LaMI

Laboratoire de Méthodes Informatiques

Une présentation du langage `mgs` (version 0.1)

Jean-Louis Giavitto, Olivier Michel, Julien Cohen

email(s) : `giavitto` *ou* `michel` *ou* `jcohen @lami.univ-evry.fr`

Document interne — Équipe Spécif

Décembre 2001

CNRS umr 8042 – Université d'Evry Val d'Essonne
523, Place des Terrasses
91000 Evry France

Une présentation du langage `mgs` (version 0.1)

Jean-Louis Giavitto, Olivier Michel, Julien Cohen

LaMI u.m.r. 8042 du CNRS
Université d'Evry Val d'Essone
91025 Evry Cedex, France.
[giavitto,michel]@lami.univ-evry.fr

Rapport interne, Décembre 2001

Résumé

Nous présentons de manière informelle le langage de programmation MGS. Le but de cette introduction est de familiariser le lecteur de manière très concrète avec la syntaxe du langage au travers de petits exemples. Nous supposons que le lecteur est déjà familiarisé avec les langages fonctionnels, la notion de syntaxe et d'interprète. Le lecteur intéressé par les motivations et les objectifs du projet MGS, ou bien par les formalismes et les techniques utilisées, doit se référer au rapport [GM01b].

MGS est un langage déclaratif qui propose d'unifier plusieurs paradigmes de programmation comme la transformation de multi-ensembles, les systèmes de Lindenmayer ou les automates cellulaires.

MGS enrichit un langage fonctionnel avec de nouveaux types de valeurs, les *collections topologiques*. Une collection correspond à un ensemble organisé d'éléments. L'organisation d'une collection peut se décrire topologiquement et permet de définir les voisins d'un élément. Une collection peut correspondre à une structure de données usuelle, comme la liste ou l'ensemble, ou bien à une structure plus exotique comme un tampon circulaire, une « grille vrillée », etc.

Afin de manipuler ces nouveaux objets, MGS introduit la notion de *transformation*. Une transformation est une fonction qui peut se définir par l'application simultanée de plusieurs règles locales. Une règle locale spécifie la transformation d'un élément de la collection *et de ses voisins*.

Ce mécanisme de calcul peut se décrire abstraitement de la manière suivante : remplacer dans une collection A une certaine sous-collection B qui vérifie certaines propriétés, par une autre collection C calculé à partir de B et de ses voisins dans A. Ce mécanisme est suffisament abstrait pour englober dans une même description la réécriture de chaînes, de multi-ensembles, le paradigme des automates cellulaires, etc..

Les auteurs de ce rapport peuvent être contacté à l'adresse suivante :

La.M.I., CNRS UMR 8042
Université d'Évry Val d'Essonne
Tour Évry 2 / 4eme etage
523 Place des terrasses de l'agora
91000 Évry Cedex France

Tel : +33 (0)1 60 87 39 04
Fax : +33 (0)1 60 87 37 89

Les interprètes `MGS` sont disponibles gratuitement à partir de l'URL `http://www.lami.univ-evry.fr/mgs` .

Versions de ce rapport :
- Mai 2002. Début de la traduction anglaise.
- Mars 2002. Corrections. Complétions du chapitre sur les GBF. Exemple de la normalisation des formules logiques. Chapitre sur les systèmes dynamiques à structure dynamique.
- Janvier 2002. Correction et ajout du chapitre sur les GBF.
- Décembre 2001. Brouillon de la version initiale.

Table des matières

Liste des Figures

Liste des Tables

Chapitre 1

Introduction

1.1 Un langage de programmation dedié à la modélisation et à la simulation des systèmes dynamiques à structure dynamique

`MGS` est l'acronyme de *"(encore) un **M**odèle **G**éneral de **S**imulation (de système dynamique)"*. Un des objectifs du projet `MGS` est de développer un *langage de programmation dédié* à la modélisation et la simulation de processus biologique à structure dynamique.

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

Les processus biologiques auxquels nous nous intéressons sont des systèmes dynamiques[1] hautement structurés et hiérarchiquement organisés, dont la structure varie au cours du temps et devant être calculée conjointement avec l'état du système. Ce type de systèmes est courant dans les modèles de croissance de plantes, en biologie du développement, dans les modèles cellulaires intégrant plusieurs échelles, dans les mécanismes de transport de protéines et de compartimentalisation, etc. Dans ce rapport, nous ne ferons qu'évoquer ces applications (Cf. chapitre **??**) pour nous concentrer sur les aspects purement « langage » du projet `MGS`.

1.2 Langage fonctionnel et collections

Parmi les abstractions fournies par `MGS` pour faciliter le développement de simulations de systèmes biologiques dynamiques, il y a les notions de *collections topologiques* et de *transformations*. `MGS` incarne ces notions dans le cadre d'un (petit) langage fonctionnel dynamiquement typé :

[1] Un système dynamique est décrit par un état qui évolue dans le temps. La suite des états dans le temps est appelé *trajectoire* du système. Nous supposerons ici que le temps est discret (processus à événement discret ou bien processus continu préalablement discrétisé). Le chapitre **??** précise ces notions.

– Les collections correspondent simplement à de nouveaux types de valeurs (qui sont adaptés à la représentation de l'état d'un système biologique complexe).
– Les transformations sont des fonctions qui agissent sur les collections et qui sont définies à l'aide d'une syntaxe particulière permettant de faciliter leur spécification à l'aide de filtres et de règles.

Une collection correspond à un ensemble organisé d'éléments. L'organisation d'une collection peut se décrire topologiquement et permet de définir les voisins d'un élément. Une collection peut correspondre à une structure de données usuelles, comme la liste ou l'ensemble, ou bien à une structure plus exotique comme un tampon circulaire, un tore, ou une discrétisation hexagonale du plan.

`MGS` est un langage fonctionnel, ce qui veut dire que les calculs se font en appliquant des fonctions qui transforment des valeurs en d'autres valeurs et non pas en modifiant un état global par effets de bord. Cependant, `MGS` possède des traits impératifs : il existe une forme limitée de variables qu'on peut utiliser « comme en `C` ».

Par « dynamiquement typé », nous voulons dire que la vérification du bon usage des types (par exemple on n'additionne pas un booléen et un tableau), n'est pas effectuée avant l'exécution du programme, mais au cours de celle-ci. On peut définir en `MGS` de nouveaux types : ces types sont utilisés dans la définition de nouvelles valeurs, des filtres, des règles et des transformations.

1.3 Plusieurs interprètes `mgs`

Dans ce rapport, la syntaxe `MGS` est donné à travers des exemples. Tous les exemples correspondent à des programmes `MGS` réels évalués par un interprète existant. Il existe plusieurs versions de cet interprète :

– deux anciennes versions :
  – l'une version réalisée en `C++`,
  – l'autre réalisée en `OCAML`.
– et une nouvelle version réalisée en `OCAML`, connu sour le petit nom de `1_MGS`.

Les constructions décrites dans ce rapport sont largement communes à ces interprètes, mais : (1) les anciens interprètes ne sont plus maintenu et, (2) nous faisons évoluer ce tutoriel pour décrire plus précisément la nouvelle version.

`MGS` est conçu comme un prototype permettant de développer et de tester de nouvelles idées en programmation. En conséquence, la syntaxe est mouvante et les interprètes ne sont pas très stables. Il existe ainsi de légères variations dans la syntaxe des programmes entre les deux versions. À terme, ces deux versions doivent converger. De plus, nous avons débuté le développement d'un compilateur qui doit réaliser l'union des deux versions. Mais il ne faut pas s'étonner si les informations dans ce rapport ne sont pas tout à fait à jour.

Les interprètes `MGS` sont disponibles à partir de la page :

`http://www.lami.univ-evry.fr/mgs`

ils sont développés sous Linux (distribution Debian, version *potato* et *woody*).

1.4 Programme, commande, définition et expression

Un programme `MGS` est une suite de demandes se terminant par un double point-virgule `;;` . Il y a trois type de demandes :

1. les expressions,
2. les définitions de types,
3. et les commandes,

Ces demandes peuvent être écrites directement au clavier, une fois qu'on a lancé un interprète (par exemple dans une fenêtre "xterm"). L'entré de l'interprète `MGS` est un éditeur-ligne. Cela veut dire que tant que l'utilisateur n'a pas frappé sur le "retour-chariot", il est possible d'éditer la ligne courante à l'aide de plusieurs commande de type "emacs". L'utilisateur dispose aussi d'un historique des lignes déjà entrées. La ligne courante est prise en compte et examinée par l'interprète uniquement après la frappe du retour-chariot. Cependant, le découpage en ligne est arbitraire et n'impose aucune contrainte à la syntaxe du langage. Par exemple on peut écrire

```
1 + ↩
  2 ;; 3 + 4 ↩
;;
```

Le symbole ↩ représente explictement un retour-chariot. Le marqueur `;;` indique la fin de l'expression `1 + 2` (dont l'écriture s'étend donc sur deux lignes). Sur la même ligne, débute une autre expression : `3 + 4`, dont la fin est reconnue la ligne d'après.

Dès que l'interprère reconnait la fin d'une commande, d'une définition de type ou bien d'une expression, il exécute l'action correspondante :

1. La reconnaissance de la fin d'une expression déclenche son évaluation et l'affichage du résultat.
2. Une définition de type ne déclenche pas de calculs mais augmente l'environnement d'exécution de l'interprète `MGS` afin de reconnaitre de nouvelles constructions (par exemple un nouveau type de valeur).
3. Enfin, les commandes ne font pas directement partie du langage `MGS` mais permettent d'interagir avec l'interprète : cela peut par exemple être l'affichage de la liste des objets connus de l'interprète, la diversion temporaire de l'entrée standard afin d'inclure un fichier, la sauvegarde des définitions connues, etc.

Une expression qui est entrée directement sous l'invite de l'interprète est dite *expression au top-level* ou bien *expression courante*. Une telle expression se termine par un marqueur `;;`. Par exemple,

```
mgs> 1 + (2 + 3) ;; (3 * 4) ;;
```

l'expression `1 + (2 + 3)` qui est entré après l'invite `mgs>` de l'interprète et dont la fin est signifiée par le terminateur `;;` est une expression au top-level. L'expression `(3 * 4)` est aussi une expression au top-level. Par contre, l'expression `(2 + 3)` qui est une sous-expression d'une expression au top-level, n'est pas une elle-même une expression au top-level.

1.4.1 Les commandes

Une commande débute toujours par `!` et se termine par `;;`. Une commande ne peut pas apparaitre dans une définition ou bien une expression. Une commande est généralement réduite à un seul mot clé entre `!` et `;;`. Voici la liste des commandes disponibles :

`help` permet d'afficher une aide sommaire (avec "more").

`legal` affiche la banière `MGS`, le numéro de version, etc.

`quit` ou bien `exit` permet de quitter l'interprète.

`list` liste le nom de toutes les fonctions et variables définies par l'utilisateur ou prédéfinies dans le système. Les fonctions prédéfinies sont agrémentées d'un petit commentaire, principalement pour rappeler leur action et l'ordre de leurs arguments.

`type` ou `collection` permet de lister le nom de toutes les définitions de types collections.

`record` énumère le nom des types d'enregistrement[2].

`trace` *f g ...* permet de tracer les fonctions dont le nom est *f*, *g*, etc.

`untrace ...` permet d'annuler un ordre de trace. Par exemple `untrace` *g* annule l'ordre de trace de la fonction de nom *g* (si celle-ci était tracée).

`include "fichier"` permet d'inclure un fichier (spécifié par une chaîne de caractère donnant un chemin d'accès, relatif ou absolu). A la suite de l'ordre d'inclusion, tout se passe comme si les entrées suivantes provenaient du fichier. A la fin du fichier, l'interprète redonne la main à l'utilisateur.

Cette commande permet d'éditer un programme avec son éditeur préféré, comme par exemple "emacs", puis d'inclure le programme obtenu.

`gc` affiche un résumé de l'utilisation mémoire par l'exécutif OCAML utilisé dans l'interprète `MGS` et démarre un cycle du glanneur de cellules.

`version` affiche un listing des numéro de version des fichiers source de l'interprète `MGS`. Cette information est à joindre aux rapport de bug.

1.4.2 Les définitions de types

Il y a plusieurs sortes de définitions de types : définition d'un nouveau type de collections, d'un nouveau type d'enregistrements, d'un nouveau type de flèches, etc. Ces différents types seront vues dans les chapitres suivants. Tout comme les commandes, une définition ne peut pas apparaître au sein d'une expression.

Une définition de type débute généralement par un mot clé indiquant la définition, comme par exemple *collection* ou bien *gbf*, puis par le nom du type, le symbole `=` et une expression de définition se terminant par un `;;` . Par exemple

```
collection MonEnsemble = set ;;
```

introduit un nouveau type de collection appelé `MonEnsemble` et dont la définition apparait à droite du signe `=` (ce type de données se comporte comme un ensemble). Voici une autre définition de type :

[2] le mot clef `state` utilisé précédemment est à présent obsolète

```
gbf grid3 = < u, v ; u + 33 v, 5 u >;;
```

qui introduit un nouveau type de collection appelé `grid3` et dont la définition apparait à droite du signe `=`. Le type `grid3` est un gbf. Les gbf constituent une famille particulière de collections, qui sera introduite au chapitre 9.

A la fin d'une définition, l'interprète manifeste qu'il a pris connaissance de la définition en réécrivant la définition du type et éventuellement, en donnant quelques informations supplémentaires. A la définition précédente, le système réagit en indiquant :

```
// Normalized form of gbf grid3 has 0 free generator
// and 1 cyclic generator:  <1, 165>
gbf grid3 = < u, v; u + 33*v = 0, 5*u = 0 >
```

1.4.3 Les commentaires

Dans le message précédant, le signe `//` indique que tout ce qui suit dans la ligne est un commentaire. Dans l'exemple précédent, ces commentaires sont produits par l'interprète, mais l'utilisateur peut aussi commenter ses propres programmes.

Pour commenter tout un bloc, il suffit d'inclure ce bloc entre `/*` et `*/`. Attention, on ne peut pas imbriquer les commentaires :

```
/* ici on débute un commentaire
qui se
termine plusieurs lignes plus loin */
```

Les signes `(*` et `*)` sont équivalents aux `/*` et `*/`. Les commentaires-lignes peuvent aussi débuter par \\ .

1.4.4 Les expressions

Une expression correspond à un calcul. Dès qu'une expression complète a été lue par l'interprète, elle est évaluée et le résultat est affiché en retour :

```
1 + 2 ;; ↩
3
```

Nous utilisons le signe ↩ pour indiquer la séparation entre ce qui a été tapé par l'utilisateur (l'expression `1 + 2`) et la valeur calculée et affichée par l'interprète : `3`.

Certaines expressions correspondent directement à des valeurs : ce sont les constantes du langage. On dispose par exemple
- de la valeur indéfinie : `<undef>`
- des booléens : `false` et `true`
- des entiers : `3`, `-265429908`

– des flottants : `3.1415`, `1.003e-27`, `-33E11`
– des chaînes de caractères : `"trois"`
– de l'ensemble vide : `set:()`
– des fonctions : `\x.x+1`
– etc.

Nous verrons au fur et à mesure les différentes valeurs de bases en `MGS`. Ces valeurs sont accompagnées par des *fonctions primitives* permettant de les manipuler. Par exemple la fonction `empty` permet de tester si une chaîne de caractères est vide. Les fonctions primitives sont souvent *surchargées* : cela veut dire que le même nom de fonction est utilisé pour désigner plusieurs fonctions différentes en réalité. Par exemple, `empty` permet de tester si une collection est vide, que ce soit un ensemble ou bien une liste.

Certaines fonctions admettent des arguments de n'importe quel type. C'est le cas de la fonction unaire `?` qui permet d'afficher la représentation textuelle d'une valeur. C'est aussi le cas de la fonction d'égalité `==` qui compare deux arguments quelconques[3] :

`1.3 == (2.6 / 2.0)` `//` renvoie la valeur booléenne *vrai*
`1.3 == "un point trois"` `//` renvoie la valeur booléenne faux
`1 == 1.0` `//` renvoie la valeur booléenne faux car un entier n'est jamais egal à un flottant

La fonction de différence se note : `!=` ou bien `~=` ou bien encore `=!=` :

`1.3 ~= "un point trois"` `//` renvoie la valeur booléenne *vrai*

[3]cependant un avertissement est émis si un des arguments est une fonction, car le test d'égalité entre fonction repose sur l'égalité de nom.

