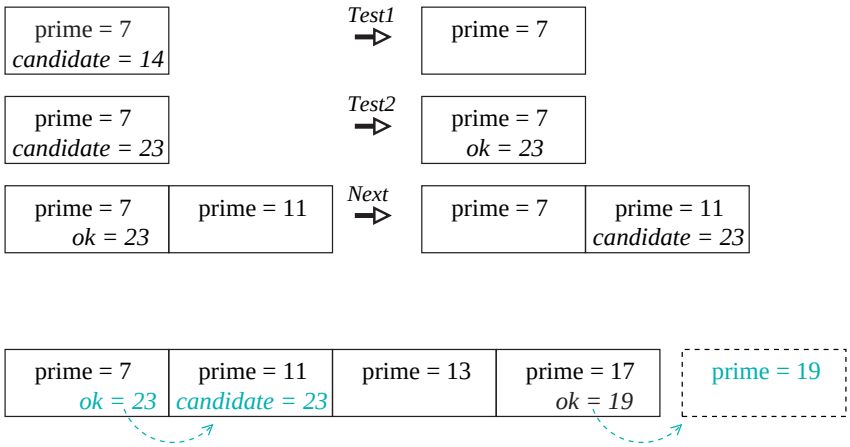


FIG. 3 – Le programme `Erato`. Quelques instanciations de règles et un fragment de séquence construit par la transformation `Erato`.

III.3.3 Croissance d'un organisme uni-dimensionnel à la façon des L-sytèmes

Nous reprenons ici un exemple paradigmatique de formalisation d'un processus de croissance en L-système [Lin68]. Les états du développement d'un organisme filamentaire « uni-dimensionnel » sont donnés à travers la définition d'un L-système décrit dans [Mic96] que nous implémenterons ensuite en MGS. Les règles décrivant l'évolution de l'organisme permettent à ses cellules de rester dans le même état, de changer d'état, de se diviser en plusieurs cellules ou bien de disparaître.

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

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

Les suffixes l et r indiquent la polarité des cellules. Un arbre de dérivation du processus de croissance est illustré par la figure 4 (empruntée à [PH92]). Les règles de changement de polarité de cet exemple sont très proches de celles de la bactérie bleue-verte *Anabaena catenula* [WMS73, dKL87].

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

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

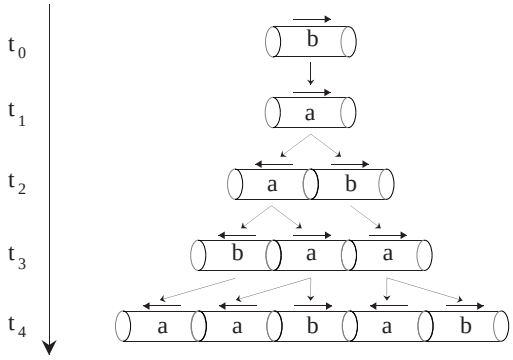


FIG. 4 – Les quatre premières dérivations du processus de croissance de la bactérie *Anabaena catenula* modélisé par un L-système (la polarité des cellules est indiquée avec une flèche).

```
  "bl"   =>   "al"   ;
} ;;
```

L'expression MGS suivante calcule les quatre premières étapes du développement :

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

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

III.3.4 Propagation de la chaleur sur une barre

On simule la diffusion de la chaleur H le long d'une fine barre de métal. L'équation différentielle partiele parabolique $\frac{\partial H}{\partial t} = \alpha \frac{\partial^2 H}{\partial x^2}$ décrivant le processus de diffusion peut être discrétisée :

$$H(x, t + \Delta t) = (1 - 2c)H(x, t) + c(H(x - 1, t) + H(x + 1, t))$$

où $H(x, t)$ est la température de l'élément x de la séquence à l'instant t et le paramètre c dépends de la caractéristique de diffusion α de la barre de métal (c est inférieur a 0.5). On peut ajouter à ce modèle des conditions aux bornes. Le programme MGS correspondant qui calcule l'évolution de la température dans la barre de métal est facilement écrit :

```
trans diffuse[c=0.25, Hleft=0, Hright=0] = {
    x / (left(x) == <undef>)  => Hleft;
    x / (right(x) == <undef>) => Hright;
    x => (1-2*c)*x + c*(right(x) + left(x))
} ;;
```

Les deux premières règles gèrent les conditions aux bornes données par les paramètres `Hleft` et `Hright`. Les opérateurs `right` et `left` permettent d'accéder aux voisins d'un élément dans une séquence. La valeur spéciale `<undef>` est retournée comme valeur associée à une position qui n'a pas de valeur. Le tracé sur la figure 5 illustre le résultat de l'exécution.

On retrouve bien le résultat obtenus par la modélisation par quanta de chaleur.

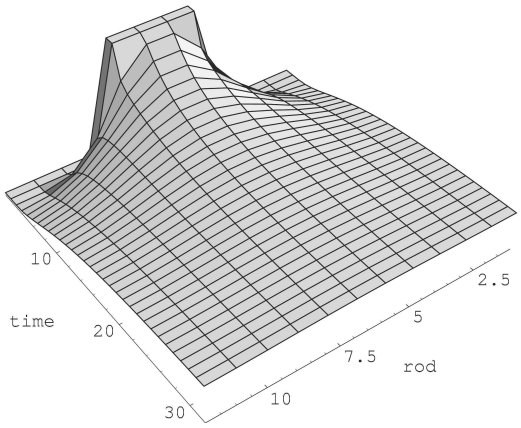


FIG. 5 – Résultat de la diffusion de la chaleur. Les extrémités de la barre sont maintenues à la température de $0°C$. Initialement, on ne chauffe que le tiers central de la barre. Le temps s'écoule de l'arrière vers l'avant.

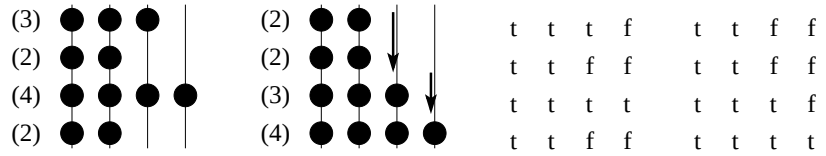
III.4 GBF

III.4.1 Tri par boulier

Le tri par boulier est un méthode originale pour trier des entiers écrits en une colonne de nombres en base unaire [ACD02]. Les nombres sont triés en laissant tomber les bits verticalement, comme le montre le schéma ci-dessous où l'on trie les nombres [3,2,4,2]. Le premier schéma représente les nombres avant le tri et le second après le tri. Les troisième et quatrième schémas correspondent à l'implémentation des deux premiers sur une grille de booléens où *true* représente un bit et *false* représente l'absence de bit. La transformation suivante permet d'implémenter ce tri :

```
trans {x |nord> y / (x=false && y=true) => y,x} ;;
```

Le tri est obtenu en itérant l'application de cette transformation jusqu'à l'obtention d'un point fixe.



III.4.2 Rotation d'un motif d'une forme spécifique

Nous proposons à présent un exemple inspiré des simulations de gaz sur réseaux, proches des automates cellulaires. Dans ces simulations on utilise des règles de la forme $\beta \Rightarrow f(\beta)$ pour spécifier les évolutions localisées de particules réparties sur un partitionnement régulier du plan. Le motif β filtre une configuration planaire comme par exemple deux particules voisines. La simulation proprement dite consiste à itérer l'application des règles sur une structure de départ.

Ici nous voulons spécifier la rotation de particules disposées en croix sur une grille carrée (Fig. 6). La grille est représentée par un GBF `grille` et la rotation est exprimée par la transformation :

```
trans rota =
    { c |est> e |nord-est> n |-est-nord> o |est-nord> s => c,s,e,n,o } ;;
```

Notons que le motif de cette règle ne filtre pas un sous-tableau mais une partie de la structure qui a la forme voulue.

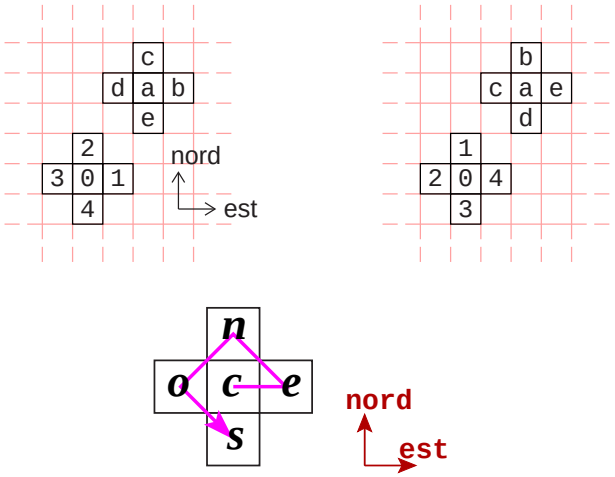


Fig. 6 – Rotation d'un motif sur une grille carrée. *nord* et *est* sont les deux générateurs.

La partie droite de la règle spécifie par quoi doivent être remplacés les éléments filtrés dans l'ordre où ils apparaissent dans le motif : c est remplacé par c, e par s, n par e, etc. Ceci correspond ici à une permutation circulaire des éléments périphériques de la partie filtrée et il en résulte une rotation globale du motif.

Pour des raisons propres aux phénomènes simulés, les physiciens préfèrent simuler les gaz sur un pavage hexagonal du plan plutôt que sur une grille carrée. La figure 7 montre une adaptation de notre exemple à ce pavage. Cette fois le motif β filtre une configuration de sept particules.

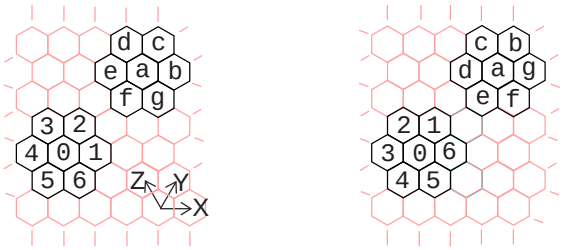


Fig. 7 – Rotation d'un motif sur un pavage hexagonal. X, Y et Z sont les générateurs.

Voici la rotation du motif sur les GBF de type `hexa` :

```
trans rota_h =
{ a |X> b |Z> c |-X> d |-Y> e |-Z> f |X> g => a,g,b,c,d,e,f ; } ;;
```

III.4.3 Le modèle d'Eden

Eden est un modèle simple de croissance [YPQ58] parfois utilisé pour modéliser la croissance de villes, de tumeurs ou de populations végétales. Le plan est partitionné en cellules vides ou occupées. À chaque étape de temps chaque cellule occupée dont une cellule voisine est libre se duplique dans cette dernière. La transformation correspondante s'écrit

```
trans Eden = { x, <undef> => x, x ;} ;;
```

Une cellule pleine est représentée par une valeur dans le GBF support de la simulation et une cellule vide correspond à une position sans valeur.

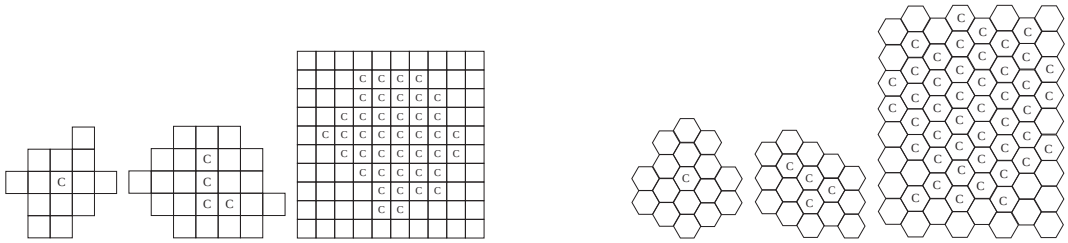


Fig. 8 – Croissance Eden sur deux GBF.

La figure 8 montre l'itération de la transformation sur deux types de GBF différents avec une seule cellule occupée au départ. La partie gauche de la figure 9 montre une variation avec trois valeurs différentes au départ, obtenue avec la même transformation.

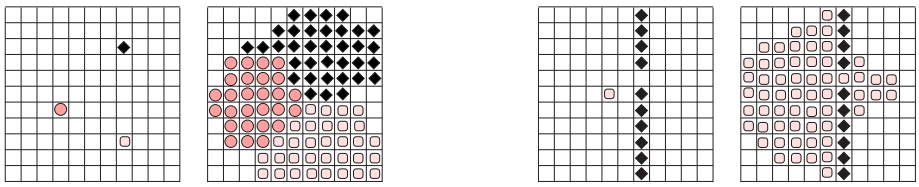


Fig. 9 – Croissance Eden avec trois espèces et avec un obstacle.

On peut aussi définir des cellules « mortes » qui ne peuvent être occupées, il faut dans ce cas adapter la transformation :

```
trans Eden2 = {x/viv(x), <undef> => x, x;} ;;
```

La partie droite de la figure 9 montre une croissance qui doit passer à travers un trou dans un mur de cellules mortes.

III.4.4 Labyrinthe

Considérons un labyrinthe représenté par un GBF où la valeur 1 symbolise l'entrée, la valeur 2 représente les couloirs et 3 représente la sortie (Figure 10). Le premier schéma est le labyrinthe

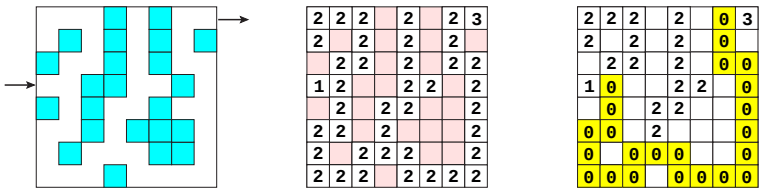


FIG. 10 – Chemins dans un labyrinthe.

à modéliser, le second est sa représentation par un GBF où les zones grisées ne possèdent pas de valeur.

La transformation ci-dessous trouve un chemin entre l'entrée et la sortie et le marque par des 0 (`\x.0` dénote la fonction `fun (x) = 0`) :

```
trans laby = { 1, (2* as X), 3 => 1, (map (\x.0) X), 3 ;};;
```

Le troisième schéma de la figure 10 est le résultat de l'application de la transformation où la solution trouvée à été mise en évidence.

III.4.5 Réaction/Diffusion de Turing sur un tore

Dans les années 1950, Turing a proposé un système simple de réaction-diffusion décrivant des réactions chimiques et une diffusion pour expliquer complètement le phénomène de morphogénèse, *i.e.* le développement de formes dans les systèmes biologiques [Tur52].

On utilise parfois ce système pour modéliser des phénomènes spécifiques tels que la coloration de la peau d'animaux (comme les tâches chez le léopard ou les rayures chez le zèbre) [Tur91].

Les équations du modèles de Turing constituent un système d'équations différentielles partielles non-linéaires de la forme :

$$\frac{\partial \overrightarrow{u}}{\partial t} = \overrightarrow{k}.\nabla^2 \overrightarrow{u} + \overrightarrow{F}(\overrightarrow{u})$$

où le vecteur $\overrightarrow{u}$ décrit à chaque instant t les concentrations spatiales de substances chimiques appelées *morphogènes*. $\overrightarrow{F}$ dénote les spécifications de la réaction.

Nous allons simuler ce système sur un tore. Nous discrétisons le tore en cellules, chacune ayant une concentration particulière en deux morphogènes dénotés par u et v. Le système d'équations suivi par le modèle est :

$$\begin{aligned} \frac{\partial u(x)}{\partial t} &= k * (\alpha - u(x)v(x)) + \text{Diff}_u \left(\sum_{y/x,y} (u(y) - u(x)) \right) \\ \frac{\partial v(x)}{\partial t} &= k * (u(x)v(x) - v(x) - c(x)) + \text{Diff}_v \left(\sum_{y/x,y} (v(y) - v(x)) \right) \end{aligned}$$

où x dénote une cellule ; $u(x)$ et $v(x)$ sont les concentrations en morphogènes u et v dans x. La fonction $c(x)$ correspond à une constante uniforme à tout le tore à laquelle on a ajouté un bruit spécifique à chaque cellule de façon aléatoire. Les variables α, k, Diff_u et Diff_v sont les paramètres de la réaction-diffusion. Enfin, $y/x, y$ désigne l'ensemble des voisins y de x, sur lesquels on somme.

Structure du système (collections)

Le tore discrétisé est modélisé par le type de GBF suivant :

```
gbf torus = < a, b ; 180 a, 60 b > ;;
```

Le tore est pavé par 60*180 soit 10800 cellules.

Chaque cellule aura pour valeur un enregistrement comportant les 3 champs `u`, `v` et `c` correspondant respectivement à la concentration en morphogène u, à celle en morphogène v et la constante de réaction c de la cellule. Pour initialiser ce tore, on utilise la transformation suivante :

```
trans init = {
  x => {
    u = u_steady,
    v = v_steady,
    c = beta_init + random(2*beta_rand) -beta_rand }
} ;;
```

À l'état initial, les concentrations en u et v sont uniformes sur les cellules du tore. La constante de réaction est calculée suivant la même base β_{init} pour toutes les cellules, à laquelle on ajoute un bruit compris entre $-\beta_{rand}$ et β_{rand}.

Dynamique du système (transformations)

Le système de Turing constitue la loi d'évolution du système. Le système d'équations vu plus haut est encodé par la transformation suivante

```
trans Turing = {
  x =>
    let laplacian =
      neighborsfold(add, {u= 0.0 - 4.0*x.u, v= 0.0 - 4.0*x.v}, x)
    in
    let du = k * (alpha - x.u*x.v) + diffu*laplacian.u
    and dv = k * (x.u * x.v - x.v - x.c) + diffv*laplacian.v
      in {
        u = x.u + dt*du,
        v = x.v + dt*dv,
        c = x.c }
} ;;
```

La fonction `neighborsfold` applique la fonction `add` à tous les voisins de x. Elle utilise un accumulateur initialisé à `{u= 0.0 - 4.0*x.u, v= 0.0 - 4.0*x.v}`. Ceci permet de calculer

un enregistrement contenant la somme des différences de concentration entre x et tous ses voisins (il y en a un suivant chaque direction a, b, $-a$ et $-b$). Ceci correspond effectivement aux formules explicitées plus haut. La fonction `add` sert juste à faire la somme entre les champs `u` et `v` de deux enregistrements.

L'application de la transformation `Turing` sur le GBF correspond à une évolution du système d'un pas dans le temps ; en l'itérant, on fait évoluer les concentrations au cours du temps.

Résultats

Les schémas de la figure 11 montrent le tore à l'état initial et après 1200 itérations de l'application de la transformation.

En chaque point du tore, l'épaisseur de l'anneau représente la concentration en v. Dans l'état initial, la concentration est uniforme sur tout le tore ; il apparaît donc lisse. Dans l'état final, l'aspect ondulé correspond aux motifs (ici, des tâches) et à l'auto-organisation de la concentration en v le long du tore sous l'effet de la diffusion des morphogènes.

FIG. 11 – Réaction/diffusion de Turing sur un tore.

III.4.6 Simulation du fourragement par une colonie de fourmis

Nous simulons ici la recherche de nourriture par une colonie de fourmis. Le comportement des fourmis est défini par des règles locales, et on peut voir émerger un comportement global.

Lorsqu'une fourmi trouve de la nourriture, elle la rapporte à la fourmilière en déposant sur son chemin une hormone odorante (une *phéromone*). Lorsqu'une autre fourmi sent une phéromone, elle suit le chemin ainsi constitué jusqu'à la source de nourriture. Plus les fourmis suivent ce chemin, plus il est renforcé en phéromones et facile à découvrir.

Chaque position de la structure de données contient un enregistrement.

```
state place = { phero_nest:float, phero_food:int } ;;

state nest  = place + { nest } ;;

state food  = place + { food:int } ;;

state ant   = place + { load_food : int } ;;

state empty = place + {~nest, ~food, , ~load_food} ;;
```

Une *place* est un enregistrement contenant deux valeurs indiquant la quantité de deux phéromones : la première correspond à l'odeur de la fourmilière et la seconde est celle indiquant un chemin à suivre.

Le nid est représenté par une place (un enregistrement) contenant un champ `nest`. La nourriture est représentée par une place contenant un champ `food`.

Les fourmis sont représentées par une place (la présence d'une fourmi n'est pas incompatible avec la présence de phéromone en un même endroit) contenant un champ représentant la quantité de nourriture portée par une fourmi.

Une place est dite vide lorsqu'elle ne contient ni fourmi, ni nourriture ni fourmilière.

La structure de donnée est une GBF `grille`. On peut également exécuter le programme sur un pavage hexagonal du plan (seule la structure de données initiale change) :

```
gbf grille = <north, east> ;;

gbf hexa = <north, neast, seast; north + seast= neast> ;;
```

Le comportement des fourmis est décrit par des règles d'évolutions locales (une transformation) :

```
trans behavior = {

(* 1 *)  f/seek(f), n:food =>
           (f+{load_food=255}),n ;

(* 2 *)  f/bring(f) , n:nest =>
           (f+{load_food=0,phero_food = 255 }),n ;

(* 3 *)  f/track(f) ,  p/( empty(p) && (p.phero_food >0) && [...] )=>
           { phero_nest = f.phero_nest,
             phero_food = f.phero_food
           },
           (f+p) ;

(* 4 *)  f/seek(f) , p/( empty(p) && (p.phero_food>0)) =>
           { phero_nest=f.phero_nest,
             phero_food = f.phero_food
           },
           (f+p) ;

(* 5 *)  f/seek(f), p/empty(p) =>
           { phero_nest=f.phero_nest,
             phero_food = f.phero_food},
           (f+p) ;

(* 6 *)  f/bring(f) , p/(  empty(p) && [...]) =>
           { phero_nest=f.phero_nest,
             phero_food = (255)},
           (f+p) ;
} ;;
```

Expliquons ces règles d'évolution :

(1). Lorsqu'une fourmi qui cherche de la nourriture en trouve, elle en ramasse.

(2). Lorsqu'une fourmi qui porte de la nourriture arrive à la fourmilière, elle dépose la nourriture (ainsi qu'une phéromone).

(3). Fourmi qui suit une piste de phéromone.

(4). Fourmi qui cherche de la nourriture et qui trouve une piste de phéromone.

(5). Fourmi qui cherche de la nourriture aléatoirement (elle n'a pas trouvé de piste de phéromone).

(6). Fourmi portant de la nourriture et la ramenant à la fourmilière par un chemin le plus direct possible (elle suit l'odeur de la fourmilière).

Dans les règles (3) et (4) les « . . . » cachent le choix de la meilleure place voisine en utilisant un itérateur sur le voisinage (`neighborfold`).

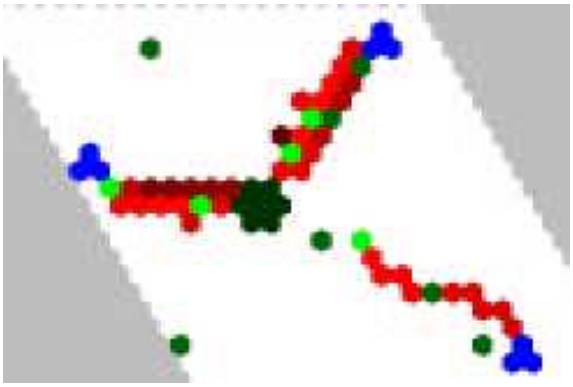


FIG. 12 – Colonie de fourmis sur un pavage haxagonal.

Les règles établissant la diffusion des molécules sur le terrain font l'objet de deux autres transformations (on encapsule les règles de chaque phénomène (comportement des fourmis, diffusion et évaporation des deux types de phéromones) dans une transformation. Les incorporer toutes dans une unique transformation mènerait à une complexification inutile des motifs afin de gérer simultanément plusieurs phénomènes.

```
STEP := 5 ;;

trans phero_food_decrease = {
    x:place => x+{phero_food=max(x.phero_food - STEP, 0)}
    } ;;


RATE := 0.9 ;;

trans gradient_nest2 = {
  n / not (nest(n)) =>
   let m =
     neighborsfold ((\x.\r.( max(x.phero_nest,r))),0,n) in
   n +  {phero_nest = m * RATE}  } ;;
```

La première transformation implémente l'évaporation de la nourriture. La seconde transformation implémente de façon très simple la diffusion de l'odeur de la fourmilière

Ce modèle a été inspiré par [Wil98a]. Le schema de la figure 12 représente l'état du système après une trentaine d'itérations sur un pavage hexagonal.

III.5 Proximas

III.5.1 Déplacement d'une grenouille sur une constellation de nénuphars

Nous simulons ici les déplacements d'une grenouille sautant de nénuphar en nénuphar à la surface d'un plan d'eau. On note d la plus grande distance que peut parcourir la grenouille en un saut. La grenouille peut donc sauter d'un nénupar à un autre si leur distance est inférieure à d.

Nous modélisons le problème en indroduisant une relation de voisinage sur les nénuphars : deux nénuphars sont voisins lorsque la distance qui la sépare est inférieure à d et une grenouille peut passer d'un nénuphar à un autre si et seulement si ce dernier nénuphar est voisin du premier.

Nous allons représenter un nénuphar par un enregistrement contenant les coordonnées de celui-ci sur le plan d'eau. Une grenouille a également des coordonnées et dispose en plus d'un *nom* servant à distinguer les grenouilles entre elles :

```
state feuille = {posx,posy} ;;
state grenouille = feuille + {nom} ;;
```

La structure de données rendant compte de la relation de voisinage que nous voulons définir est appelée dans MGS un *proxima* (voir section II.3.5, page 25). Ici, la relation caractéristique définissant le voisinage est le prédicat qui indique si la distance entre deux nénuphars est inférieure à la distance d (notée `portee_max` ci dessous) :

```
fun dist_carre (e1,e2)=
let x = (e1.posx-e2.posx) in
let y = (e1.posy-e2.posy) in (x*x) + (y*y) ;;

portee_max := 9 ;;

fun a_portee (e1,e2) =
let portee_carre =portee_max * portee_max  in
( dist_carre(e1,e2) < portee_carre) ;;

proximal amorphous = a_portee ;;
```

À présent nous donnons une transformation réalisant le déplacement de la grenouille en direction d'un point précis appelé la cible. La transformation est constituée d'une unique règle sélectionnant une grenouille et un nénuphar tels que le nénuphar est plus proche de la cible que la grenouille. Le nénuphar sélectionné est voisin de la grenouille de manière à assurer que la grenouille puisse sauter. Le déplacement de la grenouille est réalisé en déplacement le champ *nom* de l'enregistrement représentant la grenouille à celui représentant le nénuphar.

```
cible := {posx=8,posy=5} ;;

trans deplace = {
  g:grenouille , f:feuille /  (dist_carre (f,cible) < dist_carre (g,cible))
    => {posx=g.posx,posy=g.posy}, (f + {nom=g.nom})
 };;
```

Le déplacement des feuilles de nénuphar peut lui aussi être réalisé très simplement. Pour réaliser le déplacement d'un nénuphar, il suffit de modifier ses coordonnées. La relation de voisinage sera automatiquement mise à jour sans intervention du programmeur. La transformation ci-dessous réalise un déplacement uniforme de toutes les feuilles de nénuphar :

```
trans flotte = {  f:feuille => f +  {posx=f.posx-1}  };;
```

III.5.2 Exemple de routage en télécommunications

Les collections proximales trouvent également un champ d'application naturel dans les télécommunications. Par exemple, on peut simuler très simplement une constellation de relais servant à communiquer à distance, deux relais pouvant communiquer entre eux si la distance les séparant est inférieure à un seuil d. Les relais peuvent être mobiles dans le plan ou dans l'espace, ils peuvent devenir inopérants et de nouveaux relais peuvent apparaître.

Comme pour l'exemple précédent, on utilisera une collection proximale dont les éléments sont les relais et où deux relais sont voisins si leur distance est inférieure au seuil d. Ainsi pour savoir si un relais $R1$ peut dialoguer avec un relais $R2$ en utilisant des relais intermédiaires, on peut utiliser le motif suivant : `R1, x*, R2`.

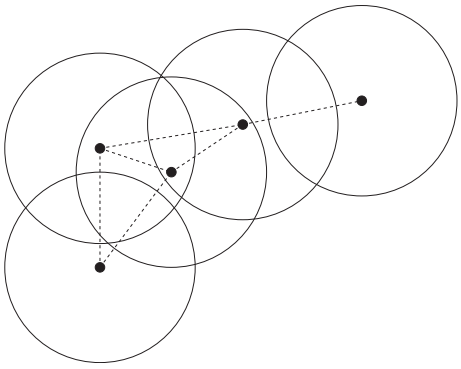


FIG. 13 – Communications à distance sur un proxima.

Sur le schema ci-dessus les point noirs sont les relais, les cercles ont un rayon égal au demi-seuil. Il y a une arête entre deux relais s'ils peuvent communiquer (leur distance est inférieure au seuil).

III.6 Graphes de Delaunay

III.6.1 Déplacement d'une nuée d'oiseaux

Le modèle que nous décrivons ici s'inspire de la simulation d'une nuée d'oiseaux proposée par [Wil98b] et des déplacements des *boids*, des créatures génériques simulées évoluant en groupe [Rey95]. L'objectif est d'imiter les déplacements d'une nuée d'oiseaux. Pour gérer les déplacements de nos « oiseaux » , nous utiliserons un algorithme très similaire à ceux des travaux cités. Le résultat de la simulation peut être rapproché également des mouvements d'un banc de poissons. Les attroupements qui apparaissent dans ce modèle ne sont ni créés, ni dirigés par un oiseau en particulier. Au contraire, chaque oiseau suit exactement les mêmes règles d'évolution, menant à l'émergence d'un groupe et d'un comportement de groupe. Le modèle est constitué de trois comportements simples décrivant les déplacements possibles d'un *boid* en fonction des positions et des déplacements des *boids* avoisinants :

séparation : lorsqu'un oiseau est trop près de l'un de ses voisins, il change de direction ;

cohésion : lorsqu'un oiseau est trop loin de ses voisins, il essaye de rejoindre rapidement son voisin le plus éloigné ;

alignement : lorsqu'un oiseau n'est ni trop près, ni trop loin de ses voisins, il adopte une direction égale à la moyenne des directions de ses voisins.

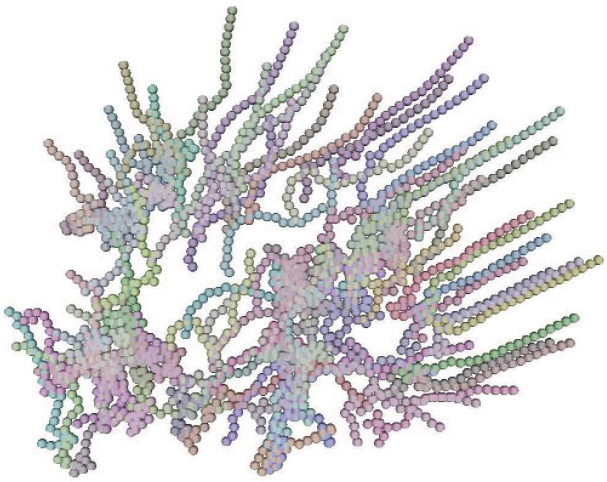


Fig. 14 – Déplacement d'une nuée d'oiseaux.

Pour simplifier la simulation, nous la réaliserons dans un plan au lieu de l'espace. Tous les oiseaux suivent une direction qui leur est propre à une vitesse fixée et identique. La vitesse d'un oiseau augmente lorsqu'il tente de rattraper ses voisins. Les voisins d'un oiseau dans cet espace 2D sont calculés selon une triangulation de Delaunay. Dans MGS, les *graphes de Delaunay* sont des collections de points (de l'espace euclidien) dont la relation de voisinage est déterminée par une triangulation de Delaunay. C'est ce type de collection que nous utilisons pour représenter les oiseaux.

L'image de la figure 14 représente la trajectoire de 50 oiseaux. À l'instant initial, les positions et les vitesses des oiseaux sont choisies aléatoirement. Après quelques étapes de simulation, la

trajectoire devient uniforme et un groupe cohérent émerge.

Chaque oiseau est représenté par un enregistrement constitué de sa position (x, y) dans l'espace euclidien et de la direction `theta` dans laquelle il se déplace. Nous déclarons ci-dessous le type d'enregistrements `Position` correspondant. Le nom de ce type peut être utilisé come un prédicat pour tester si une valeur est bien de ce type.

```
state Position = {x, y, theta};;
```

Pour définir une collection correspondant à un graphe de Delaunay en MGS, on doit donner une fonction permettant d'extraire la position euclidienne des éléments contenus dans la collection. Ici, la fonction définissant le type de graphes de Delaunay `E2` renvoit une séquence de deux éléments (on est en deux dimensions) correspondant aux coordonnées euclidiennes lues dans l'enregistrement, après avoir vérifié que la valeur est bien du type `Position`.

```
delaunay(2) E2 =
\e.if Position(e)
   then (e.x, e.y)
   else ?("bad element type for E2 delaunay type")
   fi ;;
```

Nous définissons à présent une fonction `distance` attendant deux enregistrements comportant des champs `x` et `y` :

```
fun distance(a, b) =
    let dx = a.x - b.x and dy = a.y - b.y
    in sqrt(dx*dx + dy*dy) ;;
```

Les fonctions ci-dessous seront passées en argument à la fonction `neighborsfold` dans les transformations régissant le comportement des oiseaux.

```
fun too_close (orig, a, acc) = acc || (distance(orig, a) < d_sep);;

fun too_far (orig, a, acc) = acc && (distance(orig, a) > d_bound);;

fun closest_bird(orig, a, acc) =
    let dist1 = distance(orig, a)
    in if dist1 < acc.dist then a+{dist = dist1} else acc fi ;;

fun farthest_bird(orig, a, acc) =
    let dist1 = distance(orig, a)
    in if dist1 > acc.dist then a+{dist = dist1} else acc fi ;;
```

La fonction `neighborfold` est un itérateur (un *fold*) sur les voisins d'un élément dans une collection[1].

[1]Il ne s'agit pas à proprement parler d'une fonction mais d'une *forme spéciale*. En effet, elle n'accepte pas en argument n'importe quelle valeur mais uniquement des valeurs associées à une variable provenant d'un motif filtré. Par conséquent, `neighborfold` ne peut être utilisée qu'en partie droite d'une règle de transformation ou dans une garde de motif.

La transformation `Flocking` est utilisée pour simuler une étape d'évolution sur une collection du type `E2`. Elle est composée de trois règles décrivant les comportements possibles d'un oiseau :

```
trans Flocking = {

  (* separation *)
    a / neighborsfold(too_close(a), false, a)
    => begin
          let b = neighborsfold(closest_bird(a), {dist = 2*d_sep}, a) in
          let dir = if (random(2) == 0)
                    then b.theta + 'PI_2
                    else b.theta - 'PI_2 fi
          in a + {x = a.x + speed*cos(dir),
                  y = a.y + speed*sin(dir),
                  theta = dir}
       end;

  (* cohesion *)
    a / neighborsfold(too_far(a), true, a)
    => begin
          let b = neighborsfold(farthest_bird(a), {dist = 0}, a) in
          let dir = atan2(b.y - a.y, b.x - a.x)
          in a + {x = a.x + speed2*cos(dir),
                  y = a.y + speed2*sin(dir),
                  theta = dir}
           end;

  (* alignment *)
    a => begin
            let phi = neighborsfold( (\x.\y.a.theta+y) , 0, a)
            and nb = neighborsfold( (\x.\y.y+1) , 0, a) in
            let dir = phi / nb
            in a + {x = a.x + speed*cos(dir) + random(bruit),
                    y = a.y + speed*sin(dir) + random(bruit),
                    theta = dir}
         end;
};;
```

La règle *separation* s'applique à un oiseau `a` trop près de l'un de ses voisins (deux oiseaux sont trop près si leur distance est inférieure à un « seuil d'oppression » `d_sep`). L'oiseau le plus proche est sélectionné en utilisant un `neighborfold` ayant comme argument la fonction `closest_bird` ainsi qu'une valeur initiale constituée d'un enregistrement comportant un champ `dist` plus grand que le seuil d'oppression.

Lorsque cette règle s'applique, l'oiseau adopte l'une des deux directions orthogonales à la direction de l'oiseau trop proche (une direction est un angle entre 0 et 2π).

La règle *cohesion* s'applique à un oiseau trop loin. Sa direction devient la direction menant à son voisin le plus éloigné et son déplacement est réalisé à une vitesse plus grande.

Lorsqu'aucune de ces deux règles ne s'applique, la règle par défaut effectue l'alignement de la direction de l'oiseau sur la moyenne des directions de ses voisins (plus un peu de bruit afin d'avoir une trajectoire moins régulière).

```
pre_init := map(generate_bird, iota(nb_birds, seq:()));;

init := delaunayfy(E2:(), pre_init);;
```

La fonction `generate_bird` calcule un oiseau à une position aléatoire et suivant une direction aléatoire. Une séquence initiale d'oiseaux est générée en appliquent cette fonction à une séquence d'entiers, cette séquence d'oiseaux étant transformée en un Delaunay `E2` par la primitive `delaunayfy`.

III.6.2 Croissance végétale

Nous modélisons ici la croissance d'un feuillet de cellules en interaction. Cette simulation est un exemple de système dynamique à structure dynamique utilisant des graphes de Delaunay. Nous prenons en comptes trois phénomènes : la tension et la pression entre les cellules, la diffusion et la réaction de deux produits chimiques, et la division des cellules déclenchée par la forte concentration de l'un des deux produits. Le point intéressant est la variation du voisinage d'une cellule au fil des divisions successives. La figure 15 donne une visualisation du résultat de la simulation.

Nous allons utiliser une triangulation de Delaunay pour calculer le voisinage des cellules. Nous commençons par définir le type servant à représenter une cellule :

```
state MecaCell = {x, y, z, vx, vy, vz, fx, fy, fz};;
state BioCell = {a, b, da, db, c};;
state Cell = MecaCell + BioCell;;
```

Ces déclarations spécifient trois types d'enregistrements. Le premier, contenant les champs `x`, `y`, `z`, etc., représente la position d'une cellule ainsi que sa vitesse et son accélération. Le second, contenant les champs `a`, `b`, etc., représente les deux produits chimiques et leur dérivée première. Enfin, le type d'enregistrements `Cell` contient les mêmes champs que `MecaCell` et `BioCell`. À présent nous définissons un type de graphes de Delaunay :

```
collection delaunay(3) D3 =
  \e.if Cell(e) then (e.x, e.y, e.z) else ?("bad element for D3 type") fi ;;
```

Ceci définit un type de graphes de Delaunay à trois dimensions nommé `D3`. Les mouvements des cellules résultent des forces d'élasticité et de viscosité de l'équation de la dynamique de Newton :

$$m.\frac{d^2F}{dt^2} = F_{\text{elastic}} + F_{\text{viscous}} = -k(L - L_0) - \mu\frac{dF}{dt}$$

Cette équation est paramétrée par la constante d'élasticité k, le coefficient de viscosité μ et l'espacement au repos L_0. Elle est intégrée en chaque cellule par la transformation suivante :

```
trans Meca = {
  e => let f = neighborsfold(sum(e), {fx=0,fy=0,fz=0}, e)
       in  e+{ x = e.x + dt*e.vx,
```

```
                ...
                vx = e.vx + dt*f.fx,
                ...
                fx = f.fx,
              }
}
```

La primitive `neighborsfold` est un itérateur sur les cellules voisines d'une cellule `e`. L'expression dans le `let` local calcule la somme des forces d'interaction entre une cellule `e` et ses cellules voisines. Cette somme est ensuite utilisée pour mettre à jour l'état de `e`. Rappelons que l'opérateur `+` sur les enregistrements est asymétrique : si `r` vaut `r1+r2`, alors `r.a` vaut `r2.a` si `a` est un champ de `r2`, et `r1.a` sinon.

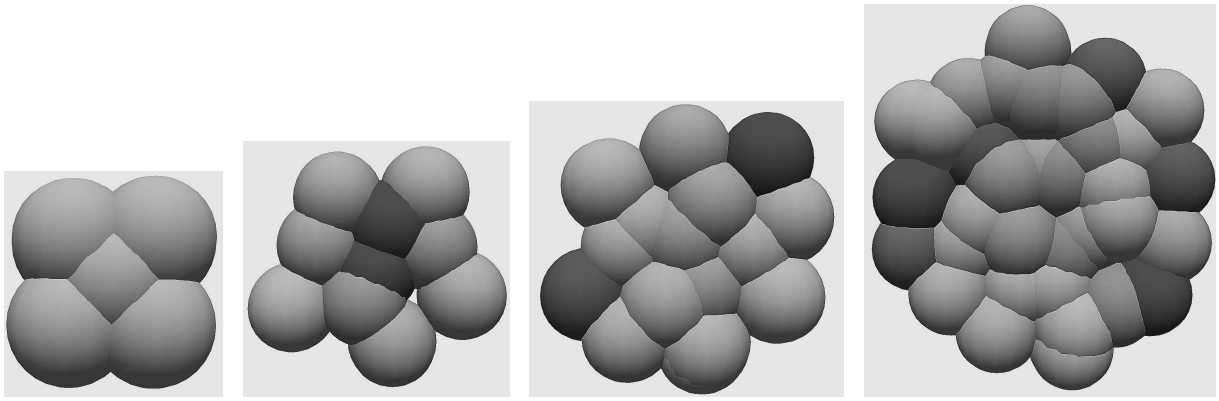


FIG. 15 – Quatre étapes dans la croissance d'un feuillet de cellules. La couleur d'un cellule est liée à la concentration du produit chimique déclenchant la division (les cellules noires sont celles qui vont se diviser).

La réaction et la diffusion des deux produits est fondée sur le modèle présenté en section III.4.5. Lorsque la concentration en `b` atteint un niveau donné, la cellule se divise. La transformation suivante implémente ceci :

```
trans Division = {
  e / e.b > lsplit => (e + {a=e.a/3, b=e.b/3}),
                      (e + {a=e.a/3, b=e.b/3, x=noise(e.x), y=noise(e.y), ...})
}
```

Le coefficient 3 utilisé dans le calcul de la répartition des concentrations dans les cellules filles est arbitraire. La fonction `noise` permet d'ajouter un peu de bruit à la simulation.

III.6.3 Chemin hamiltonien dans un graphe

Un chemin hamiltonien dans un graphe est un chemin passant une et une seule fois par chaque nœud du graphe. Si un graphe a n nœuds, le motif `(x* as X / size(X)=n)` permet de filtrer un chemin hamiltonien de ce graphe. En effet, les nœuds filtrés étant consommés (un même

nœud ne peut apparaître plusieurs fois dans un chemin filtré), on obtient bien avec ce motif un chemin passant une et une seule fois par chaque sommet du graphe, lorsqu'un tel chemin existe.

La transformation suivante trouve un chemin hamiltonien dans un graphe sans connaître à l'avance sa taille :

```
fun hamilton (c) = ( trans { x* as X/ size(X)=size(c) => e } ) (c) ;;
```

On peut l'écrire de manière plus concise en utilisant l'identificateur `self` référant toujours à la collection à laquelle la transformation est appliquée :

```
trans hamilton = { x* as X/ size(X)=size(self) => e } ;;
```

Cet exemple montre que certains problèmes s'expriment de manière très consise en MGS tout en restant lisibles. Toutefois, la concison ne doit pas faire oublier la complexité du problème. La recherche d'un chemin hamiltonien est un problème NP-complet même si une seule ligne de code suffit à le résoudre.

III.7 Exemples de programmes polytypiques

Les exemples de programmes que nous avons donnés dans ce chapitre sont en général conçus pour être appliqués sur une structure de données particulière. Toutefois, on peut également concevoir des programmes MGS qui ont un sens sur toutes les collections topologiques (nous les appellerons des programmes *polytypiques*[2]). Ce point sera plus largement discuté dans les chapitre IV et VII sur le filtrage et le typage mais regardons deux exemples de tels programmes.

III.7.1 Map

La fonction *map* est un exemple trivial de transformation qui s'écrit uniformément sur toutes les collections en MGS. Ainsi écrite, elle peut s'appliquer par exemple à des ensembles, des multi-ensembles, des séquences et aux généralisations des tableaux que nous introduisons plus loin.

```
trans map (f) = { x => f(x) ;};;
```

Tout élément d'une collection est filtré par le motif `x` et donc tous les éléments de la collection sont substitués par leur image par `f`.

III.7.2 Colonie de fourmis

L'exemple de la colonie de fourmis est polytypique. Il montre que les programmes polytypiques ne sont pas réduits à effectuer des taches très simples comme le fait la fonction *map*.

Ce programme MGS a été développé pour la première fois en ayant à l'esprit une grille carrée. Ce premier programme pouvait être exécuté sur tout autre sorte de GBF mais les résultats de la simulation étaient faussés pour la raison suivante : nous avions utilisé le fait qu'une position a quatre voisins pour calculer la moyenne de la quantité phéromone au voisinage d'une position. En comptant le nombre de voisins (à l'aide de la primitive `neighborfold`) on peut compter

[2]Le mot a été utilisé pour la première fois dans [JJ96]. Toutefois il lui était donné alors un sens plus restrictif.

le nombre de voisins et donc programmer une moyenne sans connaître *a priori* la structure du voisinage (on utilise également `neighborfold` pour sommer une valeur sur ces voisins).

Cette modification effectuée, le programme donne des résultats cohérents sur une grille carrée, sur un pavage hexagonal ou sur un maillage irrégulier du plan (proxima ou Delaunay par exemple).

Deuxième partie

L'interprète MGS

Chapitre IV

Filtrage

Les travaux présentés dans ce chapitre ont été publiés dans [GMC02a] et [CMG03].

Nous présentons dans ce chapitre le filtrage sur les collections topologiques. Les collections topologiques apportent un point de vue unifié sur les structures de données qui permet de définir le filtrage de façon uniforme. En effet, un seul langage de motifs et un seul algorithme de filtrage suffisent pour construire des fonctions définies par cas sur toutes les structures de données du langage. Ceci signifie qu'une même fonction peut opérer sur des ensembles, des tableaux, des arbres, des graphes, *etc.* La généricité des motifs et de l'algorithme de filtrage ne se fait pas au détriment de la richesse des motifs utilisables. Au contraire, le langage considéré est très riche, comme l'ont montré les exemples de transformations du chapitre précédent.

Dans la première section de ce chapitre nous précisons ce que signifie l'idée de filtrer *uniformément* sur diverses structures de données. Nous y décrivons également les caractéristiques que nous souhaitons rendre disponibles dans les motifs. Dans la section IV.2 nous posons la notion de collection topologique sur laquelle nous travaillons dans ce chapitre. La section IV.3 introduit formellement la notion de chemin et présente l'algèbre de motifs considérée. La section suivante décrit l'algorithme de filtrage générique. La notion de topologie et son rôle dans les transformations sont donnés en section IV.5. Nous illustrons en section IV.6 les notions présentées en montrant comment les GBF s'y intègrent. Nous terminons ce chapitre en essayant d'introduire de nouvelles familles de structures de données dans le cadre des collections topologiques (section IV.7) et en dressant un bref bilan (section IV.8).

IV.1 Filtrage générique et polytypisme

Les itérateurs comme *map* et *fold* sont parmi les premiers exemples de programmes que le programmeur est amené à définir à plusieurs reprises. En effet, lorsqu'on déclare un nouveau type de structures de données on a souvent besoin de les redéfinir alors que leur nature ne diffère pas d'une structure à l'autre. Les mécanismes permettant de définir des algorithmes adaptés à diverses structures de données sont actuellement l'objet de nombreux travaux de recherche. La discipline correspondante est appelée *polymorphisme structurel* [Rue92, Rue98, Gar02], *typage paramétrique* [She93], *polymorphisme de forme* (*shape polymorphism* en anglais) [JC94, JBM98, CF92], *polymorphisme intentionnel* [HM95], *polytypisme* [JJ96, Jeu00] ou *programmation générique* [Jay04, HJ03]. Les travaux autour de la notion d'itérateurs développés par exemple en C++ dans la *StdLib* peuvent également être vus comme une forme de polytypisme.

Les travaux autour du polytypisme (nous retiendrons cette appellation) se concentrent essentiellement sur les types de données algébriques. De nombreuses structures de données ne peuvent se ramener à des termes et cette approche est donc assez restrictive. Au contraire, notre approche du polytypisme n'est pas fondée sur la notion de type algébrique.

L'approche habituelle du polytypisme est la suivante. Le programmeur déclare une fonction polytypique en donnant un modèle d'algorithme et en décrivant comment celui-ci s'instancie en fonction des constructeurs associés à un type. Il s'agit d'une forme de surcharge. En effet le compilateur crée un code spécialisé de la fonction pour chaque type de structure sur laquelle elle est utilisée. La façon d'instancier le code de la fonction pour un nouveau type de structure étant décrite par le programmeur on parle de polytypisme plutôt que de simple surcharge. En effet dans la plupart des langages où le programmeur peut déclarer des fonctions surchargées, celui-ci doit donner le code de la fonction pour chaque type de structure sur laquelle elle est utilisable.

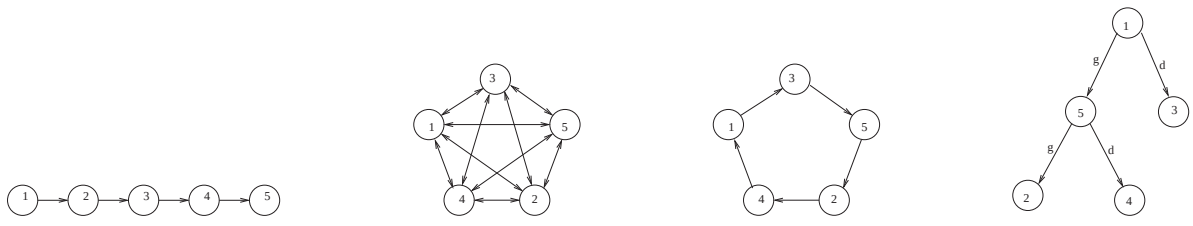
Notre approche est différente. Nous adoptons un point de vue similaire à celui de [JC94] où les données et la structure sont deux composantes d'une *structure de données* et nous considérons toutes les structures sous un aspect topologique. Ainsi le filtrage n'opère pas différemment d'une structure à l'autre. Il n'est donc pas nécessaire de produire du nouveau code pour une nouvelle structure de données. Notre point de vue du polytypisme tient donc plutôt du polymorphisme paramétrique (un seul code pour toutes les utilisations).

Toutefois il ne s'agit pas exactement de polymorphisme paramétrique car s'il est vrai qu'un seul code suffit pour filtrer sur toutes les structures de données, nous verrons qu'un code différent sera utilisé pour opérer le remplacement des parties filtrées par des nouvelles valeurs. Ainsi, une partie du polymorphisme des fonctions polytypiques reste *ad hoc*. Par ailleurs nous verrons dans le chapitre IX que nous n'hésiterons pas à adopter un algorithme de filtrage plus efficace sur certaines structures de données.

L'ajout de polytypisme dans un langage de programmation influence fortement la discipline de typage qui lui est associée. Nous ne manquerons pas de reparler de polytypisme dans le chapitre VII consacré au typage de notre langage.

IV.2 Collections topologiques et relation de voisinage

Dans ce chapitre, nous verrons les collections topologiques comme des graphes orientés, étiquetés et partiellement valués (Fig. 1).



(a) séquence (b) ensemble (c) liste circulaire (d) arbre

Fig. 1 – Collections topologiques vues comme des graphes

Les nœuds du graphe sont appelés des *positions*. Si une position p est valuée par une valeur v on dira que p possède la valeur v ou possède une valeur. Les étiquettes des arcs sont appelées des *directions*. Lorsque ce n'est pas nécessaire, nous ne reporterons pas les directions sur nos schémas.

Relation de voisinage

On dit qu'une positions p_2 est voisine d'une position p_1 s'il existe un arc $\langle p_1, p_2\rangle$ dans le graphe. Cette relation n'est pas symétrique puisque le graphe est orienté. La relation de voisinage *vois* que nous considérons est donc la relation d'adjacence du graphe : $vois(p_1, p_2)$ *ssi* $\langle p_1, p_2\rangle$ est un arc du graphe.

On dit que p_2 est voisine de p_1 selon la direction d lorsque l'arc $\langle p_1, p_2\rangle$ est étiqueté par d. Dans ce cas on dit que p_2 est un d-voisin de p_1.

On dit enfin qu'une position p_2 est *connectée* à une position p_1 si elle est en relation avec p_1 par la clôture transitive de la relation de voisinage : $vois^+(p_1, p_2)$.

IV.3 Chemins et motifs de chemins

IV.3.1 Chemins

Un *chemin* est une suite de positions distinctes $p_1, p_2, \ldots, p_n$ telle que pour tout i entre 1 et $n-1$, p_{i+1} est voisine de p_i.

Lorsque les positions successives ne sont pas voisines mais connectées, on parlera de *pseudo-chemin*.

IV.3.2 Motifs

Nous présentons un langage de motifs de chemins pour filtrer des parties de collections. En effet un chemin permet de décrire une sous-collection tout en spécifiant un ordre de parcours. La grammaire est la suivante :

Motifs :	m	::=	μ | $\mu\,\delta\,m$ | m `as` id | `(`m`)` | `(`m_1`|`m_2`)`
Motifs élémentaires :	μ	::=	q | id/e | $q\,\delta$`+` | $q\,\delta$`*`
Qualité :	q	::=	c | id | `_` | `<undef>`
Voisinage :	δ	::=	`,` | `|`d_1`|...|`d_n`>` | δ `as` id
	e	::=	expressions MGS
	d	::=	direction
	c	::=	constante
	id	::=	identificateur

Un motif de chemin m peut être un motif élémentaire μ comme une constante c, une variable gardée ou non, une variable anonyme « _ », une position sans valeur `<undef>` ou une répétition. Un motif peut également être construit par un motif élémentaire et un motif séparés par un voisinage δ, en donnant un nom à un sous-motif ou par une alternative entre deux motifs m_1 et m_2 notée `(`m_1`|`m_2`)`. Détaillons l'interprétation intuitive des motifs :

variable : Une variable de motif x filtre une position p de la collection possédant une valeur v. L'identificateur x peut alors être utilisé dans une garde ou en dehors du motif pour dénoter la valeur filtrée v.

Si la variable n'apparaît pas ailleurs, on peut utiliser le motif anonyme dénoté par le symbole « `_` » afin de ne pas nommer inutilement une position filtrée.

constante : Un motif c, où c est une constante, filtre une position dont la valeur est égale à la constante c.

vide : Le motif `<undef>` filtre une position qui ne possède pas de valeur. En pratique, on n'utilisera `<undef>` dans un motif que comme voisin d'un élément bien défini (*i.e.* différent de `<undef>`). Cette restriction fait l'objet de la discussion 1, page 73.

voisinage : Le motif $\mu\,\delta\,m$ filtre un chemin $p_0, \ldots, p_k, p_{k+1}, \ldots, p_l$ où $p_0, \ldots, p_k$ (resp. p_{k+1}, $\ldots$, p_l) est filtré par μ (resp. m) tel que p_k et p_{k+1} soient voisins selon l'une des directions de δ. Si δ est la virgule « , » alors p_k et p_{k+1} peuvent être voisins selon n'importe quelle direction. Si δ est de la forme `|`d_1`|...|`d_q`>` alors p_k et p_{k+1} doivent être voisins selon l'une des directions d_i.

Par exemple `x|e>_|n|-n>y` filtre trois positions telles que la seconde est un `e`-voisin de la première et la troisième est un `n`-voisin ou un `(-n)`-voisin de la seconde.

garde : Le motif x/e filtre une position vérifiant l'expression e. Par exemple `x,y/x>y` filtre deux positions p_0 et p_1 telles que la valeur en p_0 est plus grande que la valeur en p_1. La condition e peut être un test sur le type de la valeur filtrée comme dans $x/(x : int)$.

nommage : La construction m `as` x sert à lier x aux éléments du chemin filtré par m. On peut écrire par exemple `0,(1,x|n>3) as z,4` et `z` dénotera en dehors du motif la *séquence* composée des valeurs filtrées par `1,x|n>3`.

On peut également lier une direction. Par exemple le motif `x (, as d) y /d=nord` est équivalent au motif `x|nord>y`.

répétition : Le motif $b\,\delta*$ peut filtrer un chemin vide, une position filtrée par b ou une suite de positions, chacune filtrée par b et chacune voisine de la précédente selon l'une des directions de δ. Le symbole « `*` » (étoile) est utilisé en référence aux expressions régulières où elle exprime une répétition éventuellement nulle.

Si b est une variable, celle-ci ne sera pas liée dans le reste du motif (à moins d'être liée par ailleurs). Seul l'ensemble du chemin filtré peut être lié en utilisant un nommage comme dans $(x\,\delta*)$ `as` y. Dans cet exemple, y est lié à la *séquence* formée des valeurs du chemin filtré.

Lorsque δ est la virgule alors on peut utiliser le raccourci syntaxique $x*$.

Le motif $x\,\delta+$ filtre les mêmes chemins que $x\,\delta*$ sauf le chemin vide.

alternative : Le motif $(m_1|m_2)$ filtre les chemins filtrés par m_1 et les chemins filtrés par m_2.

pseudo chemin : Les motifs de pseudo-chemins sont une extension des motifs de chemins où l'on peut faire apparaître des *combinaisons de directions* pour exprimer le pseudo-voisinage entre deux positions filtrées.

$$\delta \;::=\; , \;\mid\; \texttt{|}\; d_1 \;\texttt{|}\; \ldots \;\texttt{|}\; d_n \;\texttt{>} \;\mid\; \delta \;\texttt{as}\; \mathrm{id}$$

où les d_i sont des combinaisons de directions de la forme $d_{i1} + \cdots + d_{im_i}$.

Par exemple le motif `x|g+g>y` filtre deux positions p_x et p_y telles qu'il existe une position p étant g-voisine de p_x et ayant p_y comme g-voisin. La position p ne fait pas partie du chemin filtré. Ce motif filtre donc deux positions pseudo-voisines sans que la position intermédiaire ne soit comprise dans le chemin filtré, contrairement à ce qu'on aurait obtenu avec le motif `x|g>_|g>y`.