Chapitre 2

Les valeurs atomiques

2.1 Les valeurs atomiques et les collections

Les valeurs comme les entiers, les chaînes de caractères, les flottants, etc., sont considérées comme *indécomposables* ou *atomique*. Les valeurs qui au contraire sont décomposables, sont appelées des *collections*. Par exemple une séquence, un ensemble, un tableau, sont des collections[1]. Il est facile de savoir si une valeur est une valeur atomique ou une collection grâce à la fonction `size` :

`size`(v)

retourne `-1` si la valeur de v est atomique. Les collections sont par contre les valeurs pour lesquelles la fonction `size` retourne un entier positif ou nul. Les types atomiques disponibles actuellement sont les suivants :

1. `undef` (cf. section 2.2) ;
2. les types des valeurs scalaires : `int` et `float` (cf. section 2.3) ;
3. les chaînes de caractères : `string` (cf. section 2.4) ;
4. les fonctions : `lambda` (cf. chapitre 3) ;

De plus, n'importe quelle valeur `MGS` peut s'interpréter comme un booléen.

2.2 La valeur indéfinie

Il existe en `MGS` une valeur particularisée et qui représente « la valeur indéfinie ». Elle se note `<undef>` ou bien `nil`. Cette valeur est retournée dans certains cas d'erreur, ou bien comme valeur d'initialisation dans certaines constructions. Il est parfois utile de distinguer

[1] Cette définition ne doit pas être prise trop littéralement : par exemple un ensemble vide ne se décompose en aucun élément (il n'en contient aucun), c'est pourtant une collection. C'est que l'ensemble vide est un cas limite des valeurs ensemblistes, qui elles, se décomposent.

Inversement, une chaîne de caractères n'est pas une valeur décomposable en `MGS`, car il n'y a pas de fonction qui permette de la parcourir et de la modifier « localement ». Ce n'est donc pas une collection (mais un autre choix aurait été possible).

plusieurs valeurs indéfinies. Cela est possible en faisant suivre le mot `undef` par un commentaire :

```
<undef:  ici un commentaire-ligne se terminant par le signe inferieur >
```

On peut comparer la valeurs indéfinie avec d'autres valeurs :

```
<undef> == <undef>
```

est une expression qui renvoie un booleen *vrai*. Toutes les valeurs indéfinies sont égales, même si le commentaire est différent :

```
nil == <undef:  ces 2 formes de valeurs indefinies sont egales>
```

Mais pour toutes les autres comparaisons `<=, <, >, >=`, ou bien pour une expression arithmétique `+, -, *, /, %`, un argument prenant la valeur `undef` provoque éventuellement une erreur et force le résultat à être `undef` :

```
<undef> + 1;; ↩
<undef:  cannot add any to any>
sin(<undef>);; ↩
   Error:  dispatch function sin, case:  undef:  not defined.
<undef:  dispatch function sin failed>
```

Le commentaire accompagnant la valeur indéfinie qui est retournée précise la cause de l'erreur. Cela est particulièrement utile quand une valeur indéfinie est propagé d'appel de fonction en appel de fonction. Dans le cas de l'addition ci-dessus, aucune erreur n'est déclaré et l'évaluation se poursuit avec une valeur indéfinie. Dans le cas de la fonction mathématique `sin` (sinus), une erreur est levé et l'évaluation se termine brutalement en rendant la main au top-level avec une valeur indéfinie.

2.3 Les valeurs scalaires

Les entiers et les flottants constituent les deux types de base des valeurs scalaires. Un flottant correspond à un *double* de la norme IEEE. Un certain nombre de constantes flottantes sont prédéfinies :

```
'E          // la base des logaritmes naturels e
'LOG2E      // le logarithme en base 2 de e
'LOG10E     // le logarithme en base 10 de e
'LN2        // le logarithme naturel de 2
'LN10       // le logarithme naturel de 10
'PI         // π
'PI_2       // π/2
```

```
'PI_4       // π/4
'1_PI       // 1/π
'2_PI       // 2/π
'2_SQRT_PI  // 2/√π
'SQRT2      // √2
'SQRT1_2    // 1/√2
```

Parmi les opérations disponibles sur les scalaires, on trouve *les opérations arithmétiques* :

```
+   *   -   /   %   min   max
```

les *opérateurs relationnels* :

```
<   <=   >   >=   ==   ~=
```

les *fonctions mathématiques* dont le résultat est un flotant :

```
cos  // cosinus (en radians)          acos  // arc cosinus
sin  // sinus (en radians)            asin  // arc sinus
tan  // tangente (en radians)         atan  // arc tangente

cosh // cosinus hyperbolique
sinh // sinus hyperbolique
tanh // tangente hyperbolique

exp  // exponentielle                 log   // logarithme naturel
                                      log10 // logarithme en base 10

sqrt // racine carré
ceil // arrondie à l'entier supérieur
floor// arrondie à l'entier supérieur

abs  // valeur absolue d'un flottant ou d'un entier
```

Dans toutes ces opérations, sauf pour l'égalité `==`, les entiers sont convertis en flottants quand cela est nécessaire. On peut donc écrire des expressions mixtes comme :

```
2 + 3.14 * fabs(sqrt(5))
```

et le résultat sera un flottant.

Les fonctions suivantes permettent de convertir un entier en flottant et vice versa : `floatof` et `intof`. Ces fonctions acceptent en arguments une chaînes de caractères pour convertir cette chaîne en entier ou flottant si c'est possible (si ce n'est pas possible, une valeur indéfinie est retournée).

Il est possible de s'encquérir du statut IEEE d'un flottant grâce à la fonction `whatfloat` appliqué à un flottant r. L'expression `whatfloat(`r`)` retourne

`0` `si` $r = 0.0$ `ou` $r = -0.0$

```
1   si r est un flottant ordinaire (aucun des autres cas)
2   si r est proche de 0 et de précision réduite
3   si r = +∞ ou r = −∞
4   si r n'est pas un nombre (résultat d'une opération indéfinie)
```

2.4 Les chaînes de caractères

Une chaîne de caractères s'écrit entre deux guillemets " . La chaîne vide s'écrit "" et correspond à une chaîne avec zéro caractère. On peut passer à la ligne lors de l'écriture d'une chaîne :

```
"voici une chaine
qui se poursuit a la ligne suivante"
```

On peut aussi inclure le caractère guillemet ou bien tabulation ou bien retour-chariot en utilisant le caractère \ comme échappement :

```
"une tabulation\tsuivie d'un saut de ligne \nun \\ et un guillemet \" pour finir";; ←
une tabulation     suivie d'un saut de ligne
un \ et un guillemet " pour finir
```

Remarquer que l'anti-slash \ se littéralise lui-même.

Quand une chaine est affichée au-top level comme valeur de retour, l'affichage se fait avec guillemets :

```
"aa\
bb";; ↩
aa
bb
```

Une chaîne de caractères peut être affichée par la fonction ? qui affiche son argument sur la sortie standard (voir le chapitre sur les entrées-sorties) :

```
?("Hello world");; ↩
"Hello world"
"Hello world"
```

Le premier "Hello world" qui s'affiche est l'affichage effectuée par la fonction ?. Le deuxième est l'affichage de la valeur retournée par l'évaluation de l'expression ?("Hello world") : la fonction ? retourne son argument.

Deux chaînes de caractères peuvent être concaténées en utilisant l'operateur + :

```
"un" + "joli" + "programme";; ↩
unjoliprogramme
```

L'opérateur `+` est surchargé de telle manière que l'addition d'une valeur quelconque avec une chaîne, concatène la chaine avec la représentation textuelle de la valeur :

```
2 + " divise " + 6 + " pour donner " + 6/2 ;;↩
2 divise 6 pour donner 3
```

2.5 Les valeurs booléennes

Le type `booléen` ne possède que deux valeurs notées `false` et `true`. Mais en `MGS` n'importe quelle valeur peut s'interpréter comme un booléen. Les règles d'interprétation suivantes sont utilisées :

- L'entier `0` représente la valeur *faux*. Toutes les autres valeurs entières sont interprétées comme *vrai*.
- Le flottant `0.0` représente la valeur *faux*. Toutes les autres valeurs flottantes sont interprétées comme *vrai*.
- La chaîne de caractère vide `""` représente *faux*, les autres chaînes représentent *vrai*.
- La valeur indéfinie `<undef>` représente la valeur *faux*.
- Quel que soit le type de la collection, la collection vide représente la valeur *faux* ; une collection non-vide est interprétée comme *vrai*.

Les opérations logiques sont le *et logique* : `&` ou bien `&&`, le *ou logique* : `|` ou bien `||` et la *négation* : `~` .

Chapitre 3

Les fonctions, le séquencement et les variables

`MGS` est un langage fonctionnel qui permet de définir des fonctions d'ordre supérieur. Cela veut dire qu'une fonction est une valeur et que comme les autres valeurs (entier, flottant, chaîne de caractères, ...) on peut les passer en argument à d'autres fonctions et les retourner comme résultat d'une fonction. Une fonction est une valeur atomique.

3.1 Les lambda-expressions

La dénotation d'une valeur fonctionnelle est fondée sur le lambda-calcul. Par exemple

```
\x.x+1
```

est une lambda-expression dont la valeur est une fonction qui, appliquée à un argument, retourne la valeur de cet argument incrémentée de 1. Les fonctions sont des valeurs comme les autres, elles peuvent donc être retournée par une fonction. Ainsi :

```
\x.\y.x+y
```

dénote une fonction qui attend une valeur X qui sera liée au nom `x` pour retourner en résultat une fonction similaire à `\y.`X`+y`.

La syntaxe de l'application de fonction est la syntaxe habituelle où les arguments sont données entre parenthèses. Par exemple, l'évaluation de

```
(\x.x+1)(2)
```

retourne la valeur `3`. Un autre exemple :

```
(\x.\y.x+y)(3)
```

décrit l'application de la fonction `\x.\y.x+y` à l'argument `3`. Le résultat de cette application est une fonction qui agit comme `\y.3+y`. Lors d'une application d'une lambda-expression, tous les arguments sont d'abord évalués (appel par valeur). Mais l'ordre d'évaluation des arguments n'est pas précisé.

La syntaxe `\x.`*exp* permet de dénoter l'asbstraction de la variable `x` sur l'expression *exp*. On peut abstraire plusieurs variables en même temps :

```
\x, y.x+y      ou bien      \(x, y).x+y
```

est un exemple d'une fonction qui retourne la somme de ses deux arguments. Si on fournit moins d'arguments que nécessaire, la fonction à appliquer est automatiquement *currifyée* et le résultat retourné est une fonction. Par exemple

```
(\(x,y).x+y)(3)
```

est une fonction qui attend un entier y (« l'argument manquant ») et retourne $3 + y$.

Fonction nommée. Une autre syntaxe est utilisée pour l'abstraction, quand il faut donner une nom à la fonction :

```
fun    Plus(x, y) = x+y
```

Cette expression crée une valeur fonctionnelle et assigne un nom à cette valeur : `Plus`. Ce nom est un alias qui peut être utilisé partout où la lambda-expression aurait pu être utilisée. Par exemple :

```
Plus(4,5)
```

est une application qui retournera la valeur `9`. Le nommage des lambda-expressions permet de définir directement des fonctions récursives :

```
fun    Factoriel(x) = if    (x == 0) then    1 else    x * Factoriel(x-1) fi
```

3.2 Les fonctions primitives

L'opérateur `+` qui apparait dans le corps de la fonction `Plus` est un exemple de *fonction primitive*. Les fonctions primitives sont des constantes fonctionnelles prédéfinies dans le langage. Il y a un riche ensemble de fonctions primitives, qui permet de gérer les valeurs de base.

Les fonctions primitives peuvent être utilisées exactement comme les fonctions nommées : on peut par exemple les passer en argument à une autre fonction.

Le nom des fonctions prédéfinies, comme le nom de tous les objets prédéfinis[1], débute par une apostrophe `'` que nous appelerons dans la suite *quote*. Ont dit que le nom (la variable, l'identificateur, ...) est *quoté*.

Certaines fonctions primitives ont une syntaxe supplémentaire pour l'application, comme par exemple l'opérateur `+` qui est la forme infixe de l'application de la fonction primitive `'addition`. Autrement dit, `3 + 4` est une abréviation de l'expression `'addition(3, 4)`.

Un identificateur quoté, comme `'addition`, peut être utilisé sans quote, tant que le nom-sans-quote n'a pas été défini par l'utilisateur. Par exemple, on peut écrire

```
'addition(3, 4)      ou bien     addition(3, 4)
```

tant que l'utilisateur ne définit pas une fonction de nom `addition`. Si l'utilisateur définit par exemple :

```
fun     addition(x, y) = x*y
```

la fonction primitive est toujours accessible à travers son nom quoté : `'addition`. En effet, l'utilisateur ne peut pas définir une fonction avec un nom quoté :

```
fun 'addition(x, y) = x*y ;; ↩
MGS: parse error
near 'addition (current token is QUOTED_IDENTIFIER -- 260)
```

3.3 Arguments optionnels

Les lambda-expressions en `MGS` possèdent une caractéristique inhabituelle : il est possible de définir des *arguments optionnels nommés qui ont une valeur par défaut*. Par exemple :

```
fun     F[a=1, b=1, c=0](x, y) = a*x + b*y + c
```

définie une lambda-expression de nom `F` avec un argument optionnel de nom `a` dont la valeur par défaut est `1`. De manière similaire, `b` et `c` sont des arguments optionnels.

Dans une application standard `F(1, 2)`, les arguments optionnels gardent leur valeurs par défaut. L'application précédente s'évalue en `3`. Mais il est possible de donner une valeur à un argument optionnel lors de l'application :

```
F[c=4](1, 2)
```

évalue le corps de la fonction `F` avec l'argument `c` lié à la valeur `4` (et le résultat est `7`). Voici un autre exemple :

```
F[c=4, a=2](1, 2)
```

[1] il existe des variables prédéfinies, des noms de type prédéfinis, etc.

dont la valeur est `8`. Comme on le voit, on peut fournir, dans un ordre quelconque, une valeur à zero, un ou plusieurs arguments optionnels lors d'une application. Le nom des paramètres optionnels est utilisé pour indiquer quel argument est valué. On peut donner une valeur à un argument inexistant : cette valeur est simplement ignorée pendant l'évaluation :

```
F[xxxx=666, c=4](1, 2)
```

s'évalue simplement en `7` comme précédement.

Certaine fonctions primitives ont des arguments optionnels. C'est le cas de la fonction `'fold` qui prend deux arguments `'zero` et `'fct`. Tant que les noms quotés ne sont pas redéfinis, ont peut écrire :

```
fold[zero = ..., fct = ...](...)
```

On retrouvera cette fonction dans la section concernant les collections monoïdales.

Il est possible de définir un argument optionnel sans lui donner explicitement une valeur par défaut. Dans ce cas, la valeur par défaut de l'argument est implicitement la valeur indéfinie `<undef>`. Dans l'exemple :

```
fun G[a](x) = x + a;;
G[a = 0](1)
G(1)
```

la fonction G fait appel à un argument optionnel `a`. La première application `G[a = 0](1)` s'évalue en `1` tandis que l'expression `G(1)` déclenche une erreur et retourne `<undef>` car

```
1 + <undef>
```

déclenche une erreur et retourne `<undef>`.

De manière similaire, l'expression

```
f[a](...)
```

doit s'interpréter comme :

```
f[a = <undef>](...)
```

3.4 Itération et point-fixe

Pour toutes les fonctions, il existe des « arguments optionels implicitement définis » qui permettent de contrôler l'itération de l'application d'une fonction. Il s'agit des arguments `'iter`, `'fixpoint` et `'fixrule`. Le plus simple est de donner un exemple. Si on définit la fonction `f` par :

```
fun f(x) = x+1
```

alors

```
f['iter = n](m)
```

calcule

$$\underbrace{\texttt{f}(\texttt{f}(\ldots(\texttt{f}}_{n \text{ fois}}(m))\ldots))$$

Par exemple, `f['iter = 33](2)` va retourner `35`. Autrement dit, la valeur de l'argument optionnel `'iter` permet de fixer le nombre de fois où on itère l'application d'une fonction. Bien sur, cet argument n'a de sens que si la fonction est unaire. Si on itère une fonction zéro fois, la valeur retournée est `<undef>`.