Le filtrage consomme les positions : une même position ne peut apparaître deux fois dans un chemin filtré. Un nom de variable ne peut être utilisé qu'une fois dans un motif excepté comme argument d'une garde (linéarité gauche au sens de [DJ90]). Ainsi le motif `x,x` n'est pas valide mais on peut écrire `x,y/y=x` pour filtrer deux positions voisines qui possèdent des valeurs identiques.

Dans la règle de transformation `x/x>0 => x+1` le motif `x/x>0` filtre un chemin composé d'une position p et `x` dénote la valeur associée à la position p dans la collection filtrée. Pour que le motif soit filtré avec succès et que cette règle s'applique il faut que la valeur à la position filtrée soit positive (`x>0`).

Comparaison avec d'autres langages de motifs

La figure 1 compare les caractéristiques du langage de motifs que nous proposons avec quelques langages dotés de filtrage : ELAN pour les moteurs de récriture (Maude et CiME ont un niveau d'expressivité similaire), TOM pour les compilateurs de filtrage et OCaml pour les langages de la famille ML.

	ELAN	TOM	OCaml	MGS
constantes/variables	oui	oui	oui	oui
gardes	oui	oui	oui	oui
répétition	non	oui	non	oui
nommage	externe	oui	oui	oui
vide (au sens de `<undef>`)	n.s.	n.s.	n.s.	oui
Voisinage				
directions	A/C	A	non A, non C	oui
voisinage quelconque	non	non	non	oui
nommage de directions	n.s.	n.s.	n.s.	oui
alternative	stratégie	non	oui	oui
combinaisons	non	non	non	oui

Tab. 1 – Expressivité des motifs MGS en comparaison avec d'autres outils

Dans le tableau de la figure 1, *n.s.* signifie qu'un tel motif n'aurait pas de sens. Par exemple, on ne peut pas filtrer de position vide dans ELAN puisqu'il n'y a pas de notion de vide dans les termes manipulés par ce langage.

IV.4 Algorithme générique de filtrage

Nous présentons dans cette section un algorithme générique pour le filtrage des motifs de chemins sur les collections. Cet algorithme est inspiré métaphoriquement de la technique de dérivation d'expressions régulières de J.A. Brzozowski [Brz64] que nous commençons par rappeler. Ensuite nous nous intéressons à un sous-ensemble minimal complet du langage de motifs puis nous détaillons l'algorithme sur ce sous-ensemble.

IV.4.1 Dérivée d'une expression régulière

Soit R une expression régulière sur un alphabet A et L_R le langage reconnu par R. La dérivée de R par rapport à un symbole a de A est notée $\frac{\partial R}{\partial a}$ et est définie par :

$$\frac{\partial R}{\partial a} = \{m \in A^* \,/\, am \in L_R\}$$

La dérivée d'une expression régulière peut aussi s'exprimer sous la forme d'une expression régulière. Par exemple :

$$\frac{\partial a.(a+b)*}{\partial a} = (a+b)*$$

Autrement dit le mot am est reconnu par $a.(a+b)*$ si et seulement si m est reconnu par $(a+b)*$.

De manière générale, un mot $m = a_1 a_2 \dots a_n$ est reconnu par une expression régulière si et seulement si $a_2 \dots a_n$ est reconnu par la dérivée de R par rapport à a_1. Ceci permet de ramener le problème de l'appartenance d'un mot à un langage au problème d'appartenance du mot vide ε à un autre langage.

L'expression R' dérivée de R peut être calculée en utilisant des règles inductives sur la structure de R. Ces règles suivent un schéma proche des règles de calcul formel de la dérivée d'une fonction réelle, d'où le terme de dérivée. Ces règles sont données dans la figure 2.

$$\frac{\partial\, a}{\partial\, a} = \varepsilon \qquad \frac{\partial\, b}{\partial\, a} = \emptyset \text{ (lorsque } b \neq a) \qquad \frac{\partial\, P*}{\partial\, a} = \frac{\partial\, P}{\partial\, a}.P*$$

$$\frac{\partial\, P.Q}{\partial\, a} = \frac{\partial\, P}{\partial a}.Q + [P].\frac{\partial\, Q}{\partial\, a} \qquad \frac{\partial\, P+Q}{\partial\, a} = \frac{\partial\, P}{\partial\, a} + \frac{\partial\, Q}{\partial\, a}$$

FIG. 2 – Dérivation d'une expression régulière

L'annulateur $[R]$ d'une expression R utilisé dans ces règles est défini par :

$$[R] = \begin{cases} \emptyset & \text{si } \varepsilon \notin L_R \\ \{\varepsilon\} & \text{si } \varepsilon \in L_R \end{cases}$$

et peut aussi être calculé en utilisant des règles simples sur la structure de R. L'annulateur est utilisé avec la dérivation pour déterminer la *décomposition canonique* des mots de L_R :

$$L_R = [R] \cup \left(\bigcup_{a \in A} a \otimes \frac{\partial R}{\partial a} \right)$$

où $a \otimes L = \{am / m \in L\}$; notons que $a \otimes \emptyset$ vaut $\emptyset$ et $a \otimes \{\varepsilon\}$ vaut $\{a\}$.

Nous allons adapter ces idées à notre problématique : un motif de chemin jouera un rôle similaire à celui de l'expression régulière et la collection un rôle similaire à celui de l'alphabet. Les différences sont les suivantes :

– Dériver une expression régulière sert à vérifier l'appartenance d'un mot à un langage. Dans notre cas nous nous préoccupons de générer tous les chemins vérifiant une propriété donnée dans une collection.

– Parcourir un mot se fait simplement de gauche à droite alors qu'il n'existe pas de parcours canonique des collections. Toutefois un chemin dénote une partie d'une collection ainsi qu'un parcours de cette partie. Le parcours induit découle de la lecture de gauche à droite du motif.
– Les motifs de chemins peuvent contenir des expressions booléennes dont la valeur n'est calculée que lors du filtrage.

IV.4.2 Chemins filtrés par un motif

Les expressions de motifs

Afin de simplifier la présentation de l'algorithme nous utiliserons une version restreinte du langage de motifs mais presque aussi expressive que la première :

$$\begin{array}{lrcl} \text{Motifs :} & m & ::= & \mu \mid \mu\ \delta\ m \\ \text{Motifs élémentaires :} & \mu & ::= & \mathrm{id}/e \mid \delta\,\texttt{*} \\ \text{Voisinages :} & \delta & ::= & \texttt{|}\ d_1\ \texttt{|}\ \ldots\ \texttt{|}\ d_n\ \texttt{>} \end{array}$$

Ce langage n'est pas équivalent au langage présenté dans la section IV.3.2 mais il permet d'exprimer la majorité des caractéristiques intéressantes de celui-ci. En effet, on peut passer d'un motif du langage complet à un motif du langage simple comme suit :
– Un motif *constante* c devient `x/x=` c où `x` est une variable fraîche ;
– Un motif *variable* `x` devient `x/true` ;
– Un motif *anonyme* « _ » devient `x/true` où `x` est une variable fraîche ;
– Le voisinage « , » se traduit par `|`d_1`|...|`d_n`>` où les d_i sont toutes les directions pouvant apparaître dans une collection (on les suppose en nombre fini) ;
– Une répétition $x\,\delta*$ se code $\delta*$ (et il n'y a plus de liaison à un nom pour la répétition) ;
– Une répétition $x\,\delta+$ s'écrit $y\,\delta\,\delta*$ où y est une variable fraîche. Rappelons que dans le motif $(x\ \delta+)\ \delta'\ m$, la variable x n'est pas liée dans m. Par conséquent nous devons utiliser une variable fraîche dans la nouvelle forme afin de ne pas *capturer* une éventuelle liaison de x préexistante.

Par exemple le motif `x,(_|d>+) |g> y` peut s'écrire avec la nouvelle syntaxe

```
(x/true) |d|g> (z/true) |d> (|d>*) |g> (y/true)
```

s'il est utilisé sur des arbres d/g.

Inclure le nommage des sous-motifs ou des répétitions à cet algorithme ne pose pas de véritable difficulté mais alourdit beaucoup sa définition. Les parenthèses ne sont pas nécessaires et n'ont donc pas été incluses dans la grammaire. Enfin, nous laissons de coté le filtrage des positions sans valeurs par le motif `<undef>`.

Discussion 1 La gestion du motif `<undef>` est complexe, par exemple le filtrage du motif `<undef>,1` peut ne pas terminer si le motif est filtré de gauche à droite alors que le filtrage de `1,<undef>` ne pose pas de problème (car il peut y avoir une infinité de positions vides dans le graphe d'une collection mais pas une infinité de valeurs). Dans [Spi03], des transformations de motifs préservant la sémantique sont proposées, permettant dans certains cas de transformer automatiquement un motif non-valide en un motif valide.

Notations

Des accolades { } sont utilisées pour noter les ensembles en énumération ou en compréhension, $\emptyset$ est l'ensemble vide et $S - e$ est égal à l'ensemble S privé de l'élément e.

[ ] est la liste vide, @ est la concaténation de deux listes. La distribution $e \otimes S$ d'un élément e sur un ensemble S de listes est définie par l'ensemble de listes $\{[e]@l \mid l \in S\}$.

Un environnement est une fonction totale d'un ensemble d'identifiants vers un ensemble de valeurs. L'augmentation d'un environnement E par un couple (i, v) est un nouvel environnement $E' = E + [i \mapsto v]$ tel que $E'(i) = v$ et $\forall j \neq i, E'(j) = E(j)$.

Dérivée d'un motif

Étant donné une collection `c`, un environnement E, et un ensemble Π de positions non encore consommées, la dérivée d'un motif m par rapport à une position p s'écrit :

$$\frac{\partial m}{\partial p}(\texttt{c}, E, \Pi)$$

Celle-ci dénote l'ensemble des chemins dans `c` commençant à la position p, filtrés par le motif m et ne passant que par des positions de Π. E est nécessaire à l'évaluation des gardes dans m. Π permet de savoir quels éléments peuvent être filtrés dans la collection. Le résultat d'une dérivation est un ensemble de chemins *i.e.* de listes de positions.

L'ensemble des chemins filtrés par un motif m dans une collection `c` est calculé par

$$\bigcup_{p \in dom(\texttt{c})} \frac{\partial\, m}{\partial p}(\texttt{c}, \epsilon, dom(\texttt{c}))$$

où ϵ est l'environnement vide et où $dom(\texttt{c})$ est l'ensemble fini des positions dans `c` qui possèdent une valeur.

Le calcul de la dérivée est défini par induction sur le motif dans la figure 3. La fonction `eval(E,c,e)` donne la valeur booléenne de l'expression `e` dans l'environnement `E` et dans la collection `c`. La fonction `vois(c` $, \Pi, \delta, p$`)` donne les positions voisines de la position p dans `c` selon les directions de δ et qui sont dans Π.

Les équations de la figure 3 s'interprètent comme suit :

1-2 : Seul le chemin vide peut être filtré dans une collection vide.

3 : `x/e` filtre le chemin $[p]$ à partir de la position p si et seulement si `e` est vérifié dans E augmenté de la liaison `x` $\mapsto p$.

4 : Les chemins filtrés par `d*` à la position p sont le chemin vide et les chemins filtrés par `x/true d d*` à la position p (`x` est une variable fraîche).

5 : Les chemins filtrés par `(x/e) d M` en p si `e` est vérifiée sont les chemins filtrés par `M` à partir des voisins de p selon la direction `d` auxquels on a ajouté p en tête. Afin que le chemin ne s'intersecte pas, p est retiré de l'ensemble Π des positions disponibles.

6 : Les chemins filtrés par `d* d' M` sont les chemins filtrés par `M` à partir des voisins de p selon `d` (répétition nulle) et les chemins filtrés par `(x/true) d (d* d' M)` (répétition non nulle).

$$\frac{\partial\,\delta*}{\partial p}(\,\mathtt{c}, E, \emptyset) = \{\,[\,]\,\} \tag{1}$$

$$\frac{\partial\,M}{\partial p}(\,\mathtt{c}, E, \emptyset) = \emptyset \quad \text{avec } M \neq \delta* \tag{2}$$

$$\frac{\partial\, id/expr}{\partial p}(\,\mathtt{c}, E, \Pi) = \text{if } eval(E + [id \mapsto p],\, \mathtt{c}, expr) \text{ then } \{\,[p]\,\} \text{ else } \emptyset \tag{3}$$

$$\frac{\partial\,\delta*}{\partial p}(\,\mathtt{c}, E, \Pi) = \{\,[\,]\,\} \cup \frac{\partial\, id/true\ \delta\ \delta*}{\partial p}(\,\mathtt{c}, E, \Pi) \quad \text{où } id \text{ est une variable fraîche} \tag{4}$$

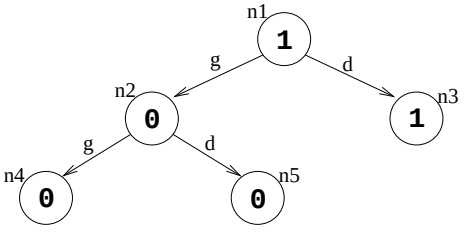
$$\begin{aligned}\frac{\partial\, id/expr\ \delta\ M}{\partial p}(\,\mathtt{c}, E, \Pi) = {} & \text{let } E' = E + [id \mapsto p] \text{ and } \Pi' = \Pi - p \text{ in} \\ & \text{if } eval(E',\, \mathtt{c}, expr) \\ & \text{then } p \otimes \left(\bigcup_{p' \in \mathtt{vois}(\mathtt{c}, \Pi', \delta, p)} \frac{\partial\, M}{\partial p'}(\,\mathtt{c}, E', \Pi') \right) \\ & \text{else } \emptyset\end{aligned} \tag{5}$$

$$\begin{aligned}\frac{\partial\,\delta*\ \ \delta'\ M}{\partial p}(\,\mathtt{c}, E, \Pi) = {} & \left(\bigcup_{p' \in \mathtt{vois}(\mathtt{c}, \Pi, \delta', p)} \frac{\partial\, M}{\partial p'}(\,\mathtt{c}, E, \Pi) \right) \cup \frac{\partial\, id/true\ \delta\ (\delta*\ \ \delta'\ M)}{\partial p}(\,\mathtt{c}, E, \Pi) \\ & \text{où } id \text{ est une variable fraîche}\end{aligned} \tag{6}$$

FIG. 3 – Calcul de la dérivée d'un motif. (On suppose dans les équations 3 à 6 que $\Pi \neq \emptyset$.)

IV.4.3 Exemple

Considérons le motif suivant : `1, _ |d> 0`. Utilisons l'algorithme pour trouver ses instances dans l'arbre `a` ci-dessous :



Le motif se réécrit en P :

$$P = \mathtt{u/u=1\ \ |d|g>}\ Q$$
$$Q = \mathtt{v/true\ \ |d>\ w/w=0}$$

L'ensemble des positions à filtrer est

$$\Pi = dom(\mathtt{a}) = \{n_1, n_2, n_3, n_4, n_5\}$$

Les chemins filtrés par P son calculés par :

$$\frac{\partial\, P}{\partial n_1}(\,\mathtt{a}, \epsilon, \Pi) \cup \frac{\partial\, P}{\partial n_2}(\,\mathtt{a}, \epsilon, \Pi) \cup \frac{\partial\, P}{\partial n_3}(\,\mathtt{a}, \epsilon, \Pi) \cup \frac{\partial\, P}{\partial n_4}(\,\mathtt{a}, \epsilon, \Pi) \cup \frac{\partial\, P}{\partial n_5}(\,\mathtt{a}, \epsilon, \Pi)$$

Calculons la dérivée à partir de n_1 :

$$\frac{\partial P}{\partial n_1}(\,\mathtt{a}, \epsilon, \Pi) = n_1 \otimes \bigcup_{p' \in \{n_2, n_3\}} \frac{\partial Q}{\partial p'}(\,\mathtt{a}, [u \mapsto n_1], \Pi')$$

où Π' vaut $\{n_2, n_3, n_4, n_5\}$. En effet, la valeur en n_1 est bien 1. La dérivée de Q par rapport à n_3 vaut $n_3 \otimes \emptyset$ soit $\emptyset$ car n_3 n'a pas de g-voisin (de fils gauche). En revanche la dérivée de G par rapport à n_2 est

$$\frac{\partial Q}{\partial n_2}(\,\mathtt{a}, [u \mapsto n_1], \Pi') = n_2 \otimes \frac{\partial\, \mathtt{w/w=0}}{\partial n_5}(\,\mathtt{a}, [u \mapsto n_1, v \mapsto n_2], \Pi'')$$

où $\Pi'' = \{n_3, n_4, n_5\}$. En effet, n_5 est le seul d-voisin de n_2. Enfin, la dérivée de `w/w=0` par rapport à n_5 est $\{\,[n_5]\,\}$ car on a bien 0 en n_5. Le résultat de la dérivée de P par rapport à n_1 est donc $\{\,[n_1, n_2, n_5]\,\}$.

Intéressons nous à présent aux dérivées de P par rapport aux autres positions. La dérivée de P par rapport à n_2 est vide puisque la condition $eval([u \mapsto n_2], \mathtt{a}, u = 1)$ échoue (la valeur en n_2 n'est pas 1). Idem en n_4 et en n_5. En revanche, en n_3 la condition n'échoue pas. La dérivée de P en n_3 est $n_3 \otimes \emptyset$ soit $\emptyset$ car n_3 n'a ni d-voisin ni g-voisin. Finalement, l'ensemble des chemins filtrés dans `a` par le motif P est le singleton $\{\,[n_1, n_2, n_5]\,\}$.

IV.4.4 Filtrage de motifs de pseudo-chemins

L'algorithme proposé calcule les pseudo-chemins filtrés par un motif de pseudo-chemins si l'on considère que `vois(c,`Π, δ, p`)` calcule les positions de Π connectées à p dans `c` suivant les combinaisons de directions de δ.

IV.5 Topologies et transformations

L'opération de filtrage est générique : elle fonctionne de la même manière quelle que soit la collection considérée. L'application d'une transformation fait également intervenir un mécanisme de remplacement des parties filtrés. Ce mécanisme diffère selon les propriétés topologiques de la collection considérée. Ce mécanisme est appelé une *substitution topologique*. Dans cette section, nous commencerons par décrire la notion de substitution topologique. Ensuite nous introduirons la notion de *topologie* qui repose sur les substitutions topologiques. Enfin nous décrirons comment ces deux notions interviennent dans le processus d'application des transformations.

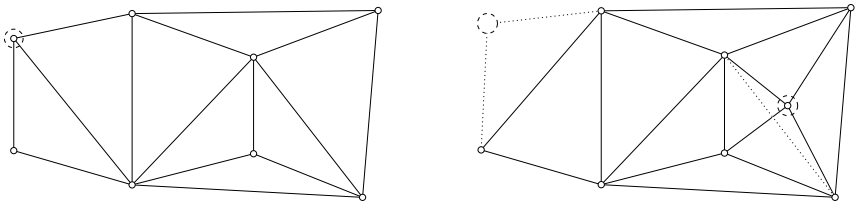
IV.5.1 Substitutions topologiques

Définition 1 Une *substitution topologique* est une fonction qui prend en arguments une collection topologique c, un chemin de c et une suite de valeurs dite *de remplacement* et renvoie une nouvelle collection topologique. La collection renvoyée doit :

- contenir les valeurs de remplacement données en arguments,
- contenir les positions de c qui ne sont pas dans le chemin passé en argument. Leurs valeurs doivent être inchangées.

Dans cette définition, rien n'est dit sur la relation de voisinage. Spécifier par exemple que lorsque deux positions non filtrées sont voisines dans la collection en argument, elles doivent être voisines

dans la collection renvoyée, serait trop restrictif. 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 récriture de graphes [Cou93].

Les substitutions topologiques peuvent ne pas être définies sur certaines entrées. On considérera que les substitutions topologiques peuvent opérer plusieurs remplacements chemins/valeurs à la fois. Certaines substitutions topologiques peuvent être définies également sur des pseudo-chemins.

Exemple : substitution newtonienne

La substitution topologique $\Psi_{\mathcal{N}}$ dite *newtonienne* prend une collection c, un chemin $p_1, \ldots, p_l$ de longueur l et une suite de l valeurs de remplacement $v_1, \ldots, v_l$ et renvoie une collection identique à c excepté aux positions p_i (i entre 1 et l) qui possèdent respectivement les valeurs v_i. Cette substitution topologique n'est pas définie si le chemin et la suite de valeurs n'ont pas le même nombre d'éléments.

Remarque : la substitution newtonienne $\Psi_{\mathcal{N}}$ accepte en argument des pseudo-chemins car la substitution se faisant point-à-point aucune ambigüité n'est possible.

Collections newtoniennes et collections leibnitziennes

La substitution newtonienne est utilisée sur des types de collections dont l'ensemble de positions et la relation de voisinage ne changent jamais. Nous qualifions ces types de collections de newtoniens car Newton pensait que l'espace existait indépendamment des phénomènes qui pouvaient y prendre place (espace absolu). À l'opposé, Leibnitz pensait que l'espace ne prenait de sens que comme support de phénomènes physiques [Jam93].

Nous qualifions de leibnitziennes les familles de collections où les positions ne sont présentes que pour porter des valeurs et sont détruites lorsqu'elles ne sont plus nécessaires.

Les séquences sont un exemple de collections leibnitziennes. Une séquence de n valeurs est représentée par une collection à n positions. Ainsi, toute position porte une valeur dans une séquence. Remarquons qu'une séquence ainsi définie est une notion différente d'un GBF à une direction, qui est une collection newtonienne plus proche de la notion de vecteur. Les ensembles, les multi-ensembles et les graphes de Delaunay sont d'autres exemples de collections leibnitziennes.

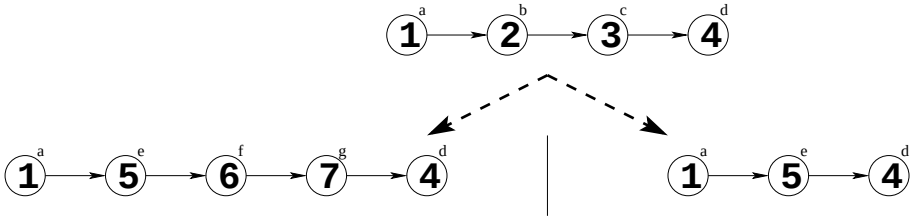
Exemple : substitution leibnitzienne $\Psi_{\texttt{seq}}$

La substitution leibnitzienne $\Psi_{\texttt{seq}}$ est utilisée sur les séquences. Elle prend un chemin $p_1, \ldots, p_n$ dans une collection c et une suite de valeurs $v_1, \ldots v_m$ et renvoie une collection c' telle que ;

- Les positions p_i $(i \leq n)$ ne sont pas dans c'.
- Les autres positions de c sont dans c'.
- Il existe des positions p'_i $(i \leq m)$ dans c' telles que p'_i possède la valeur v_i.
- Il n'existe pas d'autres positions dans c'.
- Pour $i \leq m-1$, p'_{i+1} est voisine de p'_i.
- Pour toute position p'' voisine de p_n dans c (il y en a 0 ou 1 si c est une séquence), p'' est voisine de p'_m dans c'.
- Pour toute position p'' telle que p_1 est voisine de p'' dans c, (il y en a 0 ou 1 si c est une séquence), p_1 est voisine de p'' dans c'.
- Toutes les relations de voisinage (créées) portent la direction *left*.

Lorsque n est différent de m, la collection renvoyée n'a pas le même nombre de positions (et donc de valeurs) que la collection en argument.

Exemple : Ci-dessous l'application de $\Psi_{\texttt{seq}}(\,[b,c],\,[5,6,7]\,)$ et de $\Psi_{\texttt{seq}}(\,[b,c],\,[5]\,)$ à la séquence contenant les valeurs 1, 2, 3 et 4 aux positions a, b, c et d.



IV.5.2 Topologies

Une *topologie* est la donnée d'opérateurs (dont les constructeurs) et d'une substitution topologique. L'utilisation des fonctions d'une topologie pour construire une collection induit des propriétés sur celle-ci. Par exemple une collection créée à partir de la séquence vide, du constructeur de séquences et de la substitution topologique $\Psi_{\texttt{seq}}$ est appelée une séquence et a les propriétés suivantes :

- toute position a au plus un voisin ;
- toute position est voisine d'au plus une position ;
- il n'y a pas de cycle dans la relation de voisinage ;
- il n'y a qu'une composante connexe ;
- toutes les positions possèdent une valeur ;
- tous les arcs sont étiquetés par la direction *left*.

La topologie `seq` est ainsi définie par la donnée de la séquence vide, du constructeur de séquences et de la substitution topologique $\Psi_{\texttt{seq}}$.

$$\texttt{seq} = \langle [\texttt{empty-seq},\texttt{cons-seq}], \Psi_{\texttt{seq}} \rangle$$

On pourra utiliser la lettre ρ pour dénoter une topologie. Dans ce cas, Ψ_ρ désignera sa substitution topologique.

IV.5.3 Application des règles d'une transformation

L'application d'une transformation à une collection (considérée avec la topologie ρ) consiste à appliquer les règles de la transformation de la manière suivante :

- Les instances du premier motif sont recherchées. On sélectionne un sous-ensemble maximal d'instances dont l'intersection deux à deux est vide[1]. Les positions sélectionnées sont dites *consommées*.
- On recherche les instances du motif de la seconde règle parmi les positions qui n'ont pas été consommées, puis on sélectionne un ensemble maximum de chemins ne s'intersectant pas.
- On procède ainsi de suite avec les autres règles.
- Lorsque ce processus de filtrage est terminé, on substitue les parties filtrées par les parties remplaçantes correspondantes avec la substitution topologique appropriée (σ_ρ).
- La nouvelle collection ainsi créée est retournée.

Ceci constitue la stratégie par défaut d'application des transformations. Comme nous l'avons vu dans le chapitre II, d'autres stratégies existent dans MGS.

La description de l'application des transformations donnée ci-dessus est une vue idéalisée du comportement des transformations. De nombreuses optimisations sont bien évidemment possibles lorsqu'on passe à l'implémentation. Notamment :

- mise à jour au fur et à mesure des positions consommées (voir ci-dessous),
- filtrage de plusieurs chemins *en parallèle* si le matériel le permet,
- spécialisation de l'algorithme de filtrage pour des topologies particulières (voir le chapitre IX),
- réécriture du motif en un motif équivalent plus performant, lorsque la topologie le permet [Spi03] (par exemple sur les ensembles, le motif $(1, x)$ est plus efficace que le motif $(x, 1)$).

IV.5.4 Implémentation des transformations

L'algorithme présenté en section IV.4.2 donne la sémantique des motifs : il détermine l'ensemble des chemins filtrés par un motif. Mais dans l'interprète, on ne calcule pas tous les chemins possibles : on calcule « au vol » un sous-ensemble maximal d'instances disjointes deux à deux. L'idée est d'énumérer à travers des boucles les occurences de chemins et de stopper les calculs dès qu'on a un chemin qui satisfait les gardes de sa spécification. Pour ce faire, chaque règle de la figure 3 correspond à un cas d'une fonction récursive et chaque union dans les parties droites correspond à une boucle d'itération. Une fois une instance trouvée, les positions filtrées sont *consommées* : elles ne seront pas dans l'ensemble Π de positions *disponibles* passé en argument au processus qui cherchera d'autres instances.

Notons également qu'une architecture objet permet de manipuler uniformément les collections de manière simple et indépendamment de la façon dont chaque type de collection est implémenté. Chaque topologie correspond à une *classe* fournissant les opérateurs nécessaires à l'application d'une transformation : accès aux valeurs, accès aux voisins d'une position, opération de substitution topologique, etc.

[1] Si $\mathcal{S}$ est l'ensemble des instances du motif, on sélectionne une partie $s \in \mathcal{P}(\mathcal{S})$ des instances telle que pour tout chemin i de $\mathcal{P}(\mathcal{S})$, i est soit dans s, soit en intersection avec un chemin de s.

IV.6 Application au filtrage des structures indexées

Nous avons atteint notre objectif en proposant un algorithme générique de filtrage sur les collections topologiques. Regardons à présent une mise en œuvre des techniques présentées sur le cas des GBF. Les GBF ont été introduits en section II.4.

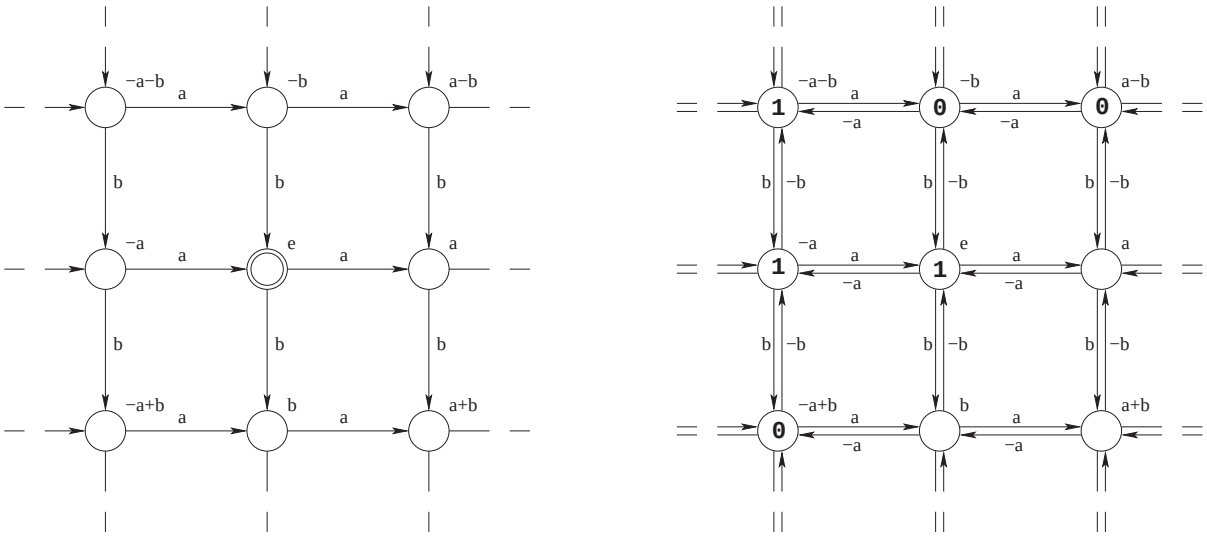
IV.6.1 Représentation topologique des GBF

Étant donné une présentation finie d'un groupe $\mathcal{G}$ le graphe de Cayley associé est le graphe (E, V) tel que E est égal à l'ensemble des éléments de $\mathcal{G}$ et il existe un arc $(e1, e2)$ dans V si et seulement si il existe un générateur u de la présentation tel que $e1 + u = e2$.

La notion de graphe de Cayley fait le lien entre les GBF et les collections topologiques :

- Les positions de la collection sont les nœuds du graphe de Cayley, soit les éléments du groupe $\mathcal{G}$.
- Les directions de la collection sont les générateurs de la présentation *et* leurs inverses.
- Les arcs de la collection sont l'ensemble des arcs du graphe de Cayley étiquetés du générateur correspondant et l'ensemble des inverses des arcs du graphe de Cayley étiquetés de l'inverse du générateur correspondant. On peut noter que sur les GBF, la relation de voisinage est symétrique.

La figure 4 illustre la correspondance entre le graphe de Cayley associé au type de GBF abélien défini par la présentation `< a,b >` et les collections topologiques.



(a) Graphe de Cayley

(b) Collection topologique

Fig. 4 – Correspondance groupe/collection

IV.6.2 Filtrage et transformations

Le filtrage sur les GBF se fait avec l'algèbre de motifs et l'algorithme de filtrage que nous avons donnés, comme sur n'importe quelle collection topologique. Afin de pouvoir appliquer les transformations, il faut choisir une substitution topologique adaptée. La substitution topologique newtonienne $\Psi_{\mathcal{N}}$ convient. En effet elle garantit que l'ensemble des positions et les voisinages ne sont pas modifiés. Ainsi les transformations préservent la structure du groupe sous-jacent aux GBF. Lorsque la partie droite d'une règle n'a pas la même longueur que la partie filtrée, une erreur est levée (nous notons cette erreur **shape**).

IV.6.3 Exemples de motifs et de dérivations

Motifs simples

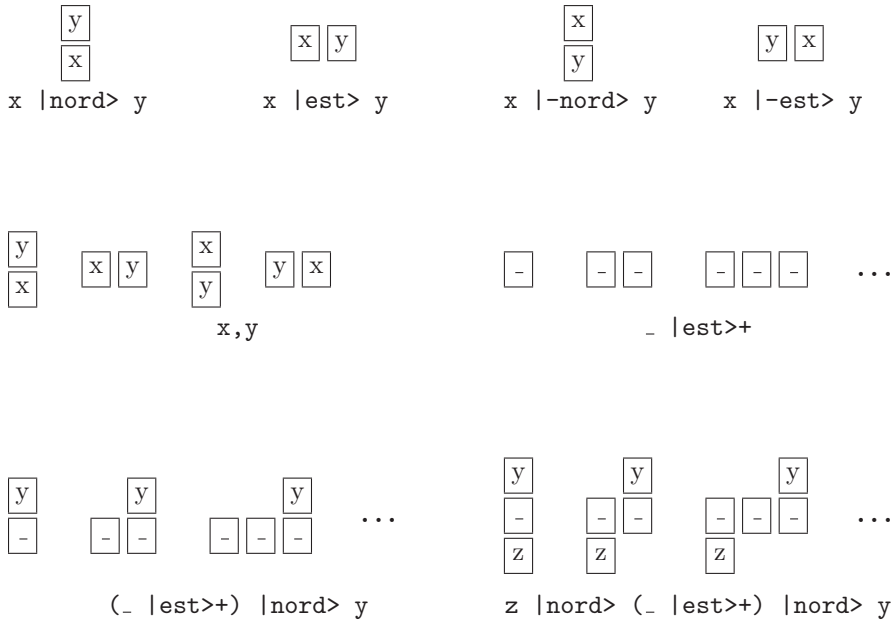
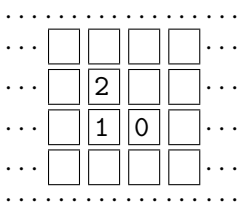


FIG. 5 – Exemples de motifs sur le GBF *grille* et parties filtrées correspondantes.

La figure 5 montre des motifs valides sur le type de GBF *grille*. Le motif `x,y` peut filtrer deux éléments disposés selon quatre configurations différentes. Les trois derniers motifs filtrent des parties dont la taille n'est pas connue à l'avance.

Calculs de dérivées

Pour illustrer l'algorithme de filtrage sur les GBF, nous allons déterminer les chemins filtrant le motif `(_, 1 |nord> x)` dans le GBF grille suivant :



Le motif est réécrit dans la nouvelle syntaxe :

$$P = \texttt{u/true |nord,est,-nord,-est> (v/v=1) |nord> (x/true)}$$

et on pose $Q = \texttt{(v/v=1) |nord> (x/true)}$.

Dans le GBF *grille* chaque position peut être exprimée comme un déplacement $n.\texttt{nord}$ $+e.\texttt{est}$ par rapport à un point origine avec $(n, e) \in \mathbb{Z}^2$. Une position sera donc notée (n, e) et l'origine est choisie arbitrairement au point ayant pour valeur 1. Nous notons $\mathtt{g}$ la collection correspondant au GBF représenté ci-dessus. Son domaine est :

$$\Pi = dom(\mathtt{g}) = \{(0,0),\ (0,1)\ , (1,0)\}$$

Les chemins filtrés par P son calculés par :

$$\frac{\partial P}{\partial (0,0)}(\mathtt{g}, \epsilon, \Pi) \cup \frac{\partial P}{\partial (0,1)}(\mathtt{g}, \epsilon, \Pi) \cup \frac{\partial P}{\partial (1,0)}(\mathtt{g}, \epsilon, \Pi)$$

Calculons la dérivée à la position $(0,0)$:

$$\frac{\partial P}{\partial (0,0)}(\mathtt{g}, \epsilon, \Pi) = (0,0) \otimes \bigcup_{p' \in \{(1,0),(0,1)\}} \frac{\partial Q}{\partial p'}(\mathtt{g}, [u \mapsto (0,0)], \Pi')$$

où Π' vaut $\{(0,1),(1,0)\}$. Le premier terme de l'union vaut $\emptyset$ car (1,0) n'a pas de `nord`-voisin :

$$\frac{\partial Q}{\partial (1,0)}(\mathtt{g}, [u \mapsto (0,0)], \Pi') = (1,0) \otimes \bigcup_{p' \in \emptyset} \frac{\partial\, x/true}{\partial p'}(\ldots) = (1,0) \otimes \emptyset = \emptyset$$

et le second terme de l'union vaut également $\emptyset$ car $(0,1)$ n'a pas de `nord`-voisin.

Nous avons donc $\frac{\partial P}{\partial (0,0)}(\mathtt{g}, \epsilon, \Pi)$ qui vaut $(0,0) \otimes \emptyset$ donc $\emptyset$. Le même résultat est obtenu pour $\frac{\partial P}{\partial (1,0)}(\mathtt{g}, \epsilon, \Pi)$. En revanche le calcul de $\frac{\partial P}{\partial (0,1)}(\mathtt{g}, \epsilon, \Pi)$ donne un résultat :

$$\frac{\partial P}{\partial (0,1)}(\mathtt{g}, \epsilon, \Pi) = (0,1) \otimes \bigcup_{p' \in \{(0,0)\}} \frac{\partial Q}{\partial p'}(\mathtt{g}, [u \mapsto (0,1)], \Pi'')$$

où Π'' vaut $\{(0,0),(1,0)\}$ et

$$\frac{\partial Q}{\partial (0,0)}(\mathtt{g}, [u \mapsto (0,1)], \Pi'') = (0,0) \otimes \frac{\partial\, x/true}{\partial (1,0)}(\mathtt{g}, [u \mapsto (0,1), v \mapsto (0,0)], \Pi''')$$

où $\Pi''' = \Pi'' - (0,0) = \{(1,0)\}$. Or on a

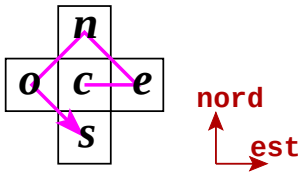
$$\frac{\partial\, x/true}{\partial (1,0)}(\mathtt{g}, \ldots, \Pi''') = \{[(1,0)]\}$$

donc le résultat final est $(0,1) \otimes ((0,0) \otimes \{[(1,0)]\})$ soit $\{[(0,1),\ (0,0),\ (1,0)]\}$, ce qui correspond au chemin attendu.

IV.6.4 Exemple d'utilisation d'un motif de pseudo-chemins

La rotation d'un motif en croix sur un GBF de type *grille* (Fig. 6) s'écrit avec un motif de pseudo-chemins :

```
trans rota =
  { c |est> e |nord-est> n |-est-nord> o |est-nord> s  =>  c,s,e,n,o } ;;
```



L'utilisation d'un motif de pseudo-chemins ne pose pas de problème puisque les valeurs sur le pseudo-chemin filtré seront substituées point-à-point par la substitution newtonienne $\Psi_{\mathcal{N}}$.

IV.7 Autres structures de données : collections proximales

Nous venons de voir comment les GBF rentrent dans le cadre des collections topologiques et bénéficient ainsi du filtrage uniforme. Voyons brièvement comment s'intègrent les structures dites proximales (présentées en section III.5).

Contrairement aux séquences ou aux GBF, la relation de voisinage sur une collection proximale dépend uniquement de la valeur de ses éléments. En effet, si c est un proximal de relation caractéristique r, alors on a :

$$p_1 \text{ est voisine de } p_2 \text{ dans } c \Leftrightarrow \begin{cases} p_1 \text{ possède la valeur } v_1 \text{ dans } c \\ p_2 \text{ possède la valeur } v_2 \text{ dans } c \\ v_1 \text{ est en relation avec } v_2, \text{ soit } r(v_1, v_2) \end{cases}$$

Notons que r est donnée par un prédicat à deux arguments, comme :

```
fun r (x,y) = distance(x,y)<10
```

La substitution topologique utilisée pour manipuler un proximal de relation caractéristique r opère de la manière suivante :

1. Considérer l'ensemble des positions de la collection en argument.
2. Y supprimer les positions consommées (les positions du chemin en argument).
3. Pour chaque valeur de l'ensemble de valeurs à ajouter, y ajouter une position possédant cette valeur.
4. Construire la relation de voisinage sur cet ensemble de positions grâce à la relation r.

Ainsi, à chaque relation r on associe une substitution topologique « proximale » $\Psi_{prox:r}$. Notons qu'il s'agit d'une substitution leibnitzienne.

De même, pour chaque relation r on peut définir un opérateur de construction $\texttt{cons}_r$ sur les proximaux paramétrés par r. Ainsi, pour toute relation r on peut considérer la topologie $\texttt{proxy:r} = \langle[\texttt{empty\_proxy}, \texttt{cons}_r], \Psi_{proxy:r}\rangle$.

Les proximaux rentrent ainsi dans le cadre que nous avons établi.

IV.8 Bilan

Nous avons vu dans ce chapitre comment filtrer uniformément sur de nombreuses structures de données, y compris des structures non-algébriques. Ceci permet notamment de gérer fonctionnellement ces structures. Avoir considéré les collections d'un point de vue topologique nous permet d'uniformiser le filtrage et de donner un sens à un même motif sur toute collection.

Dans le but d'être efficace, on n'implémentera pas le filtrage par l'algorithme tel qu'il est proposé. Celui-ci peut être vu comme une spécification de l'opération de filtrage. Nous verrons dans le chapitre consacré à la compilation des méthodes de filtrage plus efficaces. Toutefois cet algorithme a l'avantage de s'adapter à toute collection topologique. Ainsi on peut ajouter à MGS de nouveaux types de collections sans avoir à fournir d'algorithme spécialisé pour ceux-ci.

Chapitre V

SK-traduction au vol en présence de traits impératifs

Les travaux présentés dans ce chapitre ont été publiés dans [Coh05].

Dans ce chapitre nous exposons certaines des techniques utilisées pour l'implémentation de l'interprète MGS. Nous nous concentrons sur la réalisation de l'*évaluateur* qui est le module dédié à l'interprétation du noyau fonctionnel du langage. L'originalité de cette implémentation est qu'elle repose sur la réduction de l'application du langage hôte au lieu de la gérer elle-même. Un objectif de simplicité, d'orthogonalité et d'efficacité a motivé ce choix. Nous utilisons une technique de combinatorisation afin de transformer les fonctions définies par l'utilisateur en fonctions du langage hôte. Les fonctions non-strictes et les traits impératifs de MGS sont bien intégrés à ce schéma grâce à l'utilisation de *glaçons*. Nous serons donc amenés à étudier la combinatorisation SK en présence de traits impératifs.

La section V.1 détaille les raisons qui nous ont poussés à choisir cette approche. Nous introduisons dans la section suivante une implémentation *classique* d'un interprète puis nous la comparons à l'approche d'ordre supérieur en section V.3. La réalisation de notre approche par syntaxe d'ordre supérieur et combinatorisation est présentée en section V.4. Elle est ensuite étendue aux fonctions non-strictes (section V.5) et aux traits impératifs (section V.6). Les sections V.7 et V.8 concluent ce chapitre en indiquant que la preuve de correction reste à établir et en dressant un bref bilan.

V.1 Objectifs et motivations

Dans ce chapitre, nous étudions comment représenter une valeur fonctionnelle MGS par une valeur fonctionnelle du langage qui sert à implémenter notre interprète (ici OCaml). Ce type de représentation simplifie beaucoup l'écriture de l'interprète et relève de plusieurs domaines de recherche dont la logique combinatoire et la syntaxe abstraite d'ordre supérieur.

Trois problèmes se posent avec cette approche :

1. La seule manière de construire une valeur fonctionnelle comme résultat d'un calcul en Caml est par application de fonctions. Or les valeurs fonctionnelles en MGS peuvent se dénoter en utilisant une abstraction. Il faut donc pouvoir construire la fonction définie par cette abstraction uniquement par des applications. Ce point est résolu grâce à la traduction en

combinateurs.

2. Certaines expressions MGS correspondent à des fonctions non-strictes alors que le langage d'implémentation n'a que des fonctions strictes et à appel par valeur. La conditionnelle *if-then-else*, la conjonction logique et la disjonction sont des exemples de fonctions non-strictes. La solution est l'utilisation de glaçons.
3. MGS comporte des traits impératifs. Or il est délicat de mixer certains traits impératifs avec le séquencement des calculs induit uniquement par les appels fonctionnels (par exemple, comment réaliser le séquencement de deux évaluations quand on ne dispose que d'applications de fonctions et que l'ordre d'évaluation des arguments n'est pas spécifié ?). Là aussi, la solution passe par l'utilisation de glaçons.

Si l'évaluation d'un noyau fonctionnel dans un interprète est un problème classique, l'approche adoptée ici est assez peu habituelle. Cependant, elle répond aux six contraintes suivantes qui ont été posées *a priori* dans le cadre du projet MGS :

1. Nous voulons un schéma d'évaluation interprété.
   MGS est dédié au prototypage rapide de modèle de simulation. Cette activité demande de nombreux aller-retours entre les phases de conception du modèle et l'exécution de la simulation proprement dite. Un interprète et un schéma d'évaluation dynamique sont bien mieux adaptés à ce type d'activité qu'un compilateur. La compilation de MGS est un problème abordé dans la suite de ce document mais nous voulons disposer des deux outils.
2. L'évaluateur doit pouvoir s'implémenter indifféremment dans tout langage « raisonnable ».
   Afin de faciliter sa diffusion, en particulier auprès des non-informaticiens, l'interprète MGS doit pouvoir être (et a été effectivement) porté sur des environnement différents (sous Windows/Cygwin et sous MAC OS X). Par ailleurs MGS est un langage expérimental et plusieurs versions ont été développées (dont une en C++). Enfin, nous ne voulons pas que l'implémentation puisse dépendre de spécificités du langage d'implémentation.
   Si nous utilisons OCaml comme langage d'implémentation notre approche doit rester valable pour un autre langage. On verra que la stratégie d'évaluation développée dans ce chapitre ne requiert que la possibilité de passer des fonctions en paramètre à une autre fonction et de retourner des fonctions comme résultat d'une application de fonction.
3. L'évaluateur doit rester simple (tout au plus quelques centaines de lignes de code) et extensible, tout en offrant des structures de contrôle utiles comme les exceptions et des outils de trace et de débug.
   Bien que nous ne le détaillerons pas, l'introduction dans MGS d'un mécanisme d'exception à la OCaml et de possibilités de trace des appels de fonctions est réalisée par moins d'une cinquantaine de lignes de code à partir du schéma d'évaluation proposé dans ce chapitre.
4. La performance n'est pas le souci premier, mais l'évaluateur doit néanmoins être le plus efficace possible sans sacrifier l'extensibilité et la simplicité du schéma d'évaluation adopté. Notre objectif est d'être plus efficace que Mathematica (pour la partie fonctionnelle et le filtrage) et au moins aussi efficace que Python (sur l'aspect fonctionnel).
5. Le schéma d'évaluation adopté doit permettre le développement et *l'intégration immédiate* de librairies développées dans le langage d'implémentation et l'intégration de librairies développées dans un autre langage (comme C et C++).
   Ces librairies correspondent au développement des collections topologiques offertes par MGS. Elles doivent donc être efficaces (en espace et en temps) et il est donc sage de pouvoir les compiler. De plus, les développements logiciels impliqués peuvent être lourds. Ces librairies doivent donc pouvoir se concevoir et se développer de manière complètement indépendante de l'implémentation de l'évaluateur proprement dit.

6. La notion de fonction offerte par le langage doit être uniforme.

   En particulier, un programmeur MGS ne doit pas pouvoir distinguer une fonction qu'il a écrit lui-même en MGS d'une fonction « primitive » implémentée par une des librairies évoquées dans le point ci-dessus. Ainsi, une opération primitive d'ordre supérieur doit pouvoir accepter en argument aussi bien une fonction écrite par l'utilisateur en MGS qu'une autre primitive prédéfinie, et cela de manière transparente aussi bien pour le concepteur de la librairie que pour le programmeur MGS. C'est une propriété forte, difficile à assurer, mais qui permet d'écrire simplement des librairies puissantes, facilement paramétrables et extensibles.

Les points 2 et 5 nous font renoncer aux approches de type « traduction au vol vers un langage interprété » comme celle permise par Camlp4 [dR02]. La contrainte 3 exclue les approches de type « compilation vers une machine virtuelle », du moins si l'on veut que le résultat soit raisonnablement efficace[1]. La contrainte 6 ne peut pas être assurée par les approches classiques par évaluation des termes comme décrit à la section V.2.

En fait, prise au sens littéral, cette dernière contrainte impose que les fonctions écrites par le programmeur MGS soient traduites au vol en des fonctions du langage d'implémentation. Cependant, puisque le langage d'implémentation est compilé, cela n'est possible que si les fonctions sont des valeurs (ce n'est par exemple pas le cas en Fortran77 mais c'est le cas en C ou en OCaml) puisque les seules entités qui peuvent se construire lors de l'exécution sont des valeurs. Les opérations permettant de construire des valeurs fonctionnelles dans un langage de programmation ne sont pas nombreuses. Nous ne voulons pas bien sûr embarquer de manière plus ou moins explicite un interprète du langage d'implémentation dans l'interprète MGS. La *logique combinatoire* offre alors une solution : toute fonction peut se construire comme résultat de l'application d'un jeu réduit de fonctions primitives : les combinateurs.

L'idée originale développée dans ce chapitre est de faire reposer l'évaluation du noyau fonctionnel du langage source uniquement sur le mécanisme d'application de fonction du langage d'implémentation et d'associer dynamiquement à chaque fonction écrite dans le langage source, une fonction du langage d'implémentation calculée dynamiquement dans l'interprète.

Cette approche est très différente des approches d'évaluation par réduction de combinateurs proposée par [PJ87] et développées notamment dans [JS89, Hug82] (et qui relèvent de la catégorie des machines virtuelles) ou de la construction d'un réseau de fermetures dans [FL87], qui ne permet pas de faire abstraction des manipulations explicites d'environnements. A notre connaissance, l'approche que nous décrivons dans ce chapitre est nouvelle et mixe des idées issues des travaux sur la « syntaxe abstraite d'ordre supérieur » avec des notions empruntées à la logique combinatoire.

Organisation du chapitre. Dans la suite de ce chapitre, nous étudions un schéma d'évaluation qui satisfait les contraintes décrite ci-dessus. Nous ne considérons pas ici les transformations ni le filtrage car ces mécanismes sont indépendants du problème posé et sont traitées à un autre niveau. Notons toutefois que les gardes des motifs et les parties droites des règles de transformations sont vues comme des fonctions des valeurs filtrées et sont donc traitées uniformément selon le schéma que nous présentons ici.

[1] Une version de MGS a été développée avec un évaluateur correspondant à la machine virtuelle SECD [Lan65a]. Cette approche s'est révélée moins efficace que l'approche par interprétation directe des termes, comme décrit à la section V.2, du moins tant que l'on en restait à une compilation directe sans optimisation particulière. Ce schéma d'évaluation ne répondait pas non plus aux contraintes 5 et 6.

Le reste du chapitre s'organise comme suit. Nous commençons par donner une manière classique d'implémenter un interprète (section V.2) puis nous évoquons l'approche par syntaxe abstraite d'ordre supérieur (section V.3). Ensuite nous détaillons comment la combinatorisation permet d'utiliser une syntaxe d'ordre supérieur (section V.4). Dans les sections V.5 et V.6 nous étudions comment introduire les fonctions non-strictes et les traits impératifs dans ce schéma d'évaluation en utilisant des glaçons. Notons dès à présent que le chapitre suivant est consacré aux optimisations que l'on peut apporter à ce schéma.

V.2 Architecture classique d'un interprète

V.2.1 Langage considéré

L'approche adoptée est assez peu habituelle. Nous allons donc l'introduire pédagogiquement en partant de l'implémentation classique d'un interprète. Considérons le langage défini par le type `term` suivant :

```
type term =
 | Int of int
 | Var of ident
 | Abs of ident * term
 | App of term * term
 | ExtFun of value -> value
 | If of term * term * term
 | Get of memident
 | Set of memident * term
```

```
type value =
 | ValInt of int
 | ValFun of value -> value
 | ValAbs of ident * term * env

and env = (ident * term) list
```

Le type `term` définit la syntaxe abstraite du langage que nous interprétons. Ce type est exprimé dans le langage OCaml car notre interprète est implémenté dans ce langage. Toutefois nous souhaitons pouvoir adapter les idées présentées à tout langage pouvant manipuler des fonctions et ayant une stratégie d'évaluation d'appel par valeur (comme C ou C++ par exemple). Le langage choisi (ici OCaml) sera appelé le langage *hôte*.

Détaillons les constructions du langage. `Int 1` dénote l'entier 1. `Var "x"` dénote une occurrence de la variable x. `Abs ("x",`e`)` correspond à la fonction définie par l'abstraction de x dans e. Par exemple `Abs ("x",Var "x")` correspond à la fonction identité que l'on écrirait `fun (x) = x` ou bien `\x.x` en MGS. `ExtFun` f dénote une *fonction prédéfinie* du langage. On parle aussi de *fonction externe* car on peut utiliser une fonction définie dans une bibliothèque. Cette fonction doit avoir le type `value -> value` où `value` est le type des valeurs renvoyées par la fonction d'évaluation. L'addition entre entiers est un exemple de fonction prédéfinie :

```
let plus = fun (ValInt i) -> ValFun (fun (ValInt j) -> ValInt (i+j) )
```

On écrira donc `ExtFun plus` pour la dénoter dans la syntaxe considérée. Notons que toutes les fonctions sont curryfiées dans notre langage. La conditionnelle `If(`e_1`, `e_2`, `e_3`)` correspond au *if-then-else* et évalue sa branche *then* lorsque la condition e_1 est un entier strictement positif, et la branche *else* dans les autres cas. `Set(`m`,`e`)` affecte à la mémoire m la valeur de l'expression e. `Get` m renvoie le contenu de la mémoire m. Exemple :

```
Abs("x",IF(Var "x", App( App (ExtFun plus, Var "x"), Int 1), Get m))
```

est la traduction de l'expression MGS suivante : `fun (x) = if x then x+1 else m fi`.

Le langage considéré suffit à illustrer les difficultés rencontrées. Nous nous limitons aux entiers comme structures de données car l'objet de ce chapitre est l'implémentation des fonctions. Nous ne considérons pas le séquencement dans le langage car celui-ci peut être encodé avec un *if-then-else*. En effet, e_1 ; e_2 est équivalent à *if* e_1 *then* e_2 *else* e_2. En outre les autres fonctions non-strictes comme le *ou* logique paresseux peuvent également être encodées par un *if-then-else* : $e_1||e_2$ est équivalent à *if* e_1 *then true else* e_2. Enfin la sortie écran *print* ou la sortie fichier peuvent être vues comme une écriture en mémoire.

V.2.2 Évaluation

Une première approche possible pour construire la fonction d'évaluation d'un interprète est de se baser sur les équations de la sémantique dénotationnelle du langage [Mos90]. La figure 1 donne une implémentation possible qui utilise un environnement pour gérer l'instanciation des variables.

```
let store = ref []

let rec evalD env = function
 | Int i -> ValInt i
 | Var id -> assoc id env
 | Abs (id,e) -> ValAbs (id, e, env)
 | App (e1,e2) ->
        (match (evalD env e1,evalD env e2) with
           | (ValFun f,v) -> f v
           | (ValAbs (id,e,env'), v) -> evalD ((id,v)::env') e )
 | ExtFun f -> ValFun f
 | If (e1,e2,e3) ->
    (match evalD env e1 with
           | ValInt i when i>0 -> evalD env e2
           | _ -> evalD env e3)
 | Get m -> assoc m !store
 | Set (m,e) ->
    let v= evalD env e
    in begin store := (m,v) ::  !store ; v end
```

FIG. 1 – Évaluateur à la façon classique (fonction `evalD`).

Détaillons le fonctionnement de cette évaluation :

Applications et variables. L'évaluation de l'application $((\lambda x.e)\ v)$ dans un environnement *env* consiste à évaluer e sachant que x est liée à v. Ici la liaison de x à v se fait par un environnement implémenté par une liste d'associations. L'instanciation des variables associée à l'application de fonctions est donc gérée explicitement.

On remarque une différence de traitement entre les fonctions MGS écrites par l'utilisateur et les fonctions primitives : l'application des fonctions primitives est gérée par le langage

hôte alors que l'application des abstractions utilisateur nécessite la manipulation explicite de l'environnement et des variables.

Abstractions. Une abstraction pouvant être utilisée en dehors du contexte où elle est définie, on doit mémoriser la liaison des variables apparaissant dans son corps au moment de sa définition. C'est pour cela que l'environnement est stocké avec l'abstraction dans `ValAbs` (il s'agit d'une *clôture*).

Le programme suivant justifie le besoin de mémoriser l'environnement :

```
fun f (x) = (g := \y.x) ;;
f(1);;
g(0);;
```

Ici on veut que la fonction dans la mémoire g associe bien 1 à x.

Lecture/écriture mémoire. La mémoire, elle aussi, est gérée par un environnement mais alors que l'environnement pour les variables est relatif à chaque expression évaluée, la mémoire est la même pour tout le programme.

Fonctions prédéfinies. Comme les constantes entières, les fonctions prédéfinies ne sont pas modifiées par l'évaluation.

Conditionnelle. L'évaluation d'une conditionnelle n'évalue que la branche *then* ou *else* selon le résultat de l'évaluation de la condition.

Le langage hôte étant un langage fonctionnel performant, on peut avoir envie de lui déléguer totalement la tâche de réduction des applications. Ceci permettrait de gagner à la fois en abstraction et en efficacité. Ceci peut être mis en œuvre avec une syntaxe abstraite d'ordre supérieur.