Les deux autres arguments optionels sont utilisés pour calculer le point-fixe d'une fonction. Par exemple :

```
fun g(x) = 1.0 + 1.0/x ;;
g['iter = 'fixpoint](0.5)
```

calcule $1.618034\ldots$ (le nombre d'or) qui est la limite $\lim_{n\to\infty} \texttt{g}^n(\texttt{0.5})$.

Pour abbréger l'écriture, il est aussi admis d'écrire directement `g['fixpoint](0.5)` ou bien `g[fixpoint](0.5)` tant que l'identificateur `fixpoint` n'est pas redéfini. Avec cette dernière ecriture, la présence simultanée de `'fixpoint` et de `'iter`, comme dans `g['fixpoint, 'iter=33](2)`, n'a pas de sens et correspond à une erreur de programmation.

De manière générale, l'expression

```
g['fixpoint](x0)
```

calcule la suite

$$x_0,\ x_1 = \texttt{g}(x_0),\ x_2 = \texttt{g}(x_1),\quad \ldots,\quad x_{n+1} = \texttt{g}(x_n),\ \ldots$$

et retourne la valeur x_q telle que $x_{q+1} = x_q$ si cette valeur existe. Si elle n'existe pas, c'est à dire si $\texttt{g}(x_n) \neq x_n$ pour tout n, le calcule boucle (i.e. : le programme ne termine pas). La valeur retournée (si toutefois il y a une valeur retournée), est un point fixe de `g`, c'est à dire une valeur $\bar{x}$ vérifiant : $\bar{x} = \texttt{g}(\bar{x})$.

ATTENTION : le point fixe est détecté en utilisant l'égalité des valeurs `MGS` ; dans le cas d'un point fixe numérique il faut donc prendre toutes les précautions d'usage sur la convergence des itérations et la maitrise des erreurs en précision flottante.

Il n'y a pas de différence sémantique entre `'fixpoint` et `'fixrule` (autrement dit, dans un programme déterministe on peut remplacer toute occurence de `'fixpoint` par `'fixrule` et vice-versa, sans changer la valeur du résultat). Par contre, il y a une différence opérationelle dans le cas de l'application d'une transformation (Cf. le chapitre sur les transformations).

Dans le cas d'une transformation, détecter qu'un point fixe est atteint se fait de manière plus efficace en utilisant `'fixrule`. De plus, si le programme est non-déterministe, c'est à dire si deux exécutions du programme peuvent donner des résultats différents, alors `'fixpoint` et `'fixrule` ne sont pas nécessairement équivalents (on verra plus loin des constructions indéterministes).

3.5 Gestion des erreurs

Le traitement des erreurs n'est pas encore bien fixé en `MGS`. La plupart du temps, une fonction primitive renvoie la valeur `<undef>` quand un de ses arguments est erroné. Suivant la gravité de l'erreur, différents comportements peuvent advenir :

- la fonction renvoie `<undef>` et le calcul se poursuit avec cette valeur ;
- la fonction renvoie `<undef>`, un message d'erreur s'affiche et le calcul se poursuit avec cette valeur ;
- un message d'erreur s'affiche et le calcul s'interrompt ;
- un message d'erreur s'affiche et l'interprète termine la session.

Un programmeur peut lever une erreur en appelant la fonction primitive `error` :

```
error(arg)
```

affiche *arg* puis déclenche une erreur. Cette erreur est "active" pendant toute l'évaluation de l'expression courante de l'interprète (l'expression courante de l'interprète est l'expression entrée au top-level de l'interprète) mais n'affecte pas l'évaluation de l'expression suivante. Quand une erreur est active, l'appel de toutes les fonctions primitives retourne immédiatement avec la valeur `<undef>`. Le programmeur peut tester si une erreur est active grace à l'appel

```
error?()
```

qui retourne *vrai* ou *faux*.

3.6 Constructions spéciales

Certaines constructions du langage ressemblent à des fonctions prédéfinies mais exhibent un comportement différent et ne peuvent se traduire par une fonction ordinaire. Il est impossible de passer ces fonctions en arguments ou même de les utiliser dans n'importe quel contexte. Il s'agit :

1. de la mise en séquence de deux expressions : « `;` »
2. des operateurs booléens : `&` et `|` (qui peuvent s'écrire aussi `&&` et `||`)
3. de la conditionelle : `if ... then ... else ... fi`
4. de l'assertion : `!!...`
5. des opérateurs de voisinage : `left`, `right`, `neighbours`, `border?`

Mise en séquence. La séquence de 2 expressions :

e_1 `;` e_2

évalue d'abord l'expression e_1 puis évalue e_2 et retourne cette dernière valeur. L'opérateur « `;` » n'est pas une fonction et on ne peut pas le passer en argument. Il n'est pas facile d'écrire une fonction qui réalise la mise en séquence de deux expressions car `MGS` ne dit rien sur l'ordre d'évaluation des arguments d'une fonction.

Opérateur logiques et conditionelle. Le "et logique" et le "ou logique" sont des opérateurs *paresseux* : ils n'évaluent pas leur deuxième argument si le résultat final est assuré par la seule valeur du premier argument. Autrement dit, le deuxième argument d'un "et logique" n'est pas évalué si le premier argument s'évalue à *faux* (idem pour le "ou logique" si le premier argument s'évalue à *vrai*). Par exemple

```
0 & (?("hello"); 1)
```

retourne `0` sans afficher la chaîne `"hello"` au terminal.

La conditionelle est aussi paresseuse : si son premier argument est à *vrai*, le troisième argument (correspondant à la branche "fausse") n'est pas évalué et inversement, si la condition s'évalue à *faux*, le deuxième argument n'est pas évalué.

On ne peut pas passer ces constructions comme arguments d'une fonction.

Assertion. Une assertion est une expression de la forme

`!!` *exp*

qui s'évalue de la manière suivante : l'expression *exp* est évaluée et si sa valeur v est interprétée comme le booléen *vrai*, le résultat de l'assertion est v. Sinon, un message indiquant que l'assertion n'est pas vérifiée est émis et l'interprète est stoppé.

Opérateurs topologiques. Les opérateurs topologiques `left`, `right`, `neighbours` et `border?` sont des constructions spéciales qui ne peuvent apparaître que dans une règle. Il seront détaillés dans le chapitre correspondant.

3.7 Noms de fonctions, variables locales et variables globales

Il existe plusieurs manières de donner un nom à une valeur, afin de la réutiliser dans une expression ultérieure :

1. Si la valeur est une valeur fonctionnelle, cela peut être une *valeur fonctionnelle nommée*. Une valeur fonctionnelle nommée est créée lors de la définition d'une lambda nommée[2],

[2] cf. section 3.1 page 13

lors de la définition d'un type[3] ou lors de la définition d'une transformation[4].

2. Une *variable locale* permet de donner un nom à une valeur quelconque. La construction `let ...in ...` décrite ci-dessous permet d'introduire de nouvelles variables locales mais les *arguments* d'une fonction ou bien les *variables de filtre*[5] sont d'autres exemples de mecanismes qui, comme les variables locales, permettent de donner un nom local à une valeur.
3. Enfin, les *variables globales impératives* correspondent à une référence vers une valeur qu'il est possible de changer par affectation. Elles sont décrites ci-dessous.

Ces trois mécanismes de nommage correspondent à des *espace de noms* distincts. On peut donc avoir à la fois une fonction de nom *id*, une variable locale de nom *id* et une variable globale de nom *id*.

Quand on rencontre une variable dans une expression, l'identification de la variable correcte dépend du contexte :

- Si la variable est en position fonctionnelle, on regarde tout d'abord si le nom de la variable ne correspondrait pas à une valeur fonctionnelle nommée ;
- sinon, et dans le cas général où la variable n'est pas en position fonctionnelle :
  - on essaie de résoudre l'identificateur comme une variable locale ;
  - en cas d'insuccès, on essaie de résoudre l'identificateur comme une variable globale ;
  - puis on essaie finalement de résoudre cette variable comme une variable fonctionnelle nommée.

De plus, *le mode de liaison est syntaxique*[6].

Les variables locales

La construction

`let` id_1 `=` exp_1
`and ...`
`and` id_n `=` exp_n
`in` exp

correspond au *let* non récursif dans un langage comme ML. Cette construction permet de donner un nom id_i au résultat de l'évaluation de exp_i et de réutiliser cette valeur dans exp. Par exemple :

```
let x = Factoriel(4) - 20 in x*x
```

[3] cf. section 4.3 page 23
[4] cf. section 6 page 33
[5] cf. section 6.1 page 34
[6] La liaison des variables est une liaison strictement syntaxique dans le cas du compilateur `MGS`. Par contre, les interprètes sont plus laxistes en ce sens que s'il n'y a pas de définition associée à une variable apparaissant dans le corps d'une lambda-abstraction, alors la liaison de cette variable est retardée jusqu'au moment de la première application de la lambda. Autrement dit, les variables non définies restent libres mais doivent se résoudre, au moment de l'application, en une valeur fonctionnelle nommée ou bien une variable globale. Cette facilité a pour but de simplifier les redéfinitions successives lors de l'utilisation interactive de l'interprète.

s'évaluera en `16` car `x` est lié à la valeur `4`. L'évaluation de exp_1 se fait dans l'environnement englobant (`let` non récursif) et par suite :

```
let x = 1
in let x = 1 + x
   in x
```

a pour valeur `2`. On *ne peut donc pas* utiliser un let pour définir une lambda-expression anonyme récursive :

```
let fact = \x.if x == 0 then 1 else x*fact(x-1) fi
in fact(3);;
```

provoque une erreur car la variable `fact` dans le corps de la lambda-expression n'est pas connue au moment de la définition de l'abstraction et ne se résoud pas ensuite au moment de l'application `fact(3)` en une valeur fonctionnelle nommée ou bien une variable globale. En effet, la liaison introduite par le `let` est une liaison syntaxique qui est valide lors de la définition de l'expression `fact(3)` mais qui n'existe plus au moment de l'évaluation de `fact(3)`.

Les variables globales

Il est possible de définir des variables globales dont on peut changer la valeur arbitrairement, à la manière des langages impératifs. L'affectation de la valeur d'une variable est une expression spéciale dont la valeur est la valeur affectée :

```
a := 3
```

est une expression qui va affecter la valeur entière `3` à la variable globale de nom `a`. cela n'est possible qu'à la condition que la variable globale `a` « *existe* ». Une variable globale existe quand elle a été **initialisée**, c'est-à-dire affectée une première fois **au top-level**. Par exemple, l'expression

```
(1; var := "erreur")
```

déclenche une erreur si `var` n'a pas été initialisé au top-level :

```
Error: eval: cannot update unknown identifier: var
in not-at-toplevel update:
    var := "erreur"
```

Les variables globales ne sont pas typées, et donc on peut affecter successivement des valeurs de types différents à la même variable :

```
a := 1;;   a := "toto";;   a := 33.0;;
```

Voici un exemple d'utilisation où une variable globale `compteur_fact` est initialisé à `0` puis utilisé pour compter par effet de bord le nombre des appels à une fonction `f` :

```
compteur_fact := 0;;
fun f(x) = (compteur_fact := 1+compteur_fact; x+1);;
f['iter=55](0);;
?("le nombre d'appel à la fonction f est de " + compteur_fact);;
```

Chapitre 4

Enregistrements

Un *enregistrement* `MGS` est un type particulier de collection. Bien qu'étant une collection (la fonction `size` renvoient le nombre de champ dans l'enregistrement), « l'aspect collection » des enregistrements reste frustre et on peut tout aussi bien les considérer comme des valeurs atomiques.

4.1 Construction et accès

Un enregistrement `MGS` est très similaire à un enregistrement Pascal, une *struct* `C` ou bien un *record* `CAML`. Un enregistrement `MGS` associe des valeurs avec des noms symboliques appelé *champ*. Les valeurs attachées aux champs peuvent être de n'importe quel type, y compris d'autres enregistrements ou bien d'autres collections.

Construction. Un enregistrement peut se construire à l'aide des accolades :

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

est une expression qui construit un enregistrement avec deux champs : un champ de nom `a` et de valeur `1` et un champ de nom `b` et de valeur, la chaîne `"red"`. A droite du signe égal qui suit un nom de champ, peut figurer une expression arbitraitrement complexe :

```
{a = 1+2,   b = f(g(3, "red")),   c = {x=0, y='math_PI}}
```

Accès. L'accès à la valeur d'un champ se fait grâce à la notation pointée :

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

est une expression qui retourne la valeur du champ `a`, i.e. : `3`. L'accès à un champ inexistant renvoie la valeur `<undef>` sans déclencher d'erreur.

4.2 Extension d'un enregistrement

On peut fusionner des enregistrements avec l'opérateur +. L'expression

```
r + s
```

donne un nouvel enregistrement `t` possédant tous les champs de `r` et de `s`. Si un champ `a` est présent dans `s` alors `t.a` vaut `s.a`, sinon `t.a` vaut `r.a`. Cette fusion est dite asymétrique [Rém92] car la priorité en cas de « collision de champs » est donnée au deuxième argument de +. Par exemple :

```
{a = 1,   b = "red"} + {b = "yellow",   c = false}
```

a pour valeur l'enregistrement

```
{a = 1,   b = "yellow",   c = false}
```

4.3 Les types enregistrements et à quoi il servent

On peut définir un type d'enregistrement. Cependant, contrairement à un langage typé statiquement, il n'est pas nécessaire en `MGS` de définir le type d'un enregistrement avant de pouvoir créer des valeurs de ce type. En effet, la construction « accolade » précédente permet de construire n'importe quel enregistrement (de n'importe quel type) sans faire référence à son type[1].

À quoi sert alors un type en `mgs` ? De manière générale en `MGS` la définition d'un type définit un prédicat, qui possède le même nom que le type, et qui permet de tester si une valeur quelconque est de ce type[2]. Cela est vrai même pour les types prédéfinis, comme par exemple les types atomiques :

```
string("abc");; ↩
1
int("abc");; ↩
0
```

(rappelons que `0` représente la valeur *faux* en `MGS` et `1` représente la valeur *vrai*).

Le prédicat associé à un type prédéfinie ou bien construit par le programmeur, peut être utilisé dans n'importe quelle expression pour tester si une valeur est d'un type donnée. Par exemple :

[1] Certains types de collections doivent être définis avant de pouvoir créer une valeur de ce type. C'est le cas par exemple pour les GBF que nous verrons dans un prochain chapitre.

[2] Dans le cas des GBF, la définition d'un type en plus de définir ce prédicat, introduit de nouvelles constructions dans le langage, permettant de définir les valeurs GBF et les filtrer.

```
fun val_to_string(x) = if string(x) then x else ("" + x) fi ;;
```

est une fonction qui retourne son argument si celui-ci est une chaîne de caractères et qui sinon le transforme en chaîne de caractères (rappelons que l'expression `s + x` où `s` est une chaîne, retourne la concaténation de la chaîne `s` avec l'expression textuelle de la valeur de `x`).

C'est dans la définition d'une règle dans une transformation, que l'utilisation de tel prédicat prend une importance considérable, car les collections sont des collections d'éléments hétérogènes.

4.4 Définition d'un type enregistrement

Une définition de type n'est pas une expression et doit intervenir au top-level. La déclaration d'un type enregistrement prend la forme suivante :

```
state    R = {a};;
```

`state` est le mot-clé qui introduit la définition d'un type dans `MGS`. Cette déclaration spécifie que le type `R` correspond au type de tous les enregistrements qui possède au moins un champ `a`. Ainsi,

```
R({a = 1})
```

retourne la valeur *vrai*, car l'enregistrement argument du prédicat `R` possède bien un champ de nom `a`. Le type de la valeur de cet enregistrement n'a pas d'importance :

```
R({a = "une chaine"})
```

retourne aussi la valeur *vrai*. De même, la présence d'autres champs ne change pas la réponse :

```
R({a = 1, b = 3.14})
```

retourne aussi la valeur *vrai* car l'enregistrement argument du prédicat `R` possède bien un champ de nom `a` (et aussi d'autre champs, mais cela n'a pas d'importance).

Un exemple d'utilisation. Voici un exemple d'utilisation du prédicat `R` plus utile que ce qui précède :

```
fun f(r) = if R(r) then r + {a = r.a + 1} else r fi ;;
```

La fonction `f` que nous venons de définir, peut prendre n'importe quel argument. Si cet argument est un enregistrement qui possède le champ `a`, alors elle calcule un nouvel enregistrement qui possède tous les champs de l'argument, mais dont la valeur du champ `a` a été incrémenté de `1`. Par contre, si l'argument n'est pas un enregistrement contenant le champ `a`, la fonction `f` retourne cet argument.