V.3 Syntaxe abstraite d'ordre supérieur

La syntaxe abstraite d'ordre supérieur (HOAS d'après l'appellation en anglais) consiste à représenter des abstractions/liaisons de variables (λ, $\forall$, $\exists$ ou autres) dans un arbre de syntaxe par une fonction du langage hôte [PE88]. On peut transformer la syntaxe abstraite `term` donnée ci-dessus en syntaxe d'ordre supérieur en supprimant les abstractions et les variables et en faisant reposer les mécanismes d'applications de fonctions sur ceux du langage hôte :

```
type HOterm =
  | HOInt of int
  | HOApp of HOterm * HOterm
  | HOFun of value -> value
```

Un seul constructeur est nécessaire pour désigner à la fois les fonctions prédéfinies et les fonctions définies par l'utilisateur car les deux sont représentées par une fonction du type `value -> value`. Par exemple la fonction MGS `fun (x) = x+1` peut être représentée par :

```
HOFun (fun x -> HOApp (HOApp (HOFun plus, x), HOInt 1))
```

Nous n'incluons pas de construction pour la conditionnelle *if* dans cette syntaxe car elle peut être vue comme une fonction particulière. En effet le *if* peut être vu comme une fonction à trois arguments qui n'évalue pas tous ses arguments. Dans la section V.5 nous verrons que l'on peut l'encoder par une fonction ordinaire. Les traits impératifs peuvent également être encodés dans cette syntaxe (section V.6).

L'utilisation d'une syntaxe d'ordre supérieur a les avantages suivants :

- Les fonctions prédéfinies et les fonctions définies par l'utilisateur peuvent être traitées uniformément.
- Les variables ne sont plus gérées par l'interprète mais par le langage hôte. Ceci signifie notamment que l'on n'a pas besoin de gérer :
  - la portée des variables dans toutes les manipulations de l'arbre de syntaxe (les optimisations par exemple) ;
  - l'instanciation des variables lors de l'application ;
  - la propagation de substitutions ou d'environnements.

Ces avantages permettent d'avoir un interprète plus concis et donc :

- plus simple à implémenter, comportant moins d'erreurs, plus maintenable ;
- plus simple à décrire, sur lequel on peut mieux raisonner ;
- plus propice à l'expérimentation.

L'évaluation des termes en syntaxe d'ordre supérieur se fait de la manière suivante :

```
let rec evalHO = function
 | HOInt i -> ValInt i
 | HOFun f -> ValFun f
 | HOApp (t1,t2) ->
   (match (evalHO t1, evalHO t2) with (HOFun f, v) -> f v )
```

On voit bien ici que c'est le langage hôte qui gère l'application `(f v)`. La difficulté liée à cette approche est la production des valeurs fonctionnelles par l'analyse syntaxique ou comme résultat d'un calcul.

Problème P1 : production des fonctions du langage hôte. Dans Caml comme dans le λ-calcul, le seul moyen de renvoyer une fonction est de renvoyer une fonction prédéfinie ou d'en créer une par application d'autres fonctions. Un calcul ne peut pas créer une fonction « syntaxique » comme on le fait en *écrivant* un programme manuellement ou automatiquement. Ceci signifie qu'on peut aisément traduire un terme arbitraire du λ-calcul en une chaîne de caractères représentant son équivalent en Caml, mais produire la *valeur* Caml correspondante n'est pas direct.

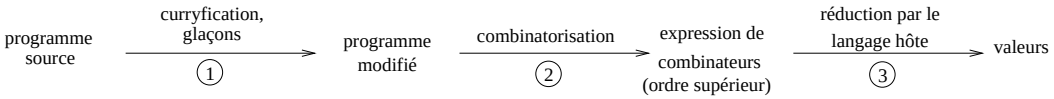
Problème P2 : intégration des fonctions non-strictes. Les fonctions non-strictes comme la conditionnelle *if-then-else*, le *ou* et le *et* paresseux ne peuvent être directement traduits en fonctions Caml. En effet, si on traduit le *if-then-else* par une fonction à trois arguments, les trois arguments seront évalués puisque Caml est un langage à appel par valeur.

Problème P3 : intégration des traits impératifs. Comme les fonctions non-strictes, les traits impératifs ne s'expriment pas tous comme des fonctions. C'est le cas du séquencement qui ne peut être vu comme une fonction à deux arguments (à moins d'avoir une garantie sur l'ordre d'évaluation des arguments). Par ailleurs même si un trait impératif peut être vu comme une fonction, comme c'est le cas du *print*, il faut veiller à ce que l'effet de bord ait lieu au « bon moment ». Par exemple, l'évaluation de l'expression

```
(fun (x) = print"0") (print "1")
```

doit imprimer 1 avant 0.

Notre réponse au problème P1 est la combinatorisation (section V.4). Celle aux problèmes P2 et P3 est l'utilisation de glaçons (sections V.5 et V.6). Grâce à ces solutions techniques, l'évaluation des termes du langage considéré suit le schéma suivant :



① Transformation des fonctions non-strictes et des traits impératifs en fonctions en utilisant des glaçons. Nous appelons cette étape *curryfication*.
② Transformations des abstractions en fonctions du langage hôte. Cette étape est appelée *combinatorisation*. La technique utilisée est la SK-traduction.
③ Évaluation en utilisant les mécanismes d'application du langage hôte.

V.4 Combinatorisation

L'utilisation de combinateurs est une manière de résoudre le problème P1. En effet, on peut transformer tout programme du λ-calcul pur en un programme équivalent[2] composé uniquement d'applications entre trois fonctions élémentaires notées S, K et I [CF58]. Cette transformation est appelée SK-traduction ou SK-compilation. Nous l'appellerons également combinatorisation. La SK-traduction est donc un moyen de créer de nouvelles fonctions par applications de fonctions.

Dans cette section nous considérons une version simplifiée du langage, sans les fonctions non-strictes et sans les traits impératifs. Les sections V.5 et V.6 s'attacheront à les intégrer dans le schéma d'évaluation que nous proposons ici.

V.4.1 SK-traduction classique

L'objet de la SK-traduction est de décomposer toute fonction en une combinaison de fonctions élémentaires [CF58]. Considérons la fonction $\lambda x.(e_1\ e_2)$. La variable x peut apparaître dans e_1 et dans e_2. Lorsqu'on abstrait x dans $(e_1\ e_2)$ on abstrait donc x dans e_1 et dans e_2. L'égalité suivante illustre bien ceci :

$$((\lambda x.e_1\ e_2)\ a) = ((\lambda x.e_1\ a)\ (\lambda x.e_2\ a))$$

C'est sur cette égalité qu'est basée ce que l'on appelle la transformation S : pour abstraire x dans une application $(e_1\ e_2)$ il est suffisant de l'abstraire dans e_1 et dans e_2 et d'utiliser une fonction élémentaire chargée de distribuer l'argument reçu par cette fonction à chacune des nouvelles abstractions puis effectuer l'application. La fonction élémentaire distribuant l'argument reçu est le combinateur S. La règle de transformation S ainsi que la règle de réduction associée sont données dans la figure 2.

Le combinateur S est la fonction $\lambda f.\lambda g.\lambda x.((f\ x)\ (g\ x))$. Le résultat de la transformation S est bien composé de plusieurs fonctions simples : $\lambda x.e_1$ ainsi que $\lambda x.e_2$ sont bien plus simples que

[2]Attention, l'équivalence dont on parle ici est à nuancer. En effet, selon [HS86, pp 30 et suivantes], le λ-calcul et la logique combinatoire ont des propriétés équivalentes. De plus, on peut définir une notion d'*équivalence faible* (que l'on notera $\equiv_{\triangleright_F}$), c'est-à-dire une relation d'équivalence sur les termes de la logique combinatoire équivalents par conversion faible.

Cependant, le λ-calcul possède une propriété appelée ξ par [CF58] qui est : si $M\beta\eta N$ alors $(\lambda x.X)\beta\eta(\lambda x.Y)$. Cette propriété est vraie car toute contraction ou substitution de variable liée faite dans X peut aussi être faite dans $(\lambda x.X)$. Or, ce n'est pas le cas en logique combinatoire. Traduite dans le langage des combinateurs, cette propriété devient : si $M \equiv_{\triangleright_F} N$ alors $(\lambda^* x.X) \equiv_{\triangleright_F} (\lambda^* x.Y)$. Ce qui n'est pas le cas (le lieur ne fait pas partie du langage). Par exemple, si $X \equiv \mathbf{S}\,xyz$ et $Y \equiv xz(yz)$, alors $X \equiv_{\triangleright_F} Y$ mais on a $(\lambda^* x.X) \equiv \mathbf{S}(\mathbf{S}\,\mathbf{S}(\mathbf{K}\,y))(\mathbf{K}\,z)$ et $(\lambda x.Y) \equiv \mathbf{S}(\mathbf{S}\,\mathbf{I}(\mathbf{K}\,z))(\mathbf{K}(yz))$, qui sont deux termes en forme normale distincte, donc non faiblement équivalents. Le lecteur intéressé se reportera à la discussion de [HS86, chap. 9, pp 87 et suivantes] qui discute ce point. Pour notre usage, l'absence de ξ n'est pas un point gênant.

	transformation	réduction	implémentation
S	$\lambda x.(e_1\ e_2) \Rightarrow S\ (\lambda x.e_1)\ (\lambda x.e_2)$	$S\ f\ g\ e \rightarrow (f\ e)\ (g\ e)$	$\lambda x.\lambda y.\lambda z.((x\ z)\ (y\ z))$
K	$\lambda x.c \Rightarrow K\ c$	$K\ c\ e \rightarrow c$	$\lambda x.\lambda y.x$
I	$\lambda x.x \Rightarrow I$	$I\ e \rightarrow e$	$\lambda x.x$

Dans la transformation K, c désigne une constante (un entier, une fonction externe ou un combinateur) ou une variable autre que x. Le symbole $\Rightarrow$ dénote une étape de combinatorisation. Le symbole $\rightarrow$ dénote une étape de réduction.

FIG. 2 – SK-traduction pure.

$\lambda x.(e_1\ e_2)$ et S est une constante du langage hôte. On dira que la transformation S fait *descendre les lambdas* car le lambda en tête du terme s'est vu déplacé dans les sous-termes de celui-ci. En itérant la transformation S pour faire descendre tous les lambdas, on arrive à des termes où les abstractions restantes sont soit de la forme $\lambda x.x$, soit $\lambda x.c$ où c est soit une constante, soit une variable autre que x. La première forme correspond à une fonction élémentaire (un combinateur) que l'on appelle I. La seconde peut être décomposée en $((\lambda v.\lambda x.v)\ c)$. La fonction $\lambda v.\lambda x.v$ est la troisième fonction élémentaire et nous l'appelons K. La forme $\lambda x.c$ se transforma donc en $(K\ c)$. La forme $(K\ c)$ signifie que l'argument attendu par ce terme est inutile et sera donc « délaissé » mais il est toutefois attendu puisque la transformation S a placé un lambda directement au dessus de c. Si le terme compilé est clos, les abstractions mises sous la forme $(K\ x)$ auront disparu à la fin du processus car x est liée par un lambda qui va « descendre » jusqu'à elle.

L'algorithme de SK-traduction consiste à appliquer les règles de transformation de la figure 2 tant qu'il reste des abstractions dans le terme. Les règles sont traditionnellement appliquées en commençant par les abstractions les plus internes du terme à transformer (stratégie *inner-most*). À la fin il ne reste donc plus que des applications entre constantes (les combinateurs S, K, I et les autres fonctions externes étant des constantes). L'algorithme peut être défini également par la fonction $\mathcal{C}$ suivante qui parcourt le terme à compiler et effectue les transformations :

$$\begin{array}{ll} \mathcal{C}[(e_1\ e_2)] = (\mathcal{C}[e_1]\ \mathcal{C}[e_2]) & \qquad \mathcal{A}\langle x, x\rangle = I \\ \mathcal{C}[c] = c & \qquad \mathcal{A}\langle x, c\rangle = (K\ c) \\ \mathcal{C}[\lambda x.e] = \mathcal{A}\langle x, \mathcal{C}[e]\rangle & \qquad \mathcal{A}\langle x, (e_1\ e_2)\rangle = S\ \mathcal{A}\langle x, e_1\rangle\ \mathcal{A}\langle x, e_2\rangle \end{array}$$

Dans la définition de $\mathcal{C}$, c désigne une constante (entière ou fonction externe) ou une variable. La fonction auxiliaire $\mathcal{A}$ compile l'abstraction d'un terme par rapport à une variable. Sous cette forme, l'algorithme applique les règles en profondeur d'abord. La fonction $\mathcal{C}$ propage la traduction tandis que la fonction $\mathcal{A}$ introduit les combinateurs (les transformations S, K et I portent toutes sur des abstractions). Puisque $\mathcal{A}$ n'est appliquée qu'à une forme compilée, la définition de $\mathcal{A}$ ne prévoit pas le cas $\mathcal{A}\langle x, \lambda y.e\rangle$. Exemple de fonctionnement de l'algorithme :

$$\begin{array}{lll} \mathcal{C}[\lambda x.((+\ x)\ 1)] = \mathcal{A}\langle x,\ \mathcal{C}[((+\ x)\ 1)]\rangle & & \\ \quad = \mathcal{A}\langle x,\ (\mathcal{C}[(+\ x)]\ \mathcal{C}[1])\rangle & & \\ \quad = \mathcal{A}\langle x,\ ((\mathcal{C}[+]\ \mathcal{C}[x])\ 1)\rangle & & \\ \quad = \mathcal{A}\langle x,\ ((+\ x)\ 1)\rangle & & \\ \quad = (S\ \mathcal{A}\langle x, (+\ x)\rangle\ \mathcal{A}\langle x, 1\rangle) & (= S\ \lambda x.(+\ x)\ \lambda x.1 &) \\ \quad = (S\ (S\ \mathcal{A}\langle x, +\rangle\ \mathcal{A}\langle x, x\rangle))\ (K\ 1)) & (= S\ (S\ \lambda x.+\ \lambda x.x)\ (K\ 1) &) \\ \quad = (S\ (S\ (K\ +)\ I))\ (K\ 1)) & & \end{array}$$

Nous appellerons SK-traduction *pure* cet algorithme de traduction. L'algorithme donné par les fonctions $\mathcal{C}$ et $\mathcal{A}$ est proche de son implémentation. Toutefois nous préférons le décrire par en ensemble de règles de récriture comme celles de la figure 2 accompagné d'une stratégie d'application (ici la stratégie *inner-most*). Par conséquent, dans la suite du document nous décrirons les modifications à cet algorithme en décrivant les modifications à l'ensemble de règles ou à la stratégie d'application.

V.4.2 Implémentation

Les combinateurs S, K et I sont implémentés par des fonctions du langage hôte du type `value -> value` et peuvent donc être utilisés comme fonctions prédéfinies dans le type `HOterm` introduit en section précédente :

```
let app f x = match f with ValFun f' -> f' x

let comb_I = fun x -> x
let comb_K = fun c -> ValFun (fun x -> c)
let comb_S = fun f -> ValFun (fun g -> ValFun (fun x ->
       app (app f x) (app g x) ))
```

Exemple : la fonction MGS définie par `fun (x) = 1` qui se SK-traduit en $(K\ 1)$ est représentée par l'expression de combinateurs `HOApp(HOFun comb_K, HOInt 1)`.

L'évaluation des expressions de combinateurs (type `HOterm`) suit le schéma d'évaluation présenté en section précédente (fonction `evalHO`).

Résultats

Le schéma d'évaluation mis en place est plus lent que le schéma classique présenté en section V.2 (voir mesures en section VI.4.1). En effet, la SK-traduction pure fait exploser la taille du terme transformé, menant ainsi à une interprétation inefficace. Toutefois, des optimisations de la SK-traduction permettent de contenir l'explosion de la taille du terme produit. Nous étudierons dans le chapitre suivant trois optimisations nous permettant d'atteindre des performances raisonnables.

Discussion 2 L'utilisation d'un type somme (ici le type `HOterm`) pour représenter les termes combinatorisés est une source d'inefficacité. En effet le programme dans cette forme passe beaucoup de temps à

- produire des valeurs *construites* comme `HOInt 0`;
- *déconstruire* des valeurs construites, par exemple pour accéder au 0 de `HOInt 0`.

L'implémentation de l'application d'un terme de ce type illustre bien la perte de temps liée à la déconstruction. La fonction `app` donnée ci-dessus effectue l'application d'un terme `f` à un terme `x`. Cette fonction effectue deux actions : elle extrait le fonction du langage hôte embarquée dans la valeur construite `f` puis elle l'applique à `x` en déléguant cette application au langage hôte. On voit bien que toute application requiert une déconstruction.

Par conséquent, il serait plus efficace de ne pas utiliser de types sommes. Par exemple on a naturellement envie d'implémenter les combinateurs comme suit :

I : `fun x -> x`
K : `fun x -> fun y -> x`
S : `fun f -> fun g -> fun x -> (f x) (g x)`

On pourrait avoir cette approche si on *compilait* vers OCaml. En effet, dans ce cas le programme produit serait une chaîne de caractères (type `string`). En revanche, lorsqu'on produit directement des valeurs du langage hôte on se heurte au problème suivant : les programmes à combinatoriser n'ont pas tous le même type. En effet on peut avoir besoin de produire la valeur `1` comme la valeur `fun x -> x`. Évidemment, une telle fonction de combinatorisation sera rejetée par le système de type du langage hôte. Deux solutions sont envisageables : utiliser un type somme ou utiliser un langage hôte faiblement typé.

Ce problème est indépendant de l'approche par syntaxe d'ordre supérieur. En effet, il est déjà présent dans l'approche classique d'implémentation des interprètes proposée en section V.2. Par conséquent nous ne reviendrons plus sur ce point dans ce chapitre.

V.5 Traitement des fonctions non-strictes

Considérons à présent la conditionnelle *if* dans le langage que nous voulons interpréter.

```
type term =
 | Int of int
 | Var of ident
 | Abs of ident * term
 | App of term * term
 | ExtFun of value -> value
 | If of term * term * term
```

Curryfication du *if*

Nous allons nous ramener au langage considéré plus tôt en transformant la construction `If (e1,e2,e3)` en une application. Après cette transformation, nous dirons que le programme est en forme *curryfiée*. Un programme curryfié est représenté dans le type `cterm` suivant :

```
type cterm =
 | CInt of int
 | CVar of ident
 | CAbs of ident * cterm
 | CApp of cterm * cterm
 | CExtFun of value -> value
```

Ce type est similaire à celui étudié en section précédente à renommage près.

Nous allons donc considérer la conditionnelle *if-then-else* comme une fonction *if* qui s'applique à trois arguments (ses trois branches). Toutefois on ne veut pas que la branche *then* et la branche *else* soient évaluées toutes les deux. Or le langage cible est par appel par valeurs, *i.e.* les arguments passés à une fonction sont réduits avant de gérer l'application. Ce qu'il faudrait ici c'est avoir une stratégie d'appel par nom. La technique habituelle pour simuler un appel par nom lorsqu'on dispose d'un appel par valeur est l'utilisation de *glaçons*.

Notion de glaçon

Un *glaçon* est une construction du langage qui permet de geler l'évaluation d'un terme jusqu'au moment où celle-ci devient nécessaire (voir [HD97] par exemple). Le terme doit alors être *dégelé*. Ici nous utiliserons une lambda abstraction comme glaçon. En effet nous considérerons qu'une abstraction est en forme réduite. Ceci signifie qu'on n'évalue pas le corps d'une abstraction tant qu'elle n'est pas appliquée. C'est donc l'application du terme gelé qui va provoquer le dégel. On dira que le corps d'une abstraction est *protégé par un lambda* pour signifier que ce corps ne sera pas évalué avant l'application de cette abstraction.

Considérons l'exemple suivant comme illustration du fonctionnement des glaçons. Soit le terme $(\Omega\ \Omega)$ où Ω est $\lambda x.(x\ x)$. Si ce terme est passé à une fonction, il sera évalué. Son évaluation ne terminant pas, la réduction de l'application ne termine pas. On peut geler ce terme avec une abstraction : $\lambda x.(\Omega\ \Omega)$. Si cette abstraction est passée à une fonction, son corps ne sera évalué que lorsque nécessaire : si la fonction recevant ce terme a besoin de la valeur de cet argument elle devra le dégeler en l'appliquant à un argument choisi arbitrairement. Par exemple en l'appliquant à 0 : $(\lambda x.(\Omega\ \Omega)\ 0)$. Cette application se réduit en l'application $(\Omega\ \Omega)$ qui sera réduite sans fin. En revanche, si la fonction n'utilise pas son argument, elle ne le dégèlera pas et elle pourra renvoyer un résultat.

Insertion de glaçons

Nous allons donc geler les branches *then* et *else* de la conditionnelle afin que ne soit évaluée que la branche nécessaire. La constante *if*, après avoir évalué son premier argument dégèlera son deuxième *ou bien* son troisième argument selon la valeur du premier. Notons que cette méthode est très classique puisqu'elle apparaît déjà dans [Lan65b].

La fonction `traduit` donnée dans la figure 3 effectue la transformation du programme initial (type `term`) vers sa forme curryfiée et comportant les glaçons (type `cterm`).

```
let rec traduit = function
 | Int i -> CInt i
 | Var id -> CVar id
 | Abs (id,e) -> CAbs (id, traduit e)
 | App (e1,e2) -> CApp (traduit e1, traduit e2)
 | ExtFun f -> CExtFun f
 | If (e1,e2,e3) ->
    let gel_e2 = Abs(dummy_id,e2)
    and gel_e3 = Abs(dummy_id,e3)
    in CApp ( CApp ( CApp (CExtFun ite, traduit e1),traduit gel_e2),
              traduit gel_e3)

and dummy_id = " "
```

Fig. 3 – Fonction de traduction vers la forme curryfiée.

La fonction `ite` aura été définie auparavant. Elle reçoit ses trois arguments *déjà évalués* (appel par valeur). Les deux derniers étant protégés par un lambda, l'évaluation ne s'est pas

propagée sous celui-ci. Si le premier argument est un entier strictement positif, la fonction constante `ite` va *dégeler* son deuxième argument en l'appliquant à un argument quelconque. Cette application va mener à l'évaluation de l'expression initialement dans la branche *then*. Dans le cas contraire, le troisième argument sera dégelé, menant à l'évaluation de la branche *else* :

```
let ite = fun e1 -> ValFun (fun e2 -> ValFun (fun e3 ->
    match e1 with
     | ValInt i when i>0 -> app e2 dummy_val
     | _ -> app e3 dummy_val ))

and dummy_val = ValInt 0
and app = function (ValFun f) -> fun v -> f v
```

Exemple de traduction

L'expression MGS `fun (x) = if x then x+1 else x+2 fi` sera curryfiée en :

```
fun (x) = (((IF x) (fun _ = x+1)) (fun _ = x+2))
```

Plus précisément, ceci correspond à la traduction de

```
Abs ("x", If(Var "x",
             App (App(ExtFun plus, Var "x"), Int 1) ,
             App (App(ExtFun plus, Var "x"), Int 2) ))
```

en :

```
CAbs ("x", CApp(CApp(CApp(CExtFun ite,
                      CVar "x"),
                  CAbs (" ", CApp (CApp(CExtFun plus, CVar "x"), CInt 1) ),
              CAbs (" ",CApp (CApp(CExtFun plus, CVar "x"), CInt 2) ))))
```

SK-traduction

La SK-traduction pure garantit qu'aucune application dans une abstraction ne sera réduite tant que cette abstraction n'est pas elle même appliquée. Pour nous en convaincre, prenons l'expression $\lambda x.(c_1\ c_2)$ comme exemple où c_1 et c_2 sont des constantes. Elle se transforme en $S\ (K\ c_1)\ (K\ c_2)$. Dans cette expression, il n'y a pas d'application de c_1 à c_2. D'après la règle de réduction de S, c_1 ne sera appliquée à c_2 que lorsque S aura reçu un troisième argument : l'argument de la fonction définie initialement.

Plus généralement, quelles que soient la forme de e_1 et la forme de e_2 dans l'expression $\lambda x.(e_1\ e_2)$, la transformation S gèle :

- l'application de e_1 à e_2 puisque e_1 n'est pas appliqué à e_2 dans $S\ \lambda x.e_1\ \lambda x.e_2$ et que l'application ne sera faite que lorsque S recevra un troisième argument ;
- les applications pouvant apparaître dans e_1 et e_2 en distribuant l'abstraction sur chacune d'elles.

Durant le processus de traduction, la transformation S *suspend* ainsi toute application se trouvant sous un lambda. Donc la SK-traduction pure garantit que rien n'est réduit sous un lambda[3]. Ceci signifie que les applications sous un glaçon ne sont réduites que lorsque le glaçon est dégelé, *i.e.* lorsque l'application gelée est appliquée à un argument factice pour la dégeler.

Remarque : le langage hôte n'évalue pas le corps des abstractions avant qu'elles ne soient appliquées. Toutefois, ceci n'est pas exploitable ici car il ne reste pas d'abstractions dans le code produit par SK-traduction. C'est pourquoi nous avons dû analyser ici le comportement après SK-traduction des applications protégées par une abstraction.

V.6 Traitement des traits impératifs

La technique des glaçons que nous avons utilisée pour geler les branches d'un *if* peut être utilisée pour intégrer des traits impératifs au langage. Nous intégrons à présent au langage considéré l'écriture et la lecture en mémoire. Comme nous l'avons vu, ces deux traits sont suffisant puisque les autres traits impératifs peuvent être simulés dans ce langage. Par exemple le séquencement peut être simulé par un *if* : l'expression `e1 ; e2` est simulée par `if e1 then e2 else e2 fi`. De même la sortie écran *print* est de même nature que l'écriture en mémoire (l'écran est une mémoire particulière).

```
type term =
 | ...
 | Get of memid
 | Set of memid * term
```

Exemple : l'expression MGS `if m1 then m2 := m1 else x fi` s'encode par `If( Get "m1", Set("m2", Get "m1"), Var "x")`. On supposera que l'espace de noms de variables est distinct de l'espace de nom des mémoires.

Comme nous l'avons fait pour le *if*, nous allons nous ramener à la syntaxe curryfiée afin d'entrer dans le cadre de la SK-traduction. Regardons comment les traits impératifs peuvent être ramenés à cette syntaxe simple (type `cterm`).

Écriture en mémoire

L'écriture mémoire `Set(m,e)` peut être vue comme une application d'une fonction constante set_m à l'expression e. La curryfication consistera donc à transformer `Set(m,e)` en `CApp(`set_m`,`e'`)` où e' est la traduction de e.

Ici il n'est pas nécessaire d'utiliser un glaçon. En effet, l'effet aura lieu nécessairement au bon moment. Plusieurs cas peuvent se présenter :

- Si l'affectation a lieu dans le corps d'une abstraction, l'application de set_m à son argument est naturellement gelée jusqu'à l'application de celle-ci, ce qui convient.
- Si l'affectation est dans une branche *then* ou *else*, un glaçon sera inséré par la curryfication du *if* et l'application de set_m à son argument sera ainsi gelée, jusqu'au bon moment.
- Si l'affectation n'est ni dans une abstraction ni dans un *if*, elle sera évaluée comme toute application qui n'est pas protégée. Ceci convient. En effet, considérons l'exemple `m :=`

[3] En réalité dans l'implémentation des applications partielles sont réduites, comme (K c) mais aucune application issue de l'expression originale ne l'est.

`f(1)`, traduite en `CApp(`set_m`, CApp(CExtFun` f`, CInt 1))`. L'application de f à 1 sera effectuée avant de passer le résultat à set_m (appel par valeur). Ceci correspond bien au résultat attendu.

Donc la transformation de l'affectation en une application suffit à garantir que celle-ci aura lieu au bon moment. La fonction set_m peut être implémentée comme suit :

```
let set m = fun v -> begin store := (m,v)::!store ; v end
```

et la curryfication est réalisée par :

```
let rec traduit = function
 | ...
 | Set (m,e) -> CApp(CExtFun (set m), traduit e)
```

Lecture en mémoire

L'affectation correspond canoniquement à une application, ce qui n'est pas le cas de l'accès mémoire. Si on considère `Get(m)` comme une constante (notée get_m), alors l'expression `fun (x) = m` sera compilée en (K get_m). Mais l'évaluation de cette forme mène à l'évaluation de get_m. On a donc évalué get_m alors qu'elle était initialement protégée par une abstraction. On l'a évaluée *au mauvais moment* car l'accès mémoire devait se faire à l'application de la fonction et non à sa définition. Ceci montre que l'on ne peut considérer comme constantes que des valeurs dont l'évaluation ne varie pas en fonction du moment où elles sont évaluées. Le moyen de déclencher l'accès à la mémoire au bon moment est de considérer get_m comme une fonction qui attend d'être appliquée pour accéder à la mémoire. On va donc transformer `Get(m)` en une application afin de garantir que l'accès mémoire sera fait au bon moment, comme pour l'affectation. La forme `Get(m)` se traduit donc en `CApp(`get_m`, `d`)` où get_m représente la fonction constante qui renvoie la valeur de la mémoire m lorsqu'elle est appliquée à n'importe quelle valeur, et où d représente un argument quelconque (*dummy*).

Exemple : l'expression MGS `fun (x) = m1` représentée par `Abs("x", Get "m1")` est curryfiée[4] en `CAbs("x", CApp(`get_{m1}`, CInt 0))` et combinatorisée en S (K get_{m1}) (K 0).

La fonction get_m peut être implémentée comme suit :

```
let get m = fun _ -> assoc m !store
```

Le *get* étant appliqué à une valeur factice, l'argument reçu peut être abandonné, d'où le « `fun _ -> ...` ». La curryfication est réalisée par :

```
let dummy_term = CInt 0

let rec traduit = function
 | ...
 | Get m -> CApp(CExtFun (get m), dummy_term)
```

[4]Ici le terme « curryfication » est un abus de langage.

Différentes façons de suspendre l'évaluation d'un terme

Nous avons rencontré deux façons différentes de parvenir à un *gel*. Le gel repose sur le fait qu'une application n'est pas réduite sous une abstraction. Pour qu'il y ait *gel* il faut donc la réunion d'une abstraction (gelante) et d'une application (gelée).

Le *if* doit geler les applications dans ses branches, pour cela elles sont placées dans le corps d'une abstraction. L'accès mémoire, quand à lui, doit être gelé s'il est dans une abstraction (naturelle ou formée par un glaçon). L'accès mémoire doit donc prendre la forme d'une application.

Témoin de performances

On peut se servir des références mémoires pour faire de la programmation récursive. Par exemple voici la fonction de Fibonacci :

```
fun fib (i) = if i-1 then fib(i-1) + fib(i-2) else 1 fi

fib := \i.(if i-1 then 1 else !fib(i-1) + !fib(i-2) )
```

La première expression correspond à la syntaxe concrète de MGS et la seconde à la syntaxe abstraite après traitement par un *pretty printer* : le symbole `:=` dénote l'affectation et `!f` dénote l'accès mémoire à `f`. Nous nous servirons de cette fonction comme témoin pour évaluer l'efficacité de notre combinatorisation dans le chapitre suivant. Sa forme combinatorisée est :

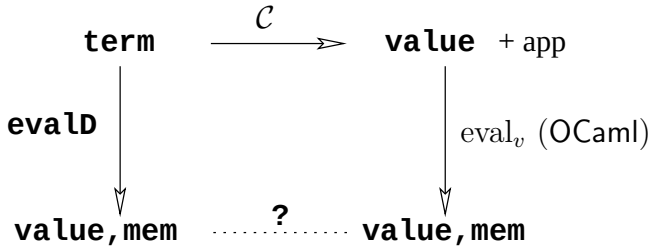
```
fib :=
((S ((S ((S (K IF)) ((S ((S (K -)) I)) (K 1)))) ((S ((S (K S))
((S ((S(K S)) ((S (K K)) (K +)))) ((S ((S (K S)) ((S ((S (K S))
((S (K K)) (K !fib)))) ((S (K K)) (K 0))))) ((S ((S (K S)) ((S ((S (K S))
((S (K K)) (K -)))) ((S (K K)) I)))) ((S (K K)) (K 1)))))))
((S ((S (K S)) ((S ((S (K S)) ((S (K K)) (K !fib)))) ((S (K K)) (K 0)))))
((S ((S (K S)) ((S ((S (K S)) ((S (K K)) (K -)))) ((S (K K)) I))))
((S (K K)) (K 2))))))) ((S (K K)) (K 1)))
```

Ici la taille du terme en forme combinatorisée est de 108 applications. La taille du terme sera utilisée comme critère mais nous regarderons également les temps d'exécution dans notre implémentation. L'implémentation sur laquelle nous effectuerons nos mesures est le noyau d'un interprète MGS complet et non une maquette dédiée uniquement à l'étude de la SK-traduction. L'évaluation de $f(30)$ prend entre 6.2 et 6.5 secondes sur notre machine (P-IV 2.4GHz, OCaml 3.06, Linux Debian woody). À titre de comparaison, l'interprète OCaml évalue $f(30)$ en 0.14 à 0.17 secondes (même programme avec des références). Il y a donc environ un facteur 40 entre les deux implémentations.

V.7 Correction

D'après [GD92], évaluer un programme du λ-calcul avec constantes sous sa forme combinatorisée avec une stratégie d'appel par valeur est équivalent à évaluer ce programme directement avec une stratégie d'appel par valeur. Ceci correspond bien à notre approche puisque notre langage source comme notre langage hôte sont à appel par valeur.

Toutefois, l'intégration des traits impératifs dans la logique combinatoire n'a pas été étudiée à notre connaissance. Nous proposons en annexe de ce document une preuve de la correction de notre approche. Nous établissons que le « ? » dans le schema-ci dessous est une relation d'équivalence, ce qui confirme les résultats obtenus en pratique avec l'interprète MGS.



V.8 Bilan

Dans ce chapitre nous avons vu comment interpréter un langage fonctionnel sans gérer nous-mêmes les mécanismes de réduction d'application. Pour cela nous avons utilisé la SK-traduction.

Il existe peu ou pas d'implémentations d'interprètes de langages de programmation fondés sur une syntaxe d'ordre supérieur. Ceci est dû à la difficulté de traduire au vol un arbre de syntaxe abstraite contenant des abstractions en un arbre de syntaxe d'ordre supérieur.

Nous avons vu comment cette approche peut-être adaptée à un langage avec fonctions non-strictes et avec des traits impératifs grâce à l'utilisation de glaçons.

Dans le prochain chapitre, nous montrons comment rendre ce schéma d'interprétation efficace.

Chapitre VI

Optimisations du schéma d'évaluation

Les travaux présentés dans ce chapitre ont été partiellement publiés dans [Coh05].

Dans ce chapitre, nous étudions comment améliorer les performances de l'interprète défini dans le précédent chapitre. Nous identifions trois sources d'inefficacité :

- Distribution systématique des lambdas : la règle de combinatorisation de $\lambda x.(e_1\ e_2)$ distribue l'abstraction de x sur e_1 et e_2 alors que x n'apparaît pas forcément dans e_1 et e_2.
- Les applications et les abstractions sont combinatorisées une par une : par exemple l'expression $\lambda x.\lambda y.(f\ 0)$ sera combinatorisé en plusieurs étapes, chacune introduisant un combinateur alors qu'on peut considérer la combinatorisation des deux abstractions en une seule étape introduisant un unique combinateur.
- Les glaçons alourdissent le terme considéré : la compilation et l'évaluation sont ainsi moins efficaces.

Nous avons également mentionné les constructeurs dans le type `value` qu'il faut sans cesse lire ou construire mais nous ne traitons pas ce problème dans ce chapitre (ce type de problèmes est abordé dans le chapitre VII sur le typage).

Pour résoudre le premier problème nous étudions les optimisations classiques de la SK-traduction et nous les adaptons à nos besoins en section VI.1. Pour le second problème nous mettons en œuvre des combinateurs n-aires dans la section VI.2. Enfin nous montrons qu'on peut se passer de glaçons en section VI.3.

VI.1 Optimisations classiques

Dans cette section nous considérons les optimisations classiques de la SK-traduction et nous étudions dans quelle mesure elles sont compatibles avec nos besoins. Nous évoquerons trois optimisations : l'η-réduction, l'utilisation de B et C, et enfin la simplification d'un S en K.

Considérons le terme $\lambda x.(M\ x)$ où $M = (\Omega\ \Omega)$ est un terme purement fonctionnel dont la réduction ne termine pas. Si on applique l'η-réduction dessus on obtient M. Ce terme n'étant plus dans une abstraction, il peut être réduit et cette réduction ne terminera pas. L'η-réduction nous a amené à évaluer un terme qui était sous un lambda dans le programme initial. Ici on voit bien que la protection par un lambda peut servir à gérer le moment où les calculs/actions ont lieu, comme nous l'avons montré dans la section V.5. En effet, on s'attend ici à voir la fonction

boucler lorsqu'on l'utilise et non pas lorsqu'on la définit. Effectuer les calculs au plus tôt ou au plus tard est un choix dans la conception d'un langage de programmation. Puisque nous sommes en présence de caractères impératifs, on se doit de respecter une politique d'évaluation *au bon moment*. Par exemple dans $\lambda x.(+\ (get_a\ 0)\ x)$ il y a un accès mémoire à a. L'accès doit se faire *au moment où la fonction est utilisée* et non pas au moment où elle est définie. Ici l'η-réduction viole ce principe puisque la fonction résultante ne serait plus définie par une abstraction mais par une application : $(+\ (get_a\ ()))$. Nous parlerons d'*évaporation des lambdas* lorsqu'un lambda disparaît, ne retardant plus l'évaluation d'un terme.

Ce phénomène apparaît aussi avec les optimisations liées à l'utilisation des combinateurs B et C de [CF58] définis par $B\,e\,f\,x \to e\,(f\,x)$ et $C\,f\,e\,x \to (f\,x)\,e$. Par exemple le terme $\lambda x.(M\ x)$ se compile en $S\ (\lambda x.M)\ I$ où M est toujours protégé, mais se compile également en $B\ M\ I$ si on utilise B (lorsque x n'apparaît pas libre dans M). Dans ce cas le lambda s'est évaporé : selon la transformation B, x n'apparaissant pas dans M, inutile de faire dépendre de x l'évaluation de M. Dans les optimisations B et C, les lambdas sont vus uniquement comme des lieurs et non comme des protecteurs[1].

Enfin ce problème apparaît encore lorsque $\lambda x.(c_1\ c_2)$ est SK-traduit en $S\ (K\ c_1)\ (K\ c_2)$ puis optimisé en $K\ (c_1\ c_2)$ (simplification d'un S en K).

Nous ne pouvons donc pas utiliser ces trois optimisations telles qu'elles sont présentées dans la littérature. Nous montrons à présent comment adapter ces optimisations à notre usage.

η-réduction

Le problème que nous voulons éviter est la réduction d'une application située dans une abstraction due à l'évaporation d'un lambda. Nous avons vu que transformer $\lambda x.(+\ (get_a\ 0)\ x)$ en $+\ (get_a\ 0)$ n'était pas correct car les applications ne sont plus gelées. Toutefois il existe des cas où aucune application n'est dégelée en enlevant un lambda. Par exemple on peut transformer $\lambda x.(+\ x)$ en $+$ ou bien $\lambda x.(y\ x)$ en y sans modifier la sémantique du programme. En fait, si c ne contient pas d'applications[2], on peut η-réduire $\lambda x.(c\ x)$.

Nous retiendrons ce critère pour pouvoir η-réduire. Toutefois, dans l'implémentation actuelle de l'interprète MGS, l'analyse est plus fine. Nous η-réduisons par exemple $\lambda x.((+\ 1)\ x)$. Certes l'application $(+\ 1)$ est dégelée mais

- ce dégel ne viole pas la sémantique des traits impératifs ou des fonctions non-strictes,
- le calcul dégelé prend un temps réduit et constant.

Une heuristique détermine quelles applications peuvent être dégelées ou non[3].

Remarque : l'η-réduction ne peut jamais s'appliquer aux lambdas jouant le rôle de glaçons dans une conditionnelle car les variables abstraites par les glaçons ne sont jamais réutilisées.

[1] Les lambdas sont des lieurs dans l'appel par nom et des lieurs/protecteurs dans l'appel par valeur, voir [HD97].

[2] On suppose qu'il n'y a aucune constante qui boucle. La non terminaison est due dans notre langage à une succession infinie de réductions. La valeur $\perp$ mentionnée au chapitre VII n'est pas une constante dans notre langage. Il s'agit d'un symbole dénotant le *résultat* d'un programme ne terminant pas. Ainsi, considérer l'exemple $\lambda x.\perp$ n'a pas de sens (on aurait pu penser que cette abstraction se compile en $(K\ \perp)$). On lui préférera l'exemple cité plus haut $\lambda x.M$ où M est $(\Omega\ \Omega)$ et contient donc au moins une application.

[3] Dans l'interprète actuel, un terme est qualifié de pur et peut être dégelé arbitrairement s'il ne contient ni traits impératifs, ni fonctions utilisateur. Sinon il sera qualifié d'impur et on ne pourra le dégeler par évaporation de lambdas. Ceci est une heuristique car on peut sans doute trouver un critère plus fin et on suppose arbitrairement que les fonctions prédéfinies coûtent peu à l'exécution.

Transformer S en K

Cette optimisation consiste à transformer $S\ (K\ c_1)\ (K\ c_2)$ en $K\ (c_1\ c_2)$. On peut aussi la voir sous la forme de la transformation $\lambda x.(e_1\ e_2) \Rightarrow K\ (e_1\ e_2)$ applicable lorsque x n'apparaît libre ni dans e_1 ni dans e_2.

L'application gelée $\lambda\_.(e_1\ e_2)$ se compile en $S\ (K\ e_1)\ (K\ e_2)$ lorsque e_1 et e_2 sont des constantes ou des variables. Comme nous l'avons vu, e_1 n'est pas appliqué à e_2 dans le terme compilé. Ici S gèle l'application tant qu'il n'a pas reçu d'argument supplémentaire (la SK-traduction pure garantit l'absence de réduction sous une abstraction). En revanche la simplification que nous examinons va dégeler l'application puisque cette expression va se transformer en $K\ (e_1\ e_2)$. Le langage hôte étant par appel par valeurs, l'application $(e_1\ e_2)$ sera évaluée avant d'être passée en argument à K. Cette optimisation n'est donc pas correcte pour notre utilisation.

Toutefois nous allons voir dans les paragraphes suivants qu'une variation autour des simplifications B et C permet de ne pas distribuer le lambda inutilement sur e_1 et e_2.

Transformer S en B ou C

Afin d'introduire les optimisations B et C, rappelons à nouveau le principe du combinateur S. Partons de l'expression $\lambda x.(e_1\ e_2)$. Pour introduire le combinateur S on part du fait qu'appliquer cette fonction à un argument v revient à :

- appliquer l'abstraction de e_1 par rapport à x $(\lambda x.e_1)$ à cet argument v,
- appliquer l'abstraction de e_2 par rapport à x $(\lambda x.e_2)$ à v
- puis appliquer le premier résultat au second.

L'introduction des lambdas se fait à la combinatorisation (transformation S) et les applications se font à la réduction (réduction S).

On peut remarquer que x n'est pas forcément utilisé à la fois dans e_1 et dans e_2. Par exemple, si x n'apparaît pas libre dans e_1 il n'est pas nécessaire de construire l'abstraction $\lambda x.e_1$ et de l'appliquer à v puisque le résultat serait e_1. C'est ici que l'on fait intervenir le combinateur B. La transformation B est similaire à la transformation S mais elle ne propage le lambda que dans e_2. De même la réduction B n'applique que l'abstraction $\lambda x.e_2$ à v avant d'appliquer e_1 au résultat de cette application. Ainsi l'expression de combinateurs résultante est plus compacte.

Le combinateur C fonctionne sur le même principe que B mais s'applique lorsque x n'apparaît pas dans e_2.

	réduction	transformation
S	$S\ f\ g\ x \rightarrow (f\ x)\ (g\ x)$	$\lambda x.(e_1\ e_2) \Rightarrow S\ (\lambda x.e_1)\ (\lambda x.e_2)$
B	$B\ e\ f\ x \rightarrow e\ (f\ x)$	$\lambda x.(e_1\ e_2) \Rightarrow B\ e_1\ (\lambda x.e_2)$
C	$C\ f\ e\ x \rightarrow (f\ x)\ e$	$\lambda x.(e_1\ e_2) \Rightarrow C\ (\lambda x.e_1)\ e_2$

La transformation B ne s'applique que lorsque x n'apparaît pas libre dans e_1 et la transformation C ne s'applique que lorsque x n'apparaît pas dans e_2. Usuellement, lorsque x n'apparaît libre ni dans e_1 ni dans e_2 on utilise l'optimisation K présentée plus tôt.

Comme nous l'avons vu plus haut, les transformations B et C participent à l'évaporation des lambdas. Dans le λ-calcul pur, ceci ne pose pas de problème car le lambda ne joue que le rôle de lieur. Mais dans notre cas il joue également le rôle de retardateur. Par conséquent nous ne pouvons appliquer ces transformations lorsque la disparition d'un lambda dégèle une application.

Ainsi nous introduisons une condition nouvelle à l'application des optimisations *à la* B et C : on peut ne pas distribuer un lambda sur une expression uniquement si elle ne contient pas d'application. Cette condition peut être affaiblie[4]. Pour marquer la différence avec les optimisations B et C habituelles, nous utiliserons une notation différente. Nous noterons $S_{\swarrow}$ le combinateur usuellement noté C (on distribue l'argument sur le terme de gauche). Nous noterons $S_{\searrow}$ le combinateur B (on distribue l'argument à droite). On pourra noter également $S_{\downarrow\!\downarrow}$ le combinateur S (on distribue l'argument sur les deux termes).

	réduction	transformation
$S_{\downarrow\!\downarrow}$	$S_{\downarrow\!\downarrow}\ f\ g\ x \rightarrow (f\ x)\ (g\ x)$	$\lambda x.(e_1\ e_2) \Rightarrow S_{\downarrow\!\downarrow}\ (\lambda x.e_1)\ (\lambda x.e_2)$
$S_{\searrow}$	$S_{\searrow}\ e\ f\ x \rightarrow e\ (f\ x)$	$\lambda x.(c\ e) \Rightarrow S_{\searrow}\ c\ (\lambda x.e)$
$S_{\swarrow}$	$S_{\swarrow}\ f\ e\ x \rightarrow (f\ x)\ e$	$\lambda x.(e\ c) \Rightarrow S_{\swarrow}\ (\lambda x.e)\ c$

Dans ces transformations, c dénote une expression sans applications. En fait, c ne peut être une abstraction (les combinateurs sont vus comme des constantes) car on applique ces transformations sur les abstractions les plus internes des termes.

Nous généralisons les idées ci-dessus au cas où e_1 et e_2 sont des constantes ou des variables différentes de x. Dans ce cas aucun des deux termes n'est abstrait par la traduction et, à la réduction, l'argument n'est distribué sur aucun des deux[5]. Nous noterons $S_{\varnothing}$ le nouveau combinateur (l'argument n'est pas distribué mais l'application est déclenchée).

	réduction	transformation
$S_{\varnothing}$	$S_{\varnothing}\ e_1\ e_2\ x \rightarrow e_1\ e_2$	$\lambda x.(c_1\ c_2) \Rightarrow S_{\varnothing}\ c_1\ c_2$

Cette transformation peut être utilisée à la place de l'optimisation $\lambda x.(e_1\ e_2) \Rightarrow K\ (e_1\ e_2)$ (simplification d'un S en K). En effet cette dernière avait l'inconvénient de ne pas geler l'application de e_1 à e_2, ce qui est réparé par la transformation $S_{\varnothing}$. Cette remarque montre que c'est le combinateur S qui joue le rôle du retardateur de réduction dans la forme combinatorisée.

```
let comb_S_left  =
      fun f -> ValFun (fun g -> ValFun (fun x -> app (app f x) g ))
and comb_S_right =
      fun f -> ValFun (fun g -> ValFun (fun x -> app f (app g x) ))
and comb_S_none  =
      fun f -> ValFun (fun g -> ValFun (fun x -> app f g ))
```

Exemple

Le terme $\lambda x.\lambda y.((+\ x)\ y)$ se combinatorise en

$$S_{\swarrow}\ (S_{\searrow}\ S_{\downarrow\!\downarrow}\ (S_{\downarrow\!\downarrow}\ (S_{\varnothing}\ S_{\varnothing}\ +)\ I))\ I$$

qui contient 8 applications au lieu de

$$((S\ ((S\ (K\ S))\ ((S\ ((S\ (K\ S))\ ((S\ (K\ K))\ (K\ +))))\ ((S\ (K\ K))\ I))))\ (K\ I))$$

[4]Dans l'interprète on utilise l'heuristique qui qualifie un terme de pur ou d'impur pour décider de distribuer le lambda ou non.

[5]Dans [PJ87] on note j la direction vers aucun des sous-termes, les autres directions sont s, b et c. Ceci apparaît dans les *chaînes de directions*.

qui contient 18 applications.

Remarque : il n'y a plus de K dans cet exemple. En effet on ne forme jamais $\lambda x.c$ par la traduction. Mais s'il y en a dans le programme source alors il y aura des K dans le programme combinatorisé.

Témoin

Nous donnons ci-dessous le résultat de la transformation de la fonction de Fibonacci en utilisant les quatre combinateurs proposés. Le combinateur $S_{\swarrow\!\searrow}$ est dénoté par `YY`, $S_{\swarrow}$ par `YN`, $S_{\searrow}$ par `NY` et $S_{\varnothing}$ par `NN`.

```
fib :=
((YN ((YY ((NY IF) ((YN ((NY -) I)) 1))) ((YY ((NY YY) ((NY (NY +))
((YY ((NY YY) ((NN (NN !fib)) 0))) ((YN ((NY YN) ((NY (NN -)) I))) 1)))))
((YY ((NY YY) ((NN (NN !fib)) 0))) ((YN ((NY YN) ((NY (NN -)) I))) 2)))))
(K 1))
```

Le terme combinatorisé est composé de 46 applications contre 108 pour la SK-compilation pure.

VI.2 Optimisation non classique : les combinateurs n-aires

L'utilisation des combinateurs avec distribution sélective que nous venons de voir est moins souple dans notre cadre que dans celui du lambda calcul pur dans lequel les optimisations B et C ont été conçues.

C'est pourquoi nous nous tournons à présent vers d'autres optimisations. Celles que nous proposons dans cette section tentent de factoriser le processus de distribution des lambdas lors de la SK-traduction *en tenant compte des applications et abstractions multiples*. Comme S à été introduit en reposant sur l'égalité $\lambda x.(e_1\ e_2)\ a = (\lambda x.e_1\ a)\ (\lambda x.e_2\ a)$, nous allons nous intéresser à des variations de S basées sur des égalités comme les deux suivantes :

$$(\lambda x.e_1\ e_2\ e_3)\ a = (\lambda x.e_1\ a)\ (\lambda x.e_2\ a)\ (\lambda x.e_3\ a)$$

$$(\lambda x.\lambda y.(e_1\ e_2))\ a\ b = ((\lambda x.\lambda y.e_1)\ a\ b)\ ((\lambda x.\lambda y.e_2)\ a\ b)$$

VI.2.1 Combinateurs vectoriels de Diller

Diller [Dil02] propose deux familles infinies de combinateurs. Dans la première, un combinateur est dénoté par une chaîne de booléens `y` et `n`. Dans la seconde, un combinateur est dénoté par des matrices de booléens `y` et `n`.

Chaînes `yn`

Exemples de règles de réduction :

$$\begin{aligned}
\mathsf{yy}\ P\ Q\ R &\to (P\ R)\ (Q\ R)\\
\mathsf{yn}\ P\ Q\ R &\to (P\ R)\ Q\\
\mathsf{ny}\ P\ Q\ R &\to P\ (Q\ R)\\
\mathsf{nn}\ P\ Q\ R &\to P\ Q
\end{aligned}$$

On peut remarquer que yy est équivalent à S (ou $S_{\swarrow\searrow}$), que yn est équivalent à C (ou $S_{\swarrow}$), que ny est équivalent à B (ou $S_{\searrow}$) et que nn est équivalent à $S_{\varnothing}$. Un y en première position de la chaîne signifie que R doit être distribué sur P. Un y en seconde position signifie que R soit être distribué sur Q. On peut remplacer un y par un n pour ne pas distribuer R sur P ou Q. Ceci permet de généraliser le combinateurs S et ses optimisations à des applications à plusieurs arguments. Pour une chaine yn de longueur n, S et ses optimisations sont généralisés aux applications $(n-1)$-aires. Si β est une chaîne yn, la réduction de β suit le schéma suivant :

$$\beta\ P_1\ P_2\ ...\ P_m\ R \rightarrow P'_1\ P'_2\ ...\ P'_m$$

où m est la longueur de β et P'_i vaut $(P_i\ R)$ si β_i est y et vaut P_i sinon.

Matrices yn

Les combinateurs S, B et C distribuent un argument sur une application simple et les chaînes yn sur une application n-aires. Les matrices yn de Diller permettent de distribuer plusieurs arguments sur des applications n-aires. Exemple :

$$\begin{smallmatrix} \mathsf{n} & \mathsf{n} & \mathsf{y} & \mathsf{n} \\ \mathsf{y} & \mathsf{n} & \mathsf{y} & \mathsf{n} \\ \mathsf{n} & \mathsf{n} & \mathsf{y} & \mathsf{y} \end{smallmatrix}\ P_1\ P_2\ P_3\ P_4\ R_1\ R_2\ R_3 \rightarrow (P_1\ R_2)\ P_2\ (P_3\ R_1\ R_2\ R_3)\ (P_4\ R_3)$$

Dans cet exemple chacune des 4 colonnes indique comment les trois arguments doivent être distribués. Une chaîne yn peut être vue comme une matrice à une ligne (un seul argument à distribuer).

Confrontations à nos besoins

Les chaînes et les matrices de Diller proposent :

- Des combinateurs pour rendre compte des applications et abstractions multiples ;
- Une stratégie de distribution des abstractions à la traduction seulement lorsque nécessaire (à la B et C) ;
- Un algorithme de combinatorisation approprié au lambda calcul pur ou avec constantes.

Dans notre contexte, la stratégie de non-distribution doit être adaptée pour être sûr que la non-distribution d'un lambda ne provoque pas le dégel d'une application. Comme pour la combinatorisation avec $S_{\swarrow}$, $S_{\searrow}$ et $S_{\varnothing}$ nous devons distibuer le lambda sur les termes contenant une application. L'algorithme de combinatorisation doit donc tenir compte de cette modification.

Considérons l'expression $\lambda x.\lambda y.(+\ (+\ x\ 1)\ (+\ y\ 2)$ comme exemple et comparons le terme obtenu par l'algorithme de Diller et par son adaptation pour notre contexte. L'algorithme de Diller produit :

$$\left(\begin{smallmatrix} \mathsf{n} & \mathsf{y} & \mathsf{n} \\ \mathsf{n} & \mathsf{n} & \mathsf{y} \end{smallmatrix}\ +\ (\mathsf{nyn}\ +\ I\ 1)\ (\mathsf{nyn}\ +\ I\ 2)\right)$$

Alors qu'avec notre restriction sur la distribution sélective des lambdas on obtient :

$$\left(\begin{smallmatrix} \mathsf{n} & \mathsf{y} & \mathsf{y} \\ \mathsf{n} & \mathsf{y} & \mathsf{y} \end{smallmatrix}\ +\ (\begin{smallmatrix} \mathsf{n} & \mathsf{y} & \mathsf{n} \\ \mathsf{n} & \mathsf{n} & \mathsf{n} \end{smallmatrix}\ +\ I\ 1)\ (\begin{smallmatrix} \mathsf{n} & \mathsf{n} & \mathsf{n} \\ \mathsf{n} & \mathsf{y} & \mathsf{n} \end{smallmatrix}\ +\ I\ 2)\right)$$

Il y a autant de combinateurs dans les deux résultats mais :

- La première matrice distribue plus de lambdas/arguments
- Les deux autres sont de taille supérieure ($3 * 2$ au lieu de $3 * 1$).

Comme l'on pouvait s'y attendre, l'expression obtenue dans notre contexte est moins efficace.

Les points négatifs de cette approche sont donc :

– Notre critère de non-distribution des lambdas étant plus restrictif que dans le cas du lambda calcul pur, l'approche de Diller dans notre contexte est moins efficace que pour un langage purement fonctionnel.
– Par ailleurs :
  • soit on représente réellement un combinateur-matrice par une matrice et son application est peu efficace (le combinateur est interprété),
  • soit on implémente un combinateur-matrice par une fonction optimisée (le combinateur est compilé).

  Le coût pour prédéfinir toutes les matrices que l'on peut raisonnablement s'attendre à rencontrer est prohibitif. En effet, pour une abstraction de p variables sur une application à q arguments il y a $2^{p*(q+1)}$ matrices possibles de $p*(q+1)$ booléens. Pour p=3 et q=3 cela fait 4096 matrices possibles. Il nous faut donc les « interpréter ».

Donc, en supposant que les combinateurs de Diller soient efficaces dans une approche compilée et pour le lambda calcul pur, il n'est pas certain qu'ils soient aussi efficaces dans notre approche interprétée et avec traits impératifs.

Cependant l'idée des combinateurs n-aires est séduisante. Nous proposons donc de reprendre l'idée des combinateurs pour applications et abstractions multiples en laissant de côté la distribution sélective des lambdas. Ainsi toutes les étapes seront plus simples et plus rapides que chez Diller :

– l'algorithme de combinatorisation ;
– la génération des combinateurs ;
– leur exécution.

Ainsi, notre approche produira théoriquement des programmes moins efficaces (il y a plus d'applications à réduire puisqu'on distribue tous les lambdas) mais, chacune des étapes étant plus rapide (notamment l'application) et plus simple, et la non-distibution des lambdas arrivant de toute façon peu souvent dans notre contexte, le compromis nous semble acceptable.

VI.2.2 Applications multiples

Les applications multiples sont souvent utilisées dans les programmes réels. Il suffit de se souvenir que la majorité des opérateurs sont binaires (+, *, = ou la concaténation et la construction de structures de données par exemple). La conditionnelle *IF* est également très utilisée et est vue comme un opérateur ternaire dans notre langage. Enfin, la SK-traduction en crée de nombreuses (avant d'en détruire la plupart si la stratégie choisie est la stratégie *inner-most*).

La SK-traduction pure de $\lambda x.(f\ e_1\ e_2)$ donne $S\ (S\ \lambda x.f\ \lambda x.e_1)\ \lambda x.e_2$ et celle de $\lambda x.(f\ e_1\ e_2\ e_3)$ donne $S\ (S\ (S\ \lambda x.f\ \lambda x.e_1)\ \lambda x.e_2)\ \lambda x.e_3$

Le forme combinatorisée d'une application à deux arguments contient deux combinateurs S. Le nombre d'applications y est de quatre : les deux initiales et deux dues à l'introduction des S. Idem pour l'application à trois arguments : trois S et six applications. Ici la linéarité du nombre de combinateurs et de nouvelles applications introduits peut sembler raisonnable. Pourtant il faut se souvenir que si ce terme est dans une abstraction, le lambda devra être distribué à travers chacune des applications du terme compilé (et dans le programme curryfié il y a plus de lambdas que dans le programme source à cause des glaçons).

Nous introduisons des combinateurs notés S_n obéissant aux règles suivantes :

	transformation	réduction
S_2	$\lambda x.(f\ e_1\ e_2) \Rightarrow S_2\ \lambda x.f\ \lambda x.e_1\ \lambda x.e_2$	$S_2\ f\ g_1\ g_2\ x \rightarrow (f\ x)\ (g_1\ x)\ (g_2\ x)$
S_3	$\lambda x.(f\ e_1\ e_2\ e_3) \Rightarrow S_3\ \lambda x.f\ \lambda x.e_1\ \lambda x.e_2\ \lambda x.e_3$	$S_3\ f\ g_1\ g_2\ g_3\ x \rightarrow (f\ x)\ (g_1\ x)\ (g_2\ x)\ (g_3\ x)$
...	...	...

Forme générale :

$$\lambda x.(f\ e_1\ e_2\ ...\ e_n) \Rightarrow S_n\ \lambda x.f\ \lambda x.e_1\ \lambda x.e_2\ ...\ \lambda x.e_n$$

$$S_n\ f\ g_1\ g_2\ ...\ g_n\ a \rightarrow (f\ a)\ (g_1\ a)\ (g_2\ a)\ ...\ (g_n\ a)$$

En utilisant les S_n, le coût d'une distribution d'un lambda sur une application multiple n'est plus linéaire avec le nombre d'applications mais constant. En effet la traduction n'introduit qu'un combinateur et ne fait apparaître qu'une application supplémentaire.

Notons que S_2 a la même sémantique que yyy chez Diller et S_3 que yyyy. Dans S_i, l'indice i fait référence à une application à i arguments. Chez Diller une chaîne de longueur i est associée à une application à i opérandes (la fonction appliquée et les $i-1$ arguments).

Exemple

En utilisant les règles de transformation S_n, le terme $\lambda x.\lambda y.(f\ e_1\ e_2)$ se transforme comme suit :

$$\begin{gathered}\lambda x.(S_2\ \lambda y.f\ \lambda y.e1\ \lambda y.e2)\\ S_3\ \lambda x.S2\ \lambda x.\lambda y.f\ \lambda x.\lambda y.e1\ \lambda x.\lambda y.e2\\ S_3\ (K\ S_2)\ \lambda x.\lambda y.f\ \lambda x.\lambda y.e_1\ \lambda x.\lambda y.e_2\end{gathered}$$

Le résultat obtenu en utilisant la SK-traduction pure est :

$$S\ (S\ (K\ S)\ (S\ (S\ (K\ S)\ \lambda x.\lambda y.f)\ \lambda x.\lambda y.e_1))\ \lambda x.\lambda y.e_2$$

VI.2.3 Abstractions multiples

Le terme $\lambda x.\lambda y.(e_1\ e_2)$ se combinatorise en $(S\ (S\ (K\ S)\ \lambda x.\lambda y.e_1)\ \lambda x.\lambda y.e_2)$ sans considérer les applications multiples et en $(S_2\ (K\ S)\ \lambda x.\lambda y.e_1\ \lambda x.\lambda y.e_2)$ sinon.

Le combinateur S' de [CF58] défini par $S'\,c\,f\,g\,x \rightarrow c\,(f\,x)\,(g\,x)$ permet de n'introduire que n combinateurs pour une abstraction sur n arguments. Par exemple $\lambda x_1.\lambda x_2.(f\,e)$ peut se combinatoriser en $S'\,S\ \lambda x_1.\lambda x_2.f\ x_1.\lambda x_2.e$. Mais on peut aller plus loin. Introduisons le combinateur S^2 qui distribue directement les deux abstractions sur les termes de l'application.

$$\lambda x.\lambda y.(e_1\ e_2) \Rightarrow S^2\ \lambda x.\lambda y.e_1\ \lambda x.\lambda y.e_2$$

Deux règles de réduction sont envisageables :

$$S^2\ f\ g\ x\ y \rightarrow (f\ x\ y)\ (g\ x\ y) \text{ et } S^2\ f\ g\ x \rightarrow S\ (f\ x)\ (g\ x)$$

De manière générale, ces deux règles ne sont pas équivalentes. Mais étant donné que les formes à réduire ne sont pas arbitraires mais ont été produites par notre algorithme de combinatorisation, les deux règles de réduction conviennent.

Le combinateur S^2 se généralise aux combinateurs S^n pour n abstractions et permet de n'introduire qu'un seul combinateur pour une abstraction multiple. L'introduction et la réduction de ces combinateurs se fait selon les règles suivantes (deux règles de réduction possibles) :

$$\lambda x_1 ... \lambda x_n.(e_1\ e_2) \Rightarrow S^n\ \lambda x_1 ... \lambda x_n.e_1\ \lambda x_1 ... \lambda x_n.e_2$$

$$S^n\ f\ g\ x \rightarrow S^{n-1}\ (f\ x)\ (g\ x)$$

$$S^n\ f\ g\ a_1\ a_2\ ...\ a_n \rightarrow (f\ a_1\ a_2\ ...\ a_n)\ (g\ a_1\ a_2\ ...\ a_n)$$

Notons que l'équivalent de S^2 chez Diller est $\begin{smallmatrix} y & y \\ y & y \end{smallmatrix}$ et celui de S^3 est $\begin{smallmatrix} y & y \\ y & y \\ y & y \end{smallmatrix}$.

Stratégie de SK-traduction

Pour que cette transformation ait un intérêt il faut changer de stratégie d'application des règles de SK-traduction. Jusqu'à présent nous les appliquions sur les abstractions les plus internes. À présent nous allons les appliquer sur les abstractions les plus externes. Ainsi :

- les formes $\lambda x_1 .... \lambda x_n.(e_1\ e_2)$ seront facilement détectées et
- les lambda les plus externes viendront se coller aux lambdas internes permettant d'utiliser moins de combinateurs, comme dans l'exemple suivant :

$$\lambda x.(\ (\lambda y.e_1)\ e_2\ ) \Rightarrow S\ \lambda x.\lambda y.e_1\ \lambda x.e_2$$

Dans la forme résultante on va pouvoir transformer les deux abstractions de $\lambda x.\lambda y.e_1$ avec un seul combinateur alors qu'en transformant les abstraction les plus internes on aurait transformé les deux abstractions séparément.

VI.2.4 Abstractions et applications multiples

On peut combiner la transformation d'abstractions multiples et d'applications multiples. Par exemple on transformera $\lambda x.\lambda y.(f\ e_1\ e_2)$ en $S_2^2\ \ \lambda x.\lambda y.f\ \ \lambda x.\lambda y.e_1\ \ \lambda x.\lambda y.e_2$. La règle de transformation correspondante est la suivante :

$$\lambda x_1.\lambda x_2 ....\lambda x_n.(f\ e_1\ e_2\ ...\ e_m) \Rightarrow$$

$$S_m^n\ \ \lambda x_1.\lambda x_2 ....\lambda x_n.f\ \ \lambda x_1.\lambda x_2 ....\lambda x_n.e_1\ \ \lambda x_1.\lambda x_2 ....\lambda x_n.e_2\ \ ...\ \ \lambda x_1.\lambda x_2 ....\lambda x_n.e_m$$

On peut donner deux règles de réduction :

$$S_m^n\ f\ g_1\ g_2\ ...\ g_m\ a_1\ a_2\ ...\ a_n \rightarrow$$

$$(f\ a_1\ a_2\ ...\ a_n)\ (g_1\ a_1\ a_2\ ...\ a_n)\ (g_2\ a_1\ a_2\ ...\ a_n)\ ...\ (g_m\ a_1\ a_2\ ...\ a_n)$$

ou

$$S_m^n\ f\ g_1\ g_2\ ...\ g_m\ a \rightarrow S_m^{n-1}\ (f\ a)\ (g_1\ a)\ (g_2\ a)\ ...\ (g_m\ a)$$

La seconde permet de réduire des applications partielles. Elle permet d'exprimer la sémantique de S_m^n en fonction de celle de S_m^{n-1} (rappelons que S_m^1 correspond à S_m). Notons que l'équivalent de S_2^2 chez Diller est $\begin{smallmatrix} y & y & y \\ y & y & y \end{smallmatrix}$

VI.2.5 K et I n-aires

Une fois qu'on a « fait descendre » toutes les abstractions au plus profond, on peut se trouver dans deux cas. On trouve la forme $\lambda x_1.\lambda x_2....\lambda x_n.x_i$ avec $1 \leq i \leq n$, soit la forme $\lambda x_1.\lambda x_2....\lambda x_n.c$ (ici c ne peut plus être une variable car un programme est clos). Commençons par généraliser K pour traiter le premier cas.

Généralisation de K

Nous introduisons des combinateurs K^n tels que :

transformation	réduction	implémentation
$\lambda x_1.\lambda x_2....\lambda x_n.c \Rightarrow K^n c$	$K^n\ c\ R_1\ R_2\ ...\ R_n \rightarrow c$	$K^n = \lambda c.\lambda x_1.\lambda x_2....\lambda x_n.c$

L'équivalent de K chez Diller est `n` et celui de K^3 est $\begin{smallmatrix}\mathsf{n}\\ \mathsf{n}\\ \mathsf{n}\end{smallmatrix}$.

La SK-traduction simple transforme $\lambda x.\lambda y.c$ en $\lambda x.K\ c$, puis en $S\ \lambda x.K\ \lambda x.c$, puis en $S\ KK\ Kc$ et après optimisation en $(K\ (K\ c))$. En fait $(K^2\ c)$ est équivalent à $K\ (K\ c)$. On peut généraliser à n. Même si $(K^2\ c)$ met autant de temps à être évalué que $K\ (K\ c)$ la forme en K^2 est avantageuse. En effet si elle se trouve sous un lambda, celui-ci n'a à être distribué que sur une application au lieu de 2 (on peut généraliser à n). L'utilisation d'un K^n a donc l'avantage de donner un code compact lorsqu'elle se trouve sous un lambda.

Remarque : comme on distribue systématiquement les abstractions sur les applications, on aura souvent des K alors que les K disparaissent si on distribue les abstractions sélectivement.

Généralisation de I

On peut voir la forme $\lambda x_1.\ ...\lambda x_n.x_i$ comme une projection à n arguments sur le $i$ème argument. Toutefois pour des raisons de simplicité nous préférons voir cette forme sous l'angle suivant : $\lambda x_1.\ ...\ \lambda x_{i-1}.(\lambda x_i.\ ...\ \lambda x_n.x_i)$. Nous introduisons des combinateurs I^n tels que :

transformation	réduction	implémentation
$\lambda x_1.\ ...\lambda x_n.x_1 \Rightarrow I^n$	$I^n\ \ R_1\ R_2\ ...\ R_n \rightarrow R_1$	$I^n = \lambda x_1.\lambda x_2.\ ...\lambda x_n.x_1$

Ainsi la forme considérée ci-dessus se compile en $\lambda x_1.\ ...\ \lambda x_{i-1}.I^{n-i+1}$ qui elle-même se compile en $(K^{i-1}\ I^{n-i+1})$. Par exemple $\lambda x_1.\lambda x_2.\ ...\lambda x_{10}.x_4$ se transforme en $(K^3\ I^7)$. Par ailleurs on peut remarquer que $K^n = I^{n+1} = \lambda x_0.\lambda x_1.\ ...\lambda x_n.x_0$.

Remarque : lorsque la même variable apparaît plusieurs fois sous un lambda dans une projection, c'est la plus interne qui est prise en compte.

VI.2.6 Témoin

Le terme combinatorisé pour la fonction de Fibonacci avec des combinateurs n-aires est donné ci-dessous (33 applications contre 108) :

```
fib :=
((((S1_3 (K1 IF)) (((S1_2 (K1 -)) I1) (K1 1))) (((S2_2 (K2 +))
(((S2_2 (K2 !fib)) (K2 0)) (((S2_2 (K2 -)) I2) (K2 1))))
(((S2_2 (K2 !fib)) (K2 0)) (((S2_2 (K2 -)) I2) (K2 2))))) (K2 1))
```

Nous donnons également dans la figure 1 une représentation sous forme d'arbre de cette expression. Cette représentation permet de retrouver la structure du terme en syntaxe abstraite.

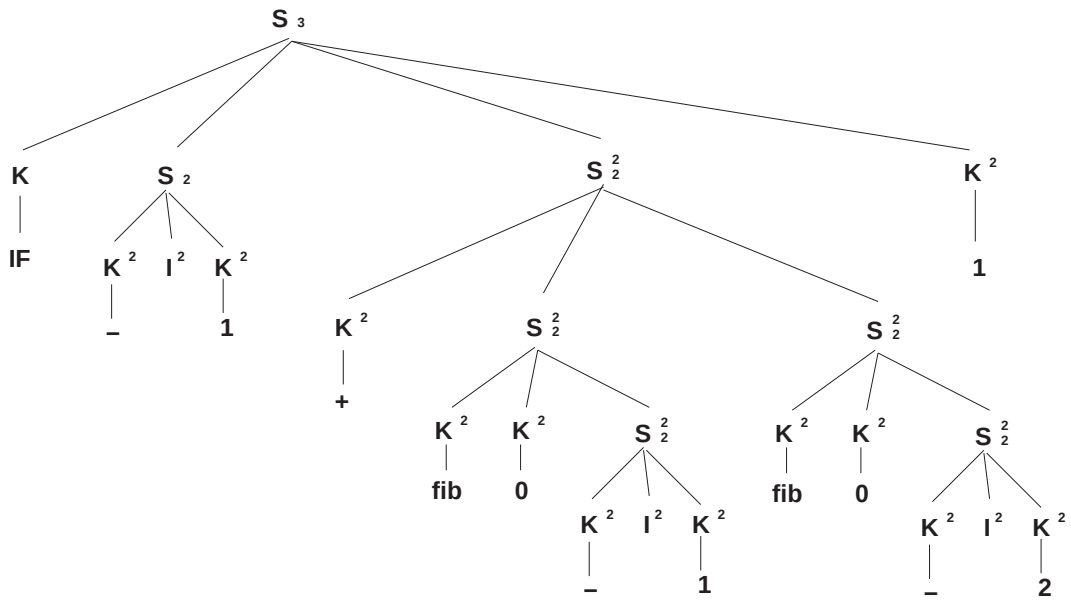


FIG. 1 – Représentation sous forme arborescente de l'expression de `fib`.

VI.2.7 Implémentation des combinateurs n-aires

Implémentation des K^n

La façon la plus simple d'implémenter K^n est la suivante :

```
let comb_K_n i = fun c -> make_k_aux c i

and make_k_aux c n =
     if n=0 then c
     else let res = make_k_aux c (n-1)
          in ValFun (fun _ -> res)
```

car $K^n = \lambda c.\lambda x_1. \ldots \lambda x_n.c$.

Cette définition récursive a l'inconvénient suivant : l'appel à `make_k_aux` n'est effectivement effectué que lorsque le combinateur est appliqué à son premier argument, c'est à dire lors de l'évaluation et non pas lors de la combinatorisation. Ceci n'est pas trop gênant car :

- le surcoût est faible ;
- on peut prédéfinir autant de K^n sans appels récursifs que l'on veut sans surcoût ;

– comme les K^n ne sont jamais séparés de leur premier argument par le processus de traduction on peut implémenter un combinateur K_c^n pour $(K^n\ c)$ (voir la sous-section *Arguments embarqués dans le combinateur* dans cette section). On est sûr que les K^n ne sont jamais séparés de leur argument car, la SK-traduction ayant lieu de la racine du terme vers ses feuilles, on est sûr de ne pas modifier un terme déjà traduit. Ceci n'est pas vrai avec la stratégie de transformation des abstractions les plus internes.

Implémentation des I^n

Nous avons vu que $I^n = K^{n-1}$. On peut donc implémenter les I^n comme on implémente les K^n.

Implémentation des S_m^n

Nous avons vu que les combinateurs S_m^n s'exprimaient en fonction de S_m^{n-1}. Par exemple $S^3 = \lambda f.\lambda g.\lambda a_1.(S^2\ (f\ a_1)\ (g\ a_1))$. On peut également exprimer les S_m en fonction de S_{m-1}. Par exemple $S_3 = \lambda f.\lambda g_1.\lambda g_2.\lambda g_3.\lambda a.((S_2\ f\ g_1\ g_2\ a)\ (g_3\ a))$.

Toutefois, la façon la plus simple d'implémenter les S_m^n est sans doute de se fonder sur la définition directe de S_m^n :

$$S_m^n = \lambda f.\lambda g_1.\lambda g_2.\ ...\lambda g_m.\lambda a_1.\lambda a_2.\ ...\lambda a_n.\big((f\ a_1\ ...\ a_n)\ (g_1\ a_1\ ...\ a_n)\ ...\ (g_m\ a_1\ ...\ a_n)\big)$$

Deux tâches sont à accomplir pour produire l'implémentation de cette fonction :
– Produire la fonction $\lambda f.\lambda g_1.\lambda g_2.\ ...\lambda g_m.\lambda a_1.\lambda a_2.\ ...\lambda a_n.\langle l_{fun}, l_{arg}\rangle$ où $l_{fun} = [f; g_1; g_2; ...; g_m]$ et $l_{arg} = [a_1; a_2; ...; a_n]$.
– Produire la fonction qui à partir de l_{fun} et l_{arg} produit l'application $(f\ a_1\ ...\ a_n)\ (g_1\ a_1\ ...\ a_n)\ ...\ (g_m\ a_1\ ...\ a_n)$.

Ensuite il n'y a plus qu'à les composer.

La fonction `construit_funs` donnée dans la figure 2 crée les $m+1$ abstractions liées à f et aux m fonctions g et les accumule dans `fun_list`. La fonction `construit_args` fait de même avec les n arguments stockés dans `arg_list`. La fonction `app_app` donnée dans la figure 3 crée

```
let rec make_S (n,m) = construit_funs [] n (m+1)

and construit_funs fun_list nb_args nb_fun = match nb_fun with
 | 0 -> construit_args (reverse fun_list) [] nb_args
 | i -> ValFun (fun f -> construit_funs (f::fun_list) nb_args (i-1))

and construit_args fun_list arg_list nb_args = match nb_args with
 | 0 -> app_app fun_list (reverse arg_list)
 | i -> ValFun ( fun a -> construit_args fun_list (a::arg_list) (i-1)
```

FIG. 2 – Construction des combinateurs S^n.

l'application voulue à partir des deux listes.

```
let app = function (ValFun f) -> function v -> f v

(* app_n f [a1;a2;...;an] = (f a1 a2 ...  an) *)
let app_n f l = List.fold_left app f l ;;

(* app_app [f1;f2] [a1;a2] = (f1 a1 a2) (f2 a1 a2) *)
let app_app fun_list arg_list = match fun_list with
 | [f] -> app_n f arg_list
 | f::r ->
    List.fold_left
     (fun f1 -> fun f2 -> app f1 (app_n f2 arg_list))
     (app_n f arg_list)
     r ;;
```

FIG. 3 – Fonctions auxiliaires pour la construction des S^n.

Arguments embarqués dans le combinateur

On peut remarquer que dans le code produit, S_n est toujours appliqué à $n+1$ arguments par construction (voir l'arbre sur la figure 1 par exemple). Ceci est vrai parce qu'on combinatorise de l'extérieur vers l'intérieur (stratégie *outer-most*), sinon l'application aurait pu être modifiée par une autre transformation. Dans ce cas on peut court-circuiter la procédure d'application standard et en faire une spécialement pour S_n. Enrichissons pour cela la syntaxe d'ordre supérieur que nous manipulons :

```
type HOterm =
 | ...
 | CombSn of int * (HOterm list)
```

Ainsi, l'expression $(((S_2^2\ (K^2\ -))\ I^2)\ (K^2\ 1))$ sera implémentée par

`CombSn (2, [HOApp(`K^2`, HOFun moins)` ; I^2 ; `HOApp(`K^2`, HOInt 1)])`

L'entier dans le constructeur `CombSn` correspond au nombre d'abstractions combinatorisées (le n dans S_m^n). Le nombre d'arguments de l'application combinatorisée (m dans S_m^n) n'est pas nécessaire puisque la liste a exactement la longueur nécessaire par construction.

Nous ne détaillerons pas plus les points pratiques de cette implémentation. Ajoutons simplement que :

- Ce style d'implémentation est mis en œuvre dans l'interprète MGS actuel et ses performances sont meilleures que l'approche proposée dans cette section pour l'implantation des S_m^n (voir section VI.4.1).
- Certes le type `HOterm` est modifié et la fonction `evalHO` est complexifiée mais il s'agit toujours d'ordre supérieur puisque c'est toujours le langage hôte qui gère la réduction des applications.
- On peut contrôler la politique d'évaluation des termes stockés dans une construction `CombSn(n,l)`. Par exemple, on peut ne les évaluer que lorsque ce « combinateur » est appliqué (à un des arguments de l'abstraction combinatorisée).

VI.3 Optimisation non classique : implémentation sans glaçons

L'utilisation des glaçons a permis de rentrer dans le cadre de la SK-traduction du λ-calcul. À présent on regarde comment se passer de glaçons en sortant un peu du cadre habituel des combinateurs dans le but d'être plus efficace.

Depuis qu'on fait rentrer les lambdas en les poussant de l'extérieur, on est sûr que pour une expression `IF(e1,e2,e3)` dans une abstraction on va rencontrer dans le code compilé une forme $(S^n_m\ (K^n\ \mathit{if})\ e'_1\ e'_2\ e_3\ ...)$. Ce qui est remarquable c'est qu'on va rencontrer un combinateur S suivi d'un $(K\ \mathit{if})$ suivi des transformations d'au moins e_1, e_2 et e_3. Avec la stratégie *inner-most* pour la combinatorisation, le S, le *if* et ses arguments auraient pu être séparés.

Ici nous allons introduire un nouveau combinateur S_{IF}. Plusieurs points motivent l'introduction de ce nouveau combinateur :

- L'idée de remplacer $(S^n_m\ (K^n\ \mathit{if})...)$ par un seul combinateur n'est pas assez intéressante car on peut déjà penser qu'avec une bonne gestion des directions le K va disparaître.
- En revanche on peut espérer ne pas faire les n applications sur la branche du *if* qui ne sera jamais évaluée. Il faut pour cela stocker les arguments et, lorsqu'ils sont tous arrivés, évaluer la condition (on a pu faire les applications partielles dessus au fur et à mesure) puis ensuite appliquer la bonne branche aux arguments (section VI.3.2).
- Une fois qu'on a ce combinateur « paresseux » on peut regarder comment se passer de l'abstraction servant de glaçon (section VI.3.3). En revanche pour les *if* n'étant pas sous une abstraction il faut trouver autre chose.

VI.3.1 Combinateur dédié au *if*

Extraction des $(S_3\ (K\ \mathit{if}))$

Le terme $\lambda x.(\mathit{if}\ e_1\ e_2\ e_3\ a_4\ ...\ a_n)$ peut se combinatoriser en

$$S_{n-3}\ (S_3\ (K\ \mathit{if})\ \lambda x.e_1\ \lambda x.e_2\ \lambda x.e_3)\ \lambda x.a_4\ ...\ \lambda x.a_n)$$

par la chaîne suivante :

$$\begin{array}{lcll} \lambda x.(\mathit{if}\ e_1\ e_2\ e_3\ a_4\ ...\ a_n) & = & \lambda x.(E\ a_4\ ...\ a_n) & \text{avec } E = \mathit{if}\ e_1\ e_2\ e_3\ a_4 \\ & \Rightarrow & S_{n-3}\ \lambda x.E\ \lambda x.a_4\ ...\ \lambda x.a_n & \\ & \Rightarrow & S_{n-3}\ (S_3\ (K\ \mathit{if})\ \lambda x.e_1\ \lambda x.e_2\ \lambda x.e_3)\ \lambda x.a_4\ ...\ \lambda x.a_n) & \end{array}$$

Cet exemple montre que dès qu'on a un *if* sous un lambda, on peut extraire une forme $(S_3\ (K\ \mathit{if}))$ sans voir exploser le reste du terme après combinatorisation. (On peut toujours séparer un S_n en S_i et S_{n-i} sans ajouter plus d'un combinateur, et de même pour S^n.)

Combinateur S_{IF}

Le terme $\lambda x.(\mathit{if}\ e_1\ e_2\ e_3)$ se combinatorise en $(S_3\ (K\ \mathit{if})\ \lambda x.e_1\ \lambda x.e_2\ \lambda x.e_3)$. Si on utilise une variante de S_3 qui ne distribue pas le lambda sur son premier argument (il s'agit de nyyy) on peut toujours obtenir (nyyy $\mathit{if}\ \lambda x.e_1\ \lambda x.e_2\ \lambda x.e_3$).

Posons un nouveau combinateur S_{IF} qui est égal à (nyyy if) et tel que :

$$\lambda x.(\mathit{if}\ e_1\ e_2\ e_3) \Rightarrow S_{IF}\ \lambda x.e_1\ \lambda x.e_2\ \lambda x.e_3$$

$$S_{IF}\ e_1\ e_2\ e_3\ a \rightarrow \mathit{if}\ (e_1\ a)\ (e_2\ a)\ (e_3\ a)$$

Ce nouveau combinateur apporte une faible amélioration en lui-même. La faible réduction de la taille du terme compilé ne justifie pas forcément l'introduction d'un combinateur supplémentaire dans le langage cible. Toutefois cette étape nous semble nécessaire pour expliquer les optimisations que nous présentons dans le reste de cette section.

Le combinateur S_{IF} se généralise aux combinateurs S_{IF}^n pour n abstractions d'un *if*.

VI.3.2 Optimisation du S_{IF}

La règle de réduction du S_{IF} donnée ci-dessus met en évidence une optimisation possible. En effet on voit que S_{IF} distribue l'argument reçu à tous les arguments di *if*. Or la distribution sur l'une des deux est nécessairement *inutile*. Ceci est d'autant plus vrai qu'on utilise S_{IF}^n. Nous proposons donc de retarder l'application des arguments jusqu'au moment où l'on saura quelle branche dégeler afin de ne pas réduire d'applications inutiles. Ceci se réalise en implémentant S_{IF} de la manière suivante :

```
let comb_S_if =
    fun e1 -> ValFun (fun e2 -> ValFun (fun e3 -> ValFun (fun a ->
      match app e1 a with
      | ValInt i when i>0 -> app (app e2 a) dummy_val
      | _ -> app (app e3 a) dummy_val )))
```

Ainsi on est sûr de réduire le minimum d'applications[6]. La règle de combinatorisation introduisant S_{IF} est identique à celle donnée ci-dessus.

Généralisation à S_{IF}^n

Ceci peut se généraliser à S_{IF}^n en stockant les arguments reçus et en les passant à la bonne branche une fois que tous ont été reçus.

Notons que si S_{IF} ou même S est utilisé alors qu'on avait l'opportunité d'utiliser un S_{IF}^n alors le résultat est toujours correct mais moins bien optimisé.

VI.3.3 Abandon des glaçons sur les branches du *IF*

Les méthodes que nous avions proposées jusqu'à présent ne nécessitaient qu'un type `HOterm` très simple où les fonctions, les fonctions non-strictes et les traits impératifs se sont intégrés. Dans la section présente, nous montrons que des optimisations sont possibles si on accepte de complexifier la syntaxe d'ordre supérieur et l'évaluation associée.

Les abstractions au dessus d'un *if* jouent le rôle de glaçons naturels. Puisque nous nous sommes donné les moyens de ne passer les arguments dégelant ces lambdas qu'à la branche concernée par le dégel, nous pouvons nous passer totalement de glaçons. Pour ce faire, nous n'introduirons plus de glaçons lors de la curryfication :

```
let rec traduit = function
 | ...
```

[6] Les applications économisées n'étaient que des applications partielles et l'économie se réduit donc à la création de clôtures par le langage hôte.

```
 | If (e1,e2,e3) ->
    CApp ( CApp ( CApp (CExtFun ite, traduit e1), traduit e2) traduit e3)
```

et nous utiliserons l'implémentation suivante du combinateur S_{IF} :

```
let comb_S_if =
    fun e1 -> ValFun (fun e2 -> ValFun (fun e3 -> ValFun (fun a ->
      match app e1 a with
      | ValInt i when i>0 -> app e2 a
      | _ -> app e3 a ))
```

La règle de combinatorisation introduisant les S_{IF} est celle qui est donnée ci-dessus (page 116).

Cette optimisation a l'avantage de ne pas complexifier artificiellement la traduction des branches d'un *if* et également de nécessiter une réduction de moins dans chaque utilisation d'un *if*. Cette approche se généralise également aux S^n_{IF}.

Remarque : Cette méthode d'abandon des glaçons n'est correcte que si :
- On n'utilise jamais un S ou un S^n là où un S_{IF} ou un S^n_{IF} aurait pu être utilisé.
- Les abstractions sont distribuées sur toute expression contenant une application. Ceci signifie que les restrictions sur l'η-réduction et l'utilisation de B et C sont toujours nécessaires.

Pour les *if* ne se trouvant pas protégés par une abstraction dans le programme source, on peut soit garder la technique des glaçons, soit faire en sorte que le *if* ne soit pas traité par le mécanisme de réduction d'application standard mais par un mécanisme d'application non-stricte. Par exemple on peut ajouter une construction dans le type `HOterm` :

```
type HOterm =
 | ...
 | HOAppIF of HOterm * HOterm * HOterm
```

On utilisera donc `HOAppIF(e1,e2,e3)` au lieu de `HOApp(HOApp(HOApp(HOFun ite, e1), e'2), e'3)` où `e'2` et `e'3` sont les versions gelées de `e2` et `e3`. L'évaluation se fera comme suit :

```
let rec evalHO = function
 | ...
 | AppIF(e1,e2,e3) ->
    (match eval e1 with
     | ValInt i when i>0 -> evalHO e2
     | _ -> evalHO e3 )
```

On aurait pu utiliser ce type de mécanisme depuis le début mais :
- On n'aurait pu le faire que pour les *if* n'étant pas sous un lambda car la SK-traduction de $\lambda x.if(e_1, e_2, e_3)$ n'est pas définie.
- L'utilisation de glaçons nous a permis de rentrer dans le cadre de la SK-traduction et de l'associer aux traits impératifs. Ceci nous a permis de raisonner tout au long des optimisations introduites et d'élaborer des règles de traduction complexes mais intuitives.
- Enfin, pour chaque trait non-strict ou impératif, il nous aurait fallu enrichir le type `HOterm` et prévoir un cas supplémentaire dans la fonction `evalHO`.

Regardons à présent comment on peut se passer des glaçons dans la combinatorisation des traits impératifs du langage.

VI.3.4 Abandon des glaçons dans les traits impératifs

L'affectation mémoire ne nécessitait déjà pas de glaçon. En revanche, pour l'accès mémoire, plusieurs cas sont à examiner si on veut se passer de l'application qui déclenche l'accès mémoire. Supposons que `Get(m)` dans le type `term` soit représenté dans la syntaxe curryfiée par une « constante » `CGet(m)`. Analysons les trois cas pouvant se présenter :

Accès mémoire sous un lambda. On peut utiliser un K *gelant* comme celui déjà évoqué en section VI.2.7, c'est-à-dire un K qui embarque son premier argument pour l'évaluer lorsque son deuxième argument est arrivé.

Dans ce cas on peut ne pas transformer un accès mémoire en une application. On pourra combinatoriser `Abs(x,CGet(m))` en K_{get-m} qui sera implémenté par `(fun _ -> assoc a !store)`.

Accès mémoire sous un *if*. Nous avons vu qu'on pouvait implémenter un *if* qui déclenche littéralement l'évaluation de la branche nécessaire. Donc si `CGet(m)` se trouve dans une branche *then* ou *else*, le code `(assoc a !store)` ne sera appelé que si nécessaire.

Accès mémoire sous rien. Si `CGet(m)` ne se trouve pas sous un lambda ni dans un *if*, l'accès à la mémoire peut se faire immédiatement. Le code `(assoc a !store)` convient donc.

VI.4 Conclusion

VI.4.1 Résultats

Le tableau de la figure ci-dessous donne des mesures de performances sur les différentes implémentations mentionnées dans ce chapitre et le précédent.

		`evalD`	SK pur	opt BCN	opt n-aires	interprète 1-MGS
fib 31	(p. 100)	4.7 - 5.0	9.3 - 9.8	5.4	6.3	0.65
ack (3,8)		6.5	> 120	88	20.5	1.84
permut 25		2.14	67.0	12.4	3.32	2.28
turing-torus 100	(p. 50)	6.08	> 120	39.7	7.45	5.11
bubble-sort	(p. 43)	11.9	10.4	9.5	28	12.65

FIG. 4 – Mesures des performances des différentes approches proposées.

La colonne `evalD` correspond à l'évaluateur « classique » présenté en section V.2. La colonne *SK pur* correspond à la SK-traduction pure proposée en sections V.4, V.5 et V.6. La colonne *BCN* correspond à l'implémentation avec distribution sélective des lambdas et avec les combinateurs $S_{\swarrow}$, $S_{\searrow}$ et $S_{\varnothing}$ (section VI.1). La colonne *n-aire* correspond à l'utilisation des combinateurs n-aires (section VI.2). Enfin la colonne *interprète* 1-MGS correspond à l'implémentation actuelle de l'interprète MGS qui utilise les trois formes d'optimisations proposées. L'implémentation de l'abandon des glaçons (section VI.3) n'a pas été réalisée indépendamment des autres optimisations et est intégrée à 1-MGS.

La fonction `permut` (troisième ligne du tableau) est définie par :

```
fun permut(a, b, c, d) = if a <= 0 then 1 else g(b, c, d, a) + g(b, c, d, a-1) fi;;
fun g(a, b, c, d) = h(b, c, d, a);;      fun h(a, b, c, d) = i(b, c, d, a);;
fun i(a, b, c, d) = j(b, c, d, a);;      fun j(a, b, c, d) = permut(a-1, b, c, d);;
```

Ces mesures permettent plusieurs obervations :
- L'interprète 1-MGS n'est jamais beaucoup plus lent que l'interprète classique alors qu'il est parfois bien plus rapide (7 fois plus rapide dans le premier test). L'amélioration apportée est donc significative.
- Nos optimisations ne permettent pas toujours un gain en performances. Par exemple, sur le cinquième test, les combinateurs n-aires sont moins performants que les combinateurs S, K et I, alors que sur le troisième test, ils permettent une accélération d'un facteur 20.

Ces premiers résultats sont donc satisfaisants et valident notre approche mais on peut remarquer qu'il est difficile de prévoir le comportement des optimisations.

Enfin, notons que nous avons rempli notre objectif d'être comparable à d'autres langages interprétés. En effet, pour `fib 31`, notre interprète est seulement 2.8 fois plus lent que le langage hôte (0.23 secondes pour l'interprète OCaml) et 46 fois plus rapide que l'interprète Mathematica (30 secondes) et 5 fois plus rapide que l'interprète Python (3.8 secondes).

VI.4.2 Bilan

Dans ce chapitre nous avons étudié trois voies d'optimisations du schéma d'interprétation présenté au chapitre précédent :
- Nous avons adapté les optimisations classiques de la SK-traduction.
- Nous avons étudié une généralisation des combinateurs à des fonctions n-aires.
- Nous avons montré que l'on pouvait se passer de glaçons.

Ces trois axes d'optimisations permettent d'avoir du code plus compact et donc plus efficace. Ces optimisations ont été étudiées indépendamment les unes des autres mais on peut les combiner. C'est ce qui est fait dans l'interprète MGS actuel qui se révèle efficace (voir tableau ci-dessus).

Troisième partie

Typage de MGS

Chapitre VII

Typage ensembliste des collections topologiques et des transformations

Les travaux présentés dans ce chapitre ont été publiés dans [Coh03c], [Coh04] et [Coh03a].

Nous présentons dans ce chapitre le typage des collections topologiques et des transformations, caractéristiques fondamentales du langage MGS. La mise en œuvre du système de types que nous proposons nous a menés à :

- créer des types pour dénoter les collections topologiques ;
- donner une règle de typage pour les transformations ;
- prendre en compte le caractère hétérogène des collections dans MGS ;
- mettre en œuvre un algorithme d'inférence automatique.

Le système de types ainsi construit permettra par exemple de donner le type $(\alpha \to \beta) \to [\alpha]\theta \to [\beta]\theta$ à la fonction *map* définie en MGS par `trans map [f] = { x => f(x) }`.

Le système que nous proposons a été implémenté dans un compilateur expérimental pour le langage MGS. Le typage statique est l'un des points essentiels qui ont permis à ce compilateur d'être bien plus performant que les différentes implémentations de l'interprète MGS.

Le chapitre est organisé comme suit. Dans la première section nous exposons l'utilité de développer un système de types spécifique à MGS et les caractéristiques qu'il doit avoir. Nous nous limitons ici à un sous-ensemble significatif de MGS contenant les collections topologiques hétérogènes et les transformations. Ce langage est décrit en section VII.2 et est muni d'une sémantique dénotationnelle. Dans la section VII.3 nous décrivons l'algèbre de types utilisée et la sémantique donnée aux types. La section VII.4 donne les règles de typage définissant la notion de programme bien typé. Nous y établissons également les propriétés des programmes bien typés. Nous étudions ensuite un algorithme de résolution de contraintes de sous-typage (section VII.5). Celui-ci est nécessaire à l'implémentation d'un algorithme de vérification de typage ou d'inférence automatique. Nous exposons l'inférence automatique de types pour notre langage en section VII.6. Pour conclure ce chapitre, nous présentons des extensions naturelles à notre système (section VII.7), nous discutons certains choix qu'il nous a fallu effectuer (section VII.8), et enfin, nous évoquons les travaux pouvant être apparentés aux nôtres et nous dressons des perspectives (sections VII.9 et VII.10).

VII.1 Objectifs

Typage statique

Lorsque l'on veut créer un nouveau langage de programmation, on est amené à proposer une implémentation de celui-ci. Faire un interprète dynamiquement typé est une solution assez rapide qui permet d'expérimenter avec le langage. Dynamiquement typé signifie que le programme exécuté contient des annotations de types qu'il faut utiliser constamment pour réaliser des opérations. C'est ce que fait l'interprète présenté au chapitre V. Par exemple lors de l'addition de deux entiers, on vérifie que les valeurs des arguments sont bien des entiers, on calcule l'entier issu de l'addition et on note que cette valeur est un entier. On a effectué trois opérations de lecture/écriture de type pour une opération « utile » d'addition. Une telle implémentation est donc inefficace.

Le but du typage statique est de déterminer au plus tôt (à la compilation par exemple) le type de toutes les valeurs et opérations effectuées pour ne pas avoir à s'en préoccuper à l'exécution.

Inférence automatique

Plusieurs moyens existent pour parvenir à un typage statique. La première méthode à avoir fait ses preuves est l'annotation de programmes. Le programmeur indique le type de certaines expressions (par exemple les variables et les fonctions en C) et par propagation de ces informations, le type de chaque expression est déduit.

L'inférence automatique des types sans annotations est de plus haut niveau puisqu'elle permet au programmeur de ne pas annoter son programme avec des types. Elle est apparue avec les langages fonctionnels et a été largement étudiée dans des langages de la famille ML. Les types sont déduits automatiquement en tenant compte du type prédéfini des constantes du langage et des constructions syntaxiques comme la définition des fonctions.

Collections hétérogènes

Notre objectif est de construire un système de types pour le langage MGS avec inférence automatique sans annotations. Le système que nous proposons ne permet pas d'éliminer la totalité des informations de types traitées à l'exécution. Il reste donc une partie de typage dynamique. Ceci est dû au caractère hétérogène des collections topologiques. Considérons par exemple le programme `(if P then 1 else true fi) :: set:()` et le programme `1::true::set:()`. Dans le premier le système de types ne peut déterminer si l'ensemble contiendra un entier *ou* un booléen. Dans le second, l'ensemble contiendra des entiers *et* des booléens. Dans les deux cas, les valeurs contenues dans l'ensemble seront étiquetées de leur type. Notons que dans le programme `hd(1::seq:())+1` aucun étiquetage n'est nécessaire puisque toutes les informations de typage peuvent être déduites de la lecture du programme et du type prédéfini des constantes `1`, `+`, `::`, `hd` et `set:()`.

Nous montrons dans le chapitre VIII que toutes les étiquettes de types peuvent être enlevées à condition de se restreindre aux collections homogènes.

Collections, transformations et polytypisme

Enfin le système de types proposé doit permettre de typer les collections et les transformations tout en exprimant le polytypisme potentiel : les transformations sont des *fonctions* définies par cas pouvant s'appliquer selon les cas :
- à toute collection topologique (polytypisme) ;
- à une collection d'une topologie donnée ;
- à une collection dont les éléments ont un type particulier ;
- à une collection ayant une topologie donnée et contenant des éléments d'un type donné.

Les *types collections* que nous proposons permettent d'exprimer tous ces cas.

VII.2 Langage considéré

Dans le chapitre présent nous nous concentrons donc sur le typage des collections topologiques hétérogènes et des transformations qui n'ont pas été étudiées auparavant. MGS intègre des caractéristiques ayant émergé récemment mais qui ont été déjà étudiées et que nous ne traiterons pas ici. Il s'agit notamment de la surcharge (voir par exemple [Fur02]), des enregistrements extensibles [Rém93b, Oho95], des traits impératifs [Wri95, Gar04] ou des arguments optionnels [FG95]. Nous écartons aussi des notions plus anciennes comme les types algébriques et les produits pour privilégier la simplicité de la présentation. Enfin nous simplifions légèrement la syntaxe des transformations et nous montrons en fin de ce chapitre qu'on peut adapter le système présenté aux transformations de MGS.

VII.2.1 Syntaxe

Le langage est restreint au λ-calcul avec constantes et *let* auquel on ajoute les transformations.

$$e \quad ::= \quad x \mid c \mid \lambda x.e \mid e\ e \mid \text{let } x = e \text{ in } e \mid \{m/e \Rightarrow e; \ldots; m/e \Rightarrow e\}$$

$$m \quad ::= \quad \mu, \ldots, \mu$$

$$\mu \quad ::= \quad x \mid x{:}b \mid x{:}r \mid * \text{ as } x \mid b* \text{ as } x \mid r* \text{ as } x$$

Dans la grammaire ci-dessus, e est une expression, c est une constante (par exemple `set:()`, `+` ou `::`) et m est un motif.

Motifs

Dans les motifs, un *motif élémentaire* μ est :
- soit une variable, filtrant une valeur quelconque (x), une valeur d'un type atomique donné ($x{:}b$) ou une valeur qui est une collection et dont la topologie est donnée ($x{:}r$),
- soit une *répétition* nommée, filtrant un nombre quelconque de valeurs voisines successivement et qui sont :
  - quelconques ($*$ as x),
  - d'un type atomique donné ($b*$ as x)
  - ou d'un type collection de topologie donnée ($r*$ as x).

On note B l'ensemble des types atomiques dans lequel on peut choisir b, on prendra par exemple $B = \{int, bool, float, string\}$. On note R l'ensemble des topologies de base dans lequel on peut choisir r, on prendra ici $R = \{set, seq, bag, grid\}$. Remarquons que les topologies de base ne sont que des symboles.

Les gardes d'un motif sont réduites à une seule expression en fin de motif. Les conditions de type sont restreintes aux types atomiques et aux topologies de base mais on peut encoder d'autres tests (voir ci-dessous). La discussion en section VII.8.1 justifie le choix de se restreindre à ces tests.

L'identificateur x dans $*$ as x dénote la *séquence* des valeurs filtrées. Une séquence est en effet suffisante pour dénoter les éléments filtrés en partie droite d'une règle ou dans la garde d'un motif tout en garantissant que la transformation puisse être appliquée sur n'importe quelle topologie (un motif filtre un *chemin*, voir la section IV.3 dans le chapitre *Filtrage*). Les règles de transformation seront notées :

$$m/g \Rightarrow e$$

où m est la partie du motif non gardée, g est la garde du motif et e est l'expression remplaçante.

Simplifications

Les simplifications de la syntaxe des transformations par rapport à la syntaxe MGS concernent les points suivants :

Directions. Les directions dans les motifs peuvent être vues comme du sucre syntaxique au niveau du typage (au niveau du filtrage elles sont importantes). Exemple :

```
{ x |nord> y => y,x }
```

peut être vu comme un raccourci pour :

```
fun (c) -> ( { x,y / nord? (x,y,c) => [y,x] } c)
```

où `nord?` est un prédicat de voisinage.

Tests de type. Un test de type de la forme $x : [int]seq$ peut s'encoder de la manière suivante dans notre langage :

```
let t = { x:int => true ; x => false } in
let all_int =
   fun c -> fold (fun (x,y) -> x && y) (t x) true in
{ x:seq / all_int x => ...  }
```

ou plus simplement, si on dispose d'un prédicat testant si une valeur est d'un type atomique particulier[1] :

```
let all_int =
   fun c -> fold (fun (x,y) -> is_int(x) && y) (t x) true in
{ x:seq / all_int x => ...  }
```

[1] Ce prédicat s'encode de la manière suivante dans notre langage :

```
let is_int = fun x ->
 let t = {x:int =>[true] ; x => [false]}
 in hd ( t (x::seq:()) )
```

Dans MGS on peut tester le type d'une valeur en dehors d'un motif comme dans l'exemple suivant : `if x:int then e1 else e2`.

Gardes. Les gardes sont rejetées à la fin du motif pour la simplicité des règles de typage. Nous donnons en fin de chapitre une version de la règle de typage des transformations qui prend en compte des gardes sur les sous-motifs.

Répétitions. La répétition arbitraire $*$ ne porte pas sur un sous-motif. Le motif $*$ as X dans notre langage restreint correspond au motif `x* as X` de MGS. On ne traite donc pas de motifs comme `(x/x=0,y/y=1)* as Z`. Un tel motif peut toutefois être encodé à l'aide de tests dans la garde.

Par ailleurs, dans les exemples que nous donnerons nous adopterons le sucre syntaxique suivant :
- les opérateurs binaires seront écrits en position infixe,
- on pourra écrire une séquence en énumérant ses éléments entre crochets au lieu d'utiliser le constructeur standard comme le montre l'exemple suivant : $[1, 2, 3]$ pour $1 :: 2 :: 3 :: \mathit{empty\_seq}$,
- on pourra omettre la garde d'un motif lorsque celle-ci est la constante *true*.

On appelle *opérateurs* les fonctions constantes données dans le langage. Parmi les constantes on trouvera les collections vides comme *empty_set* ou *empty_seq*, et les opérateurs manipulant des collections comme le constructeur[2] générique :: ou les constructeurs spécifiques aux grilles comme *nord* ou *est*.

Une règle de la forme $x \Rightarrow e$ est appelée une règle *attrape-tout* (le motif est non gardé et réduit à une variable).

Discussion 3 (Formes spéciales.) Le prédicat `nord?` dont nous parlons ci-dessus n'a de sens que sur des variables introduites par le motif. En effet, appliquer `nord?` à 1 et 2 par exemple n'a pas de sens car 1 et 2 peuvent apparaître plusieurs fois dans la collection.

Le langage MGS propose d'autres opérations qui n'ont de sens que sur des variables introduites par un motif. Vérifier que ces opérateurs ne sont utilisés qu'avec de telles variables est un problème simple qui peut éventuellement relever du typage mais que nous ne traiterons pas. Pour illustrer cette tâche, considérons la transformation `{ x => left(x) + x }`. Ici x dénote à la fois une position et sa valeur. Ce problème apparaît souvent dans les langages de programmation et n'est pas caractéristique de l'utilisation de collections ou de transformations [Str67, Str00].

VII.2.2 Topologies

Dans la section IV.5.2 une topologie est vue comme la donnée d'un ensemble de constructeurs et d'une substitution topologique.

Définition 2 L'*ensemble des directions* d'une topologie r est l'ensemble de toutes les directions pouvant apparaître dans une collection construite uniquement à partir des constructeurs de r et de sa substitution topologique.

Dans ce chapitre nous supposerons que toutes les topologies considérées ont des ensembles de directions disjoints deux à deux.

VII.2.3 Sémantique dénotationnelle

Nous donnons ici une sémantique au langage étudié afin de montrer par la suite la correction du typage que nous proposons.

[2] Ici le mot *constructeur* n'est pas utilisé au sens algébrique mais désigne une fonction ajoutant un élément à une collection.

Le langage s'évalue dans un domaine $\mathbf{D}$ que nous décrivons ici (voir [Mos90] pour une introduction aux domaines sémantiques). Pour chaque type atomique b de B il est donné un ensemble $\mathbf{B}_b$ de valeurs dites littérales (dont l'intersection deux à deux est vide). Pour chaque topologie r de R il est donné un ensemble $\mathbf{C}(\mathbf{D}, r)$ de collections à valeurs dans $\mathbf{D}$. On peut prendre simplement $\mathbf{C}(\mathbf{D}, r) = Pos_r \to \mathbf{D}$ où Pos_r est l'ensemble des positions possibles pour une collection de topologie r. Pour $r_1 \neq r_2$, l'intersection de $\mathbf{C}(\mathbf{D}, r_1)$ et de $\mathbf{C}(\mathbf{D}, r_2)$ est vide[3]. Le domaine $\mathbf{D} \to \mathbf{D}$ contient les fonctions continues totales de $\mathbf{D}$ dans $\mathbf{D}$. Les transformations sont représentées dans $\mathbf{D}$ par des fonctions strictes de $\mathbf{D} \to \mathbf{D}$. Les transformations sont continues si l'on suppose qu'elles sont déterministes. Elles le sont si l'on fixe la stratégie d'application des règles. Leur sémantique correspond à la description donnée en section IV.5.3, page 79. Notons que cette sémantique impose d'avoir une séquence comme partie droite des règles d'une transformation. En effet, les substitutions topologiques, qui sont les opérateurs substituant les valeurs filtrées par les valeurs remplaçantes, attendent une séquence de valeurs en argument. Ceci est naturel puisqu'un motif décrit un chemin, c'est à dire un parcours séquentiel de la partie filtrée. Enfin, $\mathbf{D}$ contient $\perp$, qui dénote un calcul qui ne termine pas, ainsi que les deux valeurs **wrong** et **shape**. La valeur **wrong** correspond à une erreur de type à l'exécution, par exemple lorsqu'un entier est appliqué comme une fonction. La valeur **shape** peut être vue comme une exception levée lorsqu'une transformation est amenée à rompre une topologie (par exemple, lorsque le chemin filtré et la séquence remplaçante n'ont pas la même taille dans une collection newtonienne ; c'est $\Psi_{\mathcal{N}}$ qui provoque la levée de l'exception). La définition formelle de $\mathbf{D}$ est donnée par l'équation ci-dessous :

$$\mathbf{D} = \{\perp, \mathbf{wrong}, \mathbf{shape}\} \cup \bigcup_{b \in B} \mathbf{B}_b \cup \bigcup_{r \in R} \mathbf{C}(\mathbf{D} - \{\perp\}, r) \cup (\mathbf{D} \to \mathbf{D})$$

Nous distinguons **shape** de **wrong** car les erreurs de types « classiques » seront détectées par notre système de types alors que les violations de topologie, d'une nature plus subtiles ne le seront pas (voir la section VII.10.2 à ce sujet). On note $\mathbf{T}$ l'ensemble des valeurs ne faisant pas intervenir **wrong**. Cet ensemble ne se réduit pas à $\mathbf{D} - \{\mathbf{wrong}\}$ et est défini comme suit :

$$\mathbf{T} = \{\perp, \mathbf{shape}\} \cup \bigcup_{b \in B} \mathbf{B}_b \cup \bigcup_{r \in R} \mathbf{C}(\mathbf{T} - \{\perp\}, r) \cup (\mathbf{T} \to \mathbf{T})$$

Un *environnement* est une fonction d'un ensemble d'identifiants vers l'ensemble des valeurs $\mathbf{D}$. On note $Eval(e, E)$ la sémantique de l'expression e dans l'environnement E.

VII.3 Types

Pour dénoter les collections dans notre système de types nous utiliserons un *type collection* de la forme $[\tau]\rho$ où τ est le type des éléments contenus dans la collection aussi appelé le *type contenu* de la collection et ρ symbolise sa *topologie*. Pour simplifier nous dirons que ρ est la topologie du type collection $[\tau]\rho$ et qu'une collection de ce type a ρ comme topologie. Toutefois dans ce chapitre, une topologie est un symbole. Ce symbole peut être soit une topologie de base, soit une variable de topologie que l'on pourra dénoter par la lettre grecque θ. Nous nous restreignons dans ce chapitre à $\{set,\ bag,\ seq,\ grid\}$ comme ensemble de topologies de base.

[3]Pour distinguer un ensemble à une seule position sans arcs d'un multi-ensemble à une position et également sans arc, on peut considérer que l'ensemble de positions utilisées dans les ensembles est disjoint de l'ensemble de positions utilisées dans les multi-ensembles.

Notons qu'une topologie n'est pas un type et qu'une variable de topologie ne peut être utilisée à la place d'une variable de type et *vice versa*.

Par exemple, une grille contenant des entiers a le type $[int]grid$. Le constructeur *nord* et les autres constructeurs spécifiques aux grilles ont le type $\alpha \to [\alpha]grid \to [\alpha]grid$ où α est une variable de type. Le constructeur générique :: a quant à lui le type $\alpha \to [\alpha]\theta \to [\alpha]\theta$ car il peut être utilisé avec toute collection, quelle que soit sa topologie.

On parle de collection *hétérogène* lorsque les valeurs contenues dans une collection peuvent être de types différents. L'ensemble $\{1, true\}$ contient deux valeurs de types respectifs *int* et *bool* et est donc un exemple de collection hétérogène.

Pour rendre compte de l'hétérogénéité des collections, nous utilisons des *types unions*, déjà utilisés par d'autres auteurs [Pie91, AW93, PS94, Dam94, BDd95, MW97, FCB02]. Une valeur du type union $\tau_1 \cup \tau_2$ est soit du type τ_1 soit du type τ_2. Savoir qu'une valeur est du type $\tau_1 \cup \tau_2$ ne permet pas de dire qu'elle est du type τ_1. L'entier 1 du type *int* est aussi du type $int \cup bool$.

L'ensemble $\{1, true\}$ construit par `1::true::set:()` a le type $[int \cup bool]set$. On peut lire ce type de la manière suivante : « collection de topologie *set* qui contient des valeurs de type *int* et des valeurs de type *bool* » mais il est plus juste de le comprendre ainsi : « collection de topologie *set* dont les éléments ont le type $int \cup bool$ ».

Regardons à présent des exemples de types donnés à des transformations.

VII.3.1 Type d'une transformation : exemples

Variables de topologie. La fonction *map* qui applique une fonction à tout élément d'une collection peut s'écrire $\lambda f.\{x \Rightarrow [f\ x]\}$. Cette forme s'écrit en MGS :

```
trans map (f) = { x => f(x) }
```

Ceci implémente bien un *map* car chaque élément e de la collection sera filtré par le motif x et sera remplacé par $f(e)$. Cette fonction peut s'appliquer à toute collection, indépendamment de sa topologie. Son type est $(\alpha \to \beta) \to [\alpha]\theta \to [\beta]\theta$. La variable de topologie dans ce type exprime bien que *map* s'applique à une collection de n'importe quelle topologie. Ceci exprime le polytypisme dans le système de types (voir la section VII.8.3 pour une discussion sur le polytypisme).

Invariance de la topologie par application d'une transformation. L'identité sur les collections peut s'écrire $\{x \Rightarrow [x]\}$ et a le type[4] $[\alpha]\theta \to [\alpha]\theta$. En effet, cette transformation s'applique à toute collection topologique et ne change ni sa topologie, ni son type contenu. De manière générale une transformation ne change pas la topologie de la collection à laquelle elle est appliquée.

Type contenu de la collection en argument. La transformation $\{x \Rightarrow [x+1]\}$ où $+$ est l'addition entière a le type $[int]\theta \to [int]\theta$ car une erreur de type aura lieu si la règle est appliquée à une valeur qui n'est pas une valeur du type *int*. En revanche la transformation $\{x : int \Rightarrow [x+1]\}$ a le type $[\alpha]\theta \to [\alpha]\theta$ car si des éléments non-entiers sont dans la collection, ils ne seront jamais filtrés avec succès et l'expression remplaçante ne sera pas évaluée, ne provoquant pas d'erreur de type.

[4] En toute rigueur il faudrait disposer de *topologies union* au sens de la discussion 4, page 131 pour que ce type convienne.