Maîtriser l'occurence d'un champ et sa valeur

La définition d'un type enregistrement permet de spécifier la présence d'un champ. On peut spécifier beaucoup plus finement en `MGS` l'occurence ou non d'un champ dans un enregistrement :

- on peut spécifier son absence : construction `~a`
- on peut spécifier sa présence et préciser le type de sa valeur : construction `a:T`
- on peut spécifier sa présence et préciser sa valeur : construction `a =` v

Illustrons ces possibilités par des exemples :

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

Le type `S` est le type d'un enregistrement qui doit posséder un champ `b` mais pas de champ `c`. Le type `T` est le type d'un enregistrement qui doit posséder un champ `a` dont la valeur doit être 1, et un champ `d` dont la valeur doit être une chaîne de caractères (type `string`).

Héritage entre types enregistrements

Il est possible de réutiliser la spécification de l'occurence d'un ou plusieurs champs introduite par un type enregistrement dans la définition ultérieur d'un autre type enregistrement. Cela se fait grâce à l'opérateur + entre type enregistrement qui émule ainsi une sorte d'héritage entre enregistrement. Nous parlons d'héritage car le mécanisme est très semblable au mécanisme d'héritage dans les langage à objets.

Par exemple

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

La seconde déclaration définit le type S ; un enregistrement de ce type doit satisfaire aux contraintes de R, et en plus, contenir un champ b et pas de champ c. La définition de T est une spécialisation de S (i.e. il y a un champ `a`, un champ `b` et pas de champ `c`) où de plus a doit avoir pour valeur 1 et où un champ d du type *string* doit être présent.

On voit qu'on peut affiner la spécification de l'occurence d'un champ en ajoutant des contraintes : la spécification de la présence d'un champ est une contrainte moins forte que la spécification de sa présence et du type de sa valeur, et à son tour, cette spécification est une contrainte moins forte que la spécification d'un champ et de sa valeur.

La déclaration de `S` dans

```
state R = {a} ;;
state NR = {~a} ;;
state S = R + NR ;;
```

va provoquer une erreur car les contraintes apportées par `R` et `NR` sont contradictoires : la définition de `S` requiert à la fois la présence et l'absence du champ `a`.

Une hierachie de types enregistrements

Considérons les deux déclarations de type enregistrement suivantes :

```
state R = {a, b};;
state C = {a, c};;
```

Alors l'enregistrement

```
{a=1, b=2, c=3}
```

possède à la fois le type `R` et le type `S`. Il possède aussi les types suivants :

```
state RC = R + C;;
state P1 = {a};;
state Q1 = {b};;
state R1 = {c};;
```

On voit qu'un enregistrement possède plusieurs types. Ces types peuvent s'organiser en hiérarchie correspondant à la relation d'ordre suivante : `P < Q` si toutes valeurs de type `P` est aussi une valeur de type `Q`. On dit alors `P` est une *spécialisation* de `Q`. Deux types `P` et `Q` sont *compatibles* si on peut former le type `P + Q`. Si deux types sont compatibles, alors on peut former la *borne inférieure* de deux types et c'est simplement `P+Q`.

Inversement, l'ensemble des enregistrements qui ont à la fois le type `R` et le type `C`, ont aussi le type :

```
state T = { a };;
```

et on définit donc la *borne supérieure* de `R` et `C` comme étant `T`. Il existe un type maximal, c'est simplement :

```
state Maximal = {}
```

puisque ce type est satisfait par tous les enregistrements.

Chapitre 5

Collections et transformations : une introduction

L'originalité de `MGS` réside pour une grande part dans les nouvelles façons de manipuler les collections offertes par le langage. Rappelons qu'une *collection* est une valeur décomposable en un ensemble organisé de valeurs plus simples. Calculer avec les collections correspond donc à parcourir les éléments de la collection, à les transformer et à réassembler (réorganiser) ces résultats pour créer une nouvelle collection. Ces mécanismes constituent une *transformation*.

Dans ce chapitre, nous allons parler assez abstraitement des notions de collection et de transformation, en soulignant ce qui est commun à tous les types de collections. Les chapitres suivants détailleront concrètement et collection par collection, les opérations disponibles sur chaque type[1].

5.1 Notion de collection

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

L'objectif de `MGS` est de créer et de manipuler simplement des structures de collection très variées. En conséquence, il y a beaucoup de sortes différentes de collections en `MGS`. Le programmeur peut créer de nouvelles sortes de collections, mais il ne peut pas créer de nouvelles sortes de valeurs atomiques.

Les différentes sortes de collections se distinguent par l'organisation des valeurs qui les composent. Ces organisations sont elle-mêmes classées par grandes familles. L'arbre de la figure 5.1 esquisse la hiérarchie des types de collections en `MGS`.

[1] Les notions évoquées ici sont applicables en principe aux enregistrements, mais n'ont pas été implémentés dans les interprètes. Par exemple, on ne peut actuellement pas définir de transformation agissant sur un enregistrement. C'est pourquoi nous avons préféré traiter les enregistrement dans un chapitre précédent, au même niveau que les valeurs atomiques.

Quand une valeur est une collection, elle est constituée par d'autres valeurs qui peuvent être de n'importe quel type : valeurs atomiques, ou bien d'autres collections. Les imbrications peuvent être arbitraires : on dit que les valeurs `MGS` sont des *valeurs complexes* dans le sens de [BNTW95].

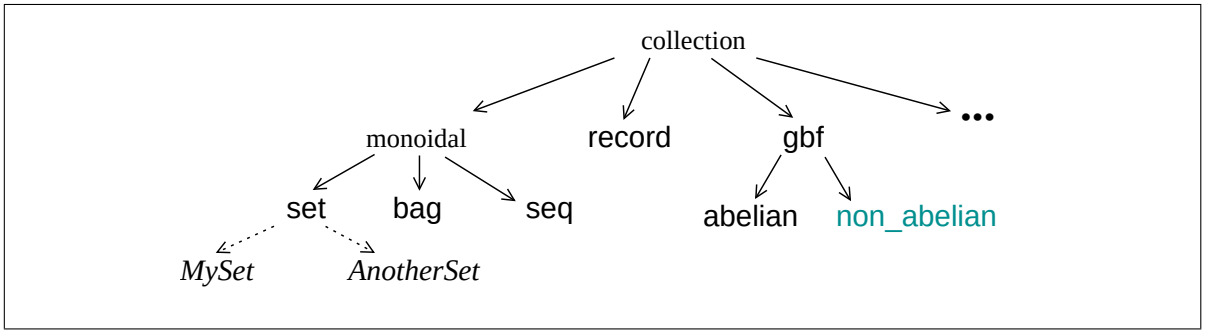


Fig. 5.1: La hiérachie des types de collections en `MGS`. *MySet* et *AnotherSet* sont des collections de type ensemble définis par l'utilisateur, cf. section 10.1. Les types `collection`, `monoidal` et `gbf` ne correspondent pas à des valeurs concrètes mais à de grandes familles de valeurs de type différents.

Nous avons mentionné comme exemple d'organisation la séquence, le tableau ou l'ensemble, mais bien d'autres sortes d'organisations sont possibles et utiles. Par exemple, si on veut modéliser et simuler un système mécanique, il va falloir associer à des éléments physiques à trois dimensions des quantités correspondant à leur position, leur vitesse, leur moment, etc. L'organisation géométrique de ces éléments physiques est donc susceptible de devenir l'organisation sous-jaccente à un ensemble de valeurs[2].

L'exemple que nous venons de donner n'est pas innocent : nous voulons introduire l'idée qu'une collection peut être appréhendé d'un point de vue géométrique, et plus généralement être examiné d'un *point de vue topologique*.

Les structures de données vues comme des espaces

Un espace, *tout comme* une structure de données, est un ensemble d'éléments (de lieux, de positions, de points) dans lequel on se déplace. La notion de déplacement suppose la notion de voisin : on va de proche en proche. Il n'est pas usuel de parler de voisinage pour une structure de données, cependant :

- dans une structure d'arbre, un nœud père est voisin de ses fils, car on peut accéder, à partir du père, à l'un de ses fils ;
- dans une matrice, à partir de l'élément d'index (i, j) on accède souvent aux éléments d'index $(i-1, j)$, $(i+1, j)$, $(i, j-1)$, $(i, j+1)$ et on les considère comme étant voisins : ce schéma d'accès se nomme d'ailleurs « voisinage de Von Neumann » ;
- dans une liste simplement chaînée, on passe séquentiellement de la tête de liste à la queue de la liste ;
- dans une structure de tampon circulaire, on va à l'élément suivant ou précédent ;
- dans un enregistrement, les différents items sont logiquement regroupés et ce regroupement a une existence tangible, puisqu'on peut le désigner par un identificateur.

[2]Des structure de données comme les *G-cartes* [Lie91] ou bien comme les *arbres CSG* (ce sont des termes qui représentent des combinaisons entre solides primitifs), permettent de représenter des organisations spatiales.

Les exemples peuvent se multiplier pour nous convaincre que la notion de voisinage logique est essentielle dans la définition des structures de données.

L'utilisation de notions topologiques pour étudier et caractériser les programmes n'est pas nouvelle. Par exemple, dans [FM97], on considère les déplacements qui consistent à suivre les pointeurs d'une structure de données `C`. Plus fondamentalement, l'étude de la notion de voisinage dans un ensemble de calculs constitue l'essence de la sémantique dénotationnelle [Vic88, Mos90, GS90].

5.2 Collections topologiques

Ces considérations nous poussent à voir une collection comme un ensemble de *positions*[3] relativement à une topologie définie par une relation de voisinage. Chaque position possède une valeur[4]. Afin de souligner l'importance de l'organisation topologique des collections nous les appellerons *collections topologiques*.

Par exemple, dans une séquence on peut définir le voisinage d'un élément comme l'élément précédent et le suivant. Nous noterons « ‚ » la relation de voisinage : $x \text{ ‚ } y$ signifie que y est un voisin de x.

Sous-collection d'une collection topologique

Pour l'instant, seule la définition d'une sous-collection nous est nécessaire. Nous dirons qu'un sous-ensemble S d'éléments d'une collection $\mathcal{C}$ est *connecté* lorsqu'on peut aller d'un élément quelconque de S à un autre élément quelconque de S, en cheminant d'élément à élément voisins dans S. (Techniquement, on demande que le quotient de S par la clôture transitive de ‚ restreint à S soit réduit à un seul élément.) Une *sous-collection* est un sous-ensemble connecté d'une collection.

Pour l'exemple des séquences, ceci nous permet de définir la sous-séquence comme un sous-ensemble connecté d'une séquence. Ceci revient à dire que les sous-séquences sont ses intervalles (une suite d'éléments contigüs). On pourrait aussi choisir de contraindre plus fortement la notion de sous-séquence en n'acceptant que les préfixes de la séquence comme sous-séquence. Cela donnerait lieu à une notion différente de séquence (un autre type de collection). Mais la seule contrainte constante valable pour tous les différents types de collections est qu'une sous-collection est un sous-ensemble connecté d'une collection.

Un projet de recherche

L'approche topologique de la notion de collection fait partie d'un long travail de recherche [GMS96] détaillé notamment dans [Gia00] qui étudie la notion de sous-collection et dans

[3] que nous appelons aussi indifféremment : lieux, places, éléments, points, cellules, simplexes.

[4] On dit aussi que chaque position est valuée, ou bien qu'elle dénote une valeur, ou encore qu'elle est étiquetée ou décorée par une valeur. Une collection *totale* est une collection dont toutes les positions possèdent une valeur bien définie et différente de `<undef>`. Une collection est *partielle* si certaine position n'ont pas de valeur bien définie.

[GM01a] qui décrit un outil générique pour la définition de relations uniformes de voisinage. Les notions topologiques nécessaires à la définition de voisinages dans les séquences ou dans d'autres structures simples sont modestes et suffisent à unifier[5] des modèles de calcul comme les L système [RS92], les P systèmes [Pau98], la CHAM [BB89] ou Gamma [BM86] et les automates cellulaires [VN66]. Néanmoins des notions de topologie plus complexes sont nécessaires, par exemple pour la modélisation de certains processus biologiques [GM01b]. L'approche topologique permet de traiter différents problèmes de façon uniforme.

5.3 Notion de transformation

La notion de voisinage dans une structure de données n'est pas une notion purement logique, perceptible pour le programmeur et invisible à l'exécution. En effet, le traitement dans un programme respecte très souvent la notion de voisinage logique qui apparaît dans les structures de données, le calcul se propageant généralement de proche en proche. Par exemple, si on considère la définition récursive de la fonction `map` sur les listes, on s'aperçoit que le traitement se propage de la tête vers la queue de la liste. Plus généralement, les traitements récursifs respectent les voisinages induits par une structure de données au point qu'il apparaît nécessaire et possible de définir automatiquement des *schémas de récursion naturellement liés à la structure de données*[6].

Cela signifie qu'un calcul sur une structure de données respecte une propriété de localité : le calcul associé à un élément dans une structure de données dépend uniquement du calcul sur les voisins[7]. Par exemple, le calcul de la valeur d'un attribut d'un sommet dans un arbre dépend des attributs des sommets fils (attribut synthétisé) ou bien du sommet père (attribut hérité) ou bien d'une combinaison des deux.

Un mécanisme pour spécifier des calculs locaux

`MGS` propose un mécanisme général, i.e. pouvant se décliner pour tous les types de collection[8], permettant de spécifier des « mécanismes de calcul locaux » sur une collection. Ce mécanisme peut se décrire de la manière suivante. Étant donné une collection C :

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

Ces trois étapes sont appelées une *transformation élémentaire* et sont illustrées à la figure 5.2.

[5] Le type de collections sous-jacent dans Gamma, la CHAM et les P systèmes est le multi-ensemble, la séquence pour les L systèmes et le tableau pour les automates cellulaires (AC). Voir le chapitre **??**.

[6] C'est par exemple tout l'objet des *catamorphismes* [MFP91, FS96, NO94] developpé pour le cas des types de données algébriques.

[7] On peut renverser cette proposition pour affirmer que si un calcul est nécessaire à un autre calcul, alors ces deux calculs sont forcément voisins dans un certain espace. Cet espace du calcul peut être abstrait, mais il est souvent matérialisé par la structure de données. Ce qui amène par exemple à structurer les programmes suivant la structure des données en entrée, ou bien la structure des données à produire comme résultat.

[8] Ce point de vue permet d'unifier dans le même cadre formel différents modèles de calcul, cf. section **??**. Pour chacun des langages il suffit de choisir la bonne organisation topologique dans la collection utilisée.

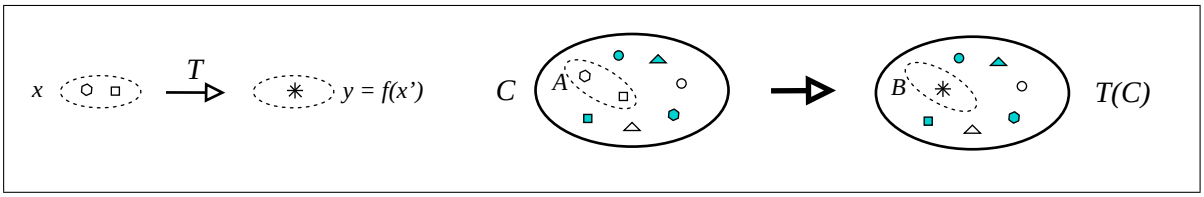


Fig. 5.2: Transformation élémentaire d'une collection. C est une collection. Une règle T spécifie qu'une sous-collection A dans C doit être remplacée par la collection B calculée à partir de A. La collection calculée dans la partie droite de la règle T dépend de la sous-collection filtrée dans sa partie gauche et éventuellement de son voisinage

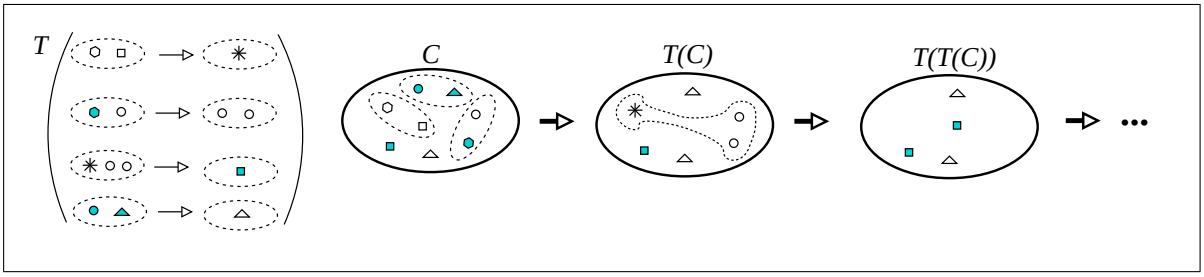


Fig. 5.3: Transformation et iteration d'une transformation. Une transformation T est un ensemble de transformations élémentaires appliquées en parallèle et indépendemment. Les transformations élémentaires n'interagissent pas entre elles, c'est-à-dire qu'elles s'appliquent à des sous-collections différentes et sans intersection. Une transformation peut être itérée un nombre quelconque de fois, par exemple pour atteindre un point-fixe.