Type contenu de la collection renvoyée. Considérons à présent la transformation $\{x : int \ / \ (x \ mod \ 2 = 0) \Rightarrow [true]\}$. Celle-ci a le type $[\alpha]\theta \rightarrow [\alpha \cup bool]\theta$ mais ce type manque de précision. Il ne porte pas l'information que des booléens apparaissent dans la collection renvoyée uniquement si la collection en argument contient des entiers. Pour traduire cette information nous utiliserons des types *conditionnels* de la forme $\tau_1?\tau_2$. Le type $\tau_1?\tau_2$ se lit « τ_1 *if* τ_2 » et vaut τ_1 lorsque τ_2 n'est pas égal au type nul noté 0 et il vaudra le type nul 0 sinon. Ainsi on peut donner le type $[\alpha]\theta \rightarrow [\alpha \cup (bool?(\alpha \cap int))]\theta$ à la transformation mentionnée ci-dessus. Ce type signifie que si la collection en argument contient des valeurs de type *int* alors la collection pourra contenir des valeurs de type *bool* car dans ce cas $\alpha \cap int$ vaut *int* et n'est pas nul. En revanche si le type de la collection en argument indique qu'elle ne contient pas de valeurs du type *int* alors le type de la collection renvoyée est le même que le type de la collection en argument. En effet, dans ce cas $\alpha \cap int$ sera nul et donc $bool?(\alpha \cap int)$ sera également nul et $\alpha \cup bool?(\alpha \cap int)$ vaudra α. Cet exemple montre que les types conditionnels permettent d'obtenir une bonne précision pour les types des transformations.

Erreur de structure. La transformation $\{x \Rightarrow [x, x]\}$ a le type $[\alpha]\theta \rightarrow [\alpha]\theta$. Elle est bien typée car elle ne provoquera pas d'erreur de type **wrong**. En revanche si elle est appliquée à une collection newtonienne (voir section IV.5.1, page 76) une erreur de structure **shape** sera levée. Le typage présenté dans ce chapitre ne nous permet pas de détecter ces erreurs. Voir le chapitre VIII à ce sujet.

Transformation *monotypique*. Toutes les transformations ne sont pas polytypiques. Par exemple le tri par boulier (section III.4.1, page 47) ne s'applique que sur des grilles[5] puisqu'il filtre des éléments voisins selon la direction *nord*. La transformation du tri par boulier s'encode comme suit dans la syntaxe de ce chapitre : `fun (c) -> ( { x,y / nord?  (x,y,c) => [y,x] } c)` .

VII.3.2 Idéaux

Une manière très intuitive de comprendre les types est de les assimiler à des ensembles de valeurs. Par exemple le type *int* dénote l'ensemble des entiers manipulables. Toutefois, on a envie que ces ensembles aient certaines propriétés. Les deux propriétés suivantes ont motivé la définition des ensembles *idéaux* pour modéliser les types [MPS86] :

1. Si une valeur x est moins précise qu'une valeur y dans le domaine des valeurs ($x \sqsubseteq y$) et si y a le type T alors on veut que x ait également le type T.
2. On veut que la plus petite borne supérieure d'une suite croissante de valeurs dans un type soit dans ce même type.

VII.3.3 Expressions de types

La syntaxe des expressions de types est la suivante :

$$\begin{array}{lll} \tau & ::= & b \mid \alpha \mid \tau_1 \rightarrow \tau_2 \mid [\tau]\rho \mid \tau_1 \cup \tau_2 \mid \tau_1 \cap \tau_2 \mid 0 \mid 1 \mid \tau_1?\tau_2 \\ \rho & ::= & r \mid \theta \end{array}$$

[5] On pourrait également considérer que la direction *nord* peut être filtrée dans toute collection mais ce n'est pas le choix que nous avons fait ici.

où $b \in B$ et $r \in R$. La sémantique des types est fondée sur la notion d'*idéal* : un type correspond à un sous-ensemble fermé par le bas du domaine des valeurs $\mathbf{D}$. On distingue types et expressions de types mais nous parlerons simplement de types lorsque le contexte permet de lever l'ambigüité. La relation de sous-typage correspond à l'inclusion ensembliste sur $\mathbf{D}$ et est notée $\subseteq$. Un type ne peut contenir **wrong** mais tout type contient $\perp$ et **shape** (il est usuel de considérer qu'une exception peut avoir tout type [Pie02]). On note $\mathbf{0_D}$ l'ensemble $\{\perp, \mathbf{shape}\}$.

Nous donnons à présent une interprétation intuitive des types avant de donner leur sémantique formelle.

- Le type *flèche*, les types atomiques et les variables de type sont interprétés comme usuellement dans les langages fonctionnels.
- Le type collection $[\tau]\rho$ dénote les collections de topologie ρ et de type contenu τ, *i.e.* dont les éléments ont le type τ. L'ensemble R des topologies a été fixé à titre illustratif mais peut être différent. Seule la présence de *seq* est nécessaire dans R. Dans la suite du document θ désigne une variable de topologie et ρ désigne indifféremment une topologie de R ou une variable de topologie. On peut dire que les topologies et les types ne sont pas de la même *sorte* au sens de [Car91]. Il s'agit d'un typage des expressions de types. Ceci permet de rejeter des expressions de type invalides comme $[bag]seq$ ou $[bool]int$.
- Les types $\tau_1 \cup \tau_2$ et $\tau_1 \cap \tau_2$ correspondent à l'union et l'intersection ensembliste des types.
- Le type 0 contient uniquement les valeurs $\perp$ et **shape**. Le type 0 est inclus dans tous les autres types car $\perp$ et **shape** appartiennent à tous les types. Le type 1 contient toutes les valeurs sauf celles faisant intervenir **wrong**. Le type 1 inclut tous les autres types. Le type $0 \rightarrow 1$ convient à toute fonction (bien typée), y compris aux transformations.
- Le type conditionnel $\tau_1?\tau_2$ vaut τ_1 lorsque τ_2 est différent de 0 et il vaut 0 sinon. Par exemple le type $int?(\tau \cap float)$ vaut int si τ contient $float$ et 0 sinon.

Un schéma de types est de la forme $\forall\alpha_1, \ldots, \alpha_n, \theta_1, \ldots, \theta_m / S.\tau$ où S est un ensemble de contraintes de types de la forme $\tau_1 \subseteq \tau_2$. La contrainte $\tau_1 = \tau_2$ est un raccourci pour la conjonction de $\tau_1 \subseteq \tau_2$ et $\tau_2 \subseteq \tau_1$. On utilisera par la suite le symbole σ pour dénoter un schéma de types et le symbole χ pour désigner indifféremment une variable de type ou une variable de topologie.

Les types récursifs peuvent s'exprimer à l'aide de contraintes. Par exemple le schéma de types $\forall\alpha/\{\alpha = [\alpha]bag \cup int\}.[\alpha]bag$ dénote les multi-ensembles pouvant contenir des entiers et des multi-ensembles contenant à leur tour des entiers et des multi-ensembles et ainsi de suite.

Discussion 4 (Topologies union.) Nous avons choisi de permettre l'union de types mais pas l'union de topologies dans la syntaxe des types. Ceci nous paraît naturel pour la raison suivante. Un type union dénote une indétermination entre deux types « précis ». Par exemple, le type $int \cup bool$ est attribué à une expression qui est un entier ou un booléen sans préjuger entre ces deux possibilités. On peut utiliser l'union entre deux types collections : le type $[int]seq \cup [int]bag$ dénote une valeur qui est une collection, soit un multi-ensemble, soit une séquence et qui contient des entiers. Ce type union exprime bien l'idée d'une collection dont la topologie est indéterminée entre deux possibilités. En utilisant des topologies union on aurait pu écrire le type $[int](seq \cup bag)$. Ceci ne pose pas de problème technique et peut se révéler nécessaire dans certains cas (voir la discussion *Limitations du typage ensembliste* en page 154). Toutefois, nous ne considérons pas ici les topologies union afin de ne pas alourdir la présentation.

Notons tout de même que, si l'on considère les topologies union, le constructeur de types collection est strict sur son deuxième argument (la topologie) et non-strict sur le premier argument (le type contenu). Par exemple, $[\tau](seq \cap bag) = [\tau]0 = 0$ alors que $[0]\rho \neq 0$. En effet, le type $[0]seq$ contient la séquence vide.

$$
\begin{array}{rcl}
[\![\alpha]\!]_s & = & [\![s(\alpha)]\!]_{id} \\
[\![b]\!]_s & = & \mathbf{B}_b \cup \mathbf{0_D} \\
[\![0]\!]_s & = & \{\bot, \mathbf{shape}\} = \mathbf{0_D} \\
[\![1]\!]_s & = & \mathbf{T} \\
[\![[\tau]\rho]\!]_s & = & \mathbf{C}([\![\tau]\!]_s, s(r)) \cup \mathbf{0_D} \\
[\![\tau_1 \to \tau_2]\!]_s & = & \left\{f \in \mathbf{T} \to \mathbf{T} | f([\![\tau_1]\!]_s) \subseteq [\![\tau_2]\!]_s\right\} \cup \mathbf{0_D} \\
[\![\tau_1 \cup \tau_2]\!]_s & = & [\![\tau_1]\!]_s \cup [\![\tau_2]\!]_s \\
[\![\tau_1 \cap \tau_2]\!]_s & = & [\![\tau_1]\!]_s \cap [\![\tau_2]\!]_s \\
[\![\tau_1 ? \tau_2]\!]_s & = & \begin{cases} [\![\tau_1]\!]_s & \text{si } [\![\tau_2]\!]_s \neq \mathbf{0_D} \\ \mathbf{0_D} & \text{sinon} \end{cases} \\
[\![\forall \chi_1, \ldots, \chi_n / S.\tau]\!]_s & = & \bigcap_{s' \in X} [\![\tau]\!]_{s'} \\
 & & \text{où } X = Sol(S) \cap \{s' | s'(\chi) = s(\chi) \text{ si } \chi \notin \{\chi_1, \ldots, \chi_n\}\}
\end{array}
$$

FIG. 1 – Sémantique des types et des schémas de types.

VII.3.4 Interprétation sémantique des types

Nous formalisons à présent la notion de type. Étant donné une expression de types τ et une substitution s des variables de type et de topologie de τ vers des expressions de types closes, la sémantique de τ dans s est définie par la fonction $[\![.]\!]_s$ donnée dans la figure 1.

Cette figure définit également la sémantique des schémas de types. On note $Sol(S)$ l'ensemble des solutions d'un ensemble de contraintes S. Remarquons que les expressions de types et les schémas de types s'évaluent dans le même domaine sémantique, les ensembles idéaux de valeurs.

À chaque constante c du langage on associe un schéma de types $TC(c)$. On suppose que TC est correct par rapport à la sémantique : pour toute constante c du langage et pour toute instanciation s des variables de type, $Eval(c, \emptyset) \in [\![TC(c)]\!]_s$.

Par la suite on dira par abus de langage qu'un type vaut 0 ou qu'il est égal au type 0 lorsqu'il dénote toujours $\mathbf{0_D}$.

VII.3.5 Sous-typage

La définition communément admise du sous-typage est la suivante : on dit que T_1 est un sous-type de T_2 lorsqu'on peut utiliser une valeur du type T_1 partout où une valeur du type T_2 peut être utilisée.

Regardons sur trois exemples ce que cela veut dire dans notre langage.

1. Seule la valeur $\bot$ a le type 0. Considérons une fonction qui a le type $T \to T'$. On peut utiliser une valeur du type T comme argument de cette fonction. Or si on lui passe $\bot$ en argument, aucune erreur de type ne sera provoquée puisqu'elle attendra que l'évaluation de cet argument termine avant de commencer à calculer (sémantique à appel par valeur). Ici on voit bien que $\bot$ peut être utilisé à la place de valeurs de n'importe quel type sans provoquer d'erreur de type. Ceci explique que 0 soit sous-type de tous les autres types.
2. Considérons une fonction du type $(int \cup bool) \to T$. Cette fonction accepte en argument des valeurs du type $int \cup bool$, c'est à dire des valeurs dont on ne sait pas si elles sont dans

$\mathbf{B}_{int}$ ou dans $\mathbf{B}_{bool}$. Si on lui passe une valeur dont on sait qu'elle est dans $\mathbf{B}_{int}$ on n'a pas envie de la voir échouer. En d'autres mots on a envie d'accepter un *int* là où un $int \cup float$ est accepté. Par conséquent on veut que *int* soit un sous-type de $int \cup bool$.

3. Il est facile de voir que si T_1 est un sous-type de T_2 alors $T \to T_1$ est également un sous-type de $T \to T_2$. À présent considérons les deux types $T_1 \to T$ et $T_2 \to T$ tels que T_1 est un sous-type de T_2. Peut-on utiliser une fonction f_1 du type $T_1 \to T$ à la place d'une fonction f_2 du type $T_2 \to T$? Non car il est plus restrictif d'accepter des arguments de type T_1 que d'accepter des arguments du type T_2. Par exemple si T_1 est 0 on sait seulement que f_1 accepte la valeur $\perp$. En l'utilisant dans un contexte où une fonction du type $T_2 \to T$ est nécessaire, on risque de passer à f_1 des valeurs différentes de $\perp$, ce qui pourra mener à une erreur de type. En revanche si on attend une fonction qui accepte des valeurs du type T_1 on peut utiliser une fonction qui attend des valeurs du type T_2, qui est plus générale. On a donc envie que $T_2 \to T$ soit un sous-type de $T_1 \to T$ lorsque T_1 est un sous-type de T_2. Ce renversement de relation de sous-typage dans la partie gauche d'une flèche est connue sous le nom de *contravariance à gauche*.

La relation de sous-typage dont nous avons besoin n'est autre que la relation d'inclusion sur les ensembles dénotés par les types. Nous la noterons $\subseteq$ sur les expressions de types comme sur les ensembles dénotés. La signification de ce symbole sera donc déduite du contexte où il est utilisé. Si τ_1 et τ_2 sont clos alors $\tau_1 \subseteq \tau_2$ *ssi* $[\![\tau_1]\!]_{id} \subseteq [\![\tau_2]\!]_{id}$. Sinon, on va s'intéresser à la question de l'existence d'une substitution s elle que $[\![\tau_1]\!]_s \subseteq [\![\tau_2]\!]_s$. Cette question est traitée en section VII.5. Auparavant, la section suivante donne les règles de typage de notre langage.

VII.4 Règles de typage

La figure 2 donne notre système de règles de typage pour le langage. La relation de typage comporte un contexte de typage Γ et un ensemble S de contraintes de type. La relation $\Gamma, S \vdash e : \tau$ se lit « dans le contexte Γ, l'expression e a le type $s(\tau)$ pour toute solution s de S ». La présence d'un ensemble de contraintes dans la relation de typage est standard dans les systèmes d'inférence de types en présence de sous-typage [FM88]. Toutes les règles sauf les règles *(trans)* et *(trans_all)* sont assez classiques. Décrivons brièvement les règles standard puis détaillons les règles *(trans)* et *(trans_all)*.

(var) : Cette règle correspond à la règle standard de Hindley/Milner.

(const) : TC donne les schémas de types des constantes.

(fun) et (app) : La règle *(fun)* correspond à celle de Hindley/Milner. Dans *(app)* la contrainte impose que le type de l'argument passé à la fonction soit un sous-type du type qu'elle attend.

(gen) et (inst) : La règle *(gen)* sert à introduire le polymorphisme paramétrique dans les types. En effet, elle exprime que si une expression a le type τ sous les contraintes S alors elle a aussi le schéma de type $\forall \chi_i / S.\tau$ où les χ_i sont des variables libres de τ. La règle *(inst)* sert à instancier les schémas de types en types. Pour utiliser cette règle, les contraintes du schéma de type doivent avoir une solution.

(let) : Le *let-polymorphisme* est obtenu en appliquant la règle *(gen)* juste après la règle *(let)*.

Les règles *(trans)* et *(trans_all)* nécessitent plus d'explications. Dans ces règles nous utilisons les deux fonctions définies ci-dessous. La fonction *Comp* s'applique à un type τ et à un motif m et renvoie un type qui vaudra 0 si le motif ne peut s'appliquer dans une collection de type contenu τ

$$\frac{}{\Gamma \cup \{x{:}\sigma\}, S \vdash x : \sigma}\ (var) \qquad \frac{\Gamma \cup \{x{:}\tau_1\}, S \vdash e : \tau_2}{\Gamma, S \vdash \lambda x.e : \tau_1 \to \tau_2}\ (fun)$$

$$\frac{}{\Gamma, S \vdash c : TC(c)}\ (const) \qquad \frac{\Gamma, S \vdash e_1 : \tau_1 \to \tau_1' \quad \Gamma, S \vdash e_2 : \tau_2}{\Gamma, S \cup \{\tau_2 \subseteq \tau_1\} \vdash e_1\ e_2 : \tau_1'}\ (app)$$

$$\frac{\Gamma, S \vdash e : \tau \text{ et } Sol(S) \neq \emptyset \text{ et } \chi_1, \ldots, \chi_n \text{ non libres dans } \Gamma}{\Gamma, \emptyset \vdash e : \forall \chi_1, \ldots, \chi_n / S.\tau}\ (gen)$$

$$\frac{\Gamma, S \vdash e : \forall \alpha_1, \ldots, \alpha_n, \theta_1, \ldots, \theta_m / S'.\tau}{\Gamma, S \cup S'[\tau_i/\alpha_i, \rho_j/\theta_j] \vdash e : \tau[\tau_i/\alpha_i, \rho_j/\theta_j]}\ (inst)$$

$$\frac{\Gamma, S \vdash e_1 : \sigma \quad \Gamma \cup \{x{:}\sigma\}, S \vdash e_2 : \tau}{\Gamma, S \vdash \text{let } x = e_1 \text{ in } e_2 : \tau}\ (let)$$

$$\frac{\Gamma \cup \gamma_\tau(p_i), S \vdash g_i : bool \quad \Gamma \cup \gamma_\tau(p_i), S \vdash e_i : [\tau_i]seq \quad (1 \leq i \leq n)}{\Gamma, S \cup \{\tau \subseteq \tau'\} \cup S' \vdash \{p_1/g_1 \Rightarrow e_1; \ldots; p_n/g_n \Rightarrow e_n\} : [\tau]\rho \to [\tau']\rho}\ (trans)$$

où S' est $\bigcup_{1 \leq i \leq n} \{\tau_i ? Comp_\tau(p_i) \subseteq \tau'\}$.

La règle suivante ne s'applique que lorsque p_n est réduit à un identificateur et g_n est l'expression *true* :

$$\frac{\Gamma \cup \gamma_\tau(p_i), S \vdash g_i : bool \quad \Gamma \cup \gamma_\tau(p_i), S \vdash e_i : [\tau_i]seq \quad (1 \leq i \leq n)}{\Gamma, S \cup S' \vdash \{p_1/g_1 \Rightarrow e_1; \ldots; p_n/g_n \Rightarrow e_n\} : [\tau]\rho \to [\tau']\rho}\ (trans\_all)$$

où S' est $\bigcup_{1 \leq i \leq n} \{\tau_i ? Comp_\tau(p_i) \subseteq \tau'\}$.

FIG. 2 – Règles de typage.

et qui vaudra un type différent de 0 sinon. On dira que cette fonction calcule la *compatibilité* du motif m avec le type τ. On définit inductivement *Comp* par :

$$\begin{array}{ll}
Comp_\tau(\mu, m) & = Comp_\tau(\mu) ? Comp_\tau(m) \\
Comp_\tau(x) & = \tau \cap 1 = \tau \\
Comp_\tau(x{:}b) & = \tau \cap b \\
Comp_\tau(x{:}r) & = \tau \cap [1]r \\
Comp_\tau(* \text{ as } x) & = \tau \cap 1 = \tau \\
Comp_\tau(b* \text{ as } x) & = \tau \cap b \\
Comp_\tau(r* \text{ as } x) & = \tau \cap [1]r
\end{array}$$

Par exemple $Comp_\tau(x_1{:}int, x_2{:}float) = (\tau \cap int)\ ?\ (\tau \cap float)$. Ainsi si τ ne contient pas int et $float$ ce type vaut 0.

La fonction γ calcule le *contexte de typage induit* par un motif m sachant que ce motif s'applique à une collection de type contenu τ. On définit inductivement γ par :

$$\begin{array}{ll}
\gamma_\tau(\mu, m) & = \gamma_\tau(\mu) \cup \gamma_\tau(m) \\
\gamma_\tau(x) & = \{x : \tau\} \\
\gamma_\tau(x : b) & = \{x : \tau \cap b\} \\
\gamma_\tau(x : r) & = \{x : \tau \cap [1]r\} \\
\gamma_\tau(* \text{ as } x) & = \{x : [\tau]seq\} \\
\gamma_\tau(b* \text{ as } x) & = \{x : [\tau \cap b]seq\} \\
\gamma_\tau(r* \text{ as } x) & = \{x : [\tau \cap [1]r]seq\}
\end{array}$$

Détaillons à présent les deux règles de typage. Pour chaque règle de transformation $m_i/g_i \Rightarrow e_i$, le type contenu de la séquence e_i est τ_i sachant que la transformation s'applique à une collection de type contenu τ et en considérant le contexte induit par m_i. Le type $\tau_i?Comp_\tau(m_i)$ vaudra 0 si les conditions de types de m_i font que le motif n'a jamais d'instances dans une collection de type contenu τ (incompatibilité), il vaudra τ_i si la règle peut s'appliquer (compatibilité).

Le type contenu de la collection renvoyée doit être un sur-type de $\tau_i?Comp_\tau(m_i)$ pour chaque i, d'où les contraintes $\tau_i?Comp_\tau(m_i) \subseteq \tau'$ car si la règle peut s'appliquer, la collection renvoyée pourra contenir des éléments de type τ_i.

Par ailleurs lors de l'application d'une transformation sans règle attrape-tout, des valeurs peuvent ne pas être filtrées et resteront dans la collection renvoyée, d'où la contrainte $\tau \subseteq \tau'$ dans *(trans)*. Cette contrainte est la seule différence entre les règles de typage *(trans)* et *(trans_all)*.

VII.4.1 Exemples

Nous donnons ici trois exemples de typage. Le premier exemple est la dérivation suivante :

$$\dfrac{\dfrac{\dfrac{}{\{x : \alpha \cap int\}, \emptyset \vdash [x; 1] : [int]seq}\,(var)}{\emptyset, \{\alpha \subseteq \alpha, (int?(\alpha \cap int)) \subseteq \alpha\} \vdash \{x : int \Rightarrow [x; 1]\} : [\alpha]\theta \to [\alpha]\theta}\,(trans)}{\emptyset, \emptyset \vdash \{x : int \Rightarrow [x; 1]\} : \forall \alpha, \theta/\{\alpha \subseteq \alpha, (int?(\alpha \cap int)) \subseteq \alpha\}.[\alpha]\theta \to [\alpha]\theta}\,(gen)$$

Ici, l'ensemble de contraintes du schéma se réduit à $\emptyset$. En effet, on peut montrer que $(int?(\alpha \cap int)) \subseteq \alpha$ est toujours vrai. Donc n'importe quelle instanciation de α et θ convient. Passons au second exemple :

$$\dfrac{\dfrac{\dfrac{\cdots}{\{x : \alpha \cap int\}, \emptyset \vdash [true] : [bool]seq} \qquad \dfrac{\cdots}{\{x : \alpha \cap int\}, \emptyset \vdash x > 0 : int}}{\emptyset, \{\alpha \subseteq \beta, (bool?(\alpha \cap int)) \subseteq \beta\} \vdash \{x : int/x > 0 \Rightarrow [true]\} : [\alpha]\theta \to [\beta]\theta}\,(trans)}{\emptyset, \emptyset \vdash \{x : int/x > 0 \Rightarrow [true]\} : \forall \alpha, \beta, \theta/\{\alpha \subseteq \beta, (bool?(\alpha \cap int)) \subseteq \beta\}.[\alpha]\theta \to [\beta]\theta}\,(gen)$$

Ici le type $\alpha \cup (bool?(\alpha \cap int))$ est la plus petite instanciation de β vérifiant les contraintes du schéma. Le type suivant est donc valable pour cette transformation : $[\alpha]\theta \to [\alpha \cup (bool?(\alpha \cap int))]\theta$.

Sans l'utilisation de types conditionnels, le type le plus précis pour cette transformation aurait été $[\alpha]\theta \to [\alpha \cup bool]\theta$ qui porte moins d'informations que le type précédent.

Enfin on peut montrer que la transformation $\{x : int \Rightarrow [true]\ ;\ x \Rightarrow [false]\}$ a le type $[\alpha]\theta \to [bool]\theta$ en utilisant la règle *(trans_all)* :

$$\frac{\dots \qquad\qquad \dots}{\dfrac{\{x:\alpha\cap int\},\emptyset\vdash[true]:[bool]seq \qquad \{x:\alpha\},\emptyset\vdash[false]:[bool]seq}{\emptyset,S\vdash\{x{:}int\Rightarrow[true]\ ;\ x\Rightarrow[false]\}:[\alpha]\theta\to[bool]\theta}\ (trans\_all)}$$

avec $S=\{bool?(\alpha\cap int)\subseteq bool; bool?\alpha\subseteq bool\}$. Le système S est toujours vrai car on a toujours $\tau?\tau'\subseteq\tau$. Donc $\emptyset,\emptyset\vdash\{x{:}int\Rightarrow[true]\ ;\ x\Rightarrow[false]\}:[\alpha]\theta\to[bool]\theta$ est vrai.

VII.4.2 Propriétés

Dans cette section nous énonçons la propriété de correction du système de types. On dira qu'un environnement E est correct par rapport à un contexte Γ et à une instanciation s des variables de type lorsque $E(x)\in[\![\Gamma(x)]\!]_s$ pour tout x lié dans Γ et E.

Lemme 1 (Correction) Soit un typage $\Gamma,S\vdash e:\sigma$, une solution s de S, un environnement E correct par rapport à Γ et s portant sur les variables libres de e. Alors $Eval(e,E)\in[\![\sigma]\!]_s$.

Idée de la preuve. Ceci se montre par induction sur la structure de l'arbre de preuve de typage. Pour toutes les règles sauf *(trans)* et *(trans_all)* c'est assez classique (modulo les sortes). Regardons la règle *(trans)* ou plutôt une forme simplifiée sans les types conditionnels dans les contraintes :

$$\frac{\Gamma\cup\gamma_\tau(p_i),S\vdash g_i:bool\quad\Gamma\cup\gamma_\tau(p_i),S\vdash e_i:[\tau_i]seq\quad(1\le i\le n)}{\Gamma,S\cup\{\tau\subseteq\tau'\}\cup\bigcup\limits_{1\le i\le n}\{\tau_i\subseteq\tau'\}\vdash\{p_1/g_1\Rightarrow e_1;\dots;p_n/g_n\Rightarrow e_n\}:[\tau]\rho\to[\tau']\rho}$$

Il s'agit essentiellement de montrer que si s est une solution de $S\cup\{\tau\subseteq\tau'\}\cup\bigcup\limits_{1\le i\le n}\{\tau_i\subseteq\tau'\}$ et si E est un environnement correct par rapport à Γ alors la fonction dénotée par $Eval(t,E)$ que nous noterons f est dans $[\![[\tau]\rho\to[\tau']\rho]\!]_s$. Ceci revient à montrer que pour toute collection c de $[\![[\tau]\rho]\!]_s$ on a $f(c)\in[\![[\tau']\rho]\!]_s$.

- f ne peut provoquer d'erreur de type car les seules expressions évaluées par l'application de f sont les gardes et les expressions remplaçantes, qui sont bien typées par hypothèse d'induction (s est également une solution de S).
- L'application de la i-ème règle va donner un chemin et une séquence de $[\![[\tau_i]seq]\!]_s$. Donc la substitution topologique va retirer certains éléments de la collection c et les remplacer par des éléments de $[\![\tau_i]\!]_s$. La substitution topologique peut également lancer une erreur de structure **shape**.
- Donc la collection renvoyée peut contenir des valeurs de la collection initiale et des valeurs d'un $[\![\tau_i]\!]_s$ pour $i\le n$. Donc la collection renvoyée est dans $[\![[\tau\cup\bigcup\limits_{i\le n}\tau_i]\rho]\!]_s$.
- Or s est solution de $\{\tau\subseteq\tau'\}\cup\bigcup\limits_{1\le i\le n}\{\tau_i\subseteq\tau'\}$ donc $[\![[\tau\cup\bigcup\limits_{i\le n}\tau_i]\rho]\!]_s\subseteq[\![[\tau']\rho]\!]_s$. Donc la collection renvoyée est bien dans $[\![[\tau']\rho]\!]_s$. □

Une preuve complète pour la règle *(trans)* peut être trouvée dans [Coh03a].

Corollaire 1 Si e est un programme (*i.e.* une expression sans variables libres) et si $\emptyset,\emptyset\vdash e:\sigma$ alors $Eval(e,\emptyset)\in[\![\sigma]\!]_{id}$, où id est la substitution identité.

Rappelons qu'à partir d'une preuve de $\Gamma, S \vdash e : \tau$ on obtient une preuve de typage de e avec $\emptyset$ comme ensemble de contraintes en appliquant la règle *(gen)*.

Corollaire 2 Si $\emptyset, \emptyset \vdash e : \sigma$ alors $Eval(e, \emptyset) \neq$ **wrong**.

Ce dernier corollaire est vrai car la valeur **wrong** n'appartient à aucun type. On dira qu'un programme est *bien typé* si il existe un schéma de type σ tel que $\emptyset, \emptyset \vdash e : \sigma$. Il est garantit qu'un programme bien typé ne provoquera pas d'erreur de type mais, comme nous l'avons déjà évoqué, on ne sait rien des erreurs de structure. Ce point est discuté en section VII.10.2 et dans le chapitre VIII.

VII.5 Systèmes de contraintes de sous-typage

Nous avons donné dans la section précédente les règles de typage permettant de prouver qu'un programme est bien typé. Pour montrer le « bon typage » d'un programme, on est amené à vérifier qu'un système de contraintes de sous-typage a une solution (lorsqu'on utilise la règle *(gen)*) ou bien qu'une substitution est solution d'un système (conditions du lemme 1). Pour automatiser l'inférence de types, il est nécessaire d'automatiser la résolution de systèmes de contraintes de sous-typage. C'est ce que nous faisons dans cette section avant de présenter l'inférence automatique en section suivante.

Nous présentons donc dans cette section une méthode de résolution de systèmes de contraintes ensemblistes fondée sur la méthode donnée par [AW92] [AW93]. La méthode de résolution donnée dans ces deux articles a été enrichie dans [ALW94] pour prendre en compte les types conditionnels puis généralisée dans [FA97] pour prendre en compte des systèmes avec des contraintes de sous-typage structurel et non-structurel. Notre adaptation consiste essentiellement à considérer les types collection et à bien choisir la sémantique des types fonctions.

VII.5.1 Conditions à l'utilisation de la méthode

La première étape de la résolution de contraintes consiste à décomposer les contraintes en contraintes plus simples. Par exemple $\tau_1 \cup \tau_2 \subseteq \tau_3$ se décompose en deux contraintes plus simples : $\tau_1 \subseteq \tau_3$ et $\tau_2 \subseteq \tau_3$. Ceci est toujours possible sur les ensembles et sur les types.

Certaines contraintes sont plus difficiles à décomposer. Il s'agit des contraintes de la forme $\tau_1 \subseteq \tau_2 \cup \tau_3$ ou de la forme $\tau_1 \cap \tau_2 \subseteq \tau_3$. En effet, sur des ensembles, la contrainte $E_1 \subseteq E_2 \cup E_3$ est équivalente à $E_1 \cap \neg E_2 \subseteq E_3$ et $E_1 \cap \neg E_3 \subseteq E_2$, où $\neg$ représente le complément ensembliste. Or, si E_2 et E_3 sont des types (*i.e.* des idéaux), $\neg E_2$ et $\neg E_3$ ne le sont pas nécessairement (même en leur ajoutant $\perp$ et **shape**, qui sont dans tout type). Par exemple, l'ensemble $1 \to 0$ est un type mais son complément n'en est pas un car ne contenant pas la plus petite fonction $\lambda x.\perp$ il n'est pas clos par le bas (ce n'est pas un idéal). De même pour la deuxième forme de contraintes difficiles : $\tau_1 \cap \tau_2 \subseteq \tau_3$ est équivalent à $\tau_1 \subseteq \tau_3 \cup \neg\tau_2$ seulement lorsque $\neg\tau_2$ est un type et à $\tau_2 \subseteq \tau_3 \cup \neg\tau_1$ lorsque $\neg\tau_1$ est un type. Ces équivalences sur les contraintes ensemblistes ne peuvent donc pas être utilisées sur des types. Nous utiliserons à la place des équivalences similaires mais qui ne sont vraies que sous certaines restrictions.

Opération de complément

Nous introduisons dans la figure 3 une opération de complément $\neg\tau$ qui n'est valide que sur les types fermés par le haut. En effet, le complément d'un type qui ne serait pas fermé par le haut ne serait pas fermé par le bas et ne serait donc pas un type. La fermeture par le haut de τ est notée $\overline{\tau}$. En pratique on est amené à calculer conjointement la fermeture et le complément, d'où la définition combinée de $\neg\overline{\tau}$ dans cette figure.

$$
\begin{array}{rcl}
\overline{0} & = & 0 \\
\overline{1} & = & 1 \\
\overline{\alpha} & = & 1 \\
\overline{b} & = & b \\
\overline{\tau_1 \cap \tau_2} & = & \overline{\tau_1} \cap \overline{\tau_2} \\
\overline{\tau_1 \cup \tau_2} & = & (\overline{\tau_1} \cap \neg\overline{\tau_2}) \cup (\overline{\tau_1} \cap \overline{\tau_2}) \cup (\neg\overline{\tau_1} \cap \overline{\tau_2}) \\
\overline{\tau_1 \to \tau_2} & = & 0 \to 1 \\
\overline{[\tau]r} & = & [\overline{\tau}]r \\
\overline{[\tau]\theta} & = & \bigcup_{r\in R} [\overline{\tau}]r \\
\overline{\tau_1 ? \tau_2} & = & \overline{\tau_1} ? \overline{\tau_2}
\end{array}
$$

$$
\begin{array}{rcl}
\neg\overline{0} & = & 1 \\
\neg\overline{1} & = & 0 \\
\neg\overline{\alpha} & = & 0 \\
\neg\overline{b} & = & \left(\bigcup_{r\in R} [1]r\right) \cup [\neg\overline{\tau}]r \cup \left(\bigcup_{b'\in B-b} b'\right) \cup 0 \to 1 \\
\neg\overline{\tau_1 \cap \tau_2} & = & (\neg\overline{\tau_1} \cap \overline{\tau_2}) \cup (\neg\overline{\tau_1} \cap \neg\overline{\tau_2}) \cup (\overline{\tau_1} \cap \neg\overline{\tau_2}) \\
\neg\overline{\tau_1 \cup \tau_2} & = & \neg\overline{\tau_1} \cap \neg\overline{\tau_2} \\
\neg\overline{\tau_1 \to \tau_2} & = & \left(\bigcup_{r\in R} [1]r\right) \cup \left(\bigcup_{b\in B} b\right) \\
\neg\overline{[\tau]r} & = & \begin{cases} \left(\bigcup_{r'\in R-r} [1]r'\right) \cup \left(\bigcup_{b\in B} b\right) \cup 0 \to 1 \text{ si } \overline{\tau} = 1 \\ \left(\bigcup_{r'\in R-r} [1]r'\right) \cup [\neg\overline{\tau}]r \cup \left(\bigcup_{b\in B} b\right) \cup 0 \to 1 \text{ sinon} \end{cases} \\
\neg\overline{[\tau]\theta} & = & \begin{cases} \left(\bigcup_{b\in B} b\right) \cup 0 \to 1 \text{ si } \overline{\tau} = 1 \\ \left(\bigcup_{r\in R} [\neg\overline{\tau}]r\right) \cup \left(\bigcup_{b\in B} b\right) \cup 0 \to 1 \text{ sinon} \end{cases} \\
\neg\overline{\tau_1 ? \tau_2} & = & (\neg\overline{\tau_1}) \cup (1 ? \neg\overline{\tau_2})
\end{array}
$$

FIG. 3 – Fermeture et complément

Illustrons la notion de fermeture par le haut notée $\overline{\tau}$ à travers plusieurs exemples :

- Le type $1 \to 0$ n'est pas fermé par le haut. En effet il existe une valeur v dans ce type (différente de $\bot$) qui est au dessous d'une autre valeur v' dans $\mathbf{D}$ qui n'est pas dans ce type. En effet, on peut prendre $v = \lambda x.\bot$ et $v' = \lambda x.true$. La valeur v est bien dans $1 \to 0$ mais v' n'y est pas et $v \sqsubseteq v'$.
- Idem pour le type $1 \to int$. On peut prendre les mêmes exemples de valeurs v et v' mais considérons un autre choix pour v et v' : $v = (\lambda x.\texttt{if}\ x = true\ \texttt{then}\ \bot\ \texttt{else}\ 2)$ et $v' = (\lambda x.\texttt{if}\ x = true\ \texttt{then}\ true\ \texttt{else}\ 2)$. On a bien $v \sqsubseteq v'$ avec v dans $1 \to int$ et v' en dehors de ce type.
- Considérons le type $[0]bag$. Il n'y a qu'une valeur différente de $\bot$ dans ce type, c'est le bag vide. Cette valeur n'a aucune valeur au dessus d'elle ni dans le type $[0]bag$, ni en dehors. Donc le type $[0]bag$ est fermé par le haut.
- Regardons $[bool]bag$. On ne peut construire de valeur différente de $\bot$ dans $[bool]bag$ possédant une valeur au dessus d'elle hors de $[bool]bag$. Donc ce type est fermé par le haut. En revanche on pourrait pour $[1 \to 0]bag$. Donc celui-ci n'est pas fermé par le haut.

La figure 3 détaille le calcul de la fermeture par le haut d'un type.

Le complément introduit ici n'est pas le complément ensembliste mais il a de bonnes propriétés : $\overline{\tau} \cup \neg\overline{\tau} = 1$ et $\overline{\tau} \cap \neg\overline{\tau} = 0$. Ce complément nous permet d'établir les équivalences ci-dessous.

Réduction des contraintes difficiles

Les deux équivalences ci-dessous permettent de réduire certaines contraintes difficiles :

$$\tau_1 \subseteq \tau_2 \cup \tau_3 \;\Leftrightarrow\; \tau_1 \cap \neg\overline{\tau_2} \subseteq \tau_3 \;\wedge\; \tau_1 \cap \neg\overline{\tau_3} \subseteq \tau_2 \tag{1}$$

$$\alpha \cap \tau_{up} \subseteq \tau_2 \;\Leftrightarrow\; \alpha \subseteq (\tau_2 \cap \tau_{up}) \cup \neg\tau_{up} \tag{2}$$

L'équivalence 1 est vraie lorsque l'union $\tau_2 \cup \tau_3$ est disjointe (pour toute instanciation des variables de τ_1 et τ_2, l'ensemble dénoté par l'intersection $\tau_2 \cap \tau_3$ est réduit à $\mathbf{0_D}$). L'équivalence 2 est vraie lorsque τ_{up} est un type clos et fermé par le haut. Il est suffisant de considérer les intersections de la forme $\alpha \cap \tau_{up}$ si les types sont en forme normale disjonctive. En effet, dans ce cas les intersections sont éliminées autant que possible et seules les intersections dont un membre est une variable de type ne peuvent être éliminées. En partie droite de cette équivalence, on peut remarquer qu'on a le type $(\tau_2 \cap \tau_{up}) \cup \neg\tau_{up}$ alors que le type $\tau_2 \cup \neg\tau_{up}$ aurait suffit. Ce choix permet d'avoir une union disjointe en partie droite de la contrainte résultante. Les opérations de fermeture et de complément utilisées dans les équivalences ci-dessous sont définies dans la figure 3. Le calcul d'une forme normale disjonctive est donné dans la figure 4.

Les équivalences 1 et 2 sont vues comme des règles de réécritures en les orientant de gauche à droite. En théorie, il est possible d'appliquer alternativement l'une puis l'autre sans fin sur une contrainte $\alpha \cap \tau \subseteq \tau'$ puisque les intersections difficiles sont transformées en unions difficiles et vice versa. Mais l'algorithme proposé en section suivante ne réduisant pas les règles de la forme $\alpha \subseteq \tau$ et $\tau \subseteq \alpha$, ce problème n'est pas rencontré.

Contraintes convenables

On dit qu'une expression de type en partie droite (resp. gauche) d'une contrainte apparaît en position *positive* (resp. *négative*). Une sous-expression de type apparaît en même position que l'expression de type qui la contient excepté pour les membres gauches d'une flèche qui apparaissent en position opposée. Par exemple dans la contrainte $\tau_1 \to \tau_2 \subseteq \tau_3 \to \tau_4$, les

$$\tau \cup \tau = \tau \qquad \tau \cap 0 = 0$$
$$\tau \cap \tau = \tau \qquad b_1 \cap b_2 = 0 \text{ si } b_1 \neq b_2$$
$$\tau \cap 1 = \tau \qquad b \cap [\tau]\rho = 0$$
$$(\alpha \cap \tau_1) \cap \tau_2 = \alpha \cap (\tau_1 \cap \tau_2) \text{ si } \tau_2 \text{ n'est pas une variable.} \qquad [\tau_1]r_1 \cap [\tau_2]r_2 = 0 \text{ si } r_1 \neq r_2$$
$$(\tau_1 \cup \tau_2) \cap \tau_3 = (\tau_1 \cap \tau_3) \cup (\tau_2 \cap \tau_3) \qquad \tau_1 \to \tau_2 \cap b = 0$$
$$[\tau_1]\rho \cap [\tau_2]\rho = [\tau_1 \cap \tau_2]\rho \qquad \tau_1 \to \tau_2 \cap [\tau]\rho = 0$$
$$\tau_1 \to \tau_2 \cap 0 \to 1 = \tau_1 \to \tau_2$$

FIG. 4 – Forme normale disjonctive d'une expression de type

expressions $\tau_1 \to \tau_2$, τ_2 et τ_3 apparaissent en position positive alors que $\tau_3 \to \tau_4$, τ_1 et τ_4 apparaissent en position négative.

Définition 3 Une contrainte est dite *convenable* si :

1. les unions en position positive sont disjointes ($\tau_1 \cap \tau_2 = 0$ pour toute instanciation) ;
2. dans une intersection $\tau_1 \cap \tau_2$ en position négative, τ_1 ou τ_2 est clos (sans variables) et fermé par le haut ;
3. aucun type conditionnel n'apparaît en position positive.

Par exemple le type $\alpha \cup int$ ne convient pas en position positive. Ceci implique que l'on pourra donner à une fonction le type $\alpha \to (\alpha \cup int)$ mais pas le type $(\alpha \cup int) \to \alpha$.

VII.5.2 Réduction des contraintes convenables

La réduction de contraintes consiste à décomposer les contraintes du système en contraintes plus « petites » et à le saturer par transitivité de $\subseteq$ jusqu'à arriver à un système dont on sait extraire les solutions.

Les équations de la figure 5 sont utilisées comme règles de réécriture en les orientant de gauche à droite. Ces équations font intervenir des ensembles de systèmes de contraintes : $\mathcal{C}$ désigne un tel ensemble de systèmes et S désigne un système (un ensemble de contraintes).

Dans les équations 3, 4 et 5, les contraintes considérées sont toujours vraies et peuvent être supprimées du système. En revanche, dans les équations 6, 7 et 8, les contraintes sont toujours fausses. Le système considéré n'a donc pas de solution et est supprimé de l'ensemble des systèmes à résoudre.

La règle 9 exprime que deux types collections sont en relation de sous-typage si leurs types contenus sont en relation de sous-typage et que leurs topologies sont égales. Les égalités entre topologie se résolvent par unification.

La règle 10 exprime la contravariance à gauche des types fonctions. Selon cette règle, on voit également qu'une fonction peut être utilisée avec le type $0 \to \tau$ lorsqu'une fonction du type $\tau_1 \to \tau_2$ est attendue. Ceci est correct car une fonction du type $0 \to \tau$ n'aura que $\bot$ ou **shape** en argument. Le langage étant à appels par valeur, la fonction ne renverra jamais de valeur ou bien elle renverra **shape**, donc on n'aura jamais de valeur du type τ utilisée à la place d'une valeur de type τ_2.

$$\mathcal{C}, S \cup \{0 \subseteq \tau\} \equiv \mathcal{C}, S \quad (3)$$
$$\mathcal{C}, S \cup \{b \subseteq b\} \equiv \mathcal{C}, S \quad (4)$$
$$\mathcal{C}, S \cup \{\alpha \subseteq \alpha\} \equiv \mathcal{C}, S \quad (5)$$
$$\mathcal{C}, S \cup \{b \subseteq \tau\} \equiv \mathcal{C} \quad \text{si } \tau \text{ est 0 ou de la forme } [\tau']\rho \text{ ou } \tau' \to \tau'' \text{ ou } b' \text{ avec } b' \neq b \quad (6)$$
$$\mathcal{C}, S \cup \{[\tau_1]\rho \subseteq \tau_2\} \equiv \mathcal{C} \quad \text{si } \tau_2 \text{ est 0 ou } b \text{ ou } \tau_2' \to \tau_2'' \quad (7)$$
$$\mathcal{C}, S \cup \{\tau_1 \to \tau_1' \subseteq \tau_2\} \equiv \mathcal{C} \quad \text{si } \tau_2 \text{ est 0 ou } b \text{ ou } [\tau_2']\rho \quad (8)$$
$$\mathcal{C}, S \cup \{[\tau_1]\rho_1 \subseteq [\tau_2]\rho_2\} \equiv \mathcal{C}, S \cup \{\tau_1 \subseteq \tau_2,\ \rho_1 = \rho_2\} \quad (9)$$
$$\mathcal{C}, S \cup \{\tau_1 \to \tau_1' \subseteq \tau_2 \to \tau_2'\} \equiv \mathcal{C}, S \cup \{\tau_2 \subseteq \tau_1,\ \tau_1' \subseteq \tau_2'\},\ S \cup \{\tau_2 \subseteq 0\} \quad (10)$$
$$\mathcal{C}, S \cup \{\tau_1 ? \tau_1' \subseteq \tau_2\} \equiv \mathcal{C}, S \cup \{\tau_1' \subseteq 0\}, S \cup \{\tau_1 \subseteq \tau_2\} \quad (11)$$
$$\mathcal{C}, S \cup \{\tau_1 \cup \tau_1' \subseteq \tau_2\} \equiv \mathcal{C}, S \cup \{\tau_1 \subseteq \tau_2,\ \tau_1' \subseteq \tau_2\} \quad (12)$$
$$\mathcal{C}, S \cup \{\tau_1 \subseteq \tau_2 \cap \tau_2'\} \equiv \mathcal{C}, S \cup \{\tau_1 \subseteq \tau_2,\ \tau_1 \subseteq \tau_2'\} \quad (13)$$
$$\mathcal{C}, S \cup \{\tau_1 \subseteq \tau_2 \cup \tau_2'\} \equiv \mathcal{C}, S \cup \{\tau_1 \cap \neg\overline{\tau_2} \subseteq \tau_2',\ \tau_1 \cap \neg\overline{\tau_2'} \subseteq \tau_2\} \quad (14)$$
$$\mathcal{C}, S \cup \{\alpha \cap \tau \subseteq \alpha\} \equiv \mathcal{C}, S \quad (15)$$
$$\mathcal{C}, S \cup \{\alpha \cap \tau_1 \subseteq \tau_2\} \equiv \mathcal{C}, S \cup \{\alpha \subseteq (\tau_2 \cap \tau_1) \cup \neg\tau_1\} \quad (16)$$

FIG. 5 – Réduction des contraintes.

La règle 11 sur les types conditionnels transforme un système en deux systèmes correspondant aux deux situations possibles. Dans le premier, τ_1' vaut 0 et la contrainte $\tau_1 ? \tau_1' \subseteq \tau_2$ devient $0 \subseteq \tau_2$, qui est toujours vrai. Dans le second, τ_1' n'est pas nul et la contrainte se transforme donc en $\tau_1 \subseteq \tau_2$. Il n'est pas nécessaire de préciser que τ_1' est non nul car la contrainte peut être vue comme une implication $(\tau_1' \neq 0) \Rightarrow (\tau_1 \subseteq \tau_2)$ soit $(\tau_1 \subseteq \tau_2) \vee (\tau_1' = 0)$.

Les règles 12 et 13 correspondent aux cas faciles sur l'union et l'intersection dans les contraintes. Les règles 14, 15 et 16 correspondent aux cas difficiles de décomposition.

Les règles 15 et 16 suffisent à gérer tout les cas avec une intersection à gauche si les types considérés sont en forme normale disjonctive.

La réduction d'une contrainte convenable par une règle de la figure 5 ne produit que des contraintes convenables.

Particularités de notre système

Les points spécifiques à notre système par rapport aux travaux originaux [AW92, AW93, ALW94] sont les suivants :

– Dans [AW93] une fonction peut être appliquée à toute valeur sans provoquer d'erreur de type car le langage qui y est présenté ne comporte pas d'opérateurs (des constantes pouvant être appliquées). Ceci se traduit dans la réduction de contraintes par le fait que $\tau_1 \to \tau_1' \subseteq \tau_2 \to 1$ est vrai quel que soit τ_2. Dans notre langage en revanche des erreurs de type peuvent apparaître lors de l'application d'opérateurs comme l'addition entière ou bien de transformations, qui ne s'appliquent qu'à des collections. Par conséquent la contrainte ci-dessus n'est pas toujours vraie dans notre système et la règle 10 est différente de la règle décomposant les types fonctions de [AW93].

– Une collection vide de topologie r peut avoir[6] le type $[0]r$. Ceci ne signifie pas que le calcul d'une valeur de ce type ne termine pas. Par conséquent, l'équation 7 jugeant $[\tau_1]\rho \subseteq \tau_2$ ne prévoit pas le cas particulier où τ_1 est nul. On dit que les types collections sont non-stricts sur le type contenu.
– Nous avons inclu une exception dans le plus petit type : $[\![0]\!] = \{\bot, \mathbf{shape}\}$.
– Nous manipulons plusieurs *sortes* dans les types.

Algorithme de réduction des contraintes

L'algorithme de réduction des contraintes convenable consiste à appliquer alternativement les deux étapes suivantes à l'ensemble de systèmes de contraintes considéré jusqu'à atteindre un point fixe :

1. Réduire toute contrainte n'étant pas *inductive* (définition ci-dessous) à l'aide des règles données ci-dessus.
2. Saturer les systèmes de contraintes par transitivité de la relation $\subseteq$. Par exemple, si $\tau_1 \subseteq \tau_2$ et $\tau_2 \subseteq \tau_3$ sont dans un système S, remplacer S par $S \cup \{\tau_1 \subseteq \tau_3\}$.

La notion de *contrainte inductive* nécessite l'introduction de la notion de *variables de surface* d'un type. Intuitivement, les variables de surface d'un type sont ses variables dont les occurences ne sont pas sous un constructeur (de type collection ou de type fonction).

Définition 4 L'ensemble des variables de surface d'un type τ, noté $\mathit{Surf}(\tau)$ est défini inductivement sur la structure des types par :

$$\begin{array}{rcl}
\mathit{Surf}(\alpha) & = & \{\alpha\} \\
\mathit{Surf}(b) & = & \emptyset \\
\mathit{Surf}([\tau]\rho) & = & \emptyset \\
\mathit{Surf}(\tau_1 \to \tau_2) & = & \emptyset \\
\mathit{Surf}(\tau_1 \cup \tau_2) & = & \mathit{Surf}(\tau_1) \cup \mathit{Surf}(\tau_2) \\
\mathit{Surf}(\tau_1 \cap \tau_2) & = & \mathit{Surf}(\tau_1) \cup \mathit{Surf}(\tau_2) \\
\mathit{Surf}(0) & = & \emptyset \\
\mathit{Surf}(\tau_1 ? \tau_2) & = & \mathit{Surf}(\tau_1) \cup \mathit{Surf}(\tau_2)
\end{array}$$

Définition 5 Étant donné un ordre total sur les variables de type, une contrainte est *inductive* si elle est de la forme $\alpha \subseteq \tau$ ou $\tau \subseteq \alpha$ et si toutes les variables de surface de τ sont *plus petites* que α.

Dans la suite de ce chapitre nous considérerons l'ordre suivant sur les variables : $\alpha_i \leq \alpha_j$ *ssi* $i \leq j$.

L'application des règles de la figure 5 jusqu'à atteindre un point fixe ne termine pas naturellement. Ce sont les deux points suivants qui font que cet algorithme termine :
– on élimine le plus d'intersections possible par la mise en forme normale ;
– on ne réduit pas toutes les contraintes réductibles (seulement les contraintes qui ne sont pas inductives).

[6] On peut décider indifféremment que $TC(\mathit{empty\_set}) = [0]\mathit{set}$ (plus précisément $\forall/\emptyset.[0]\mathit{set}$) ou que $TC(\mathit{empty\_set}) = \forall\alpha/\emptyset.[\alpha]\mathit{set}$.

Complexité de la réduction

La réduction des contraintes est l'étape coûteuse de la résolution de systèmes de contraintes ensemblistes. Nous donnons ici la complexité de ce problème.

La satisfiabilité d'un système de contraintes ensemblistes avec unions, intersections et compléments (mais sans types fonctions ou types conditionnels) est un problème NEXPTIME-complet [AW92, AKVW93]. L'ajout des types fonctions ([AW93]) n'augmente pas la complexité mais réduit le domaine des systèmes solvables par l'algorithme proposé. Si l'on se passe d'unions et d'intersections en positions difficiles (et également de types conditionnels) alors la résolution prend un temps en $O(n^3)$ [Aik99]. Donc la résolution de systèmes de contraintes dans notre système est en temps exponentiel non-déterministe.

Regardons si on peut se passer des unions et intersections en positions difficiles afin d'avoir une complexité cubique.

- Dans notre système, les unions en positions difficiles sont peu utilisées. En effet il n'est pas possible d'exprimer dans le langage une fonction nécessitant de considérer le type $int \cup bool \to \tau$ par exemple. Toutefois, considérons la transformation suivante : { `x:bool => [x] ; x => [x+1]` }. Le type $[int \cup bool]\theta \to [int \cup bool]\theta$ est correct pour cette transformation. Certes, nos règles de typage ne permettent pas de le montrer mais nous espérons pouvoir les enrichir afin de prendre en compte de telles transformations (voir section VII.10.1, page 153). Cet exemple montre bien l'utilité des unions en positions difficiles.
- Des intersections en positions difficiles sont introduites par la règle de typage des transformations via les fonctions γ et *Comp*. Les intersections introduites par γ ne sont pas nécessaires. En effet le contexte de typage $\{x : \tau_1 \cap \tau_2\}$ peut être remplacé par $\{x : \forall\alpha/\{\alpha \subseteq \tau_1; \alpha \subseteq \tau_2\}.\alpha\}$. En revanche, on ne peut pas éliminer aussi facilement les intersections introduites par *Comp*.
- Les types conditionnels permettent d'exprimer le fait qu'une règle $x_1 : b_1, x_2 : b_2 \Rightarrow s$ ne s'applique que lorsque la collection peut contenir des éléments du type b_1 et des éléments du type b_2. On peut se passer de cette information mais le typage en sera moins précis et plus de programmes corrects seront rejetés. En se passant de cette information, on élimine également les intersections en positions difficiles (point précédent).

Ceci montre qu'il est nécessaire de restreindre l'expressivité du système proposé pour atteindre une complexité théorique polynomiale.

Par ailleurs, le nombre de systèmes que nous avons à réduire *dans le pire cas* est 2^n où n est le nombre de types conditionnels dans le système initial. On trouvera dans [SA01] une étude plus fine de la complexité induite par les types conditionnels. Toutefois, cette étude est menée dans un contexte différent (contraintes d'égalité).

Regardons à présent les moyens d'arriver à une implémentation de l'inférence de types aux performances raisonnables.

Implémentation

Des travaux montrent comment implémenter une résolution de contraintes ensemblistes de façon à avoir de bonnes performances. La plupart se concentrent sur des systèmes sans contraintes difficiles ([FA96, FFSA98, FSA00]). Toutefois, il est montré dans [FA97] comment diminuer le coût de traitement des contraintes difficiles. Des implémentations fondées sur ces travaux sont disponibles sous forme de bibliothèque (Bane [AFFS98] et Banshee [Kod]). Notons que ces outils ne permettent de calculer qu'une solution particulière alors que selon les besoins, différentes

solutions peuvent être pertinentes (voir section VII.6.3).

Les travaux autour des simplifications de contraintes de sous-typage en général sont également à prendre en compte (par exemple [Pot01]).

De notre coté, nous avons réalisé une implémentation de l'algorithme proposé sans mettre en œuvre les techniques des articles cités ci-dessus. Cette implémentation est intégrée à un compilateur expérimental MGS dédié à montrer que ce langage peut être implémenté efficacement (voir le chapitre IX) et représente environ 2000 lignes d'OCaml. Sous condition de mémoïser les calculs ayant lieu de nombreuses fois (les mises en forme normale par exemple), notre implémentation nous a permis de typer de nombreux programmes MGS de petite taille dans des temps inférieurs à la demi-seconde. Dans ce cadre expérimental, notre prototype est suffisant mais il est probable que pour typer dans des temps très courts des programmes de taille réelle, comme celui développé dans [BSM04], il soit nécessaire d'implémenter les optimisations proposées dans les travaux ci-dessus.

VII.5.3 Calcul des solutions

La réduction des contraintes que nous venons de voir n'est que la première étape du processus de résolution. Les suivantes consistent à mettre le système sous une forme dont une solution est directement extractible puis en extraire une. Nous décrivons à présent les étapes de ce processus.

Soit S le système de contraintes que l'on cherche à résoudre. La première étape consiste à appliquer la réduction de contraintes à l'ensemble $\{S\}$ (rappelons que l'algorithme de réduction attend un ensemble de systèmes de contraintes en argument).

Les systèmes dans l'ensemble ont tous des solutions. L'union des solutions de ces systèmes est exactement l'ensemble de solutions de S. Si l'ensemble de systèmes est vide alors le système initial n'a pas de solution.

Système inductif

Une fois toutes que toutes les contraintes des systèmes sont inductives, on les combine de manière à obtenir des *systèmes inductifs*.

Définition 6 Un système de contraintes est inductif si on peut l'écrire sous la forme

$$\{I_i \subseteq \alpha_i \subseteq S_i\}_{i \leq n}$$

où les variables de surface de I_i et les variables de surface de S_i sont plus petites que α_i, pour tout i.

Pour obtenir un système inductif à partir d'un système de contraintes inductives, on combine les contraintes inductives de la manière suivante :
- deux contraintes inductives $\alpha \subseteq \tau_1$ et $\alpha \subseteq \tau_2$ sont combinées en une contrainte $\alpha \subseteq \tau_1 \cap \tau_2$;
- deux contraintes inductives $\tau_1 \subseteq \alpha$ et $\tau_2 \subseteq \alpha$ sont combinées en une contrainte $\tau_1 \cup \tau_2 \subseteq \alpha$.

Équations en cascade

Les systèmes inductifs sont ensuite transformés en systèmes d'*équations en cascade*.

Définition 7 Un système d'équations de types est en cascade s'il est de la forme

$$\{\alpha_i = \tau_i\}_{i \leq n}$$

où les variables de surface de τ_i sont plus petites que α_i, pour tout i.

À un système inductif $\{I_i \subseteq \alpha_i \subseteq S_i\}_{i \leq n}$ on associe le système d'équations

$$\{\alpha_i = I_i \cup (\beta_i \cap S_i)\}_{i \leq n}$$

où les β_i sont des variables fraîches plus petites que les variables du système inductif.

Élimination des variables de surface

L'étape suivante consiste à passer d'un système d'équations en cascade à un système d'équations *contractives*, dont on peut simplement extraire les solutions.

Définition 8 Une équation $\alpha = \tau$ est *contractive* si τ ne contient pas de variables de surface.

Tout système d'équations contractives a une unique solution [MPS86].

On passe d'un système d'équations en cascade $\{\alpha_i = I_i \cup (\beta_i \cap S_i)\}_{i \leq n}$ à un système d'équations contractives en

– remplaçant les β_i par des types clos et en
– remplaçant α_i par τ_i dans τ_{i+1} à τ_n.

Tout choix des β_i induit un système contractif et donc une solution particulière (deux choix différents pour les β_i peuvent toutefois donner une même solution). Ce choix peut être vu comme le degré de liberté permettant de faire varier les α_i entre leur borne inférieure et leur borne supérieure.

Selon le choix de l'instanciation des β_i on peut avoir des solutions différentes. Si on prend tous les β_i à 0 on aura la solution la plus précise d'un système (les α_i prennent leur borne inférieure). En prenant les β_i à 0 ou 1 selon que la variable α_i est covariante ou contravariante on aura une solution qui portera une information différente (voir la section VII.6.3).

Extraction des solutions

Les solutions d'un système contractif s'obtiennent par substitution. Notons toutefois que certaines solutions s'expriment comme le plus petit point fixe d'une fonction (par exemple $f(\alpha) = [\alpha]seq$ pour l'équation $\alpha = [\alpha]seq$).

Notons que [AW93] garantit que la suite de transformations successives que nous avons appliquées à un système de contraintes pour arriver à un ensemble de systèmes d'équations en cascade préserve les solutions et que toutes les solutions peuvent être atteintes en choisissant bien les β_i. Toutefois, nous n'avons pas montré que les preuves qui y sont données sont valides dans notre système. Ceci reste à faire pour valider rigoureusement notre approche.

VII.5.4 Autres approches pour la résolution des contraintes de sous-typage

La méthode que nous avons exposée juge la relation de sous-typage en jugeant la satisfiabilité d'un ensemble d'inclusions ensemblistes. Ceci permet à l'algèbre de types d'être riche (avec des

types union notamment) et nécessite une résolution de contraintes ensemblistes *dédiée* puisque les types sont des ensembles idéaux. Regardons les travaux existants dans le domaine du sous-typage non-structurel et de la résolution de contraintes ensemblistes.

De nombreux travaux étudient le sous-typage non-structurel. Comme son nom l'indique, une relation de sous-typage non-structurelle n'est pas fondée sur la structure des types. Par exemple les système de types avec sous-typage non structurel ont souvent un plus petit type 0 (noté également $\bot$) et un plus grand type 1 (ou $\top$). Notre système est non-structurel puisque nous avons un plus petit type, un plus grand type et puisqu'un type τ est plus petit qu'un type $\tau \cup \tau'$, bien que ces deux types n'ont pas la même structure.

Par exemple, [PWO97] et [JP99] montrent comment juger le sous-typage non-structurel (avec $\bot$, $\top$ et $\rightarrow$). Leur approche est inspirée de l'idée de *simulation* par des automates non-déterministes dans le domaine des systèmes concurrents et a une complexité cubique. Voir également [EST95] et [Pot98b].

Une sémantique ensembliste des types est nécessaire pour pouvoir manipuler des contraintes entre types union. On trouvera dans [CDG+97] une description de la résolution de nombreux problèmes autour de la résolution de contraintes ensemblistes, fondées sur des automates d'arbres réguliers. Les contraintes ensemblistes ont été très utilisées pour l'analyse de programmes ([Hei94] et [FF97] par exemple). Mais les travaux où les contraintes ensemblistes sont utilisées sur des types sont plus rares, principalement en raison de la difficulté de gérer les constructeurs contravariants (comme le constructeur de types $\rightarrow$). En dehors des travaux sur lesquels nous nous sommes fondés et que nous avons déjà mentionnés, nous retiendrons [FCB02] où un modèle ensembliste des types original est développé, indépendamment de la notion d'idéal.

VII.6 Inférence automatique

Nous avons vu comment décider automatiquement de la satisfiabilité d'un ensemble de contraintes de sous-typage. Ceci nous permet de proposer une inférence de types automatique, que nous décrivons dans cette section.

L'inférence automatique des types d'un programme consiste en deux étapes. En premier lieu, le schéma de type le plus général du programme (et donc ses contraintes) est calculé en appliquant les règles de typage selon une stratégie appropriée. Ensuite on calcule les solutions des contraintes générées. Nous décrivons ces deux étapes dans cette section.

Notons que les notions de type le plus général et de type principal sont différentes. Un programme e a σ comme *type le plus général ssi* $\vdash e : \sigma$ et pour tout σ', si $\vdash e : \sigma'$ alors $\sigma \subseteq \sigma'$. Il s'agit d'une propriété sémantique. Le type le plus général d'une expression est le type le plus précis qu'on puisse lui donner. Un programme e a σ comme *type principal ssi* $\vdash e : \sigma$ et pour tout σ', si $\vdash e : \sigma'$ alors σ' est une instance de σ. Il s'agit d'une propriété syntaxique. Les deux notions coïncident dans un typage simple *à la* Hindley/Milner : si un type est principal alors il est le plus général dans le modèle idéal. Dans les systèmes de types avec contraintes un programme peut avoir un type le plus général sans avoir de type principal.

VII.6.1 Production des contraintes

La stratégie d'application des règles de typage suivante définit la dérivation la plus générale modulo renommage des variables :

- On utilise des variables fraîches partout où cela est possible lorsqu'on instancie les règles de typage.
- Après avoir appliqué la règle *(let)* on applique *(gen)* pour avoir le *let-polymorphisme*.
- On applique *(inst)* après avoir appliqué la règle *(var)* ou *(const)*.
- La dérivation se termine par une application de la règle *(gen)*.
- Les règles *(gen)* et *(inst)* ne sont appliquées nulle part ailleurs.
- On applique *(trans_all)* à la place de *(trans)* partout où c'est possible.

À chaque application des règles *(app)*, *(trans)*, *(trans_all)* et *(inst)*, des contraintes sont générées. La dérivation la plus générale calcule le (schéma de) type le plus général du programme, ce schéma de types comprenant les contraintes produites.

VII.6.2 Applicabilité de la méthode de résolution

Nous avons vu que la méthode de résolution présentée en section VII.5 n'est valide que sous certaines conditions. Nous regardons à présent dans quelle mesure les contraintes produites sont convenables pour la résolution.

Lemme 2 Si TC donne des schémas de types dont le type convient en position négative alors les contraintes produites par le typage sont convenables pour la résolution.

Preuve
- Seules les règles *(const)*, *(trans)* et *(trans_all)* sont susceptibles de produire des types intersections, unions et conditionnels.
- Les types issus de TC dans une preuve de typage apparaissent en position négative dans les contraintes de la règle *(const)*. Ceci explique que les types renvoyés par TC doivent convenir en position négative.
- Les règles *(trans)* et *(trans_all)* produisent des types intersections dans les contraintes *via* les fonction *Comp* et γ. Or dans les intersections produites par *Comp* et γ, l'un des deux types est toujours un type de base ou bien un type de la forme $[1]r$. Les types atomiques et les types de la forme $[1]r$ sont clos et fermés par le haut. Donc ces intersections conviennent à gauche.
- Les règles *(trans)* et *(trans_all)* ne produisent pas d'unions.
- Les règles *(trans)* et *(trans_all)* produisent des types conditionnels en position négative, ce qui est convenable.

□

La restriction sur TC n'empêche pas de typer les opérateurs usuels des langages fonctionnels, on peut avoir par exemple le type $bool \to \alpha \to \beta \to (\alpha \cup \beta)$ pour l'opérateur *if*, en revanche on ne peut coder la surcharge avec une intersection : on ne peut avoir le type $(int \to int \to int) \cap (float \to float \to float)$ pour l'opérateur $+$.

Les contraintes produites par l'inférence étant convenables on peut appliquer la procédure de résolution.

VII.6.3 Exploitation des résultats de la résolution

Nous avons vu que la procédure de résolution permet de trouver toutes les solutions d'un système de contraintes. Or notre but étant l'implantation du programme, nous recherchons une seule solution. Nous étudions ici quelle solution est la plus adaptée à nos besoins.

Notre but ici est de récupérer des informations pour savoir comment implémenter efficacement un programme. Mais on pourrait avoir un autre but : donner le plus d'informations possible au programmeur. Par exemple, pour $\lambda x.1$, le type $\alpha \to int$ est plus intéressant pour le programmeur que $0 \to int$ (on peut donner ce type si la fonction n'est pas utilisée dans le programme typé). On peut aussi considérer le type $1 \to int$ (ici c'est équivalent à $\alpha \to int$ en théorie). Nous évoquons ici cette question puis nous discuterons ensuite comment trouver la solution la plus précise parmi les systèmes inductifs générés par la réduction des contraintes.

Choix des β_i

Une fois que le système choisi est en forme contractive, il reste à instancier les β_i. Instancier tous les β_i à 0 permet d'avoir avoir le type le plus précis. Dans le but de réaliser une implémentation efficace du programme, cette solution convient. Toutefois le typage est aussi une aide au programmeur. Dans ce cadre, la plus petite solution n'est pas toujours la plus intéressante pour lui.

Dans l'optique de rendre l'information lisible pour le programmeur, on choisira la borne supérieure d'une variable en position contravariante ($\beta_i = 1$) et la borne inférieure d'une variable en position covariante ($\beta_i = 0$) [AWP97, Pot98a]. Toutefois, nous choisirons toujours la borne inférieure dans le but d'avoir une information efficace. Notons que ce choix peut être remis en question si l'on souhaite étendre notre système à des possibilités de compilation séparée. En effet, ici, la définition et l'usage de fonctions sont dépendants.

Exemple. L'inférence sur le programme $\lambda x.x + 1$ produit le schéma de type $\forall \alpha_1, \alpha_2 / \{\alpha_1 \subseteq int, int \subseteq \alpha_2\}.\alpha_1 \to \alpha_2$ (dérivation la plus générale, type le plus général).

Si on prend $\beta_1 = \beta_2 = 0$ la solution est $0 \to int$. Ceci est correct car la fonction n'est jamais appliquée (ici le programme est réduit à cette fonction). Si on prend $\beta_1 = \beta_2 = 1$ on aura $int \to 1$ ce qui est également correct mais manque de précision. Si on prend $\beta_1 = 1$ et $\beta_2 = 0$ la solution est $int \to int$, ce qui correspond au type *intuitif* pour le programmeur.

Toutefois, on peut considérer aussi que le type $0 \to int$ apporte une information au programmeur : sa fonction est mal définie ou jamais utilisée.

Choix du système

La règle de réduction de contraintes (11) pour les types conditionnels multiplie les systèmes à résoudre. En effet un système $S \cup \{L_1?L_2 \subseteq R\}$ a une solution si et seulement si

- le système $S \cup L_2 \subseteq 0$ a une solution ou bien
- le système $S \cup L_1 \subseteq R$ a une solution.

Lorsque le premier de ces deux systèmes a une solution, cela signifie qu'une règle ne sera jamais appliquée. Lorsque le second a une solution cela signifie que le programme est bien typé que la règle puisse s'appliquer ou non. Si ces deux systèmes ont une solution, la solution du premier système porte donc plus d'information.

On peut ainsi établir un ordre partiel de « précision » sur les systèmes produits. La solution que nous utiliserons pour l'implémentation d'un programme sera choisie dans le système le plus précis. Notre expérience pratique montre qu'il existe toujours un système plus précis que les autres sans que nous n'ayons d'arguments formels à apporter (ce système reflète fidèlement le comportement du programme).

VII.6.4 Exemple complet

Regardons sur un exemple comment se déroule l'inférence automatique d'un programme. Considérons l'expression `{ x:int => [true] } (["a"])`. La première étape est la dérivation du type le plus général. Le résultat (légèrement simplifié) est le type suivant :

$$\forall \alpha_1, \alpha_2, \alpha_3, \theta / \alpha_1 \subseteq \alpha_2,\ \mathit{bool}?\alpha_1 \cap \mathit{int} \subseteq \alpha_2,\ \mathit{string} \subseteq \alpha_3,\ [\alpha_3]\mathit{seq} \subseteq [\alpha_1]\theta\}.[\alpha_2]\theta$$

Les variables α_1, α_2 et θ proviennent de la dérivation du type de la transformation : $[\alpha_1]\theta \to [\alpha_2]\theta$. De même pour les deux premières contraintes. La variable α_3 provient de la séquence `["a"]` : $[\alpha_3]\mathit{seq}$. De même pour la contrainte $\mathit{string} \subseteq \alpha_3$. Enfin la contrainte $[\alpha_3]\mathit{seq} \subseteq [\alpha_1]\theta$ provient de l'application de la transformation à la séquence.

Ensuite nous devons mettre le système de contraintes en forme inductive. Seules deux contraintes ne sont pas en forme inductive : $\mathit{bool}?\alpha_1 \cap \mathit{int} \subseteq \alpha_2$ et $[\alpha_3]\mathit{seq} \subseteq [\alpha_1]\theta$. La seconde se décompose en $\alpha_3 \subseteq \alpha_1$ et $\mathit{seq} = \theta$. La première peut se décomposer de deux manières. Nous obtenons donc deux systèmes, l'un avec la contrainte $\alpha_1 \cap \mathit{int} \subseteq 0$ (système 1, le plus précis) et l'autre avec la contrainte $\mathit{bool} \subseteq \alpha_2$ (système 2).

Le système 2 est en forme inductive (toutes ses contraintes sont inductives). Le système 1 le devient en réduisant la contrainte $\alpha_1 \cap \mathit{int} \subseteq 0$ en $\alpha_1 \subseteq \neg \mathit{int}$. Les deux systèmes ont donc une solution (en toute rigueur il faudrait avoir saturé les systèmes pour détecter des inconsistances).

$$(1) \qquad \{\alpha_1 \subseteq \alpha_2,\ \alpha_1 \subseteq \neg \mathit{int},\ \mathit{string} \subseteq \alpha_3,\ \alpha_3 \subseteq \alpha_1,\ \mathit{seq} = \theta\}$$

$$(2) \qquad \{\alpha_1 \subseteq \alpha_2,\ \mathit{bool} \subseteq \alpha_2,\ \mathit{string} \subseteq \alpha_3,\ \alpha_3 \subseteq \alpha_1,\ \mathit{seq} = \theta\}$$

VII.7 Extensions immédiates

Nous avons décrit le système de types au complet (types, relation de typage, inférence automatique). À présent nous présentons des extensions directes de nos travaux. Les constructions usuelles telles que le produit ou le *let-rec* n'ont pas été mentionnées mais leur ajout au langage et au typage se fait de manière habituelle.

VII.7.1 Gardes dans le motif.

Dans les pages précédentes nous avons restreint l'usage de la garde dans les motifs afin de rendre la présentation claire. Nous donnons à présent la règle de typage pour les transformations dont les motifs ont des gardes au niveau de chaque motif élémentaire.

Mettre des gardes au plus tôt dans le motif permet d'optimiser le processus de filtrage. Par exemple le filtrage du motif $(x/(x>0)), y$ est plus rapide que pour $x, y/x>0$.

Cette règle ne pose aucune difficulté théorique mais est assez lourde à lire à cause du jeu sur les indices et exposants des motifs élémentaire. En indice apparaît le "numéro" de la règle dans laquelle le motif élémentaire apparaît et en exposant, sa position dans le motif. Le seul point technique à prendre en compte est qu'une garde ne peut faire référence à un identificateur n'étant pas encore lié dans le motif. Par exemple dans le motif $(x/(x > y), y)$ l'identificateur y dans la garde du x ne peut faire référence au y apparaissant après.

$$\frac{\begin{array}{c}\Gamma \cup \gamma_\tau(\mu_i^1, \dots \mu_i^j), S \vdash e_i^j : bool \qquad (j \le i \le n) \\ \Gamma \cup \gamma_\tau(\mu_i^1, \dots \mu_i^{m_i}), S \vdash e_i : [\tau_i]seq \qquad (i \le n)\end{array}}{\Gamma, S \cup \{\tau \subseteq \tau'\} \cup S' \vdash \left\{ \begin{array}{c} \mu_1^1/e_1^1, \mu_1^2/e_1^2, \dots, \mu_1^{m_1}/e_1^{m_1} \Rightarrow e_1 \\ \dots \\ \mu_n^1/e_n^1, \mu_n^2/e_n^2, \dots, \mu_n^{m_n}/e_n^{m_n} \Rightarrow e_n \end{array} \right\} : [\tau]\rho \to [\tau']\rho} \; (trans)$$

où S' est $\bigcup_{1 \le i \le n} \{\tau_i ? Comp_\tau(p_i) \subseteq \tau'\}$.

Notons que si l'on considère des motifs imbriqués et la possibilité d'exprimer la répétition d'un sous-motif arbitraire (comme dans $(x, y/x < y)*$ as X) on sera amené à complexifier encore cette règle.

Afin de capturer finement les propriétés des motifs, on peut développer un système de types qui leur est propre. Un tel système a été proposé sur un sous-ensemble réduit de motifs et pour certaines topologies particulières (les monoïdes : ensembles, multi-ensembles et séquences) [Spi03].

VII.7.2 Simplification des types.

Les schémas de types générés par notre inférence peuvent contenir un nombre élevé de contraintes. On peut vouloir chercher à simplifier ces contraintes pour plusieurs raisons :

Améliorer la lisibilité : Le type $\forall \alpha, \beta, \theta / \{\alpha \subseteq \beta,\ int \subseteq \beta\}.[\alpha]\theta \to [\beta]\theta$ se simplifie en $\forall \alpha / \emptyset.[\alpha]\theta \to [int \cup \alpha]\theta$. Cette dernière présentation du type est bien plus lisible que la première.

Améliorer les performances de l'algorithme de résolution : Le système $\{\alpha \subseteq \beta; \beta \subseteq \tau\}$ se simplifie en $\{\alpha \subseteq \tau\}$ si β n'est pas une variable d'intérêt (élimination des variables inutiles). Ici, la réduction du nombre de variables entraîne la réduction du nombre de contraintes, ce qui se traduit par une diminution du temps de traitement.

Réduire l'espace des solutions : Dans l'espace des solutions aux contraintes, certaines solutions peuvent porter strictement moins d'informations que d'autres. Appelons ces solutions des solutions *inutiles*. On peut remplacer l'ensemble de contraintes par des contraintes ayant un espace de solution identique au premier, aux solutions inutiles près.

De nombreux travaux proposent des résultats sur la simplification de contraintes pouvant être utilisés dans notre système (voir [Pot01] pour une liste de travaux existants).

VII.7.3 Collections de dimension supérieure.

Les collections que nous avons traitées dans cet article peuvent être vue comme le cas particulier de dimension 1 des collections topologiques de dimension quelconque ([SMG04]). En effet, le langage MGS permet la manipulation de collection portant des valeurs sur les cellules d'une partition cellulaire d'un espace topologique. Les complexes simpliciaux ou les G-cartes sont des exemples de collections topologiques à n-dimensions. Les graphes avec des valeurs sur les arêtes sont des complexes simpliciaux de dimension 1. Un complexe simplicial de dimension 2 comporte des sommets, des arêtes et des faces triangulaires. Notre système de types se prête bien à cette extension puisque par exemple le type des collections en dimension 2 peut se décrire sous la forme $[\tau_0][\tau_1]\rho$ où τ_0 est le type des éléments sur les sommets et τ_1 est le type des éléments sur les arêtes.

VII.8 Discussions

Dans cette section, nous discutons certains choix effectués dans la conception de notre système de types.

VII.8.1 Tests de type au filtrage.

Dans notre implémentation du langage, les valeurs d'un type hétérogène (contenant une union) sont étiquetées d'une information sur leur nature. Par exemple les entiers d'une collection de type contenu $int \cup bool$ sont étiquetés par `Int`. Les collections peuvent être étiquetées de leur topologie et les fonctions par un `Fun` qui indique juste que ce sont des fonctions[7]. Ceci nous amène à envisager plusieurs sortes de tests dynamiques de type :

- Ceux qui ne demandent qu'une lecture de l'étiquette.
- Ceux qui demandent un parcours des éléments d'une collection. Par exemple on peut vouloir tester si une collection a le type $[int]seq$ lorsque son type inféré est $[int \cup bool]seq$.
- Ceux qui demandent une information relevant d'un calcul plus complexe, comme savoir si une fonction a le type $int \rightarrow int$.

Les lectures d'étiquettes ne posent pas de problème et les deux formes de tests de type possibles dans nos motifs relèvent de cette catégorie. Ajouter un test de type pour filtrer les fonctions par lecture d'étiquette ne pose pas de problème technique mais offre peu d'intérêt puisqu'on ne pourrait garantir pour celles-ci que le type $0 \rightarrow 1$.

Nous avons écarté les tests de la seconde catégorie. En effet, le temps de calcul induit peut être inattendu pour le programmeur. De plus le programmeur peut coder ce test dans le langage étudié (voir page 126).

Les tests de la dernière catégorie posent des problèmes théoriques. En effet, à moins de marquer chaque fonction par l'ensemble de ses types possibles par un moyen auxiliaire (annotation du programmeur par exemple) la sémantique du langage dépendrait d'une inférence de type, elle-même fondée sur cette sémantique. Cette question technique est abordée dans [FCB02].

VII.8.2 Types sommes ou types unions ?

Les types sommes et les types sommes polymorphes ([Gar98]) permettent un style de programmation assez proche de celui qu'on obtient avec des types unions. Toutefois, s'il existe des cas où l'étiquetage des valeurs par un constructeur est un avantage pour le programmeur, il en existe où cet étiquetage est pesant. Dans le type `I of int | B of bool` par exemple, les constructeurs semblent superflus puisqu'il n'y a pas nécessité de distinguer deux sens donnés à un même type (contrairement au type `I1 of int | I2 of int`). Nous avons rencontré un exemple similaire pour représenter les valeurs MGS dans le chapitre V sur l'implémentation de l'interprète. Par ailleurs, dans un contexte où des types complexes comme des enregistrements sont utilisés, il est rarement nécessaire d'apporter une information supplémentaire à un type pour indiquer son interprétation. Les nombreux programmes MGS existants (voir le chapitre III) ont montré qu'un test de type est aussi expressif qu'un filtrage sur un constructeur dans ce cas, tout en étant moins lourd. Nous pensons donc qu'il est bénéfique pour le programmeur d'avoir le choix d'utiliser des constructeurs ou non.

[7] Ceci est similaire à l'implémentation des valeurs proposée au chapitre V mais n'est mis en œuvre que pour les types unions.

VII.8.3 Polytypisme.

Le polytypisme, ou programmation générique, consiste à pouvoir écrire des programmes opérant indifféremment sur diverses structures de données partageant une propriété commune. L'approche classique du polytypisme repose sur le filtrage des structures algébriques en utilisant leurs constructeurs ([Jan00, Jay04, Jeu00, HJ03] par exemple). Notre approche est différente puisqu'elle repose sur la présence d'une relation de voisinage dans toute collection.

Nous avons donné des exemples de programmes polytypiques en section III.7, nous avons vu les notions existantes de polytypisme et le fonctionnement du filtrage uniforme en section IV.1, revenons à présent sur la manière dont nous avons abordé le polytypisme dans notre système de types.

Nous avons rendu compte du polytypisme à la manière du polymorphisme paramétrique, *i.e.* on a une variable de topologie qui se comporte comme une variable de type dans un système *à la* Hindley/Milner. D'ailleurs il s'agit de let-polytypisme.

Par ailleurs, on peut envisager d'avoir du polytypisme borné (on parle usuellement de polymorphisme borné lorsque les variables de types sont accompagnées de contraintes) pour désigner les transformations s'appliquant à toutes les topologies sur lesquelles une direction donnée (*nord* par exemple) a un sens. Il faut pour cela faire l'hypothèse que les directions des topologies ne sont pas disjointes.

Il est également nécessaire d'envisager du polytypisme borné pour traiter les topologies union, au sens donné dans la discussion 4, page 131.

VII.8.4 Typage mixte statique/dynamique

Le système d'inférence de types que nous proposons peut être qualifié de statique car il associe un type à chaque expression du langage avant l'exécution. Toutefois les types union peuvent être vus comme une manière d'intégrer du typage dynamique à un système statique. En effet lors de l'exécution une valeur d'un type union sera accompagnée d'une étiquette indiquant comment elle est implémentée, comme dans les langages dynamiquement typés.

Cette mixité est également présente dans d'autres travaux. Par exemple, les types dynamiques sont directement accessibles au programmeur dans certaines versions de Caml (CAML [LM94], G'CAML [Fur02]). Le *soft typing* [CF91, ALW94] exploite également cette mixité : des types dynamiques ou des types union sont utilisés là où le système d'inférence n'a pas réussi à garantir l'absence d'erreurs de type dans le programme. Par exemple le programme `(if true then 1 else false) + 1` sera rejeté par un système statique mais accepté dans un système à typage souple. (Le typage souple ne rejette aucun programme.)

Si l'on veut avoir un système de types complètement statique (sans informations de types gérées à l'exécution) on ne pourra pas traiter les collections hétérogènes. Par exemple l'expression `oneof(1::true::set:())` n'a ni le type *int* ni le type *bool* car l'opérateur de construction `::` est commutatif sur les ensembles. Nous proposons dans le chapitre VIII un système complètement statique pour notre langage restreint aux collections homogènes.

Une autre manière d'avoir un typage complètement statique serait d'utiliser des types sommes (voir la section VII.8.2 à ce sujet)

VII.9 Travaux apparentés

Les travaux [AW93, ALW94] fournissent une méthode d'inférence de types en présence de types union qui sert de fondement à notre système de types. En effet, les types union sont la clé de notre représentation de l'hétérogénéité. Notre travail se démarque fortement du leur par le point de vue original sur le typage des structures de données et la puissance du processus de filtrage que nous typons. Les travaux [FCB02] proposent également un système de types fondé sur un modèle ensembliste pour un langage avec filtrage sur le type des valeurs où l'hétérogénéité des données est représentée par des types union. Toutefois la puissance de leurs motifs les contraint à imposer une déclaration du type des fonctions par le programmeur.

D'autres systèmes de types permettent de vérifier des propriétés plus fortes dans des langages à base de règles. Par exemple Structured Gamma [FLM98] munit le langage Gamma d'un système de types dédié à la vérification de l'intégrité de certaines structures de données.

De nombreux langages proposent un filtrage plus expressif que celui de ML. Citons TOM [MRV03] pour le filtrage associatif, ELAN [MK98], MAUDE [CDE+03] et CiME [CM96] pour le filtrage associatif et commutatif, CDUCE [FCB02] et G'CAML [Fur02] pour le filtrage sur le type et Generic Haskell [HJ03] pour le filtrage polytypique. Ces trois derniers langages sont munis d'un système de type original. Le langage que nous avons typé ici propose toutes ces caractéristiques, ainsi que du filtrage sur des structures non algébriques.

Le système de types proposé ici ainsi que le système de [Coh03c] ont été intégrés à un compilateur MGS. Ceci a permis d'éliminer l'étiquetage des valeurs par leur type lorsque celui-ci est inutile et a apporté de grandes améliorations des performances. Ceci nous incite à intégrer d'autres optimisations fondées sur les types telles que l'élimination des règles inutiles (voir section IX.3) et à affiner les règles de typage des transformations pour obtenir des types plus précis (voir ci-dessous).

VII.10 Perspectives

Nous décrivons ici des perspectives à court terme relatives au système de types présenté.

VII.10.1 Des types plus précis.

Analyse des motifs

Nous avons vu que notre système permettait d'inférer des types assez précis. Nous donnons ici trois cas où des types plus précis existent.

- Le type inféré pour la transformation $\{x\!:\!int \Rightarrow [true]\}$ est $[\alpha]\theta \rightarrow [\alpha \cup (bool?(\alpha \cap int))]\theta$. Ce type ne porte pas l'information que la collection renvoyée ne contient plus d'entiers.
- Le type inféré pour la transformation $\{x : int \Rightarrow [x]; x : int \Rightarrow [true]\}$ est $[\alpha]\theta \rightarrow [\alpha \cup (bool?(\alpha \cap int))]\theta$. Or la deuxième règle ne s'applique jamais car tous les entiers sont consommés par la première donc le type $[\alpha]\theta \rightarrow [\alpha]\theta$ qui est plus précis convient.
- Le type inféré pour la transformation $\{x\!:\ int \Rightarrow [x{+}1]; x \Rightarrow [x{+}.1.0]\}$ est $[float]\theta \rightarrow [float]\theta$ à cause de l'opération flottante de la seconde règle. Pourtant la collection peut contenir des entiers sans provoquer d'erreur. Le type le plus précis serait $[\alpha]\theta \rightarrow [\alpha]\theta$ avec la contrainte $\alpha \subseteq int \cup float$.

Ces trois exemples montrent qu'une analyse plus fine des règles permettrait de gagner en précision dans les types inférés. Ces améliorations nécessitent d'intégrer le complément ensembliste (ou une forme approchée) au système de types. Ceci se fait simplement puisque nous disposons déjà de l'opération de complément sur les types fermés par le haut. Les tests de types dans les motifs étant limités à des types fermés par le haut (types atomiques, tests de topologies pour les collections et éventuellement tests pour savoir si une valeur est une fonction), des compléments de types non fermés par le haut ne semblent pas nécessaires pour réaliser les analyses décrites ci-dessus.

Limitations du typage ensembliste

Par ailleurs l'utilisation des types union est source d'une autre perte d'information, indépendante du typage des collections et des transformations. L'exemple suivant met en évidence ce problème. Considérons une fonction *map* du type $(\alpha \to \beta) \to [\alpha]\theta \to [\beta]\theta$. Appliquons-la à une fonction f du type $\alpha \to \alpha$ et à une collection c du type $[int]seq \cup [bool]seq$. La collection étant une séquence soit d'entiers, soit de booléens, nous savons que le résultat sera une séquence contenant soit des entiers, soit des booléens. Or notre système de types saura uniquement indiquer que le résultat a le type $[int \cup bool]seq$, ce qui porte moins d'information que le type $[int]seq \cup [bool]seq$. Le problème est dû au type que nous donnons à *map*. Celui-ci indique que le deuxième argument de *map* doit avoir un type de la forme $[\theta]\alpha$. Or le plus petit type de c ayant cette forme est $[int \cup bool]seq$. C'est ce type qui sera utilisé pour calculer le type renvoyé, d'où la perte d'information. Un exemple similaire peut être donné pour le système de [AW93].

Notons que ceci est plus gênant si on considère une collection c du type $[int]seq \cup [bool]bag$. En effet, dans notre système ce type ne peut s'écrire sous la forme $[\alpha]\theta$. Pour pallier à ce problème il suffit d'utiliser des topologies union comme on utilise des types union. La collections c pourrait alors avoir le type $[int \cup bool](seq \cup bag)$.

Cette fuite d'informations est une question que nous n'avons pas résolue pour le moment.

VII.10.2 Erreurs de structure newtonienne.

Notre système de types garantit l'absence d'erreurs de types au sens où
- aucune valeur n'étant pas une fonction ne peut être appliquée comme une fonction
- et aucune fonction ou opérateur attendant des valeurs d'un type particulier ne sera appliquée à une valeur d'un type ne convenant pas.

Toutes les erreurs d'un programme ne sont pas des erreurs de type. Par exemple, les divisions par 0 ne peuvent généralement pas être détectées par les systèmes de types utilisés. Dans MGS nous rencontrons une erreur qui n'est pas à proprement parler une erreur de type mais qui semble détectable par des méthodes de typage. Il s'agit des erreurs de structure newtonienne (**shape**) survenant lorsqu'une transformation tente de modifier l'organisation des éléments d'une collection à structure rigide. Un système fondé sur le système que nous proposons ici et permettant de détecter les erreurs de structure est proposé dans le chapitre suivant.

VII.10.3 Topologies complexes.

Dans ce chapitre nous avons considéré les topologies comme des symboles. Si on considère une topologie comme la donnée d'un ensemble d'opérateurs (les constructeurs et la substitution topologique) comme au chapitre IV, celle-ci induit des propriétés sur les collections produites.

Par exemple, une grille est une collection dont chaque élément a au plus quatre voisins, et dont les direction commutent et sont inverses deux à deux ($nord\ x\ y \Leftrightarrow sud\ y\ x$). Une topologie peut donc également être considérée du point de vue des propriétés. Nous proposons au chapitre suivant un moyen de garantir dans le système de types qu'une structure newtonienne est préservée par une transformation, *i.e.* que le motif et la séquence remplaçante des règles de la transformation ont toujours la même taille.

On peut imaginer des propriétés complexes sur les topologies où cette vérification n'est pas suffisante. Prenons comme exemple un damier *int-bool*, soit une grille dont les éléments booléens n'ont que des entiers comme voisins et *vice-versa*. L'analyse ici doit être plus fine. Elle ne peut être réalisée par notre système de types mais cette sorte d'analyse peut être traitée par un système de types, voir par exemple le système de [Mon94] pour cette propriété particulière.

La variété des collections topologiques/structures de données et des propriétés dont on peut avoir besoin de les équiper garantit une richesse dans les travaux potentiels sur le typage des collections et des transformations.

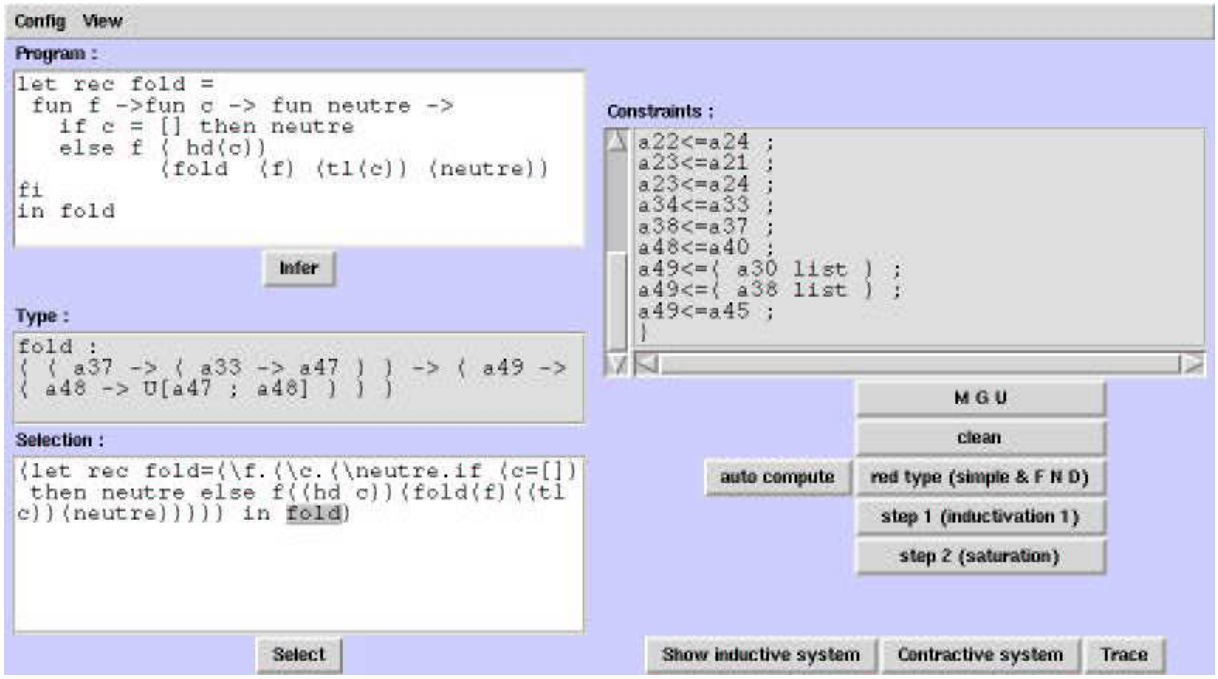


FIG. 6 – Version visuelle du module d'inférence de types.

Chapitre VIII

Variantes sur le système de types

Les travaux présentés dans la première section de ce chapitre ont été publiés dans [Coh03c], et [Coh04].

Nous avons vu dans le chapitre précédent comment donner un système de types puissant au langage MGS. Nous nous tournons à présent vers deux évolutions possibles de ce système. La première évolution est orientée vers l'optimisation : optimisation du code produit mais aussi simplification et accélération de l'inférence de types. Cette évolution est fondée sur la remarque que si on se limite aux collections homogènes (qui contiennent des valeurs de même nature) on peut :

- éliminer toutes les informations de types à l'exécution, le typage est alors complètement statique ;
- abandonner la relation de sous-typage fondée sur l'inclusion ensembliste, on n'utilisera plus de types union.

Ceci nous amènera également à proposer un algorithme d'inférence de types *à la* Damas/Milner. Ce système de types *simple* (sans sous-typage) est présenté en section VIII.1.

La seconde évolution est orientée vers la vérification. Le système de types du chapitre précédent ne permet pas de vérifier l'absence d'*erreurs de structure* à l'exécution. Nous proposons d'aborder ce problème en ajoutant une information sur la taille des collections dans leur type. Cette modification du type collection est faite de manière à modifier le moins possible notre système de types afin de pouvoir utiliser l'algorithme d'inférence déjà établi. Ce système est présenté en section VIII.2.

VIII.1 Typage simple

Par typage simple nous entendons un typage sans relation de sous-typage. Le typage simple a de nombreux avantages : la mise en œuvre est simple, bien connue et efficace. Il est utilisé dans de nombreux langages (dans OCaml par exemple même s'il existe des moyens d'avoir du sous-typage, notamment sur les objets). Enfin, il permet de produire du code dénué d'informations de types et donc plus efficace que du code dynamiquement typé.

Intégrer les collections topologiques et les transformations dans un système de type simple permet également de montrer qu'elles peuvent s'intégrer dans un langage préexistant, qu'il soit muni de sous-typage ou non.

Nous commençons dans cette section par regarder l'influence du typage simple sur le langage à considérer. Ensuite nous montrons comment on peut se ramener à un typage simple à partir du système présenté dans le chapitre VII. Enfin, nous détaillons un système plus proche du style Damas/Milner puis nous revenons sur les caractéristiques de l'approche simplement typée.

VIII.1.1 Collections homogènes

Comme discuté en section VII.8.4, nous nous restreignons aux collections homogènes pour pouvoir développer un typage simple. Simple signifie sans sous-typage et donc sans types union. Cette absence de types union nous mènera à ne pas pouvoir accepter d'expressions comme `1::true::seq:()` (collection hétérogène) ou comme `if p then 1 else false`.

Les tests de types deviennent également inutiles dans le langage (ou du moins ils ne peuvent avoir la même utilité qu'avec des collections hétérogènes). Nous les laisserons donc de coté.

L'homogénéité des collections doit être garantie par le système de types si l'on veut garantir l'absence d'erreurs de type à l'exécution. Ceci impose de vérifier que les transformations préservent bien l'homogénéité des collections. Par exemple la transformation { `x/x=0 => [true]` } renvoie une collection hétérogène dès que la collection à laquelle elle est appliquée contient des entiers nuls et des entiers non nuls. Elle doit donc être rejetée.

Il y a deux façons de garantir qu'une transformation préserve l'hétérogénéité :
- Soit la transformation ne change pas le type des éléments contenus dans la collection à laquelle elle est appliquée. La transformation { `x/x=0 => [-1]` } est dans ce cas.
- Soit elle filtre avec succès tous les éléments de la collection et les remplace par des éléments d'un même type. La transformation { `x/x=0 => [true] ; x => [false]` } est dans ce cas.

Pour vérifier ce dernier point, nous nous limiterons à vérifier que la transformation a une règle *attrappe-tout*, c'est à dire une règle dont le motif est de la forme x (ou bien _) et qui filtre donc tout élément.

VIII.1.2 Variation sur le typage ensembliste

Le système de types du chapitre VII peut être transformé en système simple en remplaçant les contraintes de sous-typage par des égalités dans les règles de typage. Dans les règles *(trans)* et *(trans_all)* cette modification garantit que le type des éléments remplaçants est le même pour toutes les règles. Dans la règle *(trans)* elle garantit en outre que le type contenu de la collection argument est le même que celui de la collection renvoyée.

L'inférence automatique peut se faire en résolvant par unification les égalités produites. Ceci est plus efficace que la méthode de réduction de contraintes de sous-typage (en théorie ce problème est DEXPTIME-complet [KM89], donc meilleur que la résolution de contraintes ensemblistes mais est encore difficile, en pratique, les cas coûteux ne sont pas rencontrés comme en témoigne la grande satisfaction des utilisateurs de l'inférence *à la* Damas/Milner). L'unification ne pose ici aucun problème puisque les types sont des termes simples (*i.e.* ni associatifs, ni commutatifs, *etc.*).

VIII.1.3 Variation sur le système de Damas/Milner

Nous proposons de reformuler ce système dans un style *à la* Damas/Milner sans contraintes en partie gauche du symbole $\vdash$ et avec un algorithme fondé sur W [DM82]. Cet algorithme

produit les contraintes et les résout en une seule passe.

Règles de typage

Les règles de typage sont données dans la figure 1. Comme usuellement la fonction *Gen*

$$\frac{\Gamma(x) \leq \tau}{\Gamma \vdash x : \tau}\,(var - inst) \qquad \frac{TC(c) \leq \tau}{\Gamma \vdash c : \tau}\,(const - inst)$$

$$\frac{\Gamma \cup \{x : \tau_1\} \vdash e : \tau_2}{\Gamma \vdash (\lambda x.e) : \tau_1 \to \tau_2}\,(fun) \qquad \frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}\,(app)$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \cup \{x : Gen(\tau_1, \Gamma)\} \vdash e_2 : \tau_2}{\Gamma \vdash (let\ x = e_1\ in\ e_2) : \tau_2}\,(let)$$

$$\frac{\Gamma_i \vdash e_i : [\tau]seq \quad \Gamma_i \vdash g_i : bool \quad (1 \leq i \leq n)}{\Gamma \vdash \{m_1/g_1 \Rightarrow e_1; \ldots; m_n/g_n \Rightarrow e_n\} : [\tau]\rho \to [\tau]\rho}(trans)$$

La règle ci-dessous ne peut être utilisée que si g_n est `true` :

$$\frac{\Gamma_i \vdash e_i : [\tau']seq \quad \Gamma_i \vdash g_i : bool \quad (1 \leq i \leq n)}{\Gamma \vdash \{m_1/g_1 \Rightarrow e_1; \ldots; m_n/g_n \Rightarrow e_n\} : [\tau]\rho \to [\tau']\rho}(trans\_all)$$

où $\Gamma_i = \Gamma \cup \gamma_\tau(m_i)$ et avec γ définie par :

$$\begin{array}{ll} \gamma_\tau(x) & = \{x : \tau\} \\ \gamma_\tau(* \text{ as } x) & = \{x : [\tau]seq\} \\ \gamma_\tau(\mu, m) & = \gamma_\tau(\mu) \cup \gamma_\tau(m) \end{array}$$

FIG. 1 – Règles de typage simple.

généralise un type τ en schéma de type

$$\forall \alpha_1, \ldots, \alpha_n, \theta_1, \ldots, \theta_m.\tau$$

où les variables de type et de topologie quantifiées sont les variables du type qui ne sont pas liées dans le contexte de typage.

Inférence automatique

La figure 2 donne l'algorithme W d'inférence automatique basé sur l'algorithme de Damas/Milner. Dans cet algorithme, φ dénote la substitution courante (dans un style impératif). Les fonctions $fresh_t$ et $fresh_r$ donnent des variables de type et des variables de topologie fraîches. Si l'algorithme échoue le programme est mal typé et sinon il calcule le type *principal* au sens de [DM82] du programme considéré.

$W(\Gamma \vdash e) =$ (* cas originaux - Damas/Milner *)
if $e = x$
  let $\forall \alpha_1, \ldots, \alpha_n, \theta_1, \ldots, \theta_m . \tau = \Gamma(x)$
  let $\alpha'_1, \ldots, \alpha'_n = \mathit{fresh}_t, \ldots, \mathit{fresh}_t$
  let $\theta'_1, \ldots, \theta'_m = \mathit{fresh}_r, \ldots, \mathit{fresh}_r$
  return $\tau[\alpha_1 \leftarrow \alpha'_1, \ldots, \alpha_n \leftarrow \alpha'_n, \theta_1 \leftarrow \theta'_1, \ldots, \theta_m \leftarrow \theta'_m]$

if $e = \mathit{fun}\ x \rightarrow e$
  let $\alpha = \mathit{fresh}_t$
  let $\tau = W(\Gamma \cup x : \forall . \alpha \vdash e)$
  return $\alpha \rightarrow \tau$

if $e = e_1\ e_2$
  let $\tau_1 = W(\Gamma \vdash e_1)$
  let $\tau_2 = W(\Gamma \vdash e_2)$
  let $\alpha = \mathit{fresh}_t$
  do $\varphi \leftarrow \mathit{mgu}(\varphi(\tau_1) = \varphi(\tau_2 \rightarrow \alpha)) \circ \varphi$

if $e = \mathit{let}\ x = e_1\ \mathit{in}\ e_2$
  let $\tau_1 = W(\Gamma \vdash e_1)$
  let $\sigma = \mathit{Gen}(\varphi(\tau_1), \varphi(\Gamma))$
  return $W(\Gamma \cup \{x : \sigma\} \vdash e_2)$

---

if $e = \{p_1/g_1 \Rightarrow e_1;\ \ldots;\ p_n/\mathtt{true} \Rightarrow e_n\}$ (* cas supplémentaires pour *)
  let $\alpha, \beta = \mathit{fresh}_t, \mathit{fresh}_t$ (* traiter les transformations *)
  let $\theta = \mathit{fresh}_r$
  for $i = 1..(n-1)$
    let $\tau_i = W\big(\Gamma \cup \gamma_\alpha(p_i) \vdash g_i\big)$
    do $\varphi \leftarrow \mathit{mgu}\big(\{\varphi(\tau_i) = \mathit{bool}\}\big) \circ \varphi$
    let $\tau'_i = W\big(\Gamma \cup \gamma_\alpha(p_i) \vdash e_i\big)$
    do $\varphi \leftarrow \mathit{mgu}\big(\{\varphi(\tau'_i) = \varphi([\beta]\mathit{seq})\}\big) \circ \varphi$
  let $\tau'_n = W\big(\Gamma \cup \gamma_\alpha(p_n) \vdash e_n\big)$
  do $\varphi \leftarrow \mathit{mgu}\big(\{\varphi(\tau'_n) = \varphi([\beta]\mathit{seq})\}\big) \circ \varphi$
  return $[\alpha]\theta \rightarrow [\beta]\theta$

if $e = \{p_1/g_1 \Rightarrow e_1;\ \ldots;\ p_n/g_n \Rightarrow e_n\}$ $(g_i \neq \mathtt{true})$
  let $\alpha = \mathit{fresh}_t$
  let $\theta = \mathit{fresh}_r$
  for $i = 1..n$
    let $\tau_i = W\big(\Gamma \cup \gamma_\alpha(p_i) \vdash g_i\big)$
    do $\varphi \leftarrow \mathit{mgu}\big(\{\varphi(\tau_i) = \mathit{bool}\}\big) \circ \varphi$
    let $\tau'_i = W\big(\Gamma \cup \gamma_\alpha(p_i) \vdash e_i\big)$
    do $\varphi \leftarrow \mathit{mgu}\big(\{\varphi(\tau'_i) = \varphi([\alpha]\mathit{seq})\}\big) \circ \varphi$
  return $[\alpha]\theta \rightarrow [\alpha]\theta$

FIG. 2 – Algorithme d'inférence automatique W.

VIII.1.4 Résultats

Le typage simple des collections et des transformations a les avantages suivants :
– On dispose de la même forme de polytypisme qu'avec notre système initial.

– Les performances de l'inférence automatique sont bonnes et ce style d'inférence est très répandu dans les langages fonctionnels.
– Aucun test de type n'est nécessaire à l'exécution et le code produit peut donc être non étiqueté. Le code produit est donc très efficace.
– L'hétérogénéité peut toujours être obtenue avec des types sommes (« l' univers » sera donc défini à la main) ou avec des enregistrements extensibles.

Par ailleurs, puisque ce système peut être vu comme une modification du système du chapitre précédent (en remplaçant $\subseteq$ par $=$), on peut utiliser la preuve de correction de ce dernier si l'on souhaite établir une preuve de correction pour le système simple.

Les inconvénients de ce système sont relatifs à l'absence de types union et ont été discutés en section VII.8.2, page 151.

VIII.2 Vérification de l'absence d'erreurs de structure

Dans le chapitre précédent, nous avons vu les topologies comme de simples symboles. Toutefois une topologie induit un ensemble de propriétés. Dans cette section, nous illustrons l'utilisation du système de types pour vérifier des propriétés topologiques en considérant la vérification de la structure des collections newtoniennes.

Il s'agit ici de vérifier qu'une transformation ne renverra pas l'erreur **shape** en tentant de changer la structure d'une collection de topologie newtonienne. Cette vérification se réduit à vérifier que pour chacune des règles de la transformation, le motif et l'expression remplaçante ont *toujours* la même taille. Ceci signifie que *pour tout* chemin filtré avec succès par un motif, le chemin filtré et la séquence de remplacement *correspondante* ont la même longueur.

Avant de nous pencher sur la question nous posons une limite à notre champs d'action : nous devons tenter de ne pas modifier le langage MGS. Ceci signifie notamment que :
– Le programmeur ne doit pas avoir besoin d'annoter son programme pour assister le typeur.
– Nous ne devons pas restreindre significativement l'expressivité des motifs, ni restreindre la syntaxe des expressions de remplacement.

Nous serons ainsi amenés à rencontrer les deux difficultés suivantes :
– On ne peut connaître à l'avance la longueur d'un chemin filtré par un motif à cause de la répétition arbitraire (le symbole $*$).
– L'expression de remplacement est une expression libre du langage (dont la valeur doit être une séquence).

Comme avec la plupart des systèmes de types on peut espérer être correct mais pas complet. Par exemple le programme $\{* \text{ as } x \Rightarrow [1]\}(0 :: \mathit{emptygrid})$ ne provoque pas d'erreur de structure mais nous le considérerons ici comme hors de notre portée. Nous tenterons plus modestement de garantir que le programme $\{* \text{ as } x, y \Rightarrow y :: (\mathit{map}\ f\ x)\}$ ne provoque pas d'erreur (avec f une constante du type $\alpha \to \alpha$ par exemple).

Nous explorons ici une voie tirant parti du caractère ensembliste de notre système de types (un type dénote un ensemble). En effet, nous proposons de dénoter la taille d'une collection par un type dénotant l'ensemble des valeurs possibles de cette taille. Les *types-taille* sont basés sur les constructeurs *Zero* et *Suc* et les opérateurs ensemblistes comme l'union. Ainsi les contraintes produites par le typage d'une expression sont résolues par une variation de l'algorithme de résolution du chapitre précédent.

Il s'agit d'un travail en cours et beaucoup de questions ne sont pas encore résolues.

VIII.2.1 Proposition avec des tailles union

Ici, nous intégrons les tailles dans le système de types du chapitre précédent. Cette approche a l'avantage de s'intégrer naturellement dans le système déjà existant sans mettre en œuvre des techniques additionnelles.

Types collections

Nous enrichissons la grammaire des types en embarquant une *taille* s dans les types collection : $[\tau]\rho(s)$. Les tailles dénotent des ensembles d'entiers et sont de la forme suivante :

$$s \ ::= \ \mathit{Zero} \mid \mathit{Suc}(s) \mid \eta \mid s \cup s \mid s \cap s \mid 0 \mid 1 \mid c \mid s + s$$

où η est une variable de taille, 0 est la taille vide (*i.e.* l'ensemble vide), 1 est l'ensemble de toutes les tailles, et c est une constante appartenant à est un ensemble infini de constantes symboliques. L'opérateur d'addition + est associatif commutatif. On suppose qu'on sait mettre les tailles en forme normale. Cette opération nécessite la réduction de l'addition selon l'addition des entiers de Peano, lorsque possible. Nous reviendrons plus tard sur l'utilité des constantes symboliques.

Par exemple, la taille $\mathit{Suc}(\mathit{Suc}(\mathit{Zero}) \cup \mathit{Zero})$ dénote l'ensemble d'entiers $\{1, 2\}$. Le type $[\mathit{int}]\mathit{seq}(\mathit{Suc}(\mathit{Zero}))$ dénote l'ensemble des séquences d'entiers comportant exactement un élément. Enfin, le type $\forall\eta/\emptyset.[1]\mathit{seq}(\mathit{Suc}(\eta))$ dénote l'ensemble des séquences non-vides.

Taille d'un motif

La fonction *length* calcule la *taille* d'un motif et la fonction γ est modifiée comme suit :

$\mathit{length}(x)$	$= \mathit{Suc}(\mathit{Zero})$	$\gamma_\tau(x)$	$= \{x{:}\tau\}$
$\mathit{length}(x : b)$	$= \mathit{Suc}(\mathit{Zero})$	$\gamma_\tau(x{:}b)$	$= \{x{:}\ \tau \cap b\}$
$\mathit{length}(x : r)$	$= \mathit{Suc}(\mathit{Zero})$	$\gamma_\tau(x{:}r)$	$= \{x{:}\ \tau \cap [1]r(1)\}$
$\mathit{length}(* \text{ as } x)$	$= c_x$ où $c_x \in C$ et est fraîche	$\gamma_\tau(* \text{ as } x)$	$= \{x{:}[\tau]\mathit{seq}(c_x)\}$
$\mathit{length}(\mu, m)$	$= \mathit{length}(\mu) + \mathit{length}(m)$	$\gamma_\tau(\mu, m)$	$= \gamma_\tau(\mu) \cup \gamma_\tau(m)$

On voit que l'on attribue une longueur symbolique c_x au motif $*$ as x. Pourquoi ne pas lui attribuer plutôt une variable de taille ν_x. Pour répondre à cette question, penchons nous sur le rôle des variables.

Pour dire qu'une expression est bien typée on est amené à résoudre un système S de contraintes. Ceci revient à juger s'il existe une instanciation des variables du système telle que les contraintes du système soient vérifiées. Ceci revient donc à juger la formule

$$\exists\chi_1, \ldots, \chi_n.\phi_1 \wedge ... \wedge \phi_m$$

où les χ_i sont les variables de S et les ϕ_i sont ses contraintes.

À présent supposons que je veuille exprimer la contrainte que le motif $(*$ as $x)$ a la même taille que la séquence vide *empty_seq*. Si j'utilise une variable η_x pour la longueur du chemin filtré par le motif, je peux écrire la contrainte suivante : $\eta_x = \mathit{Zero}$. Le processus de résolution va nous amener à juger $\exists\eta_x.(\eta_x = \mathit{Zero})$. Ceci est évidemment vrai. Or ceci ne signifie pas que lors d'une application d'une transformation où ce motif apparaît, il aura une longueur nulle. La formule qu'il nous aurait fallu juger est $\forall\eta_x.(\eta_x = \mathit{Zero})$ qui est fausse. En effet la contrainte qu'un chemin filtré par $(*$ as $x)$ a toujours la même longueur que la séquence vide est insatisfiable.

Cet exemple montre que si on utilise des variables pour dénoter la longueur d'un chemin filtré par une répétition $*$ il faut leur donner une sémantique différente des autres variables : il faut les quantifier universellement. C'est pour cela que nous utilisons des constantes symboliques. En effet, ces symboles n'étant pas des variables, ils ne peuvent être instanciées et ne sont donc pas quantifiées existantiellement. Considérons quelques exemples pour nous convaincre que l'utilisation de ces symboles simule l'utilisation de variables universellement quantifiées.

L'égalité $c_1 = c_2$ est fausse tout comme $\forall \eta_1, \eta_2.(\eta_1 = \eta_2)$ lorsque c_1 et c_1 sont deux constantes symboliques différentes. L'égalité $c = 1$ est fausse tout comme la formule $\forall \eta.(\eta = 1)$. L'égalité $c = \eta$ est fausse comme la formule[1] $\exists \eta.(\forall \eta_0.(\eta_0 = \eta))$.

Typage

Les règles de typage (sauf celles pour les transformations) ne changent pas par rapport à celles du chapitre précédent. Seule la fonction TC doit être modifiée. Par exemple : $TC(\mathit{empty\_seq}) = [0]\mathit{seq}(\mathit{Zero})$ ou indifféremment $TC(\mathit{empty\_seq}) = \forall \alpha/\emptyset.[\alpha]\mathit{seq}(\mathit{Zero})$. Donnons également le type du constructeur générique :

$$TC(::) = \forall \alpha, \theta, \eta_1, \eta_2/\{\mathit{Suc}(\eta_1) \subseteq \eta_2\}.\alpha \to [\alpha]\theta(\eta_1) \to [\alpha]\theta(\eta_2)$$

$$TC(::) = \forall \alpha, \theta, \eta/\emptyset.\alpha \to [\alpha]\theta(\eta) \to [\alpha]\theta(\mathit{Suc}(\eta))$$

Les deux versions sont équivalentes. Notons que nous ne considérons pas dans cette section les constructeurs idempotents comme celui des ensembles. Nous nous focalisons ici sur les collections newtoniennes et également sur les séquences puisqu'elles sont nécessaires en partie droite des règles de transformations.