Une *transformation*, sans l'adjectif « élémentaire », consiste en l'application *en parallèle* et dans la même collection, de plusieurs transformations élémentaires indépendantes. Par « transformations élémentaires indépendantes » nous voulons signifier que les sous-collections qui sont objet de chaque transformation élémentaire, ne s'intersectent pas.

Une transformation est une fonction comme une autre, qui prend un seul argument (qui doit être une collection) et qui calcule une nouvelle collection. On peut donc itérer une transformation[9]. Si on interprète une collection comme l'état d'un système dynamique (regroupant d'une certaine manière l'état des sous-systèmes), et une transformation comme une fonction d'évolution (définie par l'évolution des sous-systèmes et leur couplage), alors l'itération d'une transformation correspond tout simplement à faire évoluer le système au cours du temps et à calculer la trajectoire du système[10].

5.4 Motifs, règles et transformations

Une transformation sera donc définie par la donnée d'un ensemble de transformations élémentaires, et de certaines informations permettant de gérer leurs applications en parallèle (priorité, stratégie de désambigüation, etc.). Pour spécifier une transformation élémentaire, il faut préciser :

[9] Cf. section 3.4 page 15.

[10] Voir la note 1 page 1 et le chapitre **??**.

1. comment sélectionner A
2. comment est calculé B
3. les contraintes de la substitution de A par B.

Le point 1 sera réalisé par l'utilisation de *motifs*, appelés encore *filtres*, permettant de définir la sous-collection à sélectionner. On associera à chaque motif une expression décrivant le calcul à réaliser pour obtenir le point 2. L'association d'un motif et d'une expression (cette expression pouvant faire référence aux éléments filtrés par le motif), correspond à une *règle*. Le dernier point, correspond simplement à des contraintes structurelles induites par le type de la collection sur laquelle s'applique la transformation.

Ainsi, une transformation est simplement une fonction spécifiée à travers des règles permettant de filtrer des sous-collections. Cette façon de définir des fonctions est habituelle dans un langage comme ML, où les règles permettent de filtrer les valeurs correspondant à un type de données algébrique. En `MGS`, le langage de motif permet de filtrer des structures plus générales, grâce au point de vue topologique qui est adopté.

Chapitre 6

La spécification d'une transformation en mgs

Dans le chapitre précédent, nous avons vu qu'une structure de données correspondait à une relation de voisinage entre les éléments de la structure (collection topologique). Cela permet de définir une sous-collection comme un sous-ensemble connecté d'une collection. Une transformation est une fonction qui calcule une nouvelle collection en appliquant des transformations élémentaires. Enfin, une transformation élémentaire substitue une sous-collection par une autre. Une transformation élémentaire est spécifiée par une règle constituée d'un motif permettant de sélectionner une sous-collection par filtrage, et d'une expression calculant la sous-collection à substituer.

Concrètement en `MGS`, une transformation est un ensemble de règles noté :

```
trans    T = { ...    regle;    ...   }
```

Le mot-clef `trans` introduit la définition d'une transformation et joue un rôle similaire au mot-clef `fun` pour les lambda-abstraction. Ce mot-clef est suivi du nom de la transformation (ici `T`), du signe `=` puis de la spécification du corps de la transformation qui est un ensemble de règles données entre accolades. Lorsque la transformation est réduite à une seule règle on peut remplacer les accolades par une paire de parenthèses et omettre le point-virgule terminant la règle.

Une règle prend la forme suivante :

```
motif  =>  expression
```

ou bien

```
nom  =   motif  =>  expression
```

où *motif* « filtre » une sous-collection A de la collection C à laquelle la transformation est appliquée. A sera remplacée dans C par une collection B calculée par *expression*. La deuxième forme correspond à une règle nommée : dans cette version du langage, le nom d'une règle est

simplement ignoré et sert uniquement de commentaire. Il existe d'autres sortes de règles que nous verrons plus loin.

Informellement, quand une transformation `T` est appliquée à une collection, la collection résultante est construite à partir des sous-collections transformées par `T` (celle qui sont filtrée par les motifs des règles de `T`), et des sous-collections laissées invariantes (celles qui ne sont pas filtrées par les motifs des règles de `T`).

6.1 Les motifs

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

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

où p, p' sont des motifs, x représente une variable de motif, P est un prédicat et exp est une expression dont la valeur sera interprétée comme un booléen. Les explications qui suivent donnent une sémantique informelle de ces motifs :

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

Enregistrement : les motifs `{ ... }` sont utilisées pour filtrer un élément qui est un enregistrement. Le contenu des accolades, semblable à la spécification d'un type enregistrement[1], peut être utilisé pour filtrer un enregistrement devant (ou ne devant pas) contenir un champ (éventuellement contraint par une certain valeur de champ ou un type). Par exemple,

```
{ a, b:string   , c=3, ~d }
```

est un motif qui filtre un enregistrement contenant les champs `a`, `b` de type `string` et `c` de valeur `3`, éventuellement d'autres champs, mais sans champ `d`.

Voisin : `p,p'` est un motif qui filtre deux collections connectées `p` et `p'`.
Par exemple `x,y` filtre deux éléments connectés (i.e. `x` doit être voisin de `y`). La relation de connectivité dépend du type de la collection sur laquelle le filtre est utilisé. Dans tous les cas, `x` et `y` correspondent à des positions distinctes de la collection (mais ces positions peuvent avoir la même valeur).

Liaison : une liaison `p as x` donne le nom `x` à la collection filtrée par `p`. Ce nom peut être utilisé partout ailleurs dans la règle.
Ce nom peut être utilisé par exemple en partie droite d'une règle pour accéder à la valeur filtrée par le motif `p`. Mais la variable `x` peut aussi apparaitre dans la suite du motif, par exemple dans une garde (cf. ci-dessous) ou bien comme dans le motif `x,x` qui filtre deux éléments connectés de *même* valeur. En effet, chaque occurrence de `x` dans une règle doit dénoter la même valeur.

[1] Voir section 4.4, page 25.

Garde : le motif `p / exp` filtre la collection filtrée par `p` si l'expression *exp* s'évalue à *vrai* (lors du filtrage). L'expression *exp* peut utiliser des variables de filtre. Par exemple :

```
y / y>3
```

filtre un élément `y` pourvu que `y` soit supérieur `3`.

Typage : le motif `p:P` est du sucre syntaxique pour `((p as x) / P(x))` où `x` est une nouvelle variable de motif. Par exemple, supposons que `MyRecord` soit le type d'un enregistrement déclaré par le programmeur. Alors `x:MyRecord` filtre un enregistrement de type `MyRecord`.

Répétition : le motif `p+` (resp. `p*`) correspond à une itération `p, ..., p` et filtre donc une sous-collection non vide d'éléments filtrés par `p` (resp. une sous-collection pouvant être vide).

Par exemple, `x+` est un motif équivalent à

$$\texttt{x}_1\texttt{, x}_2\texttt{, ..., x}_n$$

où n peut prendre une valeur entière supérieure à 1 et où les $\texttt{x}_i$ sont des variables de filtre inutilisée ailleurs. Autre exemple, `(x/x>3)+ as X` est équivalent à

$$\texttt{((x}_1\texttt{ / x}_1\texttt{ > 3), ..., (x}_n\texttt{ / x}_n\texttt{ > 3)) as X}$$

et filtre donc une sous-collection non-vide d'éléments tous supérieur à 3.

Certaines constructions de motifs n'existent que pour une sorte de collection. Nous les verrons dans les chapitres correspondants.

Un exemple complet. Le motif

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

sélectionne une sous-collection `S` composée d'entiers inférieurs à 3, et telle que le cardinal de `S` soit inférieur à 5 et que la somme des éléments de la collection soit supérieure 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).

La somme des élément de la collection est obtenue en utilisant la fonction `'fold[+]`. Cette forme est une abréviation de la forme générale de la fonction primitive `'fold` qui prend deux arguments optionnels :

$$\texttt{'fold['zero=}c\texttt{,'fct=}f\texttt{]}$$

(c est une valeur quelconque et f est une fonction binaire) et qui appliquée à une collection d'éléments $x_1, \ldots, x_n$ va calculer

$$f(\ldots f(f(c, x_1), x_2), x_n)$$

On peut utiliser la fonction `'fold` pour définir un motif équivalent au précédent :

```
x+ as S /   'fold['zero=true, 'fct = \(x, y).  int(x) & (x > 3) & y]
          & (5 > 'fold['zero=0, 'fct = \(x, y).  1 + y])
          & (10 < 'fold['zero=0, 'fct = \(x, y).  1 + y])
```

Le premier `'fold` de la gxsarde correspond à la vérification que tous les éléments de `S` sont des entiers strictement supérieur à 3 ; le second `'fold` calcule simplement le nombre d'éléments dans la collection (et est équivalent à `'size`) ; le troisième `'fold` de la garde calcule la somme de tous les éléments de `S`. Ce motif est donc équivalent au précédent (ce qui ne veut pas dire que l'efficacité du filtrage soit la même pour les deux motifs).

6.2 Les règles

Une transformation est un ensemble de règles. Lorsqu'une transformation est appliquée à une collection, la stratégie standard d'évaluation consiste à appliquer en parallèle le plus possible de règles. Une règle peut être appliquée quand son motif filtre une sous-collection. Quand deux règles s'appliquent, les sous-collections filtrées sont sans intersection : on parle d'*application indépendante* des règles. Attention, la même règle peut s'appliquer plusieurs fois, si les applications sont indépendantes.

Voici un exemple de règle qui s'applique à une sous-collection réduite à un seul élément :

```
x:int => x*x
```

Cette règle qui sélectionne un élément entier dans une collection et qui le remplace par son carré. Si cette règle est la seule règle d'une transformation `T`, alors la transformation `T` appliquée à une collection C quelconque retournera une collection de même structure que C et dont les éléments seront les carrés des éléments de C.

Application maximale des règles

Pour donner un exemple destiné à préciser ce que nous entendons par « appliquer en parallèle le plus possible de règles », il nous faut introduire la notation permettant de spécifier une *séquence* (i.e. un ensemble de valeurs organisé linéairement). La séquence de n éléments $a_1, \ldots, a_n$ se note simplement en `MGS` par l'énumération de ses éléments, dans l'ordre et séparé par des virgules. Par exemple, `1, 2, 3` dénote une séquence de trois éléments dont le premier est `1` et le dernier `3`.

Supposons qu'on applique la transformation `S`

```
trans S = {
    R1 =  x / x > 0 => x-1;
    R2 =  x, y / y > 0 => 0, (y+1) ;
}
```

à la séquence `11, 22, 33`. Même en satisfaisant aux contraintes données précédemment, il y a trois manières possibles d'appliquer les règles $\mathtt{R}_1$ et $\mathtt{R}_2$:

$$\underbrace{\mathtt{11}}_{\mathtt{R}_1}, \underbrace{\mathtt{22}}_{\mathtt{R}_1}, \underbrace{\mathtt{33}}_{\mathtt{R}_1} \longrightarrow \mathtt{10, 21, 32} \tag{6.1}$$

$$\underbrace{\mathtt{11, 22}}_{\mathtt{R}_2}, \underbrace{\mathtt{33}}_{\mathtt{R}_1} \longrightarrow \mathtt{0, 23, 32} \tag{6.2}$$

$$\underbrace{\mathtt{11}}_{\mathtt{R}_1}, \underbrace{\mathtt{22, 33}}_{\mathtt{R}_2} \longrightarrow \mathtt{10, 0, 34} \tag{6.3}$$

En effet, toute autre combinaison d'application, comme par exemple

$$\underbrace{\mathtt{11}}_{\mathtt{R}_1}, \mathtt{22, 33} \longrightarrow \mathtt{10, 22, 33}$$

n'applique pas le plus possible de règles : il reste toujours une sous-collection laissée invariante (dans l'exemple `22, 33`) dans laquelle on aurait pu appliquer au moins encore une règle.

Bien que la combinaison (6.1) présente plus d'occurences d'application d'une règle (trois applications de règles contre deux pour les autres), les trois combinaisons présentent un parallélisme maximal dans le sens où il n'existe aucune sous-collection laissée invariante.

La stratégie standard d'application maximale ne spécifie pas laquelle des trois combinaisons (6.1), (6.2) ou (6.3) est effectivement choisie. Cette stratégie est donc *sous-déterminée* (i.e. : elle laisse ouverte plusieurs posssibilités). Mais sa définition impose quand même une propriété très forte :

> *Si aucune règle ne s'applique lors de l'application d'une transformation* `T` *à une collection C, alors c'est qu'aucun motif ne peut filtrer une sous-collection dans C.*

Règles exclusives et inclusives

Plusieurs mécanismes permettent de moduler la stratégie standard d'application maximale et il est possible de contrôler dans une certaine mesure la façon dont les règles sont appliquées dans une transformation.

Les règles dont il est question précédemment sont des règles exclusives. Une *règle exclusive* monopolise la sous-collection filtrée *i.e.* cette dernière ne peut avoir d'intersection avec une sous-collection filtrée par une autre règle.

Les *règles inclusives* n'ont pas cette contrainte. Elles sont principalement utilisées pour modifier des parties différentes d'une valeur comme un enregistrement[2]. Une règle inclusive prend la forme

[2]Actuellement, seul un motif filtrant un unique enregistrement est accepté en partie gauche d'une règle inclusive. Mais l'idée générale est d'étendre les règles inclusives à tout objet complexe, comme par exemple une collection imbriquée (une collection imbriquée A est une collection qui est un élément d'une collection B, ce qui est à distinguer d'une sous-collection C de B composée de plusieurs éléments de B). Les règles inclusives permettent de traiter des objets complexes en spécifiant des traitements sur les parties indépendantes de ces objets.

```
r +=> r'
```

où le symbole de flèche `+=>` indique que la règle est inclusive, où `r` filtre un enregistrement et où `r'` s'évalue en un enregistrement. Une telle règle remplace un enregistrement `R` (filtré par `r`) par l'enregistrement `R+r'`. Le concept de règle inclusive peut sembler *ad-hoc*. C'est en fait un moyen très puissant de réduire significativement l'explosion combinatoire de la spécification des transformations quand les valeurs à transformer sont composée de sous-valeurs largement indépendantes et possédant chacune leurs propres règles d'évolution.

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

```
{x as    v}   +=>   {x = v+1}
{y as    v}   +=>   {y = 2*v}
```

sont déclarée inclusives (car on utilise le symbole `+=>` plutôt que le symbole `=>`) et filtrent respectivement un enregistrement avec un champ `x` et un enregistrement avec un champ `y`. Elles peuvent donc toutes les deux s'appliquer à l'enregistrement `{x=2, y=3}` par exemple. Ainsi si l'on applique les deux règles ci-dessus à `{ x=2, y=3, z=0 }` on obtient `{ x=3, y=6, z=0 }`. Le résultat est calculé ainsi :

$$\Big(\{\ \texttt{x=2, y=3, z=0}\ \} + \{\ \texttt{x=2+1}\ \} \Big) + \{\ \texttt{y=2*3}\ \}$$

ou bien

$$\Big(\{\ \texttt{x=2, y=3, z=0}\ \} + \{\ \texttt{y=2*3}\ \} \Big) + \{\ \texttt{x=2+1}\ \}$$

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

6.3 Spécification des propriétés de la règle

Une règle possède plusieurs propriétés dont le fait d'être inclusive ou exclusive n'est qu'un exemple. Certaines de ces priorités permettent de contrôler la stratégie d'application, d'autres propriétés permettent de moduler la manière de construire la collection résultat de la transformation, ou de tracer l'application d'une règle.

De manière générale, la spécification de ses propriétés se fait en utilisant une flèche composée de la forme

```
={  ...  }=>
```

Le symbole `={` est appelé un « début de flèche ». Le symbole `}=>` est appelé une « fin de flèche » et on peut aussi mettre un nombre quelconque de `=`. Le « corps de la flèche », ici `...` correspond à des spécifications de propriétés de la règle en cours de définition. La spécification d'une propriétés consiste à donner une valeur à un *attribut prédéfini*, et prend donc toujours la forme :

nom = *exp*

La valeur de *exp* *au moment de la définition* de la transformation[3] est utilisé pour donner sa valeur à l'attribut *nom*. On peut spécifier plusieurs propriétés en même temps, en les séparant par des virgules. Nous ne verrons dans ce rapport que les cinq attributs suivants :

1. `exclusive` [4]
2. `priority`
3. `trace`
4. `flat`
5. `on`

Voici un exemple de spécification composée :

```
=={ priority=2, flat=0, trace=1 }==>
```

Il est possible de donner un nom à une flèche composée en utilisant une déclaration simillaire à une définition de type :

`arrow` *df* `= ={ ... }=>`

Cette déclaration, qui doit apparaître au top-level, définit un nom de flèche *df* qui est un alias pour la flèche spécifiée en partie droite.

Le nom de la flèche *df* doit être composé d'un début de flèche d et d'une fin de flèche f. La grammaire en annexe détermine exactement comment un nom de flèche doit être composé. Voici quelques exemples de noms valides :

```
=F=>        ==Toto==>        -eval->        _rho_>        #=>
|==>>       :=>              :--G-->        *==>          ^=>
```

Priorité d'une règle : attribut `priority`

Les règles exclusives sont appliquées avant les règles inclusives. Un élément filtré par une règle exclusive ne peut pas être candidat à l'application d'une autre règle (inclusive ou exclusive).

Par ailleurs, ont peut associer une priorité (un entier) aux règles pour spécifier leur précédence dans chacune des classes de règles. La stratégie est d'essayer d'appliquer les règles de plus forte priorité en premier. La priorité par défaut est 0. On spécifie une priorité de la manière suivante :

motif `=={priority=`n`}==>` *expression*

[3] sauf pour la propriété `on`.

[4] Nous avons déjà vu cet attribut. En effet, la syntaxe `+=>` est une abréviation de `={exclusive=0}=>`.

Quand deux règles de même priorité peuvent s'appliquer, `MGS` applique la première règle qui est définie. Par exemple dans les deux transformations suivantes :

```
trans S = {                    et      trans T = {
    R  =   x => x+1;                       R' =   y => y-1;
    R' =   y => y-1;                       R  =   x =>{ priority=10 }=> x+1;
}
```

la règle `R'` ne s'applique jamais.

Mais il faut remarquer que spécifier un ordre total entre les règles ne suffit pas à définir une stratégie déterministe d'application des règles, car il faut encore spécifier l'ordre de parcours des éléments de la collection (dans l'exemple précédent, les règles s'appliquant à un seul élément, il n'y a pas de problème). Le point de vue adopté en `MGS` est donc que le programmeur doit se fonder sur la seule propiété réellement assurée et enoncée page 37 : « si aucune règle ne s'applique sur une collection C, alors c'est qu'aucun motif ne peut filtrer une sous-collection dans C. ».

Trace d'une règle : attribut `trace`

Pour faciliter l'analyse et le déverminage des transformations, il est possible de demander à ce que l'application de chaque règle soit tracée, en spécifiant la propriété :

```
...  ={ trace=exp }=> ...
```

si *exp* est une expression qui s'évalue à *vrai* (au moment où la règle est définie), alors chaque tentative d'application de la règle générera une trace sur la sortie standard.

Voici un exemple. On définit une règle qui incrémente de 1 un élément, à la condition que celui-ci soit supérieur à 3 :

```
trans T = (x/(x>3) ={trace=1}=> x+1) ;;
```

L'attribut `trace` de la règle indique qu'un message de trace doit être émis à chaque tentative d'application. On applique à présent cette transformation à une séquence de 4 éléments (les entiers successifs de 2 inclus à 5, Cf. le chapitre suivant pour la notion de séquence) :

```
T((2, 3, 4, 5));;
```

Voici les messages émis par l'interprète quand il évalue l'expression précédente :

```
// try (x /x > 3) ={flat, trace}==> x + 1;
// fail.

// try (x /x > 3) ={flat, trace}==> x + 1;
// fail.
```

```
// try (x /x > 3) =={flat, trace}==> x + 1;
// match.
// resulting value:  5

// try (x /x > 3) =={flat, trace}==> x + 1;
// match.
// resulting value:  6

2, 3, 5, 6
```

La dernière ligne correspond à l'affichage du résultat.

Substitution et applatissement : attribut `flat`

Cet attribut contrôle la manière dont la sous-collection produite en partie droite d'une règle remplace la sous-collection filtrée dans le résultat. Soit la règle

`x =>` *cte*

qui remplace un élément `x` dans une collection C par une constante *cte*. Supposons par exemple que C soit une sequence et que *cte* soit aussi une séquence ; pour fixer les idées C est la séquence de 3 éléments `(1, 2, 3)` et *cte* est la séquence de deux éléments `(11, 12)`. La règle précédente admet deux interprétations.

1. On remplace un élément `x` par *un* élément *cte* ; comme *cte* est une séquence, le résultat est une imbrication de séquences. On obtient donc une séquence de 3 éléments, chaque élément étant une séquence de 2 entiers : `((11,12), (11,12), (11,12))`.
2. Ou bien, on peut considérer que `x` filtre une sous-sequence réduite à un élément de C, et que cette sous-collection est substitué par *cte*. La sequence *cte* est donc une sous-sequence du résultat et on doit donc insérer les éléments de *cte* au même niveau que les autres éléments du résultat. On obtient donc une séquence *plate* de 6 entiers : `(11, 12, 11, 12, 11, 12)`.

Ces deux interprétations sont possibles quel que soit le type de collection, et quel que soit le motif exact en partie gauche *quand* l'expression en partie droite d'une règle possède le même type que la collection sur laquelle s'applique la transformation.

La seconde interprétation est l'interprétation par défaut. Elle correspond à la propriété `flat` à *vrai*. On peut donc forcer la première interprétation en indiquant une propriété `flat` à false.

Voici un exemple où on utilise les deux comportements à la fois. La transformation

```
trans T = {
    x/x>3  ={flat=0}=>  x,(x+1);
    y/y<3  ={flat=1}=>  y,(y+1);  // ici la propriété flat a sa valeur par défaut
};;
```

va remplacer tous les éléments x d'une séquence qui sont supérieurs à 3 par une séquence imbriquée de 2 éléments (x et son successeur), et va insérer à la place de tous les éléments y inférieur à 3 deux éléments (y et son successeur). Ainsi, l'application de `T` à la liste `(1, 4, 2)` a pour résultat `(1, 2, (4, 5), 2, 3)` .

Règles conditionnelles : attribut `on`

La valeur de la propriété `on` est une expression qui est évaluée au moment où la règle est susceptible de s'appliquer[5], mais avant filtrage. Si la valeur obtenue est *vrai*, alors la règle devient une règle candidate. Le mécanisme de filtrage et le mécanisme de priorité permettra de choisir laquelle des règles candidates s'appliquera effectivement. Si la valeur retournée par l'évaluation de la propriété `on` est fausse, cette règle est tout simplement ignorée. Cette stratégie est illustrée à la figure 6.1.

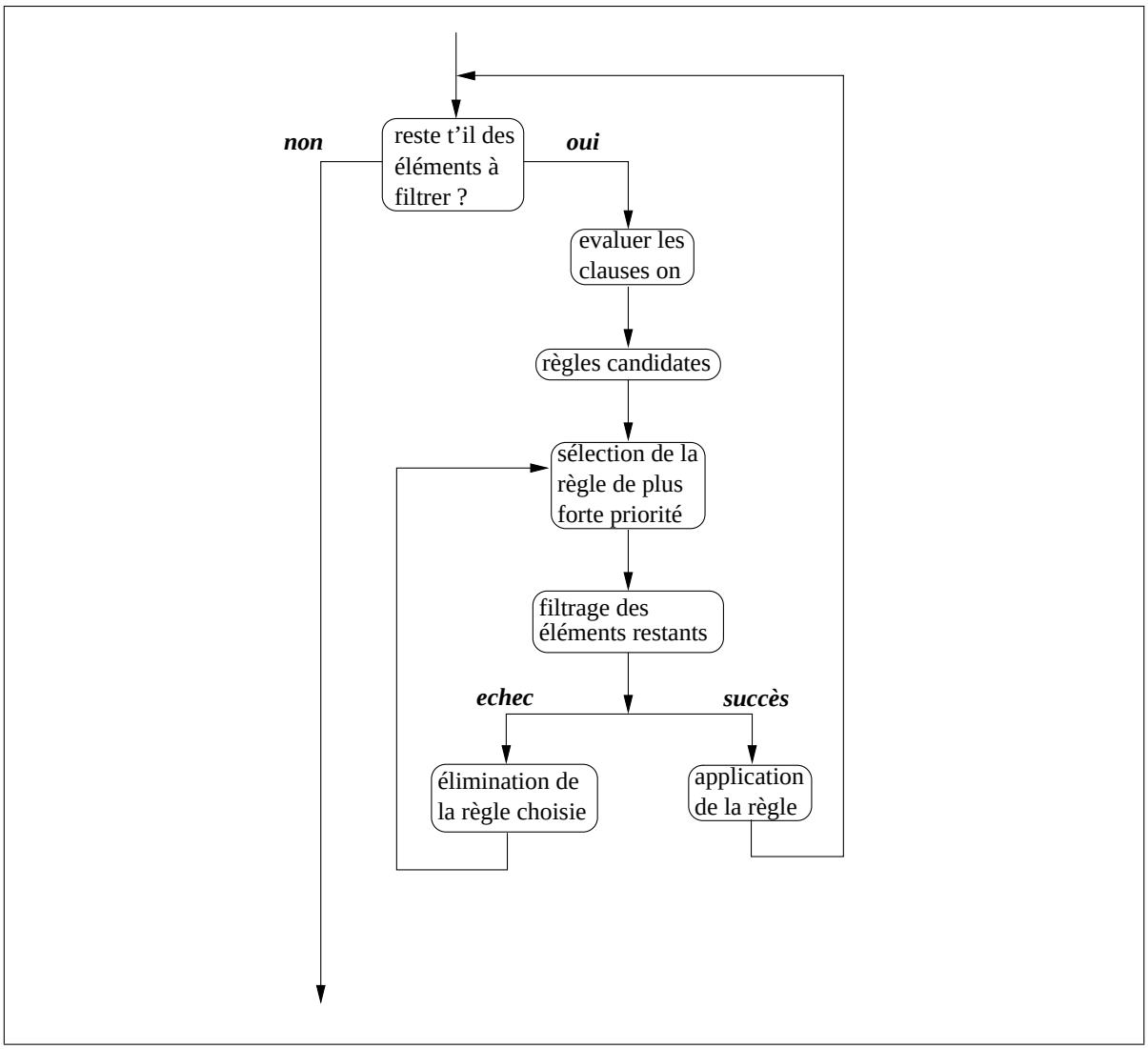


Fig. 6.1: Algorithme d'application des règles.

[5] contrairement à l'évaluation des autres propriétés.

L'expression associée à la propriété `on` ne peut pas faire référence à des variables du motif de la règle, puisqu'elle est évaluée avant le filtrage. Mais elle peut faire référence à des arguments de la transformations, Cf. ci-dessous.

Voici un exemple où une variable globale `flag` est utilisée pour modifier le comportement d'une transformation :

```
flag := <undef>;;

trans T = {
    R1 =    x ={ on = flag }==> x+1;
    R2 =    y / y > 3  ==>  y-1;
}
```

La propriété `on` de la règle `R1` est évaluée chaque fois qu'on peut encore filtrer un élément dans une collection. Dans l'expression suivante

```
(flag := 0; T((1, 2, 3, 4)));;    // s'évalue en (1, 2, 3, 3) car R1 n'est pas active
```

à chaque évaluation de la propriété `on`, on obtient la valeur *faux* et la règle `R1` est donc ignorée, permettant l'application éventuelle de la règle `R2`. Par contre, dans cet exemple :

```
(flag := 1; T((1, 2, 3, 4)));;    // s'évalue en (2, 3, 4, 5)
```

les règles `R1` et `R2` sont toutes les deux actives, mais la règle `R2` n'est jamais activée à cause des priorités (la règle `R1` s'applique toujours).

6.4 Les transformations

Les transformations sont des fonctions comme les autres mais qui agissent sur les collections et qui sont définies par un ensemble de règles. Comme pour les fonctions, ont peut définir des arguments optionnels et comme pour les fonctions, ont peut itérer l'application d'une transformation.

Variables locales et argument par défaut

Les arguments optionnels sont définis de la même manière que pour les fonctions. `MGS` n'est pas un langage purement fonctionnel, et les arguments optionnels doivent être vus comme des variables locales « impératives » qui peuvent être modifiées par les règles d'une transformation ou bien le corps d'une fonction. De telles variables peuvent être utilisées pour « transferer de l'information » entre applications de règles.

Par exemple, on peut exprimer une condition à l'application d'une règle, comme dans la transformation

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

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

Gérer l'application des transformations

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

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

$T[n]\,(c)$	dénote l'application de `n` étapes de transformation à `c`
$T[\,$`'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

D'autres stratégie d'applications d'une transformations sont imaginables[6] mais ne sont pas encore implémentés.

[6]Par exemple : le *mode stochastique* où l'ordre d'application des règles exclusives est choisi au hasard et où la probabilité pour qu'une règle soit choisie est proportionnelle à sa priorité, et le *mode asynchrone* où une seule règle exclusive est appliquée à chaque étape de transformation.

Chapitre 7

Collections monoïdales

7.1 Les structures d'ensemble, de multi-ensemble et de sequence

Plusieurs types de collections topologiques sont disponibles en `MGS`. Dans ce chapitre, nous nous focalisons sur les *ensembles*, les *multi-ensembles* et les *séquences* :

- Les séquences correspondent à des listes ou bien des vecteurs de taille variable. Les éléments d'une séquence sont organisés par une relation d'ordre total définissant le premier élément de la séquence, le deuxième élément, etc.
- Les multi-ensembles correspondent à des ensembles d'éléments où les répétitions sont possibles[1]. Ce type de structure est aussi appelé un *sac* ou bien un *bag* en anglais.
- Les ensembles correspondent à la notion mathématique d'ensemble d'éléments. Chaque élément est présent au plus une fois dans l'ensemble.

Ensembles, multi-ensembles et sequences comme monoïdes. Ces trois type de collections sont qualifiées de *monoïdales* car elles peuvent être vue comme les valeurs d'un monoïde. Un monoïde est un ensemble (ici l'ensemble des valeurs $\mathcal{V}$) muni d'une opération associative : *join* que nous noterons en `MGS` simplement par une virgule : « ,+.1667em ». En effet :

- une séquence correspond à un opérateur join sans propriété spéciale (autre que l'associativité) ;
- un multi-ensemble correspond à un operateur join *commutatif*[2] (i.e. l'ordre des éléments dans la séquence n'a pas d'importance) ;
- un ensemble correspond à un opérateur join à la fois commutatif et *idempotent*[3].

En effet, supposons que l'opérateur « ,+.1667em » soit commutatif. Alors la séquence

```
1,2,3
```

peut aussi s'écrire

[1] On peut formaliser un multi-ensemble M d'éléments pris dans $\mathcal{V}$ par une fonction totale $Ind_M : \mathcal{V} \longrightarrow \mathbb{N}$, l'indicatrice de M, qui associe à chaque élément de $\mathcal{V}$ le nombre de ses occurences dans M. La fonction Ind_M prend la valeur 0 sauf en un nombre fini d'éléments de $\mathcal{V}$.

[2] Un opérateur binaire $\star$ est commutatif si $x \star y = y \star x$ pour tout x et tout y.

[3] Un opérateur binaire $\star$ est idempotent si $x \star x = x$ pour tout x.