À présent regardons comment garantir qu'une transformation ne viole pas la topologie d'une collection newtonienne. Si pour chaque règle d'une transformation on peut vérifier que la taille du motif est *égale* à la taille de la séquence de remplacement alors elle préserve les topologie newtonienne. Ceci signifie que les appels à la fonction de substitution topologique $\Psi_{\mathcal{N}}$ par la transformation ne provoqueront pas l'erreur **shape**. L'égalité est jugée en comparant syntaxiquement les formes normales des tailles.

La règle de typage est la suivante :

$$\frac{\Gamma \cup \gamma_\tau(p_i), S \vdash g_i : \mathit{bool} \quad \Gamma \cup \gamma_\tau(p_i), S \vdash e_i : [\tau_i]\mathit{seq}(\eta_i) \quad (1 \leq i \leq n)}{\Gamma, S \cup \{\tau \subseteq \tau'\} \cup S' \cup S'' \vdash \{p_1/g_1 \Rightarrow e_1; \ldots; p_n/g_n \Rightarrow e_n\} : [\tau]\rho(\eta) \to [\tau']\rho(\eta)}\ (\mathit{trans} - \mathcal{N})$$

où S' est $\bigcup_{1 \leq i \leq n} \{\tau_i ? \mathit{Comp}_\tau(p_i) \subseteq \tau'\}$ et S'' est $\bigcup_{1 \leq i \leq n} \{\mathit{length}(p_i) = \eta_i\}$.

Rappelons que pour une règle $*$ as $x/g \Rightarrow e$, les expression e et g sont typées dans le contexte de typage induit par le motif, par conséquent x est considéré avec le type calculé par $\gamma(*$ as $x)$. Ainsi, si x apparaît dans e sa taille sera bien c_x.

Une transformation bien typée selon cette règle de typage sera dite *newtonienne*. Notons que l'application d'une transformation newtonienne ne change pas le nombre d'éléments d'une collection newtonienne.

[1] En revanche la formule $\forall \eta_0.(\exists \eta.(\eta_0 = \eta))$ est vraie. Mais il est facile de se convaincre que ce c'est pas celle-ci que nous voulons exprimer. Ceci montre le besoin de formaliser notre approche. Voir [CL89] pour une formalisation de systèmes de contraintes avec variables universellement quantifiées et [PR03] pour des exemples d'utilisation de telles contraintes dans un système de types.

Notons également que si on veut étendre notre vérification aux motifs *disjoints* (comme le motif $m_1 \mid m_2$), les deux branches d'un motif disjoint devront avoir la même taille.

Résolution des contraintes

L'algorithme de résolution de systèmes de contraintes ensemblistes peut être simplement enrichi pour tenir compte des contraintes sur les tailles. Par exemple on considérera la réduction suivante :

$$\{[\tau_1]\rho_1(s_1) \subseteq [\tau_2]\rho_2(s_2)\} \equiv \{\tau_1 \subseteq \tau_2 \ ; \ \rho_1 = \rho_2 \ ; \ s_1 \subseteq s_2\}$$

On enrichira également la procédure de simplification des types (mise en forme normale) avec des simplifications de tailles. On observera notamment que $[\tau]\rho(0) = 0$. En effet, la taille notée 0 dénote l'ensemble vide. Il existe bien des collections sans éléments (de type $[int]seq(Zero)$ par exemple) mais aucune sans taille.

Exemple

La transformation *map* définie par $\lambda f.\{x \Rightarrow [f\ x]\}$ est newtonienne. En effet ici la taille du motif est $Suc(Zero)$ tout comme la taille de la séquence remplaçante $(f\ x) :: empty\_seq$. Considérons simplement que son type est $(\alpha \to \beta) \to [\alpha]\theta(\eta) \to [\beta]\theta(\eta)$.

Considérons à présent la transformation $\{* \text{ as } x, y \Rightarrow y :: (map\ f\ x)\}$ où f a le type $int \to int$. Cherchons à montrer que cette transformation a le type $[int]\theta(\eta) \to [int]\theta(\eta)$. La taille du motif est $c_x + Suc(Zero)$. La taille de la séquence remplaçante est $Suc(c_x)$. En effet x a le type $[int]seq(c_x)$ d'après le contexte de typage, $(map\ f\ x)$ a donc le type $[int]seq(c_x)$ également et $(y :: (map\ f\ x))$ a le type $[int]seq(Suc(c_x))$. La taille du motif après mise en forme normale est bien égale à la taille de la séquence remplaçante. Donc la transformation est bien newtonienne.

Résultats

L'avantage de cette méthode est qu'elle s'adapte simplement au système déjà existant. Les points faibles sont les suivants :

- La récursion polymorphe est nécessaire dans ce cadre pour donner un type satisfaisant aux fonctions récursives (et donc une annotation du code semble nécessaire [Hen93]). En effet dès qu'on ajoute une information de taille aux types, toutes les fonctions dont la récursion se fait sur une structure de donnée deviennent récursivement polymorphes. Par exemple dans la définition `let rec len c = if empty c then 0 else 1 + len (rest c) fi` la fonction `len` est appliquée à `rest(c)` qui n'a pas le même type que `c` puisque pas la même taille.
  Notons toutefois que la plupart des fonctions récursives peuvent s'exprimer sans récursion à partir d'itérateurs qui peuvent être prédéfinis et dont le type sera donné par TC.
- Il faut étudier formellement les opérateurs d'addition et des constantes symboliques pour être sûr que leur intégration ne pose pas de problèmes théoriques.
- On ne peux contraindre deux arguments à avoir exactement le même nombre d'éléments. Par exemple on a envie de donner le type $[seq]int(\eta) \to [seq]int(\eta) \to [seq]int(\eta)$ à une fonction qui prend deux listes de même longueur et renvoie la liste composée des sommes des éléments des deux listes deux à deux. Ce type ne convient pas : il permet d'appliquer cette fonction à deux listes de taille s_1 et s_2 en instanciant η à $s_1 \cup s_2$. Ce problème était

déjà présent dans le système de type initial : dans le type $\alpha \to \alpha \to \tau$ on ne peut forcer les deux arguments à avoir un même *type de base*. Ceci signifie que notre système de types permettant des types hétérogènes ne peut être utilisé dans un cadre homogène. (Dans le chapitre suivant un système de types pour les collections homogènes est proposé.)

Encore une fois, il s'agit d'un travail en cours et beaucoup de choses ont besoin d'être formalisées. Les résultats pratiques obtenus sont toutefois prometteurs. Nous avons en effet implémenté le système présenté et l'exemple de typage donné ci-dessus fonctionne réellement. Par ailleurs des programmes ne faisant pas intervenir de transformations mais provoquant des erreurs de structures peuvent être rejetés comme c'est le cas pour le programme `oneof(seq:())`.

VIII.2.2 Autres approches possibles

L'approche que nous avons explorée est à rattacher à la discipline des types dépendants [Xi98]. Plusieurs autres approches peuvent être envisagées pour détecter statiquement la taille des collections manipulées. La plus répandue est l'utilisation de formules de Presburger comme dans [Gia92], [HPS96] ou [CK01]. On peut également avoir recours à l'interprétation abstraite comme dans [JS97]. Enfin, l'utilisation de types union permet une approche différente de la nôtre pour l'analyse des tailles des collections. Par exemple le type $Cons(int, Nil) \cup Nil$ peut dénoter les listes vides ou à un élément du type int.

Notons que l'approche que nous avons développée ne permet pas de donner un type précis à la définition récursive de la concaténation, alors que les systèmes de [CK01] et [JS97] le permettent.

Quatrième partie

Conclusions et perspectives

Chapitre IX

Vers la compilation de MGS

Le typage de programmes MGS nous a paru être la première étape nécessaire au développement d'un compilateur pour ce langage. En effet, il permet d'adresser deux problèmes inhérents à l'interprète :

1. Le typage dynamique nécessite de manipuler constamment des étiquettes indiquant le type des valeurs.
2. Le filtrage tire peu parti des informations de types et ne peut être pleinement optimisé ni compilé en fonction de la topologie de la collection sur laquelle on filtre.

Nous rendons compte dans ce chapitre de solutions explorées pour remédier à ces sources d'inefficacité. Ces explorations se sont concrétisées par un générateur de code (que nous appellerons picoMGS dans la suite de ce chapitre) ayant les caractéristiques suivantes :

Génération de code OCaml. Un programme OCaml réalisant le programme MGS est produit et peut être compilé.

Suppression des informations de types. L'inférence de types est exploitée pour générer un programme sans étiquettes de types (typage simple, collections homogènes) ou mélangeant des parties de programme avec et sans étiquettes (typage ensembliste, collections hétérogènes).

Spécialisation du filtrage. Les informations de types sont exploitées pour produire un algorithme de filtrage spécialisé pour une topologie particulière.

Nous noterons que cet outil n'est qu'une plate-forme d'étude et que :
- Les collections disponibles sont limitées aux monoïdes.
- Seules les séquences ont fait l'objet d'une compilation de motifs (génération d'un algorithme de filtrage particulier au motif et à la topologie).
- Les primitives et fonctionnalités disponibles dans l'interprète MGS ne sont pas toutes implémentées (notamment, pas de surcharge).

Toutefois, ce développement laisse présager de bons résultats quant au développement futur d'un compilateur MGS vu les performances obtenues (voir les tableaux pages 171 et 185).

Ce chapitre est organisé en trois sections. La première montre comment le code produit par notre générateur comprend des expressions munies de leur type et d'autres sans informations de type. La cohabitation entre ces deux domaines est possibles grâce à des conversions très simples. La seconde section montre que l'on peut produire un algorithme de filtrage pour chaque motif d'une transformation et ce pour chaque topologie sur laquelle la transformation doit s'appliquer. Elle montre également comment on peut envisager l'interfaçage d'un compilateur MGS avec

un compilateur de filtrage externe. Enfin, la troisième section montre que d'autres optimisations fondées sur les types sont possibles pour permettre une implémentation efficace du langage MGS.

IX.1 Mixité du typage statique/dynamique

Dans cette section nous décrivons comment notre générateur de code produit des programmes OCaml où cohabitent des expressions sans informations de types et des expressions étiquetées de leur type.

Par convention, les types associés aux expressions MGS seront écrits dans une police Times italique (*bool* par exemple) et les types OCaml seront écrits dans une police Courier gras (`bool` par exemple).

Expressions étiquetées

Nous utilisons le type somme ci-dessous pour représenter des valeurs étiquetées de leur type dans le langage cible OCaml.

```
type expr =
  B of bool
| I of int
| F of float
| S of string
| U               (* unit *)
| Seq of expr list
| Set of set
| Bag of bag
| Fun of (expr -> expr)

and set = Empty_s | Node_s of set * expr *       set * int
and bag = Empty_b | Node_b of bag * expr * int * bag * int
```

L'application est implémentée directement par une extraction de la fonction sous le constructeur `Fun` suivie d'une application de celle-ci comme le montre l'exemple suivant : l'expression MGS

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

est implémentée par le code généré :

```
Fun (fun user_var_x ->
 let localmgs_0 = (I 1) in
  (match user_var_x with
     Fun f -> f localmgs_0
   | _ -> raise Wrong )
);;
```

Exemple de collection hétérogène : l'expression MGS

```
1 ::  true ::  seq:() ;;
```

est implémentée par :

```
cons (I 1,cons (B true,Seq []));;
```

Notons que `cons` est un constructeur spécifique aux séquences. En effet les constructeurs ne sont pas surchargés dans le sous-ensemble de MGS que nous traitons. Par ailleurs, `cons` a le type `expr*expr->expr`.

Les expressions OCaml du type `expr` seront dites *étiquetées*. Notons qu'on ne peut pas représenter de collection hétérogène sans passer par un type somme.

Expressions sans étiquettes

Les expressions OCaml qui ne sont pas étiquetées sont dites *simples*. La traduction sans étiquettes de la fonction MGS donnée ci-dessus correspond à la fonction simple suivante :

```
fun user_var_x -> (((user_var_x :  (int) -> ('a0))) (1));;
```

ou, si l'on dépoussière le code et que l'on enlève l'annotation de type qui n'est pas nécessaire :

```
fun user_var_x -> user_var_x 1 ;;
```

Le tableau 1 présente les différences de performances entre du code non-étiqueté et du code étiqueté (les deux codes ont été générés par picoMGS). Notons que la différence de performances entre les deux codes pour *bubble-sort* est moins grande que pour *fibonacci* ou *ackerman* car ce programme recourt intensément au filtrage.

	code simple	code étiqueté
`fibonacci(40)`	2.2 sec.	22 sec.
`ackerman (3,11)`	0.5 sec.	46 sec.
`bubble-sort (iota(500)@iota(500))`	0.45 sec.	0.62 sec.

Tab. 1 – Comparaison des performances entre du code étiqueté et du code non étiqueté.

Passage du code simple au code étiqueté

Considérons l'exemple de l'expression MGS `e1::e2` où `e1` est une expression de type *int* (par exemple `1+1`) et `e2` est une expression de type [*float*]*seq* (par exemple `1.1::seq:()`). L'arbre de syntaxe de la figure 1 est annoté du résultat de l'inférence de types sur cette expression. L'expression `e1::e2` ne peut être implémentée par une expression simple (il s'agit d'une collection hétérogène). Par conséquent, une manière simple d'implémenter cette expression consiste à l'implémenter entièrement par du code étiqueté (y compris les sous-expressions `e1` et `e2`). Toutefois, il est plus efficace d'implémenter `e1` par du code simple puis convertir l'expression OCaml de type `int` en expression de type `expr`.

Nous essayons de systématiser cette approche dans notre générateur de code.

À chaque type on associe une *implémentation* : soit le type `expr`, soit un type simple OCaml (qui n'est pas un type somme). Les types union sont implémentés par du code étiqueté (type

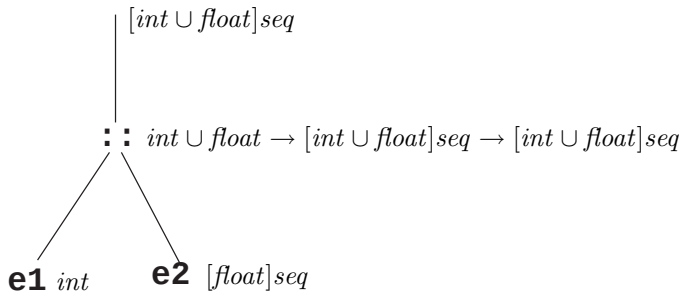


FIG. 1 – Termes étiquetés par leur type.

`expr`). Les types clos et sans union sont implémentés par du code simple. Pour les types non-clos, les variables de types sont non bornées et sont implémentées par une variable de type OCaml, les variables de types bornées sont implémentées par le type `expr`, de même que les types collections comprenant une variable de topologie. En effet, dans le code généré le polytypisme est traité par une forme de surcharge. On peut implémenter les collections de façon homogène et éviter la surcharge mais cela n'a pas été fait dans notre prototype. Dans notre exemple l'implémentation de l'expression `e1::e2` est dynamiquement typée mais les implémentations de `e1` et de `e2` sont statiquement typées.

Une fois les implémentations choisies pour chaque sous-expression, on insère des conversions entre le monde simple et le monde étiqueté. Dans cet exemple, il faut faire passer `e1` et `e2` dans le monde étiqueté (FIG. 2).

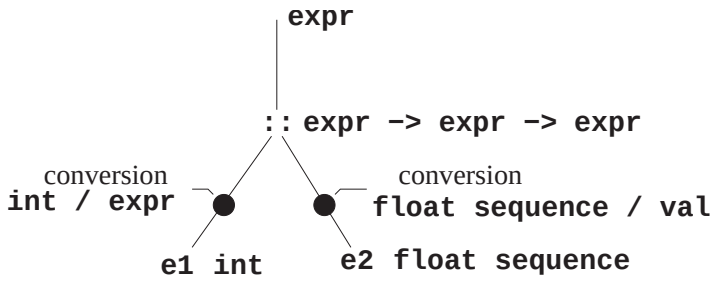


FIG. 2 – Termes étiquetés par leur mode d'implémentation.

Ici la conversion d'un `int` en `expr` consiste à insérer un constructeur `I`. En revanche la conversion de la séquence consiste à convertir tous ses éléments puis à insérer un constructeur `Seq`.

On peut remarquer que seul le contenu de la séquence nécessite d'être étiqueté. En effet, on peut représenter une séquence hétérogène par le type `expr list`. Ceci nécessiterait de prévoir plusieurs sortes de conversions et d'avoir des expressions mixtes qui ne sont ni dans le monde simple, ni dans le monde étiqueté. Ceci n'a toutefois pas été pris en compte dans notre prototype.

Passage du code étiqueté au code simple

Considérons à présent l'exemple suivant : `size(e)` où `e` a le type $[int \cup float]seq$. Le typage donne le résultat indiqué sur la figure 3.

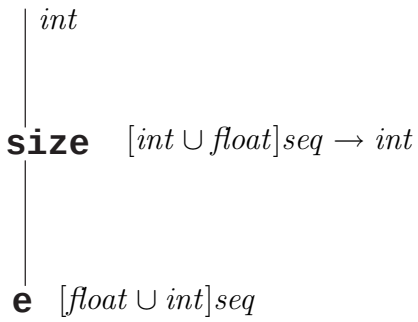


FIG. 3 – Termes étiquetés par leur type (2).

L'implémentation de l'expression `e` sera étiquetée puisque son type contient une union. Idem pour `size`. Toutefois, le résultat de l'application étant un *int* on pourra l'implémenter statiquement. Donc on va effectuer une conversion entre le monde dynamique et le monde statique (voir figure 4).

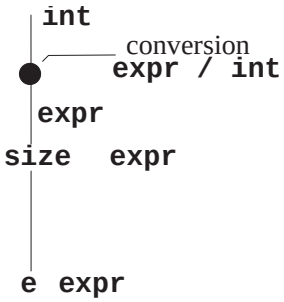


FIG. 4 – Termes étiquetés par leur implémentation (2).

Ici la conversion se fait avec le code suivant : `match XXX with Int i -> i` où `XXX` est le code correspondant à l'expression considérée.

On peut remarquer que `size` est un opérateur du langage et que quel que soit son argument, il renverra toujours un *int* (`size` a le type $\forall\alpha, \theta/\emptyset.[\alpha]\theta \rightarrow int$). On peut alors implémenter une version de `size` avec le type `expr->int` (figure 5). Pour voir si ce choix est intéressant, envisageons deux cas :

Cas 1 : le résultat attendu par l'application de `size` à une collection doit être étiqueté. Il suffit dans ce cas d'appliquer le constructeur `I` au résultat. Cette conversion n'est pas coûteuse car le constructeur aurait été de toute manière été inséré par la version de `size` ayant le type `expr`.

Cas 2 : le résultat attendu doit être (ou peut être) simple. Dans ce cas on a économisé l'étiquetage *et* la conversion du monde étiqueté au monde simple.

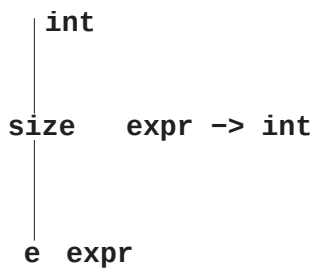


FIG. 5 – Termes étiquetés par leur implémentation (3).

Cette solution est donc satisfaisante dans tous les cas. L'arbre étiqueté par les implémentations correspondantes est représenté ci-dessus. Toutefois, l'opérateur `size` n'est ni dans le monde simple, ni dans le monde étiqueté. Mais il s'agit ici d'une expression atomique (sans sous-expression) et prédéfinie dans le langage. La mise en œuvre d'un tel opérateur hybride est donc plus simple que la manipulation de collections hybrides (partiellement simples, partiellement étiquetées).

Bilan

	picoMGS	picoMGS	picoMGS	OCaml `ocamlopt`	MGS interprète
typage	dynamique	ensembliste	simple		dynamique
étiquetage	oui	mixte	non		oui
fibo(35)	1.57 s	0.18 s	0.18 s	0.18 s	4.5 s
het (35)	1.57 s	0.18 s	X	X	4.5 s

FIG. 6 – Influence du typage et de l'étiquetage sur les performances.

Nous venons de voir que la coexistence dans un programme d'expressions étiquetées et d'expressions non-étiquetées était possible et simple à mettre en œuvre. Cette idée n'est pas nouvelle et a été proposée dans le domaine du *soft typing* pour la première fois [CF91, Wri94, ALW94]. Toutefois l'optique du *soft typing* est légèrement différente de la nôtre puisqu'elle consiste à accepter tous les programmes et à utiliser du code muni de son type pour vérifier dynamiquent le bon typage du code exécuté lorsque le système de types n'a pas permis de garantir l'absence d'erreur de type. Dans notre cas, le code muni de son type est essentiellement destiné à permettre l'implémentation de collections hétérogènes. Notons que la problématique de savoir lorsque l'on peut représenter une valeur sans l'accompagner d'une information sur sa structure a été traitée dans [Ler92, Ler97].

La production de code mixte simple/étiqueté nous permet de bénéficier de bonnes performances dans les parties du programme ne nécessitant pas d'informations de types à l'exécution tout en ne rejetant pas les collections hétérogènes. Ceci n'est évidemment rendu possible que grâce à une inférence de types lors de la « compilation ».

La figure 6 montre que le code mixte peut avoir les même performances que le code simple. La fonction *het* est définie par :

```
fun het (x) = true ::  fibo(x) ::  seq:() ;;
```

Cette fonction produit une collection hétérogène contenant un booléen et un entier. Le code mixte produit pour *het(35)* a les mêmes performances que le code simple produit pour *fib(35)*. Ceci montre l'utilité de la mixité dans le code produit : une grande classe de programme est acceptée (la fonction *het* n'est pas acceptée par OCaml) tout en fournissant de bonnes performances lorsque possible.

Notons que nous n'avons pas étudié les propriétés du code mixte. En particulier, il existe peut être des programmes qui seraient plus efficaces avec du code étiqueté qu'avec du code mixte. En effet les coûts de passage entre les deux mondes n'est pas négligeable pour les collections.

IX.2 Compilation du filtrage

Dans l'interprète, le motif est interprété : il s'agit d'une structure de données que l'on parcourt et en fonction de ce qui est lu, on va effectuer un traitement spécifique. Cette approche suit de près la définition de l'algorithme de filtrage vue au chapitre IV.

Or la structure du motif est connue avant de commencer à effectuer le filtrage. On peut associer un algorithme de filtrage particulier à chaque motif. Cette approche est particulièrement intéressante en compilation : le motif est interprété une seule fois à la compilation et l'algorithme spécialisé pour *un* motif et *une* topologie est utilisé à l'exécution du programme.

Nous avons suivi cette démarche dans picoMGS. L'objectif était de démontrer quantitativement les bénéfices d'une telle approche. Dans cette section, nous présentons notre étude de la compilation des motifs sur les séquences. La compilation des motifs sur d'autres topologies n'ont pas été traitée dans picoMGS.

Dans cette section, nous commençons par présenter la compilation de motifs sur les séquences telle que réalisée dans picoMGS (sous-section IX.2.1). Le reste de la section présente la façon dont nous pensons déléguer la compilation du filtrage à un outil spécialisé : TOM. La sous-section IX.2.2 justifie notre choix d'utiliser TOM, la sous-section IX.2.3 présence brièvement cet outil et enfin, la sous-section IX.2.4 présente comment TOM peut être utilisé dans notre démarche de compilation du filtrage de MGS.

IX.2.1 Compilation des motifs sur les séquences

La figure 7 présente l'algorithme généré par picoMGS pour filtrer le motif `x,y` sur une séquence, considéré dans la règle `x,y => [x+y]`. La partie droite de la règle est nécessaire à la génération de l'algorithme car en cas de succès, il renvoie un couple constitué de la dernière position filtrée et de la séquence remplaçante (les valeurs des éléments filtrés sont nécessaires pour la calculer). On notera que dans le sous-ensemble de MGS considéré, l'addition n'est pas surchargée. Dans la règle que nous considérons il s'agit de l'addition entière. Par conséquent les éléments filtrés sont des entiers et on peut se placer dans le monde simple, où les valeurs ne sont pas étiquetées. Dans l'algorithme généré, le filtrage de `x,y` dans la séquence `c` implémentée par un tableau et à partir d'une position `pos` consiste simplement à vérifier que la position `pos` n'est pas consommée puis à filtrer le reste du motif (soit `y`) à partir de la position voisine notée `pos'`. Filtrer `y` en `pos'` consiste à vérifier que `pos'` n'est pas consommée. Si c'est le cas on indique que

```
if libre pos
then
  let x0 = c.(pos)
  and pos' = pos+1
  in (if libre pos'
      then
        let x1 = c.(pos')
        and pos'' = pos'+1
        in Succes (pos''-1, ((x0+x1)::[]) )
      else Echec_quantite (* il n'y a plus d'elements libres au voisinage *) )
else Echec_quantite (* il n'y a plus d'elements libres au voisinage *)
```

FIG. 7 – Algorithme de filtrage généré par picoMGS pour la règle `x,y => [x+y]`.

le filtrage est en succès et on renvoie la dernière position consommée et la séquence remplaçante. Si l'une des deux positions était déjà consommée, le filtrage est en échec.

La répétition du processus de filtrage à partir d'autres positions non consommées de la séquence est gérée en dehors de la définition de cet algorithme et est de manière identique que le motif soit interprété ou compilé.

```
if libre pos
then
  let x0 = c.(pos)
  and pos' = pos+1
  in (if ((x0)<(0))
      then (Succes (pos'-1, (((x0)+(1))::([]))))
      else Echec_qualite)
else Echec_quantite
```

FIG. 8 – Algorithme de filtrage généré par picoMGS pour la règle `x/x<0 => [x+1]`.

La figure 8 donne l'algorithme généré pour la règle `x/x<0 => [x+1]`.

Les algorithmes filtrant les motifs `x,y` et `x/x<0` sont très simples. Des algorithmes plus complexes doivent être produits pour filtrer des répétition (`*`). Les algorithmes produits doivent garantir que si une solution existe alors elle est trouvée. La figure 9 donne l'algorithme pour la règle `x, y+, z => [0]`. Le filtrage de `y+,z` se décompose en deux parties. La fonction `filtre_motif_interne` filtre le sous-motif répété (ici `y`) et la fonction `filtre_reste` filtre le sous-motif apparaissant après le motif répété (ici `z`). Ces deux fonctions sont passées à la fonction `itere_plus` prédéfinie qui va répéter le filtrage du sous-motif jusqu'à ce qu'une instance du motif soit trouvée ou qu'un échec soit déclaré parce que toutes les répétitions possibles ont été tentées. En effet, si pour une instance de `y+` le filtrage du motif `z` échoue, rien n'assure qu'il échouera pour une instance plus grande de `y+` (penser par exemple au motif `y+,z/z=0`). Notons que dans le motif `(`M`)+,` M' les variables introduites dans M ne sont pas accessibles depuis M'.

La figure 10 donne l'algorithme généré pour la règle `( (x,y/y<x)+ , (x,y/x<y)+ )+ => [0]`. Cette figure montre la taille importante des algorithmes produits. Par ailleurs, on peut voir sur le code associé au motif `x/x<0` (FIG. 8) que le code produit est grossier (la liaison `let pos'=pos+1` est inutile par exemple). Notre générateur d'algorithmes de filtrage pour les

```
if libre pos
then
  let x0 = c.(pos)  and pos' = pos+1
  in (
      let filtre_motif_interne i = (
        if libre i
        then
          let x1 = c.(i)  and i' = i+1
          in (Succes (i'-1, ((()))))
        else Echec_quantite )
      and
        filtre_reste i = (
          if libre i
          then
            let x2 = c.(i)  and i' = i+1
            in (Succes (i'-1, ((0)::([]))))
          else Echec_quantite )
      in (Sequence.itere_plus filtre_motif_interne filtre_reste pos')
    )
else Echec_quantite
```

FIG. 9 – Algorithme de filtrage généré par picoMGS pour la règle `x, y+, z => [0]`.

séquences fait une centaine de lignes d'OCaml pour produire des algorithmes naïfs. Or une caractéristique fondamentale de MGS est la possibilité d'utiliser des transformations. Le filtrage est une opération coûteuse dans un programme constitué de transformations et l'utilisation d'algorithmes naïfs n'est pas acceptable. En effet, un programme produit par notre compilateur MGS est bien meilleur que sa version interprétée mais bien moins performant qu'un programme équivalent compilé par un moteur de récriture (ELAN ou Maude) ou un compilateur de motifs (TOM) (voir par exemple la comparaison entre picoMGS et TOM sur le *swap sort*, figure 14, page 185). Il est donc naturel de chercher à améliorer les algorithmes produits.

IX.2.2 Choix effectué

Pour disposer d'un filtrage très performant on a le choix entre :

1. Produire un programme dans un langage spécialisé dans le filtrage comme ELAN ou Maude (pour les termes).
2. Implémenter les méthodes connues pour filtrer efficacement certaines structures de données (par exemple les méthodes mises en œuvre dans les outils ci-dessus ou bien des algorithmes classiques comme Boyer/Moore sur les séquences).
3. Utiliser un compilateur de motifs disponible pour le langage cible que nous avons choisi.

Dans les trois cas une étude spécifique est nécessaire pour chaque sorte de collection disponible dans MGS. Par exemple ELAN permet de filtrer sur des séquences ou des multi-ensembles mais pas sur des tableaux.

La première approche nécessite de disposer d'un langage cible différent pour chaque sorte de collection sur laquelle on veut filtrer. D'une part, ceci ne permet pas de créer facilement des

```
let filtre_motif_interne i = (
  let filtre_motif_interne i = (
    if libre i
    then
      let x0 = c.(i)  and i' = i+1
      in (if libre i'
          then
            let x1 = c.(i')  and i'' = i'+1
            in (if ((x1)<(x0))
                then (Succes (i''-1, ((()))))
                else Echec_qualite)
          else Echec_quantite )
    else Echec_quantite )
  and
    filtre_reste i = (
      let filtre_motif_interne i = (
        if libre i
        then
          let x2 = c.(i)  and i' = i+1
          in (if libre i'
              then
                let x3 = c.(i')  and i'' = i'+1
                in (if ((x2)<(x3))
                    then (Succes (i''-1, ((()))))
                    else Echec_qualite)
              else Echec_quantite )
        else Echec_quantite )
      and
        filtre_reste i = (Succes (i-1, ((()))))
      in (Sequence.itere_plus filtre_motif_interne filtre_reste i))
  in (Sequence.itere_plus filtre_motif_interne filtre_reste i))
and
  filtre_reste i = (Succes (i-1, ((0)::([]))))
in (Sequence.itere_plus filtre_motif_interne filtre_reste pos)
```

FIG. 10 – Algorithme de filtrage généré par picoMGS pour la règle `( (x,y/y<x)+ , (x,y/x<y)+ )+ => [0]`.

programmes utilisant des structures de données différentes. D'autre part, ceci nécessite de savoir compiler tout le langage MGS vers plusieurs langages cibles.

La seconde approche est plus attractive mais elle nécessite un important travail d'étude, d'implantation, de mise au point et de maintenance des méthodes disponibles dans la littérature. Elle impose également de modifier le compilateur pour intégrer de nouvelles optimisations.

La troisième approche est la plus séduisante. En effet elle suppose que tous les efforts d'implémentation d'un filtrage efficace sont concentrés dans un outil externe. Ceci a les avantages suivants :

– Un seul langage cible est nécessaire (en supposant que les outils choisis soient conçus pour

ce langage).
- Le compilateur devient automatiquement plus performant lorsque l'outil compile le filtrage plus efficacement.
- Notre expertise du filtrage sur des structures peu connues peut être diffusée indépendamment de la plate-forme MGS si on suppose que nous développons nos algorithmes directement dans l'outil au lieu de les développer dans notre compilateur.

Ces trois avantages nous ont menés à choisir TOM, que nous présentons dans la section suivante[1], pour la suite de notre étude.

Dans le reste de cette section, nous présentons brièvement TOM puis nous présentons comment on peut l'utiliser pour compiler les motifs MGS.

IX.2.3 Le compilateur de filtrage TOM

TOM est un compilateur de filtrage développé à l'INRIA au sein du projet PROTHEO [MRV01, MRV03]. Ce développement fait suite à ELAN, un langage de programmation basé sur la réécriture doté d'un compilateur performant [Mor99].

Cet outil permet d'étendre un langage de programmation avec des opérateurs de filtrage, associatifs ou non. Ceci permet donc de programmer par filtrage dans des langages impératifs (C, Java et Eiffel) et de disposer de filtrage associatif dans les langages fonctionnels, ceux-ci disposant déjà de filtrage simple en général (TOM peut être utilisé avec OCaml).

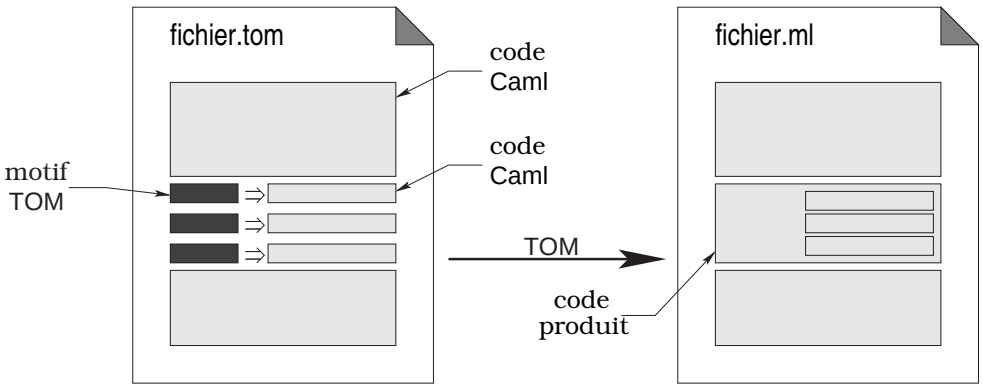


FIG. 11 – TOM est non intrusif : il transforme un motif en algorithme de filtrage.

Un exemple très simple d'utilisation de TOM est l'implémentation de la fonction `List.iter`. Dans la bibliothèque standard OCaml, la fonction `List.iter` est définie par :

```
let rec iter f = function
   [] -> ()
 | a::l -> f a; iter f l
```

[1]Une collaboration avec l'équipe de développement de TOM nous a permis de concrétiser le premier de ces trois points (TOM fonctionne maintenant avec OCaml). Nous espérons concrétiser dans un futur proche la réalisation du troisième point.

Cette fonction applique la fonction f à chaque élément d'une liste (il ne s'agit pas de *map*, on ne renvoie pas de liste). Le filtrage de Caml ne permet de filtrer que la tête de la liste et la queue (ou plusieurs éléments en tête en imbriquant les motifs). Donc la définition de `iter` doit être récursive car on veut toucher tous les éléments de la liste.

Le filtrage de TOM permet de trouver les instance d'un motif n'importe où dans une liste (pas uniquement en tête) et garantit que toutes les instances sont trouvées. Ceci permet de définir `iter` beaucoup plus succinctement :

```
let iter f l =
 %match (list l) {
  conc (l'*,a,l''*) -> { f a }
}
```

Simplifions la syntaxe utilisée afin de comparer les deux implémentations :

Code Caml :

```
let rec iter f l=
 match l with
   [] -> ()
 | a::l' -> f a ; iter f l'
```

Code TOM :

```
let iter f l=
 match l with
   l' @ [a] @ l'' -> f a
```

L'action effectuée par TOM sur le code ci-dessus est le remplacement du morceau de code correspondant au filtrage par du code Caml pur réalisant effectivement le filtrage.

Décrivons plus précisément le filtrage proposé par TOM en le comparant à celui proposé par MGS :

- TOM propose du filtrage simple (ou syntaxique) ou modulo associativité (séquences ou tableaux selon une dimension). Pour le moment, le filtrage modulo commutativité n'est pas traité, on ne peut donc pas filtrer facilement sur des ensembles ou des multi-ensembles. De même, le filtrage sur des structures n'étant pas des termes n'est pas traité. Filtrer sur une matrice nécessite par exemple un encodage. TOM propose également le filtrage sur les documents au format XML.
- L'expressivité des motifs dans cette limite est comparable aux motifs de MGS. Par exemple sur la séquence on peut filtrer deux éléments voisins avec le symbole « , » ou une sous-séquence avec le symbole « * ». La garde s'exprime ou bien en définissant de nouveaux opérateurs de filtrage, ou bien dans la partie droite de la règle.
- La stratégie de filtrage de TOM est différente de celle de MGS : toutes les instances sont trouvées alors que MGS manipule des instances disjointes.
- La sémantique des règles est également différente : dans TOM, la partie droite d'une règle dénote une action à effectuer et non une partie venant remplacer l'instance du motif.

IX.2.4 Utilisation de TOM pour le filtrage des séquences

Sémantiques différentes

TOM réalise un filtrage différent de celui de MGS. La traduction de code MGS en code TOM n'est donc pas directe. En effet, si $m \Rightarrow e$ est une règle TOM, e dénote une action à

effectuer pour chaque instance de m dans la structure considérée. Si deux instances du motif sont trouvées et qu'elles se recouvrent, l'action e sera effectuée deux fois alors qu'en MGS, les instances sélectionnées doivent être disjointes. Ceci implique que le code TOM +OCaml que l'on va générer doit gérer explicitement la consommation des éléments filtrés.

Cette gestion des éléments consommés peut se faire soit en déclarant un opérateur de filtrage particulier dans TOM (nous le noterons `libre`), soit en ajoutant un test dans la partie droite des règles TOM.

Tentative avec des listes OCaml

L'utilisation d'une liste de OCaml comme structure de donnée sur laquelle appliquer le filtrage de TOM nécessite d'associer un booléen à chaque élément au sein de la liste pour indiquer s'il est consommé ou non. Pour éviter de modifier la liste et donc de recommencer le filtrage à zéro à chaque fois qu'un élément est consommé (ce qui entraînerait de consulter de nombreuses fois les valeurs déjà consommées), on peut utiliser des références OCaml pour porter le booléen.

Nous avons commencé par explorer cette voie afin de montrer que l'on pouvait interfacer picoMGS avec TOM.

Le code TOM+OCaml de la figure 12 réalise l'équivalent de la transformation MGS { `x,y/y<x => [y,x]` } modulo la construction de la nouvelle séquence après le filtrage, d'un coût généralement négligeable par rapport au coût du filtrage. Les performances de cette implémentation sont données dans le tableau de la figure 14, colonne 5.

```
let rec bub l =           (* l est modifiee en place *)
%match(list l) {
   conc(x*,libre(e1),libre(e2),z* ) -> {
     match (‘e1,‘e2) with (v1,r1),(v2,r2) ->
       if v2<v1
       then
 begin
   r1:=Replace(v2::v1::[]) ;
   r2:=Destroy
 end
       else ()}
};;
```

FIG. 12 – Filtrage sur une séquence implémentée par une liste.

Tentative avec des tableaux

Pour réaliser efficacement le filtrage sur une séquence, il est naturel d'utiliser un tableau comme nous l'avons fait en section IX.2.1 car ceci permet un parcours plus rapide de la structure et une gestion des positions consommées en dehors de la structure. Le code TOM +OCaml réalisant ceci pour la transformation { `x,y/y<x => [y,x]` } est donné en figure 13.

Toutefois, TOM étant encore en cours de développement, le filtrage des tableaux par TOM n'est pas disponible pour le langage cible OCaml au moment où ce document est produit. Nous

```
let tri_elementaire t =      (* t est une séquence représentée par un tableau *)
  let s = Array.make (Array.length t) true
  and modifications = ref []
  in begin
      (* debut du filtrage *)
     %match (array t) {
        conc(x*,libre(e1),libre(e2),y* ) -> {
          if (e2<e1)
          then
            begin
              let pos1 = ‘pos(e1) and pos2 = ‘pos(e2) in
                modifications :=
                     ( [pos1;pos2], [t.(pos2);t.(pos1)] ) :: !modifications ;
                s.(pos1) <- false ;
                s.(pos2) <- false ;
            end
      }
     };
       (* fin du filtrage *)
       construit_res t modifications
    end
```

FIG. 13 – Filtrage sur une séquence implémentée par un tableau.

n'avons donc pas eu la possibilité de mesurer les résultats de cette version.

IX.2.5 Bilan

Nous avons vu dans cette section que l'on pouvait créer automatiquement un algorithme de filtrage spécialisé pour un motif et type de collections. Ceci permet d'envisager le développement d'un compilateur MGS complet et performant. Notons que disposer d'une version typée du langage permet :

- de savoir pour quelle topologie le motif est compilé ;
- de rendre significatives les différences de performances entre le filtrage compilé et le filtrage interprété (rien ne sert d'optimiser le filtrage si la plate-forme est fondamentalement inefficace).

Toutefois, la compilation du filtrage doit être finement réalisée afin que MGS puisse être comparé à d'autres langages fondés sur le filtrage en termes de performances. Nous avons envisagé de concrétiser cet aspect en utilisant un compilateur de filtrage. Notre étude préliminaire montre que cette voie a plusieurs avantages, notamment :

- les algorithmes efficaces sont développés et mis au point une seule fois (*non-redondance*) ;
- les améliorations apportées à ces algorithmes sont disponibles immédiatement dans MGS (*non-obsolescence*) ;
- le « savoir-faire » de différentes équipes est centralisé dans un seul outil et rendu disponible à la communauté (une collaboration impliquant l'auteur et visant à proposer du filtrage sur les GBF dans TOM a débuté) (*centralisation*).

La voie que nous avons explorée ici est donc prometteuse. Encore une fois, il s'agit d'un travail en cours et il reste encore à implémenter la traduction automatique d'un motif MGS vers un motif TOM d'une part (le code TOM +OCaml des figures 12 et 13 a été écrit à la main) et à envisager cette approche sur d'autres topologies d'autre part.

La figure 14 présente les mesures permettant de comparer les approches mentionnées dans cette section pour le filtrage sur les séquences. On peut observer que :

- Le code compilé est plus performant que le code interprété, même avec du typage dynamique et un motif interprété (colonnes 1 et 2).
- La compilation du motif permet des gains importants aussi bien lorsque la compilation est naïve (colonnes 3 et 4) que lorsqu'elle est déléguée à TOM (colonnes 3 et 5, en notant qu'on peut faire mieux que dans la colonne 5 en utilisant des tableaux).
- TOM peut produire du code très efficace s'il est bien utilisé. La comparaison des colonnes 4 et 6 montre que le code TOM que l'on écrirait *intuitivement* peut être extrêmement efficace (sur le crible) ou extrêmement mauvais (sur le bubble-sort).

IX.3 Optimisations du filtrage fondées sur les types

Nous avons vu en première section de ce chapitre que l'inférence de types lors de la compilation permet de se passer de nombreux tests de type lors de l'exécution (les lectures d'étiquettes). Elle permet également des optimisations dans le processus d'application des transformations. Nous avons vu dans la section précédente la spécialisation du filtrage en fonction de la topologie.

Nous présentons dans cette section d'autres optimisations possibles fondées sur l'analyse de types à la compilation.

Règles inutiles. Considérons la règle $x : int \Rightarrow [x+1]$ et une collection c.

Si l'analyse de types indique que c ne contient pas d'entiers alors la règle ne peut s'appliquer. De manière générale, pour une règle $m/g \Rightarrow e$ et si τ est le type de la collection, la règle ne peut s'appliquer si $\mathcal{V}oisin(m, \tau)$ vaut 0 (*incompatibilité* au sens du chapitre VII). On sait donc dès la compilation qu'il est inutile d'essayer d'appliquer cette règle.

Tests de types inutiles Considérons à nouveau la règle $x : int \Rightarrow [x+1]$ et une collection c. Si l'analyse de types indique que les éléments contenus dans c sont de type int alors le test de type est inutile pendant l'exécution.

Filtrage en fonction du type On peut envisager d'implémenter les collections de manière à faciliter la recherche de valeurs dont on connaît le type. Par exemple on peut implémenter les ensembles par des sous-ensembles dont les valeurs ont toutes le même type (l'idée de choisir automatiquement l'implémentation des données en fonction des calculs qui sont faits dessus apparaît dans [SSS81]). Dans ce cas une optimisation du filtrage est possible et elle englobe les deux précédentes : on ne cherche à instancier un motif élémentaire que dans la partie de la collection qui contient les valeurs du type approprié.

Les deux premières optimisations proposées sont assez directes à implanter mais on peut penser qu'elles s'appliquent peu dans les programmes réels. La dernière optimisation s'applique plus souvent mais il peut être difficile d'implémenter les collections en voulant optimiser à la fois l'accès aux valeurs en fonction de leur type et l'accès aux valeurs en fonction de la relation de voisinage. En effet, le processus de filtrage est fortement lié à la topologie des collections puisqu'un motif filtre des valeurs voisines.

Quoi qu'il en soit, on voit que l'inférence de type est la source de nombreuse optimisations pouvant être implémentées dans un compilateur MGS.

Ceci laisse présager de bonnes performances pour un futur compilateur. Bien que très partielle, notre maquette de compilateur, picoMGS, a permis de montrer que la compilation de programmes MGS était envisageable et pouvait donner des programmes efficaces. Le programme picoMGS est constitué de 9000 lignes d'OCaml, incluant l'algorithme d'inférence de types et ses variantes présentées dans ce manuscrit, la génération de code dynamiquement typé, statiquement typé ou mixte et enfin la compilation des motifs sur les séquences.

FILTRAGE	1	2	3	4	5	6
bubble-sort `(iota(300)@iota(300))`	1.1	0.18	0.12	0.09	0.07	16
bubble-sort `(iota(500)@iota(500))`	3.2	0.62	0.45	0.35	0.27	St.Ov.
bubble-sort `(iota(3000)@iota(3000))`	123	26	24	21.7	29.8	St.Ov.
`(iota(3000)@iota(3000))`	0.3	<0.05	<0.05	<0.05	<0.05	<0.05
crible 300	2.4			0.93		0.03
crible 1000	512			87		0.65

Les durées sont données en secondes. Les mesures ont été effectuées sur un PC doté d'un Pentium-4 à 2.4 GHz. Le code OCaml est compilé par le compilateur natif version 3.06.
Descriptif des colonnes :

1. Interprète 1-MGS
2. Code généré par picoMGS avec motif interprété et typage dynamique.
3. Code généré par picoMGS avec motif interprété et typage statique (code non étiqueté).
4. Code généré par picoMGS avec motif compilé et code non étiqueté.
5. Code TOM+OCaml implémentant une transformation MGS (implémentation par des listes modifiées en place, compilation optimisée).
6. Code TOM+OCaml, sémantique différente de MGS

FIG. 14 – Comparatif des performances du filtrage sur les séquences.

Conclusion

IX.4 Résumé des travaux

Nous avons défini dans ce travail un algorithme générique de filtrage sur les collections topologiques, un mécanisme de traduction à la volée d'expressions fonctionnelles en termes de combinateurs ainsi qu'un système original de types permettant la prise en compte de fonctions polytypiques. Ces travaux s'intègrent dans le cadre du projet MGS en permettant un traitement uniforme et efficace des transformations agissant sur les collections topologiques.

Plus précisément, nous avons commencé par définir la notion de collection topologique. Ensuite, suivant le point de vue unifié apporté par ce type de données, il a été naturel de définir un langage de motifs ainsi qu'un algorithme de filtrage unique permettant de prendre en compte toute la classe des collections topologiques. De plus, nous avons fait en sorte que la généricité des motifs et de l'algorithme de filtrage ne se fasse pas au détriment de la richesse des motifs utilisables. Au contraire, le langage considéré est très riche, comme l'ont montré les nombreux exemples de transformations proposés.

Puis, nous avons exposé certaines des techniques utilisées pour l'implémentation de l'interprète MGS. Nous nous sommes concentrés sur la réalisation de l'*évaluateur* qui est le module dédié à l'interprétation du noyau fonctionnel du langage. L'originalité de l'implémentation proposée est qu'elle repose sur la réduction de l'application du langage hôte au lieu de la gérer elle-même. Un objectif de simplicité, d'orthogonalité et d'efficacité a motivé ce choix. Une technique de combinatorisation a été utilisée afin de transformer les fonctions définies par l'utilisateur en fonctions du langage hôte. Les fonctions non-strictes et les traits impératifs de MGS sont bien intégrés à ce schéma grâce à l'utilisation de *glaçons*. En conclusion de cette présentation, nous avons complété nos travaux par le détail de plusieurs optimisations significatives qui ont été implémentées.

Un interprète dynamiquement typé permet d'expérimenter et valider les outils théoriques développés dans un langage. Cependant, il est intéressant, surtout pour des applications gourmandes en temps de calcul, de permettre la compilation des programmes. Pour ce faire, il est fondamental d'aborder les problèmes liés au typage. Nous nous sommes intéressés à la définition d'un système de types statique. Nous avons réalisé un algorithme d'inférence de types en présence de sous-typage non-structurel. Ce système de types permet la prise en compte de fonctions polytypiques, étendant naturellement le polymorphisme structurel désormais standard dans les langages fonctionnels. On peut remarquer qu'il subsiste dans le système développé une partie de typage dynamique. En effet, du fait du caractère hétérogène des collections topologiques, il n'est pas possible de supprimer entièrement les informations de types.

IX.5 Poursuite des travaux

Les travaux que nous avons décrits dans ce document comportent de nombreux manques. En effet, pour des raisons de temps, un certain nombre de problèmes n'ont pas trouvé de réponse entièrement satisfaisante.

IX.5.1 Filtrage

Spécialisation du filtrage en fonction de la topologie. Le filtrage peut être très simplement optimisé sur certaines topologies en récrivant le motif considéré en un motif ayant les mêmes instances mais plus simple à filtrer. Par exemple, sur un ensemble il est plus rapide de filtrer le motif `1,x` que le motif `x,1` (avec une stratégie de gauche à droite). Cette idée a déjà été abordée par [Spi03]. Cette approche reste toutefois à formaliser et à généraliser.

Par ailleurs nous avons montré que la spécialisation de l'algorithme de filtrage en fonction du motif *et* de la topologie permet également d'améliorer les performances. Cette spécialisation reste à étudier et à concrétiser. Là aussi, des voies sont données dans [Spi03].

Architecture modulaire. Nous avons vu que la génération d'un algorithme de filtrage performant pour chaque topologie et chaque motif est une tâche difficile. Nous avons esquissé l'idée que cette tâche doit être réalisée indépendamment des autre mécanismes en œuvre dans MGS et ainsi nous avons envisagé de déléguer la compilation du filtrage à un outil externe. Ceci permettrait d'une part de diviser la conception du compilateur MGS en deux tâches différentes (mais non indépendantes) et, d'autre part, d'associer les compétences acquises au sein du projet MGS aux compétences de l'équipe du projet TOM. Cette approche reste à mettre pleinement en œuvre et une collaboration a débuté dans ce sens.

Enrichissement des stratégies. Plusieurs stratégies d'applications des règles dans une transformation existent déjà dans l'interprète MGS. Des projets comme Maude et ELAN ont formalisé la notion de stratégie et développé des stratégies qui ne sont pas encore exprimables en MGS. Par exemple la recherche de toutes les instances d'un motif (pouvant éventuellement s'intersecter) nécessite actuellement un encodage dans une transformation. Pour la modélisation de phénomènes physiques, par exemple pour les *gaz sur réseau* ou pour réaliser les méthodes dites de Monte-Carlo, on a besoin de stratégies d'application stochastique. Cette dernière est difficile à spécifier et à réaliser correctement. Par exemple, l'algorithme de filtrage proposé dans cette thèse doit être adapté pour ce type de problème afin de ne pas introduire de biais systématique dans les occurences d'applications des règle.

IX.5.2 Interprète

L'utilisation de l'interprète MGS pour l'exécution de nombreux programmes de nature variée, dont certains de grande taille ([BSM04]) a permis une validation par la pratique des techniques originales mises en œuvre dans son implémentation et que nous avons présentées dans cette thèse.

L'étude formelle des propriétés des principes mis en œuvre a débuté par la preuve de la correction de la combinatorisation (annexe A) et peut être approfondie. Il reste également à étudier finement l'impact des différentes optimisations proposées dans cette thèse sur les performances de l'interprète.

IX.5.3 Typage

Sécurité. Des travaux ont été initiés dans le but de vérifier l'absence d'erreurs survenant lors l'application des transformation sur des collections newtoniennes (à structure rigide). Les résultats obtenus ne sont pas encore pleinement satisfaisant, trop peu de programmes sont garantis sans erreurs de structure (il ne s'agit pas d'erreurs de types au sens classique du terme). Ces travaux doivent donc être poursuivis d'une part et généralisés afin de détecter les transformations violant des propriétés propres à d'autres topologies.

Performances. Nous avons vu que le système de types que nous proposons pouvait être amélioré en affinant l'analyse effectuée sur les motifs. Le typage des motifs doit donc être étudié spécifiquement afin d'accepter plus de programmes et d'obtenir des types plus précis (et donc une implémentation plus efficace).

Par ailleurs on peut se demander comment tirer parti de l'inférence de types pour éliminer le plus d'informations de types possible à l'exécution. Nous avons montré que toutes les informations de types sont inutiles lorsqu'on se restreint aux collections homogènes et qu'une réalisation mixte valeurs étiquetées/valeurs non-étiquetées est nécessaire dans le cas de collections hétérogènes. On peut toutefois se demander s'il existe des possibilités intermédiaires. Par exemple, sous certaines conditions on peut prévoir le type de chaque valeur dans une séquence (en particulier, `hd(1,true,seq:())` est un entier alors que `hd(1,true,bag:())` n'est pas forcément un entier en raison de la commutativité du constructeur des ensembles).

Enfin, dans l'optique de développer un compilateur utilisable, l'implémentation de l'inférence de types doit être particulièrement soignée en raison de la complexité de la résolution de contraintes ensemblistes. Là aussi, on peut envisager une réalisation utilisant un solveur externe comme Banshee [Kod]. Mais, cet outil n'ayant pas toute l'expressivité nécessaire, une adaptation dans un sens (affaiblir les types) ou dans l'autre (enrichir Banshee) est requise.

Modularité. Notre implémentation actuelle de l'inférence de types ne permet pas la compilation séparée. En effet le type le plus précis d'un programme ne permet pas la réutilisation de ses fonction en dehors de celui-ci. Pour traiter cette question il faut modifier la façon dont les solutions sont choisies dans l'espace des solutions d'une part et gérer finement l'implémentation du polymorphisme et notamment du polytypisme. Ceci reste à aborder.

Aide à la conception. Nous avons vu que les types produits par notre algorithme d'inférence ne sont pas lisibles par le programmeur en raison du trop grand nombre de contraintes générées. Le typage devient une aide réelle pour le programmeur lorsque les types sont lisibles. De nombreux travaux, comme [Pot01], montrent comment simplifier les contraintes afin de rendre les types lisibles (et également de rendre l'implémentation de la résolution plus efficace). Ces travaux doivent être pris en compte pour rendre le typage *utile* à l'activité de programmation. Un effort est également à faire pour rendre compréhensibles les messages d'erreurs de types.

Richesse. Notre étude s'est concentré sur le typage des collections topologiques et des transformations. Toutefois, MGS est doté d'autres caractéristiques nécessitant une prise en compte dans le système de types. Ces caractéristiques comprennent la surcharge, les enregistrements extensibles et les argument optionnels. Ces points doivent être pris en compte par le système de type si l'on veut créer un compilateur pour le langage MGS dans sa totalité.

IX.5.4 Compilation

Nous avons développé un prototype de générateur de code permettant d'étudier des techniques qui seront mises en œuvre dans un compilateur MGS. Si ce prototype nous a permis d'estimer la faisabilité des approches envisagées, les mesures pour estimer les gains apportés par chaque technique ne sont pas suffisantes. Il faut systématiser le processus de *benchmark* et s'assurer que les jeux de tests ont des caractéristiques variées.

Par ailleurs les choix de conception pour un futur compilateur MGS n'ont pas encore été faits. Nous n'avons fait que donner des pistes. D'autres pistes auraient pu être explorées : les liens entre l'interprète et le compilateur ou les techniques modernes de compilation (*just in time* par exemple).

IX.5.5 Applications

Il faut poursuivre le développement d'applications pour tester et valider les concepts introduits. Par ailleurs, MGS se révèle adapté à d'autres domaines d'applications que ceux évoqués dans ce document. Nous en citerons quatre pour lesquels un travail a déjà débuté.

Une des motivations du projet MGS est de développer des outils permettant de simplifier et de prototyper la simulation de processus de développement. Nous en avons donné quelques exemples simples au cours de ce travail. Nous aimerions nous attaquer à présent à une application plus détaillé de la croissance d'une tumeur. Cette application a un grand intérêt du point de vue médical s'il est possible d'intégrer des parties du réseau métabolique et du réseau de régulation génétique de la cellule dans ces simulations. Par exemple, il serait alors possible de rechercher systématiquement, en tenant compte des informations issues du génotype du sujet, la combinaison de médicaments anti-tumoraux qui réduit la propagation de la tumeur au minimum.

MGS a été récemment utilisé pour mettre en évidence une faille bien connue dans le protocole à clé publique Needham-Schroeder [MJ04]. Cet exemple sert d'application-test dans le domaine de validation de protocoles cryptographiques et les mécanismes offerts par MGS se sont révélé très bien adaptés. Ce type d'applications, tout comme les applications de *model-checking*, correspondent à des recherches dans de très grands espaces d'états. Une approche possible est d'étendre MGS avec des collections intentionnelles permettant de décrire ces espaces d'état « de manière paresseuse » et de ne les développer que si besoin est.