`1,3,2` *ou bien* `2,1,3` *ou bien* `2,3,1` *ou bien* `3,1,2` *ou bien* `3,2,1`

en faisant jouer la commutativité. Un élément d'un monoïde commutatif sur $\mathcal{V}$ représente donc bien un multi-ensemble d'éléments de $\mathcal{V}$. La propriété d'idempotence permet de supprimer les doublons et donc de « normaliser » un multi-ensemble en un ensemble.

Un traitement uniforme. En `MGS`, le traitement des ensembles, des multi-ensembles et des séquences est très similaire car il repose sur la notion de monoïde. C'est pourquoi ces trois types de collections sont regroupés dans un même chapitre.

7.2 Enumérer les élements d'une collection monoïdale

Les collections en `MGS` sont des collections complexes, c'est-à-dire qu'elle peuvent contenir des éléments de n'importe quel type. Les trois collections monoïdales ne contenant aucun élément se notent :

```
():set   ou bien   set:() // l'ensemble vide
():bag   ou bien   bag:() // le multi-ensemble vide
():seq   ou bien   seq:() // la séquence vide
```

L'operateur *join* est noté en `MGS` par « `,`+.1667em », quel que soit le type de la collection monoïdale. C'est le type des arguments de « `,`+.1667em » qui permet de décider quelle est la bonne interprétation :

join	`x`	c de type C
`x`	`x,x` : séquence de 2 éléments	`x,`c : une collection de type C
c de type C	c`,x` : une collection de type C	c,c : une collection de type C

Dans le tableau précédent, C représente une collection monoïdale et `x` représente un élément qui n'est pas du type C. Par exemple :

```
1, set:()
```

représente un ensemble contenant un seul élément (un singleton) : `1`. Si `x` n'est pas une collection monoïdale, alors `x,x` construit une séquence de 2 éléments. Par exemple :

```
1, 2 // est équivalent à :  1, 2, seq:()
```

construit une séquence dont le premier élément vaut `1` et le second `2`.

L'opérateur join est associatif à droite et donc :

```
u, ..., x, y, z     s'interprète comme     u, (..., (x, (y, z))...)
```

autrement dit :

```
0, 1, 2, 3, 4, 5, ():set
```

s'interprète comme `0, (1, (2, (3, (4, (5, ():set)))))` et représente donc l'ensemble des entiers positifs ou nuls jusqu'à 5 compris. Pour construire une séquence d'éléments (qui ne sont pas des collections), il suffit donc de les énumérer avec des virgules :

```
0, 1, 2, 3, 4, 5
```

construit la liste des entiers positifs ou nuls jusqu'à 5.

Comme join est une fonction `MGS`, ses arguments sont des expressions arbitraires, et on peut écrire par exemple :

```
2, (2+2), (2*2), bag:()
```

expression qui va construire un multi-ensemble où la valeur `2` apparaît une fois et la valeur `4` apparaît 2 fois. L'interprète `MGS` affiche :

```
2, 4, 4, ():bag
```

comme résultat de l'évaluation de cette expression. Et `2, (2+2), (2*2), set:()` va construire un ensemble de deux éléments qui s'affichera comme `2, 4, ():set` (il n'y pas de répétition d'éléments dans un ensemble).

Opérateur join et liste d'arguments

L'utilisation de la virgule comme opérateur join induit une ambiguité avec la notation utilisée pour l'application des fonctions. En effet

```
f(1, 2, ():X)
```

(où `X` est `seq`, `bag` ou `set`) peut s'interpréter comme

1. l'application de la fonction `f` aux 3 arguments `1`, `2` et `():X` ;
2. ou bien comme l'application de la fonction `f` à un seul argument `1,2,():X` qui est une collection de type `X` construite avec deux opérateurs join.

C'est la première interprétation qui est utilisée par défaut. Pour forcer la seconde interprétation, il suffit d'utiliser une paire de parenthèses supplémentaires, ce qui force la construction de la collection :

```
f( (1,2,():X) )
```

Par exemple :

```
g(1, 2, (10, 11, 12), set:())
```

s'interprète comme une fonction à 4 arguments : les deux premiers arguments sont les entiers 1 et 2, le troisième argument est une séquence de longueur 3 et le dernier argument est l'ensemble vide.

Construction de collections imbriquées

L'expression :

```
():bag, ():set
```

va construire un ensemble : c'est l'argument à droite de la virgule qui impose son type quand cet argument est une collection. Le résultat sera donc un ensemble contenant un élément (à savoir le multi-ensemble vide). On a là un exemple de collection *imbriquée*, i.e. une collection qui est un élément d'une autre collection.

Cependant l'opérateur join ne permet pas de construire une séquence de séquences, ou bien un multi-ensemble de multi-ensembles ou bien un ensemble d'ensembles. En effet, si par exemple `x = 1,2,3` et `y = 4,5,6` sont deux séquences d'entiers, alors

```
x, y = 1, 2, 3, 4, 5, 6
```

est une une séquence d'entiers. Pour construire une séquence de deux éléments dont chacun est une séquence d'entiers, il ne faut pas utiliser l'opérateur join, mais l'opérateur *cons* noté en `MGS` par `::` . L'expression `x::`C où `x` est une valeur quelconque et C une collection monoïdale quelconque, construit une nouvelle collection contenant les éléments de C plus la nouvelle valeur `x`. Par exemple, l'expression

```
x ::  y ::  ():seq
```

est interprétée comme `x :: (y :: ():seq)` car l'opérateur cons est associatif à droite et a pour valeur

```
(1, 2, 3), (4, 5, 6), ():seq
```

qui est une séquence de deux séquences. Remarquons que `x::y` s'évalue en `(1, 2, 3), 4, 5, 6` qui est une séquence de quatre éléments, le premier étant une séquence et les suivant étant des entiers.

7.3 Opérations sur les collections monoïdales

De nombreux opérateurs sont disponibles pour manipuler une collection monoïdale. Ils sont généralement surchargées pour agir uniformément sur n'importe lequel des trois types de collections monoïdales. Dans la description ci-dessous, on suppose que que `c` est une collection monoïdale, `e` est une valeur quelconque et `n` est un entier positif. Nous omettons aussi l'apostrophe dans le nom de la fonction.

`c @ c'` ***(append)*** concatène les deux collections `c` et `c'` qui doivent avoir le même type. Si ce type est ensemble, alors `@` réalise l'union ensembliste, si ce type est multi-ensemble,

alors on a l'union des multi-ensemble[4] et si ce type est séquence, alors on a la concaténation des séquences.

`bag(e)` est le prédicat qui retourne *vrai* si `e` est un multi-ensemble.

`bagify(e)` si `e` n'est pas une collection monoïdale, alors construit un multi-ensemble avec un seul élément qui est `e`. Si `e` est une collection, alors `bagify(e)` construit un multi-ensemble formé par les éléments qui sont dans `e`.

`e::c` ***(cons)*** permet de construire une collection en rajoutant un élément `e` à une collection `c` (cf. le paragraphe ci-dessus).

`delete(e,c)` retourne une nouvelle collection contenant les éléments de `c` excepté la première occurence de `e` si elle existe. Par exemple `delete(1, (1, 2, 3, 1))` calcule la collection `2, 3, 1`.

`empty(c)` retourne une valeur *vrai* si `c` est une collection qui ne possède aucun élément.

`hd(c)` ***(head)*** retourne un élement de la collection `c`. Voir le paragraphe 7.4.

`iota(n,c)` retourne la collection `0, 1, ..., (n-1), c`. Par exemple `iota(10, set:())` construit l'ensemble des entiers de 0 à 9 inclus alors que `iota(10, seq:())` construit la séquence des entiers successifs de 0 à 9.

`member(e,c)` est un prédicat qui prend la valeur *vrai* si l'élément `e` est présent dans la collection `c`. Si la collection `c` est un multi-ensemble, la valeur renvoyée par member est le nombre d'occurence de `e` (remarquez que si `e` n'apparait pas dans le multi-ensemble `c`, alors cette valeur est zéro, ce qui est bien interprété comme la valeur *faux*).

`c,c'`**+.2222em** `e,c`**+.2222em** `c,e`**+.2222em** `e,e` ***(join)*** permet de construire des collections monoïdales par énumération des éléments (cf. le paragraphe ci-dessus).

`monoidal(e)` est le prédicat qui retourne *vrai* si `e` est une collection monoïdale : `monoidal(x) = seq(x) | set(x) | bag(x)`.

`seq(e)` est le prédicat qui retourne *vrai* si `e` est une séquence.

`sequify(e)` si `e` n'est pas une collection monoïdale, alors construit une séquence avec un seul élément qui est `e`. Si `e` est une collection, alors `sequify(e)` construit un multi-ensemble formé par les éléments qui sont dans `e` et pris dans un ordre arbitraire (mais si `e` est une séquence, alors `sequify(e)` retourne `e`).

`set(e)` est le prédicat qui retourne *vrai* si `e` est un ensemble.

[4] Si Ind_M est l'indicatrice du muti-ensemble M et si Ind_N est l'indicatrice du multi-ensemble N, alors `\x.` Ind_M`(x) + `Ind_N`(x)` est l'indicatrice de M `@` N.

`setify(e)` si `e` n'est pas une collection monoïdale, alors construit un ensemble avec un seul élément qui est `e`. Si `e` est une collection, alors `setify(e)` construit un ensemble formé par les éléments qui sont dans `e`.

`setkind` construit une collection d'un type donné, cf. section 10.1.

`size(c)` retourne le nombre d'éléments dans la collection `c`. Les collections « vides » ont une taille de zéro.

`tl(c)` ***(tail)*** retourne une collection de même type que `c` et obtenue à partir de `c` en supprimant un élement. Voir le paragraphe 7.4.

Les fonctions suivantes ne s'appliquent qu'à une séquence `s` :

`last(s)` retourne le dernier élément de `s`.

`reverse(s)` retourne une nouvelle séquence dont les éléments sont les éléments de `s` pris dans l'ordre inverse.

`take(s, n)` retourne le $\texttt{n}^{\texttt{eme}}$ élément de la séquence `s`. La numérotation des éléments débutent par 1. Si `n` est négatif ou nul, alors la recherche d'un élément se fait « à partir de la fin et à rebours ». Par exemple `take(1, 2, 3, 4), 2)` retourne l'entier `2` et `take(1, 2, 3, 4), -2)` retourne l'entier `3`.

7.4 Récursion primitive sur les collections monoïdales

Les deux opérateurs `hd` et `tl` agissent sur n'importe quel type de collection monoïdale. Leur nom provient des fonctions similaires sur les listes qui existent en LISP : *head* et *tail*. Ils sont tels que

C `=` `hd(`C`)` `,` `tl(`C`)`

Ces deux opérateurs ne peuvent s'appliquer qu'à une collection qui n'est pas vide. Avec le prédicat `empty`, ils permettent de programmer une forme de *récursion primitive* sur la structure d'une collection monoïdale :

```
fun Iter(e,g) = \C. if empty(C) then e
                                else g(hd(C), Iter(e,g)(tl(C)))
                    fi;;
```

Une expression `Iter(e,g)` définit une fonction h (une lambda-abstraction) qui prend une collection pour argument. La valeur de h sur une collection vide est donnée par `e`. La fonction `g` combine l'élément obtenu par `hd` à partir de la collection initiale et le résultat obtenu en

appliquant h sur la sous-collection obtenue par application de `tl`. Plus synthétiquement, la fonction h vérifie les deux équations :

$$h(\,X\texttt{:()}) \;=\; \texttt{e}(7.1)$$

$$h((a\,\texttt{,}C)) \;=\; \texttt{g}(a, h(C))$$

où *X* est un type de collection monoïdale `seq`, `set` ou `bag`. Notez que la fonction h est une fonction unaire et que donc $h((a\,\texttt{,}C))$ dénote l'application de la fonction h à un argument qui est une collection construite par join de a et de C. Par contre `g` est une fonction binaire et $\texttt{g}(a, h(C))$ dénote l'application de `g` aux deux arguments a et C.

Les fonctions h créées par application de la fonction `Iter` peuvent être *polymorphes* en ce sens qu'elles peuvent s'appliquer indifférement à des séquences, des multi-ensembles et des ensembles[5]. Par exemple :

```
fun SUM(C) = Iter(0, \(x,y).x+y)(C);;
```

définit une fonction polymorphe `SUM` qui agit sur toutes les collections monoïdales (dont les éléments sont des scalaires) pour calculer la somme de tous les éléments. Par exemple

```
SUM((1, 2, 3, 2, 1, seq:()))    s'évalue en   9
SUM((1, 2, 3, 2, 1, set:()))    s'évalue en   6
SUM((1, 2, 3, 2, 1, bag:()))    s'évalue en   9
```

Homomorphismes. Les fonctions h qu'on peut définir avec la fonction `Iter` sont tellement importantes et communes, qu'elles portent un nom : ce sont les *homomorphismes* (de liste, d'ensemble ou de multi-ensemble suivant le type de la collection argument). C'est pourquoi la fonction `Iter` que nous avons définie est en fait une fonction primitive en `MGS` appelée `'fold` :

h `= fold['zero=e, 'fct=g]`

La fonction `fold` est appelé un *itérateur* car il permet de « traverser » une structure de donnée en faisant des calculs au passage.

Il existe un autre itérateur de collections en `MGS` : c'est la fonction `map` qui applique une fonction à chaque élément d'une collection et qui retourne la collection des résultats :

[5] Afin d'avoir la fonction `Iter` « la plus polymorphe possible », il est judicieux de définir une variante `Iter'` où l'argument `e` est une fonction à appliquer sur la collection vide plutôt que directement une valeur :

$h(\,X\texttt{:()}) = \texttt{e}(\,X\texttt{:()})$

Cet astuce permet de faire dépendre le type du résultat du type de l'argument. Par exemple

`Iter'((\x.x), (\(x,y).`*f*`(x),y))`

correspond à la fonction `map(`*f*`)` qui applique la fonction unaire *f* à chaque élément d'une collection et retourne la collection des résultats.

```
map[\x.x+1]((1, 2, 3, seq:()))   s'évalue en   2, 3, 4, seq:()
map[\x.x+1]((1, 2, 3, set:()))   s'évalue en   2, 3, 4, set:()
map[\x.x+1]((1, 2, 3, bag:()))   s'évalue en   2, 3, 4, bag:()
```

Noter que `map[f]` est l'abréviation de `map['fct=f]`. En un certain sens, `map` est un itérateur plus général que le `fold` car il s'applique à n'importe quel type de collection et pas seulement aux collections monoïdales ; mais c'est aussi un itérateur plus spécialisé car dans le cas des collection monoïdale, il peut s'écrire à partir du `fold`.

Exemples d'utilisation. La fonction `SUM` se définit simplement comme

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

La fonction `'size` peut se définir comme

```
fold['zero=0, 'fct=\(x,y).1+y]
```

La fonction `hd` sur les séquences, se définit aussi en terme de `fold` :

```
fold['zero=<undef>, 'fct=\(x,y).x]
```

Le prédicat qui vérifie que tout élément d'une collection monoïdale vérifie une condition *P* donnée :

```
fold['zero=1, 'fct=\(x,y).y&P(x)]
```

Le prédicat qui vérifie qu'il existe au moins un élément d'une collection monoïdale vérifie une condition *P* donnée :

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

La fonction qui calcule l'élément maximal dans collection :

```
fold['zero=<undef>, 'fct=\(x,y).if undef(y) then x else max(x,y) fi]
```

La fonction qui calcule l'ensemble des parties d'un ensemble se programme simplement[6] grâce à `fold` et `map` :

[6]Cette fonction se programme comme donné ci-dessus dans l'interprète OCAML. Sa définition doit être un peu adaptée pour l'interprète C++ qui ne permet pas l'occurence de variables liées extérieurement dans une lambda-définition imbriquée (occurence de `a` dans définition de `\c.(a,c)` sous l'abstraction `\(a,C)`). Mais en utilisant la curryfication d'une fonction auxilliaire le remède est simple et le code devient :

```
fun aux(a,c) = a,c
fun Power(x) = fold['zero = ((():set)::():set),
                    'fct=(\(a,C). (C , map[aux(a)](C))](x);;
```

```
fun Power(s) = fold['zero = (():set) ::  ():set,
                    'fct=\(a,C). (C , map[\c.(a,c)](C))](s);;
```

Par exemple `Power((1,2,3,():set))` retourne la réponse :

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

La valeur de `'zero` est l'ensemble singleton qui contient l'ensemble vide comme élément. En changeant simplement la valeur de l'argument `'zero` pour la sequence-singleton contenant la séquence vide, on obtient les « parties d'une séquences » qui sont les séquences formées à partir des éléments de la séquence initiales pris dans l'ordre. Ainsi

```
fold['zero=():seq::():seq, 'fct=\(a,C).(C,map[\c.(a,c)](C))]((1, 2, 3))
```

s'évalue en :

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

7.5 La relation de voisinage dans les collections monoïdales

L'opérateur join et ses propriétés induisent directement la relation de voisinage entre les éléments d'une collection monoïdale. C'est pourquoi ce n'est pas par hasard que nous avons utilisé la virgule en `MGS` et la notation `,` dans la section 5.2.