La notion de collection topologique parait bien adaptée à la définition de processus d'auto-assemblages. Nous avons d'ailleurs développé, dans le cadre des P-systèmes, une étude permettant de caractériser le domaine couvert par un processus de tuilage dans un espace décrit par un GBF [GMC03]. Outre les applications nombreuses en biologie (construction et évolution des membranes, étude du cytosquelette cellulaire, etc.), l'étude et la simulation des processus d'auto-assemblage est un problème critique dans le domaine de la nano-ingénierie.

Enfin, nous avons débuté une collaboration avec Frédéric Gruau de l'équipe ALCHEMY à Orsay (LRI/INRIA) pour la modélisation et la simulation de *blobs* [GLRT04]. Le « blob-computing » est un nouveau modèle de calcul amorphe utilisant les ressources d'un réseau de très nombreuses unités de calcul simples, les éléments du calcul se déplaçant de manière irrégulière sur le support physique. Le développement de ce type d'architecture matérielle et de l'algorithmique associée demande de très nombreuses simulations.

IX.6 Perspectives à moyen et long terme

Enfin, pour clore ce chapitre, nous évoquerons plusieurs directions de recherche sur le long terme ouvertes par ce travail.

Les collections de dimensions supérieures. Nous avons évoqués dans la section I.4.3 que le bon cadre théorique pour la formalisation des collections topologiques était celui de la topologie algébrique, et que la structure des interactions correspondait à un complexe simplicial. Il est possible à partir de là de généraliser les collections topologiques et de considérer qu'elles ne correspondent plus à des graphes valués mais à des *complexes de chaînes*. Cet objet mathématique permet de décrire un espace et d'attacher à chaque brique élémentaires de cet espace une valeur.

Cette étude a été entreprise dans le travail en cours de la thèse d'Antoine Spicher. Cependant, cette généralisation, qui permet de coder très simplement les lois de la physique, pose du point de vue du typage et de l'analyse des motifs, de très nombreuses questions. Par exemple, peut-on s'assurer par un système de types, que la topologie d'une collection ne changera pas par l'application d'une transformation ? Cette question est sans doute trop difficile. Mais la topologie algébrique a développé plusieurs invariants : peut-on alors formuler des conditions nécessaires et/ou suffisantes sur une collection pour vérifier que ces invariants soient conservés à travers une transformation ?

L'apport de la topologie algébrique ne se réduit pas à la construction d'espace de plus grande dimension. Une piste est l'utilisation des notions de fibres et de produits amalgamés pour décrire la structure des chemins spécifiés par un motif dans une collection topologique. Toutes les informations sur cette structure sont en effet susceptibles de faciliter la recherche des occurences d'un chemin.

Une logique des programmes MGS. Dans ce travail de thèse, il est principalement question de programmes. Cependant, un langage de programmation devrait être accompagné d'outils de preuve permettant d'établir des propriétés vérifiées par un programme. Par exemple, l'une des grande force des types algébriques est que l'on peut utiliser un principe d'induction qui permet de raisonner dessus.

Le projet Gamma est un très bon exemple qui montre que de tels outils de preuve peuvent être développés, y compris pour des structures de données « exotiques ».

Un autre exemple de cette ligne d'attaque est donné par les travaux sur les algèbres de programmes [BDM97]. Par exemple, [Bir84] établit que sur les listes, la composition de deux *map* est toujours un *map* et plus généralement, que la composition de deux homomorphismes de listes est encore un homomorphisme de listes (loi de fusion). Ils nous semble que cette approche est particulièrement attractive pour l'étude des transformations : quelles sont leurs propriétés algébriques, à quelle classe de règles doit-on se restreindre pour s'assurer d'une propriété donnée, etc.

Transformation et réécriture. Nous avons évoqués dans le premier chapitre qu'une transformation spécifiait un certain système de réduction abstrait. Bien que nos collections topologiques puissent être assimilées à des graphes, une transformation ne correspond pas à une réécriture de graphe.

Nous espérons pouvoir étudier les relations entre les transformations et la réécriture. Nous avons vu que la notion de transformation permettait par exemple de définir une notion de

réécriture de tableau. A l'inverse, nous sommes convaincu que des concepts utiles, définis et étudiés dans le domaine de la réécriture, peuvent nous permettre de mieux comprendre et de maîtriser les transformations. Par exemple, peut-on définir une notion d'ordre bien fondé à partir des motifs, à quoi correspond un motif linéaire, peut-on relier les transformation à une théorie équationnelle, etc.

Cette problématique ouvre aussi le domaine de la réécriture aux applications qui ont motivé initialement le développement du projet MGS : les applications de simulation de systèmes dynamiques. Plusieurs éléments [GMM04] permettent de penser que l'application de notions de réécriture à la modélisation et la simulation de systèmes dynamiques est un domaine très prometteur. De ce point de vue, les notions de confluence, de terminaison ou de normalisation sont sans doute moins importantes que des propriétés comme l'atteignabilité ou la linéarité. Nous sommes convaincus qu'il y a là l'opportunité de développer de nouveaux concepts et de nouveaux outils en tirant parti de notre connaissance des systèmes de réécriture, et en prenant en compte ces nouvelles applications et ces nouveaux objets.

-oOo-

Annexe A

Correction de la combinatorisation

Dans cette annexe, nous montrons que la combinatorisation proposée au chapitre V est correcte sur un langage minimal. Nous considèrerons le λ_v-calcul (λ-calcul avec appel par valeurs) tel que présenté dans [GD92] auquel nous ajouterons la conditionnelle *if-then-else* et les traits impératifs. Nous montrerons qu'un terme de ce calcul est *équivalent* à sa forme combinatorisée.

A.1 Le langage considéré

A.1.1 Syntaxe

Nous définissons ici la syntaxe du langage que nous considérons.

Domaines syntaxiques :

$$\begin{array}{rr} x \in \mathit{Vars} & \text{(variables)} \\ c \in \mathit{Consts} & \text{(constantes)} \\ op \in \mathit{Fun\text{-}Consts} & \text{(fonctions constantes/opérateurs)} \\ V \in \mathit{Vals} & \text{(valeurs)} \\ F \in \mathit{Funs} & \text{(fonctions)} \\ M \in \Lambda_{vs} & \text{(termes)} \\ m \in \mathit{Adds} & \text{(adresses mémoire)} \end{array}$$

Syntaxe :

$$\begin{array}{rl} F & ::= op \mid \texttt{fun}\ x \to M \mid get_m \mid set_m \\ V & ::= x \mid c \mid F \\ M & ::= V \mid M\ M \mid \texttt{if}\ M\ \texttt{then}\ M\ \texttt{else}\ M \end{array}$$

Pour chaque adresse mémoire m, get_m est l'opérateur d'accès à la valeur en m et set_m est l'opérateur d'affectation en m. L'ensemble des constantes contient une constante particulière notée ( ) et prononcée « *void* » en anglais.

A.1.2 État mémoire

Un *état mémoire* s est une fonction totale de l'ensemble d'adresses mémoire vers l'ensemble de valeurs. On note $s + \{m \leftarrow V\}$ l'état mémoire qui vaut vaut $s(m')$ en m' si $m' \neq m$ et V sinon.

A.1.3 Réduction

Nous définissons à présent la sémantique de notre langage. Nous commençons par définir les réductions possibles en tête de terme. La relation de réduction élémentaire en tête, notée $\xrightarrow{\varepsilon}$ indique comment réduire un terme dans un état mémoire pour donner un nouveau terme dans un nouvel état mémoire.

$$\begin{array}{rcll}
(\texttt{fun}\ x \to M)\ V\ /\ s & \xrightarrow{\varepsilon} & M\{x \leftarrow v\}\ /\ s & (\beta) \\
set_m\ V\ /\ s & \xrightarrow{\varepsilon} & (\,)\ /\ s + \{m \leftarrow V\} & (\delta_{set-m}) \\
get_m\ V\ /\ s & \xrightarrow{\varepsilon} & s(m)\ /\ s & (\delta_{get-m}) \\
op\ V\ /\ s & \xrightarrow{\varepsilon} & \delta(op, V)\ /\ s \ \text{ si } \delta(op, V) \text{ existe} & (\delta) \\
\texttt{if}\ \mathit{true}\ \texttt{then}\ M_1\ \texttt{else}\ M_2\ /\ s & \xrightarrow{\varepsilon} & M_1\ /\ s & (\mathit{cond} - \mathit{true}) \\
\texttt{if}\ \mathit{false}\ \texttt{then}\ M_1\ \texttt{else}\ M_2\ /\ s & \xrightarrow{\varepsilon} & M_2\ /\ s & (\mathit{cond} - \mathit{false})
\end{array}$$

On peut remarquer que l'application d'un opérateur ne modifie pas l'état mémoire.

La réduction d'un terme peut également avoir lieu en profondeur dans celui-ci. La règle suivante définit la relation de réduction et force les réductions élémentaires à n'avoir lieu qu'à certains endroits autorisés. Ainsi, pour réduire une application, il n'est autorisé de réduire le membre droit de celle-ci que lorsque le membre gauche est une valeur. On force donc l'évaluation d'une application à commencer par réduire le membre gauche de celle-ci.

$$\frac{M_1\ /\ s_1 \ \xrightarrow{\varepsilon}\ M_2\ /\ s_2}{\Gamma(M_1)\ /\ s_1 \ \to\ \Gamma(M_2)\ /\ s_2}(\text{contexte})$$

$$\begin{array}{rcl}
\Gamma & ::= & \bullet \\
& | & \Gamma\ M \\
& | & V\ \Gamma \\
& | & \texttt{if}\ \Gamma\ \texttt{then}\ e_1\ \texttt{else}\ e_2
\end{array}$$

Cette définition des contexte garantit également que la branche *then* et *else* ne sont pas réduites avant que la condition n'ait été évaluée et que le choix d'évaluer l'une des deux branches n'ait été fait avec la règle $(\mathit{cond} - \mathit{false})$ ou $(\mathit{cond} - \mathit{true})$.

A.1.4 Évaluation

On dit qu'une expression M est en *forme normale* lorsqu'il n'existe pas de couple M'/s' tel que $M\ /\ s\ \to\ M'\ /\ s'$ (cette propriété est indépendante du choix de s).

Évaluer un terme consiste à le réduire jusqu'à obtenir une forme normale. Lorsque la réduction d'une expression (dans un état mémoire s) donne une forme normale qui n'est pas une valeur, on dira que son évaluation (dans s) bloque.

A.1.5 Combinatorisation

On note [.] la fonction de combinatorisation. Celle-ci procède à la SK-traduction simple et à la « curryfication » de la conditionnelle. Les combinateurs I, K, S et IF sont considérés comme des opérateurs dans la définition de la combinatorisation ci-dessous. En revanche, pour la réduction, on les considèrera comme des abstractions pour simplifier la présentation. Ceci ne

pose pas de problème car en tant qu'abstraction ou en tant qu'opérateurs, il s'agit de valeurs. La combinatorisation est définie inductivement comme suit :

$$
\begin{aligned}
[C] &= C && \text{(comb-atom)}\\
[M\,N] &= [M]\,[N] && \text{(comb-app)}\\
[\texttt{fun}\ x \to x] &= I && \text{(comb-}I\text{)}\\
[\texttt{fun}\ x \to C] &= K\,C && \text{(comb-}K\text{)}\\
[\texttt{fun}\ x \to (M\,N)] &= S\,[\texttt{fun}\ x \to M]\,[\texttt{fun}\ x \to N] && \text{(comb-}S\text{)}\\
[\texttt{if}\ M_1\ \texttt{then}\ M_2\ \texttt{else}\ M_3] &= [IF\,M_1\,(\texttt{fun}\ \_ \to M_2)\,(\texttt{fun}\ \_ \to M_3)] && \text{(comb-}IF\text{)}\\
[\texttt{fun}\ x \to \texttt{fun}\ y \to M] &= [\texttt{fun}\ x \to [\texttt{fun}\ y \to M]] && \text{(comb-prof)}
\end{aligned}
$$

où C est une constante, un opérateur, get_m, set_m ou une variable (différente de x dans la règle (comb-K)). Le symbole $\_$ dénote une variable fraîche. Notons que la combinatorisation ne préserve pas les valeurs (si V est une valeur, $[V]$ ne l'est pas forcément). Les abstractions correspondant aux combinateurs sont les suivantes (on rappelle que ( ) est une constante).

$$I : \texttt{fun}\ x \to x$$

$$K : \texttt{fun}\ x \to \texttt{fun}\ y \to x$$

$$S : \texttt{fun}\ f \to \texttt{fun}\ g \to \texttt{fun}\ x \to (f\ x)\ (g\ x)$$

$$IF : \texttt{fun}\ x \to \texttt{fun}\ y \to \texttt{fun}\ z \to \texttt{if}\ x\ \texttt{then}\ y\ (\,)\ \texttt{else}\ z\ (\,)$$

A.2 Relation d'équivalence

Nous définissons à présent une relation d'équivalence sur les termes. Certaines règles définissant cette équivalence sont reprises de [GD92]. Nous commençons par donner quelques propriétés usuelles :

$$
\begin{aligned}
M &\simeq M && && \text{(réflexivité)}\\
M \simeq N &\Rightarrow N \simeq M && && \text{(symétrie)}\\
M \simeq M' \text{ et } M' \simeq M'' &\Rightarrow M \simeq M'' && && \text{(transitivité)}\\
M \simeq M' &\Rightarrow M\,N \simeq M'\,N && && \text{(app-comp-1)}\\
M \simeq M' &\Rightarrow N\,M \simeq N\,M' && && \text{(app-comp-2)}\\
M \simeq M' &\Rightarrow \texttt{if}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2 \simeq \texttt{if}\ M'\ \texttt{then}\ N_1\ \texttt{else}\ N_2 && && \text{(cond-comp-1)}\\
M \simeq M' &\Rightarrow \texttt{if}\ N_1\ \texttt{then}\ M\ \texttt{else}\ N_2 \simeq \texttt{if}\ N_1\ \texttt{then}\ M'\ \texttt{else}\ N_2 && && \text{(cond-comp-2)}\\
M \simeq M' &\Rightarrow \texttt{if}\ N_1\ \texttt{then}\ N_2\ \texttt{else}\ M \simeq \texttt{if}\ N_1\ \texttt{then}\ N_2\ \texttt{else}\ M' && && \text{(cond-comp-3)}
\end{aligned}
$$

$$M \simeq N \Rightarrow \texttt{fun}\ x \to M \simeq \texttt{fun}\ x \to N \quad (eq-\xi)$$

Deux fonctions sont équivalentes si elles sont équivalentes « en tout point » (règle reprise de [GD92]) :

$$F_1\ x \simeq F_2\ x \Rightarrow F_1 \simeq F_2 \quad (\ x \notin FV(F_1) \cup FV(F_2)\ ) \quad (eq-\zeta')$$

Si un terme M se réduit en un terme N sans changer l'état mémoire alors M et N sont équivalents :

$$(\forall s.\ M\ /\ s\ \rightarrow\ N\ /\ s) \Rightarrow M \simeq N \quad (eq - red)$$

Si deux expressions dans un même état mémoire se réduisent en deux expressions équivalentes entre elles *en donnant un même état mémoire*, alors elles sont équivalentes :

$$(\forall s\ \exists s'.\ M_1\ /\ s\ \rightarrow\ N_1\ /\ s' \text{ et } M_2\ /\ s\ \rightarrow\ N_2\ /\ s' \text{ et } N_1 \simeq N_2) \Rightarrow M_1 \simeq M_2 \quad (eq - mod - red)$$

L'équivalence est définie comme la plus petite relation satisfaisant les propriétés données ci-dessus.

Les équivalences engendrées par une étape de réduction s'étendent simplement à plusieurs étapes de réduction par transitivité :

Lemme 3 (*eq-*$\beta*$) $(\forall s\ \exists s'.\ M\ /\ s\ \rightarrow^*\ N\ /\ s) \Rightarrow M \simeq N$

Lemme 4 (*eq-mod-red*$*$)

$$(\forall s\ \exists s'.\ M_1\ /\ s\ \rightarrow^*\ N_1\ /\ s' \text{ et } M_2\ /\ s\ \rightarrow^*\ N_2\ /\ s' \text{ et } N_1 \simeq N_2) \Rightarrow M_1 \simeq M_2$$

Notons que la relation d'équivalence que nous définissons ici ne requière qu'une notion d'identité simple (ou syntaxique) sur les états mémoire. Par exemple, dans la règle de réduction (β), l'état avant réduction est *identique* à l'état après réduction (il s'agit du même objet mathématique). Cette définition peut paraître restrictive mais elle est suffisante pour montrer que la combinatorisation est correcte.

A.3 Correction

À présent nous énonçons la correction de la combinatorisation puis nous la démontrons.

Lemme 5 (Correction de la combinatorisation) Pour tout M on a $M \simeq [M]$.

A.3.1 Preuve

Nous allons montrer ceci par induction sur la définition de la fonction de combinatorisation [.].

Expression atomique : règle (comb-atom). Si M est une constante, un opérateur, set_m, get_m ou une variable alors $[M]$ vaut M par définition.

Application : règle (comb-app). Si M est de la forme $M_1\ M_2$ alors $[M]$ vaut $[M_1]\ [M_2]$. Par hypothèse d'induction, on a $[M_1] \simeq M_1$ et $[M_2] \simeq M_2$. Donc, par remplacement d'égal à égal (règles (app-comp-1) et (app-comp-2)), on a bien $[M_1\ M_2] \simeq M_1\ M_2$.

Conditionnelle : règle (comb-*IF*). Si M est de la forme `if` M_1 `then` M_2 `else` M_3 alors, $[M]$ vaut $IF\ [M_1]\ [\texttt{fun}\ x \rightarrow M_2]\ [\texttt{fun}\ x \rightarrow M_3]$ où x est fraîche. Pour montrer que M et $[M]$ sont équivalentes, utilisons la règle $(eq - \beta)$. Nous allons montrer que le résultat de l'évaluation de M dans un état mémoire s est équivalent au résultat de l'évaluation de $[M]$ dans s (*i.e.* les valeurs résultantes sont équivalentes et l'état mémoire est le même).

Réduisons M dans s. On doit commencer par réduire M_1. Trois cas sont possibles :

1. l'évaluation ne termine pas ;
2. l'évaluation bloque, *i.e.* on arrive sur une forme normale M_1' qui n'est pas une valeur et dans un état mémoire s_1' ;
3. l'évaluation termine et donne V_1 / s_1.

Dans le premier cas, l'évaluation de M dans s ne termine pas. Dans le second cas, l'évaluation de M dans s bloque dans l'état mémoire s_1'. Dans le troisième cas on a :

$$\texttt{if } M_1 \texttt{ then } M_2 \texttt{ else } M_3 \ / \ s \ \rightarrow^* \ \texttt{if } V_1 \texttt{ then } M_2 \texttt{ else } M_3 \ / \ s_1 \tag{1}$$

À présent, regardons $[M]$. Par *H.I.* on a $[M_1] \simeq M_1$, $[\texttt{fun } x \rightarrow M_2] \simeq \texttt{fun } x \rightarrow M_2$ et $[\texttt{fun } x \rightarrow M_3] \simeq \texttt{fun } x \rightarrow M_3$. Donc :

$$IF\ [M_1]\ [\texttt{fun } x \rightarrow M_2]\ [\texttt{fun } x \rightarrow M_3] \simeq IF\ M_1\ (\texttt{fun } x \rightarrow M_2)\ (\texttt{fun } x \rightarrow M_3) \tag{2}$$

Dans $IF\ M_1\ (\texttt{fun } x \rightarrow M_2)\ (\texttt{fun } x \rightarrow M_3)$, l'expression M_1 est réduite en premier (IF est une valeur). Nous retrouvons les trois issues possibles pour l'évaluation de M_1. Si l'évaluation de M_1 ne termine pas, alors l'évaluation de $[M]$ ne termine pas[1]. Si l'évaluation de M_1 bloque dans l'état s_1' alors l'évaluation de $[M]$ bloque également dans cet état. Dans le cas restant, l'évaluation de M_1 donne V_1 dans l'état s_1 et on a donc :

$$IF\ M_1\ (\texttt{fun } x \rightarrow M_2)\ (\texttt{fun } x \rightarrow M_3)\ /\ s \ \rightarrow^* \ IF\ V_1\ (\texttt{fun } x \rightarrow M_2)\ (\texttt{fun } x \rightarrow M_3)\ /\ s_1 \tag{3}$$

Les expressions IF et V_1 étant des valeurs, l'application de gauche peut être réduite (β-réduction car IF est en fait une abstraction) :

$$IF\ V_1\ (\texttt{fun } x \rightarrow M_2)\ (\texttt{fun } x \rightarrow M_3)\ /\ s_1 \ \rightarrow \ E_1\ (\texttt{fun } x \rightarrow M_2)\ (\texttt{fun } x \rightarrow M_3)\ /\ s_1 \tag{4}$$

où E_1 est $\texttt{fun } y \rightarrow \texttt{fun } z \rightarrow \texttt{if } V_1 \texttt{ then } y\ (\,)\ \texttt{else } z\ (\,)$. Les expressions E_1 et $(\texttt{fun } x \rightarrow M_2)$ étant des valeurs, l'application est réduite :

$$E_1\ (\texttt{fun } x \rightarrow M_2)\ (\texttt{fun } x \rightarrow M_3)\ /\ s_1 \ \rightarrow \ E_2\ (\texttt{fun } x \rightarrow M_3)\ /\ s_1 \tag{5}$$

où E_2 est $\texttt{fun } z \rightarrow \texttt{if } V_1 \texttt{ then } (\texttt{fun } x \rightarrow M_2)\ (\,)\ \texttt{else } z\ (\,)$. Les expressions E_2 et $\texttt{fun } x \rightarrow M_3$ étant des valeurs, l'application de gauche peut être réduite :

$$E_2\ (\texttt{fun } x \rightarrow M_3)\ /\ s_1 \ \rightarrow \ \texttt{if } V_1 \texttt{ then } (\texttt{fun } x \rightarrow M_2)\ (\,)\ \texttt{else } (\texttt{fun } x \rightarrow M_3)\ (\,)\ /\ s_1 \tag{6}$$

Or on a $(\texttt{fun } x \rightarrow M_2)\ (\,) \simeq M_2\{x \leftarrow (\,)\}$ et comme x n'apparait pas dans M_2, $M_2\{x \leftarrow (\,)\}$ vaut M_2. De même, $(\texttt{fun } x \rightarrow M_3)\ (\,) \simeq M_3$. Donc, en remplaçant d'égal à égal (règles (cond-comp-2) et (cond-comp-3)) on a :

$$\texttt{if } V_1 \texttt{ then } (\texttt{fun } x \rightarrow M_2)\ (\,) \texttt{ else } (\texttt{fun } x \rightarrow M_3)\ (\,) \simeq \texttt{if } V_1 \texttt{ then } M_2 \texttt{ else } M_3 \tag{7}$$

Par (1) nous avons :

$$M \ / \ s \ \rightarrow^* \ \texttt{if } V_1 \texttt{ then } M_2 \texttt{ else } M_3 \ / \ s_1$$

et par (3), (4), (5) et (6) nous avons :

$$IF\ M_1\ (\texttt{fun } x \rightarrow M_2)\ (\texttt{fun } x \rightarrow M_3)\ /\ s \ \rightarrow^* \ \texttt{if } V_1 \texttt{ then } (\texttt{fun } x \rightarrow M_2)\ (\,) \texttt{ else } (\texttt{fun } x \rightarrow M_3)\ (\,)\ /\ s_1$$

donc la règle $(eq - mod - red)$ permet de déduire que :

$$M \simeq IF\ M_1\ (\texttt{fun } x \rightarrow M_2)\ (\texttt{fun } x \rightarrow M_3)$$

en considérant également (7). Or, on a vu que par (2) nous avons :

$$[M] \simeq IF\ M_1\ (\texttt{fun } x \rightarrow M_2)\ (\texttt{fun } x \rightarrow M_3)$$

Donc on a bien $[M] \simeq M$.

[1] Dans ce cas et dans le cas où l'évaluation bloque, il faudrait compléter la preuve pour montrer l'équivalence.

Abstraction : règle (comb-K). M est de la forme $\texttt{fun}\ z \to C$ où C est soit une variable différente de z, soit un opérateur, soit un set_m, soit un get_m. Dans ce cas, $[M]$ vaut $K\,C$. On cherche à montrer que $\texttt{fun}\ z \to C \simeq (\texttt{fun}\ x \to \texttt{fun}\ y \to x)\,C$. Or on a :

$$(\texttt{fun}\ x \to \texttt{fun}\ y \to x)\,C\ /\ s\ \to\ \texttt{fun}\ y \to C\ /\ s$$

(on suppose ici que C n'est pas la variable y pour éviter les considérations sur l'alpha-conversion lors de l'application des substitutions). Donc on a $(\texttt{fun}\ x \to \texttt{fun}\ y \to x)\,C \simeq \texttt{fun}\ y \to C$. Il reste donc à montrer que $\texttt{fun}\ y \to C \simeq \texttt{fun}\ z \to C$. Nous avons supposé que C n'est pas y et C ne peut être z. Par conséquent, on peut montrer par la règle $(eq-\zeta')$ que cette équivalence est bien vérifiée. Ceci montre que si $[M]$ est de la forme $K\,C$ alors $M \simeq [M]$.

Abstraction : règle (comb-I). M est de la forme $\texttt{fun}\ y \to y$ et $[M]$ vaut I. Or I vaut $\texttt{fun}\ x \to x$. On montre simplement que $\texttt{fun}\ x \to x \simeq \texttt{fun}\ y \to y$ par la règle $(eq-\zeta')$. Ceci montre que si $[M]$ est de la forme I alors $M \simeq [M]$.

Abstraction : règle (comb-S). M est de la forme $\texttt{fun}\ x \to (N_1\,N_2)$, on pose $N_1' = [\texttt{fun}\ x \to N_1]$ et $N_2' = [\texttt{fun}\ x \to N_2]$. Dans ce cas, $[M]$ vaut $S\,N_1'\,N_2'$. Par hypothèse d'induction, on a $[\texttt{fun}\ x \to N_1] \simeq \texttt{fun}\ x \to N_1$ et $[\texttt{fun}\ x \to N_2] \simeq \texttt{fun}\ x \to N_2$. Donc, $[M] \simeq S\,(\texttt{fun}\ x \to N_1)\,(\texttt{fun}\ x \to N_2)$. Il nous faut donc montrer que $\texttt{fun}\ x \to (N_1\,N_2) \simeq S\,(\texttt{fun}\ x \to N_1)\,(\texttt{fun}\ x \to N_2)$.

Commençons par réduire le terme de droite. Les termes S et $\texttt{fun}\ x \to N_1$ sont des valeurs (S est une abstraction) donc on peut réduire l'application gauche :

$$S\,(\texttt{fun}\ x \to N_1)\,(\texttt{fun}\ x \to N_2)\ /\ s\ \to\ E_1\,(\texttt{fun}\ x \to N_2)\ /\ s \tag{8}$$

où E_1 est $(\texttt{fun}\ g \to \texttt{fun}\ z \to ((\texttt{fun}\ x \to N_1)\ z)\ (g\ z))$. Là aussi les deux parties de l'application sont des valeurs et on peut donc la réduire :

$$E_1\,(\texttt{fun}\ x \to N_2)\ /\ s\ \to\ E_2\ /\ s \tag{9}$$

où E_2 est $\texttt{fun}\ z \to ((\texttt{fun}\ x \to N_1)\ z)\ ((\texttt{fun}\ x \to N_2)\ z)$. On a donc par (8) et (9) :

$$S\,(\texttt{fun}\ x \to N_1)\,(\texttt{fun}\ x \to N_2) \simeq \texttt{fun}\ z \to ((\texttt{fun}\ x \to N_1)\ z)\ ((\texttt{fun}\ x \to N_2)\ z)$$

Il reste donc à montrer que :

$$\texttt{fun}\ x \to (N_1\,N_2) \simeq \texttt{fun}\ z \to ((\texttt{fun}\ x \to N_1)\ z)\ ((\texttt{fun}\ x \to N_2)\ z)$$

Montrons ceci par la règle $(eq-\zeta')$). Nous appliquerons les deux termes à la variable fraîche X. Commençons par le terme de droite :

$$(\texttt{fun}\ z \to ((\texttt{fun}\ x \to N_1)\ z)\ ((\texttt{fun}\ x \to N_2)\ z))\,X\ /\ s\ \to\ ((\texttt{fun}\ x \to N_1)\ X)\ ((\texttt{fun}\ x \to N_2)\ X)\ /\ s \tag{10}$$

(car z n'apparaît pas dans N_1 ni dans N_2). Or on a $(\texttt{fun}\ x \to N_1)\,X \simeq N_1\{x \leftarrow X\}$ car on a :

$$(\texttt{fun}\ x \to N_1)\,X\ /\ s\ \to\ N_1\{x \leftarrow X\}\ /\ s$$

De même on a $(\texttt{fun}\ x \to N_2)\,X \simeq N_2\{x \leftarrow X\}$. Donc, par remplacement d'égal à égal on a :

$$((\texttt{fun}\ x \to N_1)\ X)\ ((\texttt{fun}\ x \to N_2)\ X) \simeq (N_1\{x \leftarrow X\})\,(N_2\{x \leftarrow X\}) \tag{11}$$

Regardons à présent le terme $(\texttt{fun}\ x \to (N_1\, N_2))\, X$:

$$(\texttt{fun}\ x \to (N_1\, N_2))\, X\ /\ s\ \to\ (N_1\, N_2)\{x \leftarrow X\}\ /\ s \tag{12}$$

Le terme $(N_1\, N_2)\{x \leftarrow X\}$ est égal à $(N_1\{x \leftarrow X\})\,(N_2\{x \leftarrow X\})$ (par la définition de l'application d'une substitution). Par conséquent on a :

$$(\texttt{fun}\ x \to (N_1\, N_2))\, X \simeq (N_1\{x \leftarrow X\})\,(N_2\{x \leftarrow X\}) \tag{13}$$

Par (10), (11), (12) et (13) on a donc :

$$(\texttt{fun}\ x \to (N_1\, N_2))\, X \simeq (\texttt{fun}\ z \to ((\texttt{fun}\ x \to N_1)\ z)\ ((\texttt{fun}\ x \to N_2)\ z))\, X$$

Donc par $(eq - \zeta')$ on a bien :

$$\texttt{fun}\ x \to (N_1\, N_2) \simeq \texttt{fun}\ z \to ((\texttt{fun}\ x \to N_1)\ z)\ ((\texttt{fun}\ x \to N_2)\ z)$$

Cette équivalence est celle que nous cherchions à montrer. Par conséquent, si M est de la forme $\texttt{fun}\ x \to N_1\, N_2$ alors on a bien $[M] \simeq M$.

Combinatorisation en profondeur : règle (comb-prof). M est de la forme $\texttt{fun}\ x \to \texttt{fun}\ y \to N$. Par définition, $[\texttt{fun}\ x \to \texttt{fun}\ y \to N] = [\texttt{fun}\ x \to [\texttt{fun}\ y \to N]]$. Or, $[\texttt{fun}\ y \to N]$ est soit un opérateur (I), soit une application (de la forme $K\, C$ ou $S\, N_1\, N_2$). On peut donc appliquer l'hypothèse d'induction à $\texttt{fun}\ x \to [\texttt{fun}\ y \to N]$ et établir que :

$$[\texttt{fun}\ x \to [\texttt{fun}\ y \to N]] \simeq \texttt{fun}\ x \to [\texttt{fun}\ y \to N]$$

Toujours par hypothèse d'induction on a $[\texttt{fun}\ y \to N] \simeq \texttt{fun}\ y \to N$. Donc, par remplacement d'égal à égal on obtient :

$$\texttt{fun}\ x \to [\texttt{fun}\ y \to N] \simeq \texttt{fun}\ x \to \texttt{fun}\ y \to N$$

Ceci permet de montrer que $[\texttt{fun}\ x \to \texttt{fun}\ y \to N] \simeq \texttt{fun}\ x \to \texttt{fun}\ y \to N$ et donc que si M est de la forme $\texttt{fun}\ x \to \texttt{fun}\ y \to N$, alors $M \simeq [M]$.

Ce dernier cas conclue la preuve par induction.□

Bibliographie

[ACD02] Joshua J. Arulanandham, Cristian S. Calude, and Michael J. Dinneen. Bead-Sort : A natural sorting algorithm. *EATCS Bull*, 76 :153–162, 2002.

[AFFS98] Alexander Aiken, Manuel Fähndrich, Jeffrey S. Foster, and Zhendong Su. A toolkit for constructing type- and constraint-based program analyses. In *Second International Workshop on Types in Compilation (TIC'98)*, 1998.

[Aik99] Alexander Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, (35) :79–111, 1999.

[AKVW93] Alexander Aiken, Dexter Kozen, Moshe Vardi, and Edward Wimmers. The complexity of set constraints. In *Proceedings of Computer Science Logic 1993*, pages 1–17, 1993.

[ALW94] Alexander Aiken, T.K. Lakshman, and Edward Wimmers. Soft typing with conditional types. In *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 163–173, 1994.

[AW92] Alexander Aiken and Edward Wimmers. Solving systems of set constraints. In *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 329–340, 1992.

[AW93] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the Seventh ACM Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, 1993.

[AWP97] Alexander Aiken, Edward Wimmers, and Jens Palsberg. Optimal representations of polymorphic types with subtyping. In *Theoretical Aspects of Computer Software (TACS)*, 1997.

[Bak78] Theodore P. Baker. A technique for extending rapid exact-match string matching to arrays of more than one dimension. *SIAM Journal on Computing*, 7(4) :533–541, 1978.

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

[BCM88] Jean-Pierre Banâtre, Anne Coutant, and Daniel Le Métayer. A parallel machine for multiset transformation and its programming style. *Future Generation Computer Systems*, 4 :133–144, 1988.

[BDd95] Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. Intersection and union types : Syntax and semantics. *Information and Computation*, 119 :202–230, 1995.

[BDM97] Richard Bird and Oege De Moor. *Algebra of Programming*, volume 100 of *International Series in Computer Science*. Prentice Hall, 1997.

[BFM01] Jean-Pierre Banâtre, Pascal Fradet, and Daniel Le Métayer. Gamma and the chemical reaction model : Fifteen years after. *Lecture Notes in Computer Science*, 2235 :17–44, 2001.

[BFR04] Jean-Pierre Banâtre, Pascal Fradet, and Yann Radenac. High-order chemical programming style. In *Unconventional Programming Paradigms - Challenges and Research Issues for New Programming Paradigms (UPP'04)*, Mont Saint-Michel, 2004. ERCIM, University of Rennes. Pre-proceedings, the final proceedings will be published by Springer in the LNCS series in 2005.

[Bir77] Richard S. Bird. Two dimensional pattern matching. *Information Processing Letters*, 6(5) :168–170, 1977.

[Bir84] Richard S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21(3) :239–250, 1984.

[BLM92] J.-P. Banâtre and D. Le Métayer, editors. *Research directions in high-level parallel programming languages*, volume 574 of *Lecture Notes in Computer Science*. Springer Verlag, 1992.

[BM86] Jean-Pierre Banâtre and Daniel Le Métayer. A new computational model and its discipline of programming. Technical Report RR-0566, INRIA, 1986.

[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.

[Brz64] Janusz A. Brzozowski. Derivatives of regular expressions. *JACM*, 11(4) :481–494, 1964.

[BSM04] Damien Boussié, Antoine Spicher, and Olivier Michel. Simulation du déplacement du nématode *ascaris suum* en MGS. Technical report, LaMI, 2004.

[Car91] Luca Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, pages 431–507. Springer-Verlag, Berlin, 1991.

[CDE+03] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. The Maude 2.0 system. In R. Nieuwenhuis, editor, *Rewriting Techniques and Applications (RTA 2003)*, volume 2706 of *Lecture Notes in Computer Science*, pages 76–87. Springer-Verlag, 2003.

[CDG+97] Hubert Comon, Max Dauchet, Remi Gilleron, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications. Available on : `http://www.grappa.univ-lille3.fr/tata`, 1997.

[CF58] Haskell B. Curry and Robert Feys. *Combinatory Logic*, volume I. North Holland, 1958.

[CF91] Robert Cartwright and Mike Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI'91)*, pages 278–292. ACM Press, 1991.

[CF92] Robin Cockett and Tom Fukushima. About Charity. Yellow Series Report No. 92/480/18, Department of Computer Science, The University of Calgary, 1992.

[CK01] Wei-Ngan Chin and Siau-Cheng Khoo. Calculating sized types. *Higher-Order and Symbolic Computation*, 14(2/3) :261–300, 2001.

[CL89] Hubert Comon and Pierre Lescanne. Equational Problems and Disunification. *Journal of Symbolic Computation*, 7 :371–425, 1989.

[CM96] Evelyne Contejean and Claude Marché. CiME : Completion Modulo E. In Harald Ganzinger, editor, *7th International Conference on Rewriting Techniques and Applications*, volume 1103 of *Lecture Notes in Computer Science*, pages 416–419. Springer-Verlag, 1996.

[CMG03] Julien Cohen, Olivier Michel, and Jean-Louis Giavitto. Filtrage et règles de réécriture sur des structures indexées par des groupes. In *Journées Francophones des Langages Applicatifs (JFLA 2003)*. INRIA, 2003.

[Coh03a] Julien Cohen. Typage des collections topologiques hétérogènes et des transformations. Research Report LaMI-89-2003, LaMI, 2003.

[Coh03b] Julien Cohen. Typing rule-based transformations over topological collections. In J.-L. Giavitto and P.-É. Moreau, editors, *4th International Workshop on Rule-Based Programming (RULE'03)*, pages 50–66, 2003.

[Coh03c] Julien Cohen. Typing rule-based transformations over topological collections. In J.-L. Giavitto and P.-É. Moreau, editors, *Electronic Notes in Theoretical Computer Science*, volume 86. Elsevier, 2003. Revised version of [Coh03b].

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

[Coh05] Julien Cohen. Interprétation par SK-traduction et syntaxe abstraite d'ordre supérieur. In O. Michel, editor, *Journées Francophones des Langages Applicatifs (JFLA 2005)*, pages 17–34. INRIA, 2005.

[col95] collectif. *Encyclopædia Universalis*, volume 10, chapter Groupes (Mathématiques), page 987. 1995.

[Cou93] Bruno Courcelle. Réécriture de graphes : orientation bibliographique. Spring school on rewriting, Odeillo, May 1993, unpublished, available on author's web page, 1993.

[Dam94] Flemming Damm. Type inference with set theoretic type operators. Technical Report 838, IRISA, 1994.

[Dil02] Antoni Diller. Efficient multi-variate abstraction using an array representation for combinators. *Information Processing Letters*, 84(6) :311–317, 2002.

[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.

[dKL87] C. G. de Koster and Aristid Lindenmayer. Discrete and continuous models for heterocyst differentiation in growing filaments of blue-green bacteria. *Acta Biotheoretica*, 36 :249–273, 1987. Kluwer Academic.

[DM82] Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *Principles of Programming Languages*, pages 207–212, 1982.

[dR02] Daniel de Rauglaudre. *CamlP4 - Reference Manual*. INRIA, 2002.

[ES79] Manfred Eigen and Peter Schuster. *The Hypercycle : A principle of natural self-organization*. Springer, 1979.

[EST95] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *Proceedings Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)'95*, pages 169–184, 1995.

[FA96] Manuel Fähndrich and Alexander Aiken. Making set-constraint program analyses scale. In *The CP'96 Workshop on Set Constraints*, 1996.

[FA97] Manuel Fähndrich and Alexander Aiken. Program analysis using mixed term and set constraints. In *Proceedings of the 4th International Static Analysis Symposium*, pages 114–126, 1997.

[FB94a] Walter Fontana and Leo Buss. "The arrival of the fittest" : Toward a theory of biological organization. *Bulletin of Mathematical Biology*, 1994.

[FB94b] Walter Fontana and Leo Buss. What would be conserved if "the tape were played twice" ? *Proceedings of the National Academy of Sciences USA*, 91 :757–761, 1994. reprinted in : "Complexity : Metaphors, Models, and Reality", G. A. Cowan, D. Pines, and D. Meltzer, editors, pp. 223–244, Addison-Wesley, Reading, MA, 1994.

[FB96] Walter Fontana and Leo Buss. *Boundaries and Barriers*, chapter The barrier of objects : from dynamical systems to bounded organizations, pages 56–116. Addison-Wesley, 1996.

[FCB02] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic Subtyping. In *Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science*, pages 137–146. IEEE Computer Society Press, 2002.

[FF97] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming Language Design and Implementation*, pages 235–248. ACM Press, 1997.

[FFSA98] Manuel Fähndrich, Jeffrey Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 85–96, 1998.

[FG95] Jun P. Furuse and Jacques Garrigue. A label-selective lambda-calculus with optional arguments and its compilation method. Technical Report RIMS Preprint 1041, Research Institute for Mathematical Sciences, Kyoto University, 1995.

[FL87] Marc Feeley and Guy Lapalme. Using closures for code generation. *Journal of Computer Languages*, 12(1) :47–66, 1987.

[FLM98] Pascal Fradet and Daniel Le Métayer. Structured Gamma. *Science of Computer Programming*, 31(2-3) :263–289, 1998.

[FM88] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. In *Proceedings of the Second European Symposium on Programming*, pages 155–175. North-Holland Publishing, 1988.

[FSA00] Manuel Fähndrich, Zhendong Su, and Alexander Aiken. Projection merging : Reducing redundancies in inclusion constraint graphs. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 81–95, 2000.

[Fur02] Jun P. Furuse. *Extensional polymorphism : theory and application*. PhD thesis, Université Paris 7, 2002.

[Gar98] Jacques Garrigue. Programming with polymorphic variants. In *ML Workshop*, 1998.

[Gar02] Jacques Garrigue. Simple type inference for structural polymorphism. In *9th Workshop on Foundations of Object-Oriented Languages*, 2002.

[Gar04] Jacques Garrigue. Relaxing the value restriction. In *International Symposium on Functional and Logic Programming*, volume 2998 of *Lecture Notes in Computer Science*, Nara, 2004. Springer-Verlag.

[GBFM04] J.-L. Giavitto, J.-P. Banâtre, P. Fradet, and O. Michel, editors. *Pre-proceedings of Unconventional Programming Paradigms - Challenges and Research Issues for New Programming Paradigms (UPP'04)*, Mont Saint-Michel, 2004. ERCIM – NFS, University of Rennes. The final proceedings will be published by Springer in the LNCS series in 2005.

[GD92] John Gateley and Bruce F. Duba. Call-by-value combinatory logic and the lambda-value calculus. In *Mathematical Foundations of Programming Semantics, 7th International Conference, 1991*, volume 598 of *Lecture Notes in Computer Science*, pages 41–53. Springer, 1992.

[GGMP02] Jean-Louis Giavitto, Christophe Godin, Olivier Michel, and Przemislaw Prusinkiewicz. *Biological Modeling in the Genomic Context*, chapter Computational Models for Integrative and Developmental Biology. Hermes, 2002.

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

[Gia92] Jean-Louis Giavitto. Typing geometries of homogeneous collections. In *2nd International Workshop on Array Manipulation (ATABLE)*, Montréal, 1992.

[Gia03] Jean-Louis Giavitto. Topological collections, transformations and their application to the modeling and the simulation of dynamical systems. In *Rewriting Technics and Applications (RTA'03)*, volume 2706 of *Lecture Notes in Computer Science*, pages 208–233, Valencia, 2003. Springer.

[GLRT04] Frédéric Gruau, Yves Lhuillier, Philippe Reitz, and Olivier Temam. Blob computing. In *Proceedings of the first conference on Computing frontiers*, pages 125–139. ACM Press, 2004.

[GM01a] Jean-Louis Giavitto and Olivier Michel. MGS : A programming language for the transformations of topological collections. Technical Report LaMI-61-2001, LaMI, 2001.

[GM01b] Jean-Louis Giavitto and Olivier Michel. MGS : A rule-based programming language for complex objects and collections. In M. van den Brand and R. Verma, editors, *Electronic Notes in Theoretical Computer Science*, volume 59. Elsevier Science Publishers, 2001.

[GM02] 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 of *Lecture Notes in Computer Science*, pages 137–150, Himeji, Japan, 2002. Springer-Verlag.

[GMC02a] Jean-Louis Giavitto, Olivier Michel, and Julien Cohen. Pattern-matching and rewriting rules for group indexed data structures. *ACM SIGPLAN Notices*, 37(12) :76–87, 2002.

[GMC02b] Jean-Louis Giavitto, Olivier Michel, and Julien Cohen. Une présentation du langage MGS (tutoriel). Technical Report LaMI-84-2002, LaMI, 2002.

[GMC03] Jean-Louis Giavitto, Olivier Michel, and Julien Cohen. Accretive rules in Cayley P Systems. In *Membrane Computing : International Workshop, WMC-CdeA 2002*, volume 2597 of *Lecture Notes in Computer Science*, pages 319–338. Springer-Verlag Heidelberg, 2003.

[GMCS04a] Jean-Louis Giavitto, Olivier Michel, Julien Cohen, and Antoine Spicher. Computation in space and space in computation. Research Report LaMI-103-2004, LaMI, 2004. Long version of [GMCS04b].

[GMCS04b] Jean-Louis Giavitto, Olivier Michel, Julien Cohen, and Antoine Spicher. Computation in space and space in computation. In *Unconventional Programming Paradigms - Challenges and Research Issues for New Programming Paradigms (UPP'04)*, Mont Saint-Michel, 2004. ERCIM, University of Rennes. Pre-proceedings, the final proceedings will be published by Springer in the LNCS series in 2005.

[GMM04] Jean-Louis Giavitto, Grant Malcolm, and Olivier Michel. Rewriting systems and the modelling of biological systems. *Comparative and Functional Genomics*, 5 :95–99, 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.

[HD97] John Hatcliff and Olivier Danvy. Thunks and the λ-calculus. *Journal of Functional Programming*, 7(3) :303–319, 1997.

[Hei94] Nevin Heintze. Set-based analysis of ML programs. In *Proceedings of the 1994 ACM conference on LISP and Functional Programming*, pages 306–317. ACM Press, 1994.

[Hen93] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(2) :253–289, 1993.

[Hen94] Michael Henle. *A combinatorial introduction to topology*. Dover publications, New-York, 1994.

[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.

[HJ03] Ralph Hinze and Johan Jeuring. Generic Haskell : practice and theory. In *Generic Programming*, volume 2793 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[HM95] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 130–141. ACM Press, 1995.

[HPS96] John Hugues, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Symposium on Principles of Programming Languages*, pages 410–423, 1996.

[HS86] J. Roger Hindley and Jonathan P. Seldin. *An Introduction to Combinators and the λ-calculus*. Cambrigde University Press, 1986.

[Hug82] John Hughes. Super Combinators—A New Implementation Method for Applicative Languages. In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*, pages 1–10, Pittsburgh, 1982.

[Itk76] Uri Itkis. *Control Systems of Variable Structure*. Wiley, 1976.

[Jam93] Max Jammer. *Concepts of Space : The History of Theories of Space in Physics (third, enlarged edition)*. Dover Publications, 1993. first published in 1954.

[Jan00] Patrik Jansson. *Functional Polytypic Programming*. PhD thesis, Chalmers University, 2000.

[Jay04] C. Barry Jay. The pattern calculus. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(6) :911–937, 2004.

[JBM98] C. Barry Jay, Gianna Bellè, and Eugenio Moggi. Functorial ML. *Journal of Functional Programming*, 8(6) :573–619, 1998.

[JC94] C. Barry Jay and J. Robin B. Cockett. Shapely types and shape polymorphism. In D. Sannella, editor, *Programming Languages and Systems - ESOP '94 : 5th European Symposium on Programming, Edinburgh, U.K.*, volume 788 of *Lecture Notes in Computer Science*, pages 302–316. Springer Verlag, 1994.

[Jeu92] Johan Jeuring. The derivation of a hierarchy of algorithms for pattern matching on arrays. In G. Hains and L. M. R. Mullin, editors, *Proceedings ATABLE-92, Second international Workshop on Array Structures*, 1992.

[Jeu00] J. Jeuring, editor. *Workshop on Generic Programming*. Utrecht University, 2000.

[JJ96] Johan Jeuring and Patrik Jansson. Polytypic programming. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming, Second International School*, volume 1129 of *Lecture Notes in Computer Science*, pages 68–114. Springer-Verlag, 1996.

[JP99] Trevor Jim and Jens Palsberg. Type inference in systems of recursive types with subtyping. Available on authors' web page, 1999.

[JS89] Simon L. Peyton Jones and Jon Salkild. The spineless tagless G-machine. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 184–201. ACM Press, 1989.

[JS97] C. Barry Jay and Milan Sekanina. Shape checking of array programs. In *Proceedings of CATS'97 (Computing : The Australasian Theory Symposium)*, pages 113–121, 1997.

[KM89] Paris C. Kanellakis and John C. Mitchell. Polymorphic unification and ML typing. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 105–115. ACM Press, 1989.

[Kod] John Kodumal. BANSHEE : A toolkit for building constraint-based analyses. Web page. `http://banshee.sourceforge.net`.

[Lan65a] Peter J. Landin. An abstract machine for designers of computing languages. In *Proceedings of the IFIP Congress*, pages 438–439, 1965.

[Lan65b] Peter J. Landin. Correspondence between ALGOL 60 and Church's lambda-notation : part I. *Commununications of the ACM*, 8(2) :89–101, 1965.

[Lar02] Valérie Larue. Structures de données indexées par un groupe : représentation graphique et extension au cas non abélien. Master's thesis, Université d'Évry Val d'Essonne, 2002.

[Ler92] Xavier Leroy. Unboxed objects and polymorphic typing. In *19th Symposium Principles of Programming Languages*, pages 177–188. ACM Press, 1992.

[Ler97] Xavier Leroy. The effectiveness of type-based unboxing. In *Workshop on Types in Compilation '97*. Technical report BCCS-97-03, Boston College, Computer Science Department, 1997.

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

[LM94] Xavier Leroy and Michel Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4) :431–463, 1994.

[Man01] Vincenzo Manca. Logical string rewriting. *Theoretical Computer Science*, 264(1) :25–51, 2001.

[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 (JFLA'02)*, Anglet (France), 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.

[MJ04] Olivier Michel and Florent Jacquemard. An analysis of the Needham-Schroeder public-key protocol with MGS. In G. Mauri, G. Păun, and C. Zandron, editors, *Preproceedings of the Fifth Workshop on Membrane Computing (WMC5)*, pages 295–315. EC MolConNet - Universita di Milano-Bicocca, 2004.

[MK98] Pierre-Étienne Moreau and Hélène Kirchner. A compiler for rewrite programs in associative-commutative theories. In *Proceedings of Algebraic and Logic Programming - Programming Language Implementation and Logic Programming, ALP/PLI LP'98*, volume 1490 of *Lecture Notes in Computer Science*, Pisa, 1998. Springer.

[Mon94] Bernard Monsuez. Polymorphic typing of heterogeneous lists. Technical report, École Normale Supérieure, 1994.

[Mor99] Pierre-Étienne Moreau. *Compilation de règles de réécriture et de stratégies non-déterministes*. PhD thesis, Université Henri Poincaré – Nancy 1, 1999.

[Mos90] Peter D. Mosses. *Handbook of Theoretical Computer Science*, volume B, chapter Denotational semantics, pages 577–631. Elsevier Science Publishers, 1990.

[MPS86] David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71 :95–130, 1986.

[MRV01] Pierre-Étienne Moreau, Christophe Ringeissen, and Marian Vittek. A pattern-matching compiler. In D. Parigot and M. G. J. van den Brand, editors, *Proceedings of the 1st International Workshop on Language Descriptions, Tools and Applications*, volume 44, Genova (Italy), 2001. Electronic Notes in Theoretical Computer Science.

[MRV03] Pierre-Étienne Moreau, Christophe Ringeissen, and Marian Vittek. A pattern matching compiler for multiple target languages. In G. Hedin, editor, *12th Conference on Compiler Construction, Warsaw (Poland)*, volume 2622 of *Lecture Notes in Computer Science*, pages 61–76. Springer-Verlag, 2003.

[MSR91] Eric Mjolsness, David H. Sharp, and John Reinitz. A connectionist model of development. *Journal of Theoretical Biology*, 152(4) :429–454, 1991.

[MW97] Simon Marlow and Philip Wadler. A practical subtyping system for Erlang. In *2nd International Conference on Functional Programming*, 1997.

[Oho95] Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6) :844–895, 1995.

[Pat94] R. Paton, editor. *Computing With Biological Metaphors*. Chapman & Hall, 1994.

[PE88] Frank Pfenning and Conal Elliot. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language Design and Implementation*, pages 199–208. ACM Press, 1988.

[PH92] Przemyslaw Prusinkiewicz and James Hanan. L-systems : from formalism to programming languages. In G. Rozenberg and A. Salomaa, editors, *Lindenmayer Systems, Impact on Theoretical Computer Science, Computer Graphics, and Developmental Biology*. Springer-Verlag, 1992.

[Pie91] Benjamin C. Pierce. Programming with intersection types, union types, and polymorphism, 1991. Avaible on author's web page.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[PJ87] Simon L. Peyton-Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[Pot98a] François Pottier. *Synthèse de types en présence de sous-typage : de la théorie à la pratique*. PhD thesis, Université Paris 7, 1998.

[Pot98b] François Pottier. A framework for type inference with subtyping. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 228–238, 1998.

[Pot01] François Pottier. Simplifying subtyping constraints : A theory. *Information & Computation*, 170(2) :153–183, 2001.

[PR03] François Pottier and Didier Rémy. The essence of ML type inference (extended version). Unpublished manuscript, 2003.

[PS94] Marc Pantel and Patrick Sallé. Typage souple pour le langage FOL. In *Journées Francophones des Langages Applicatifs (JFLA 94)*, pages 21–51. INRIA, 1994.

[PĬ00] Gheorghe Păun. Computing with membranes. *Journal of Computer and System Sciences*, 1(61) :108–143, 2000.

[PWO97] Jens Palsberg, Mitchell Wand, and Patrick O'Keefe. Type inference with non-structural subtyping. *Formal Aspects of Computing*, 9 :49–67, 1997.

[Rau03] Erik M. Rauch. Discrete, amorphous physical models. *International Journal of Theoretical Physics*, 42 :329–348, 2003.

[Rém93a] Didier Rémy. Syntactic theories and the algebra of record terms. Research Report 1869, INRIA, Rocquencourt, 1993.

[Rém93b] Didier Rémy. Type inference for records in a natural extension of ML. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993.

[Rey95] Craig Reynolds. Boids, background and update. Web page `http://www.red3d.com/cwr/boids/`, 1995.

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

[Rue92] Karl Fritz Ruehr. *Analytical and Structural Polymorphism Expressed using Patterns over Types*. PhD thesis, University of Michigan, 1992.

[Rue98] Fritz Ruehr. Structural polymorphism. In *Workshop on Generic Programming (WGP'98)*, 1998.

[SA01] Zhendong Su and Alexander Aiken. Entailment with conditional equality constraints. In *European Symposium on Programming*, pages 170–189, 2001.

[She93] Tim Sheard. Type parametric programming. Technical Report CS/E 93-018, Oregon Graduate Institute of Science and Technology, Department of Computer Science and Engineering, Portland, 1993.

[SM04] 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, 2004. Draft proceedings published as a technical report of the Institute of Computer Science of the University of Kiel. The post-proceedings will be published as an LNCS volume.

[SMG04] 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 *Lecture Notes in Computer Science*, Amsterdam, 2004. Springer.

[Spi03] Antoine Spicher. Typage et compilation de filtrage de chemins dans des collections topologiques. Master's thesis, Univerité d'Évry Val d'Essonne, 2003.

[SSS81] Edmond Schonberg, Jacob T. Schwartz, and Micha Sharir. An automatic technique for selection of data representations in SETL programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 3(2) :126–143, 1981.

[Str67] Christopher Strachey. Fundamental concepts in programming languages. Unpublished, In Proceedings of the 1967 International Summer School in Computer Programming, Copenhagen, Denmark, available as [Str00], 1967.

[Str00] Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1-2) :11–49, 2000. Republished from [Str67].

[TN87] Tommaso Toffoli and Margolus Norman. *Cellular Automata Machine*. MIT Press, Cambridge MA, 1987.

[Tur52] Alan M. Turing. The chemical basis of morphogenesis. *Philosophical Transactions of the Royal Society of London*, Series B : Biological Sciences(237) :37–72, 1952.

[Tur91] Greg Turk. Generating textures for arbitrary surfaces using reaction-diffusion. In T. W. Sederberg, editor, *Computer Graphics (SIGGRAPH '91 Proceedings)*, volume 25, pages 289–298, 1991.

[Var79] Francisco J. Varela. *Principle of Biological Autonomy*. McGraw-Hill/Appleton & Lange, 1979.

[VN66] John Von Neumann. *Theory of Self-Reproducing Automata*. University of Illinois Press, 1966.

[WA85] William W. Wadge and Edward A. Ashcroft. *Lucid, the Data flow programming language*. Academic Press U.K., 1985.

[Wil98a] Uri Wilensky. NetLogo ants model. Web page `http://ccl.northwestern.edu/netlogo/models/Ants`. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL., 1998.

[Wil98b] Uri Wilensky. NetLogo flocking model. Web page `http://ccl.northwestern.edu/netlogo/models/Flocking`. Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL., 1998.

[WMS73] M. Wilcox, G. J. Mitchison, and R. J. Smith. Pattern formation in the blue-green algae, *Anabaena*. I. Basic mechanisms. *Journal of Cell Science*, 12 :707–723, 1973.

[Wri94] Andrew K. Wright. *Practical Soft Typing for Scheme*. PhD thesis, Rice University, 1994.

[Wri95] Andrew K. Wright. Simple imperative polymorphism. *LISP and Symbolic Computation*, 8(4) :343–355, 1995.

[Xi98] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, Pittsburgh, 1998.

[YPQ58] H. P. Yockey, R. P. Platzman, and H. Quastler, editors. *Symposium on Information Theory in Biology*. Pergamon Press, New York, London, 1958.