Topologie des ensembles. Dans un ensemble, un élément `x` est voisin de n'importe quel autre élément.

Topologie des multi-ensembles. La topologie des multi-ensembles est la même que la topologie des ensembles : deux éléments quelconques sont voisins. La différence est que deux éléments quelconques n'ont pas forcément une valeur différente.

Topologies des sequences. La topologie des séquence est la topologie naturellement attendue : si la séquence possède au moins deux éléments, alors tous les éléments à l'exception du premier et du dernier ont deux voisins (appelé le voisin *gauche* et le voisin *droit*). Le premier et le dernier élément n'ont qu'un seul voisin (un voisin droit pour le premier élément et un voisin gauche pour le dernier). Si la séquence est réduite à un singleton, alors ce singleton n'a pas de voisin.

Ces relations de voisinage sont naturellement induites par l'operateur join : deux éléments `x` et `y` sont voisins dans une collection monïdale C si on peut écrire C sous la forme :

C `=` A`, x, y,` B

où A et B sont des collections de même type que C. Par exemple, en utilisant l'associativité et la commutativité de l'opérateur join dans les ensembles, on a :

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

ce qui montre que `1` et `2` sont voisins, que `2` et `3` sont voisins et que `1` et `3` sont voisins.

7.6 Transformation d'une collection monoïdale

Ce sont les relations de voisinages ci-dessus qui sont utilisées dans le filtrage d'une collection monoïdale. Il n'y a rien à ajouter par rapport à ce qui a été introduit dans le chapitre 6. Nous nous contenterons donc d'illustrer le mécanisme des transformation sur les collections monoïdales par trois exemples. D'autres exemples plus complets sont développés dans la section 8

Element maximal

La transformation suivante

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

peut être utilisée sur n'importe quel type de collection monoïdale et son point-fixe retourne une collection (de même type) contenant les éléments maximaux de la collection argument. Par exemple

```
MAX['fixrule]((1, 2, 2, 1, 0, 2, set:())) ⟶ 2, set:()
MAX['fixrule]((1, 2, 2, 1, 0, 2, bag:())) ⟶ 2, 2, 2, bag:()
MAX['fixrule]((1, 2, 2, 1, 0, 2, seq:())) ⟶ 2, 2, 2, seq:()
```

Remarquons que si on recherche les éléments maximaux dans un ensemble ou un multi-ensemble, alors on peut simplifier la transformation :

```
trans MAX2 = ( x,y / (x > y) => x )
```

en effet, dans un ensemble ou un multi-ensemble, si on a `x,y` alors on a aussi `y,x`, ce qui permet d'énumerer tous les cas de comparaisons avec une seule règle (mais ce n'est pas le cas dans les séquences).

7.7 Map et fold

Il est facile de définir une transformation qui réalise la fonction `map`. Nous allons détailler pas à pas la solution finale. Une première version qui vient à l'esprit est la suivante :

```
trans MAP1 = ( x => f(x) )
```

où `f` est la fonction qu'on veut appliquer à chaque éléemnt de la collection. Mais la fonction `MAP1` doit être réécrite pour chaque nouvelle fonction `f`. Pour pallier ce problème, il suffit de faire de `f` un argument ; les transformations ne peuvent prendre que des arguments optionnels (outre la collection sur laquelle elle s'applique), d'où une seconde version :

```
trans MAP2[fct] = ( x => fct(x) )
```

qu'on peut utiliser par exemple ainsi :

```
MAP2[fct=\x.x+1]((1, 2, 3)) ⟶ 2, 3, 4
```

Mais cette version possède aussi le léger défaut suivant : si on l'applique en oubliant de fournir une valeur à l'argument optionnel `fct`, celui-ci garde sa valeur par défaut, qui implicitement a pour valeur `<undef>` et donc l'évaluation de

```
MAP2((1, 2, 3))
```

provoque des erreurs du type :

```
Error:  eval:  in expression
    fct(x)
the expression in functionnal position:
    fct
evaluates to
    <undef>
which is not a function/state/transformation/collection/cloture
```

ce qui n'est pas très informatif. Cet inconvénient se corrige aisément en définissant explicitement la valeur par défaut de l'argument `fct` :

```
trans MAP4[fct = \x.error("erreur MAP: pas d'argument fct present")]
= ( x => fct(x) )
```

avec cette version `MAP4`, le message d'erreur est plus explicite. Cette fonction n'est cependant pas tout à fait correcte. En effet, supposons que la fonction `fct` renvoie une séquence (et que la transformation est appliquée à une séquence) ; comme par exemple dans :

```
MAP4[fct = \x.():seq]((1, 2, 3))
```

L'évaluation de cette expression calcule la séquence vide `():seq` alors que le résultat attendu est une séquence de 3 éléments, chacun étant une séquence vide :

```
(():seq) ::  (():seq) ::  (():seq) ::  ():seq
```

En effet, la stratégie de substitution par défaut, Cf. section 6.3 page 41, consiste à insérer la collection produite en partie droite de la règle, et non de remplacer simplement l'élément filtré en partie gauche. Pour cela, il suffit de spécifier la valeur fausse pour l'attribut `flat` de la règle :

```
trans MAP[fct = \x.error("erreur MAP: pas d'argument fct present")]
= ( x =={ flat=0 }==> fct(x) )
```

ce qui constitue notre version finale. Remarquons qu'avec une transformation pour coder la fonction `map`, on n'a pas la même syntaxe pour le passage de l'argument fonctionnel et de la collection. On ne peut pas non plus currifyer l'application (l'application partielle ne peut se faire sur les arguments optionnels). On peut utiliser une fonction auxilliaire servant d'interface si on veut retrouver exactement l'écriture habituelle de l'application :

```
fun map(f, l) = MAP[fct=f](l)
```

Trie d'une séquence

Une procédure de tri simillaire au « tri-bulle » est immédiate à définir en `MGS` :

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

Cette transformation n'est pas exactement un tri-bulle, car l'inversion de 2 éléments prend place à des positions a priori arbitraires (en parallèle) et l'élément maximal, quand il n'est pas à la bonne place, ne remonte pas (comme une bulle) vers la fin de la collection comme dans le trie-bulle séquentiel.

Chapitre 8

Exemples de programme `mgs` sur les collections monoïdales

The following examples are freely inspired by examples given for Γ, P systems, L systems and the `81/2` language [Mic96a].

8.1 Convex Hull

The convex hull of a set P of points in the plane is defined to be the smallest convex polygon containing them all. It is easy to show that the vertices of the convex hull of P are elements of P. The program to compute the convex hull considers a point X and a triple of points U, V and W and eliminates X if it falls inside the triangle U, V, W.

We first define a record *Point* which has a field x and a field y. We define also two variables named *true* and *false* for convenience (however each value can be interpreted as a boolean when needed as in the `C` programming language).

```
state Point = {x, y} ; ;
false := 0 ; ;   true := ~false ; ;
```

A point X falls inside the points U, V and W iff it exists α, β and γ between 0 and 1 such that : $\alpha U + \beta V + \gamma V = X$ and $\alpha + \beta + \gamma = 1$. This gives a linear system of three equations with three unknowns α, β and γ wich can be solved using the determinant method. This explains the function *inside* defined below. The function *det* computes a 3×3 determinant ; the function *check* tests if a value is between 0 and 1 ; and *inside2* is an auxilliary function that does the

real work.

```
fun check(d) = (d >= 1)||(d <= 0) ;;
fun det(a,b,c,d,e,f,g,h,i) =
    a*(e*i - h*f) - d*(b*i - h*c) + g*(b*f - e*c) ;;
fun inside(X,U,V,W) =
    inside2(X,U,V,W, det(U.x,V.x,W.x,U.y,V.y,W.y,1,1,1)) ;;
fun inside2(X,U,V,W,d) =
    if d == 0 then false
    else if check(det(X.x,V.x,W.x,X.y,V.y,W.y,1,1,1)/d) then false
    else if check(det(U.x,X.x,W.x,U.y,X.y,W.y,1,1,1)/d) then false
    else if check(det(U.x,V.x,X.x,U.y,V.y,X.y,1,1,1)/d) then false
    else true
    fi fi fi fi ;;
```

The function *inside* is used in the guard of the transformation :

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

To test our program, we compute the convex hull of various points lying inside the square delimited by $(0,0)$ and $(1,1)$, including the four corners :

```
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 : ()
)) ;;
```

computes the expected result :

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

8.2 Eratosthene's Sieve on a Set

The idea is to generate a set with integers from 2 to N (with transformation *Generate* and *Succeed*) and to replace an x and an y such that x divides y by x (transformation *Eliminate*).

The results is the set of prime integers.

$$\begin{array}{rcl}
\texttt{trans}\ \mathit{Generate} & = & \{x, \mathit{true}\}\ \texttt{=>}\ x, \{x+1, \mathit{true}\}; \\
\texttt{trans}\ \mathit{Succedd} & = & \{x, \mathit{true}\}\ \texttt{=>}\ x; \\
\texttt{trans}\ \mathit{Eliminate} & = & (x, y\ /\ y\ \texttt{mod}\ x = 0)\ \texttt{=>}\ x;
\end{array}$$

With this program, the expression

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

computes the primes up to N (and we can turn this expression into a function by abstracting on N).

8.3 Eratosthene's Sieve on a Sequence

The idea is to refine the previous algorithm using a sequence. Each element i in the sequence corresponds to the previously computed ith prime P_i and is represented by a record $\{\mathit{prime} = P_i\}$. This element can receive a candidate number n, which is represented by a record $\{\mathit{prime} = P_i, \mathit{candidate} = n\}$. If the candidate satisfies the test, then the element transforms itself to a record $r = \{\mathit{prime} = P_i, \mathit{ok} = n\}$. If the right neighbor of r is of form $\{\mathit{prime} = P_{i+1}\}$, then the candidate n skips from r to the right neighbor. When there is no right neighbor to r, then n is prime and a new element is added at the end of the sequence. The first element of the sequence is distinguished and generates the candidates.

$$\begin{array}{rcl}
\texttt{trans}\ \mathit{Eratos} = \{ & & \\
\quad \mathit{Genere1} & = & n : \mathit{integer}\ /\ \tilde{}\texttt{right}\ n\ \texttt{=>}\ n, \{\mathit{prime} = n\}; \\
\quad \mathit{Genere2} & = & n : \mathit{integer}, \{\mathit{prime}\ \texttt{as}\ x, \tilde{}\mathit{candidate}, \tilde{}\mathit{ok}\} \\
 & & \texttt{=>}\ n+1, \{\mathit{prime} = x, \mathit{candidate} = n\}; \\
\quad \mathit{Test1} & = & \{\mathit{prime}\ \texttt{as}\ x, \mathit{candidate}\ \texttt{as}\ y, \tilde{}\mathit{ok}\}\ /\ y \quad \text{mod}\ x = 0\ \texttt{=>}\ \{\mathit{prime} = x\}; \\
\quad \mathit{Test2} & = & \{\mathit{prime}\ \texttt{as}\ x, \mathit{candidate}\ \texttt{as}\ y, \tilde{}\mathit{ok}\}\ /\ y \quad \text{mod}\ x <> 0 \\
 & & \texttt{=>}\ \{\mathit{prime} = x, \mathit{ok} = y\}; \\
\quad \mathit{Next} & = & \{\mathit{prime}\ \texttt{as}\ x1, \mathit{ok}\ \texttt{as}\ y\}, \{\mathit{prime}\ \texttt{as}\ x2, \tilde{}\mathit{ok}, \tilde{}\mathit{candidate}\} \\
 & & \texttt{=>}\ \{\mathit{prime} = x1\}, \{\mathit{prime} = x2, \mathit{candidate} = y\}; \\
\quad \mathit{NextCreate} & = & \{\mathit{prime}\ \texttt{as}\ x, \mathit{ok}\ \texttt{as}\ y\}\ \texttt{as}\ s\ /\ \tilde{}\texttt{right}\ s \\
 & & \texttt{=>}\ \{\mathit{prime} = x\}, \{\mathit{prime} = y\}; \\
\}
\end{array}$$

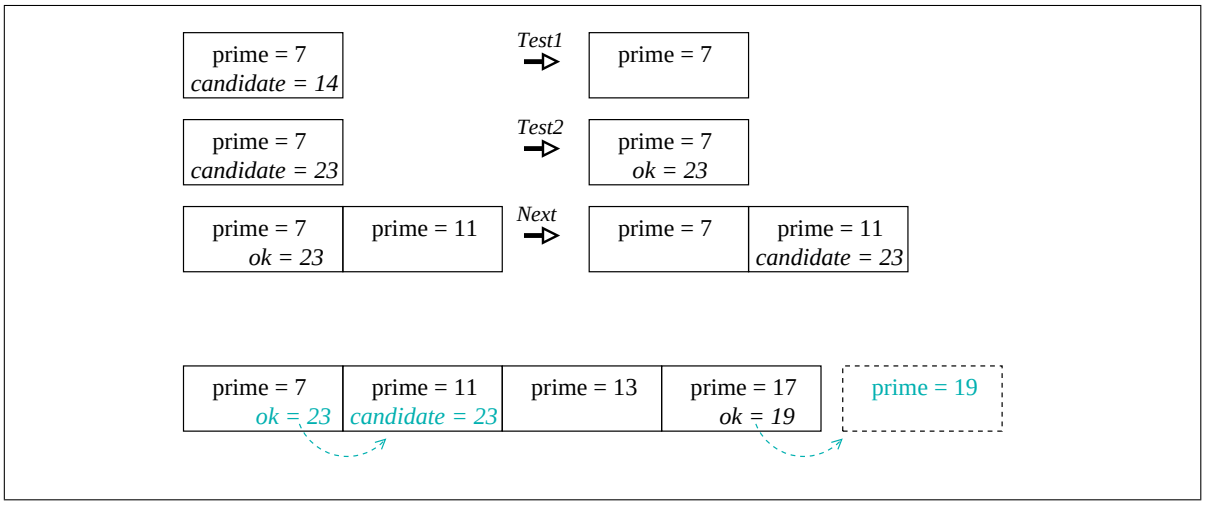


Fig. 8.1: The *Eratos* program. Some rule instantiations and a fragment of the sequence built by the transformation *Eratos*.

We have given an explicit name to each rule. See an illustration on Fig. 8.1. The expression $Eratos[N]((2, \texttt{seq} : ()))$ executes N steps of the Eratosthene's sieve. For instance $Eratos[100]((2, \texttt{seq} : ()))$ computes the sequence : $42, \{candidate = 42, prime = 2\}, \{ok = 41, prime = 3\}, \{prime = 5\}, \{prime = 7\}, \{prime = 11\}, \{prime = 13\}, \{ok = 37, prime = 17\}, \{prime = 19\}, \{prime = 23\}, \{prime = 29\}, \{prime = 31\}, \texttt{seq} : ()$.

8.4 Maximum Segment Sum

Consider the problem of finding the segment of maximal sum in a sequence of numbers. For instance, in sequence ¡1, 2, -3, 2, 2, -1¿ the maximum segment sum is the segment ¡2, 2¿. This optimization problem can be solved by dynamical programming. The corresponding algorithm is easely stated in `MGS`.

We first transform a sequence of numbers into a sequence of records. A record at position p has a field *val* which records the number at position p in the initial sequence, a field *sum* which holds the sum of the current computed maximal segment endings at position p and a field named *indices* which contains the positions of the elements of the current segment ending at p. Initially, the current segment that ends at position p also begins at position p. Thus :

$$\begin{aligned} \texttt{trans}\, init\, [p = 0] \quad &= \quad (x/\ \texttt{record}(x)) \\ &\Rightarrow\ (p := p + 1; \{val = x, sum = x, indices = (p, \texttt{set} : ())\}) \end{aligned}$$

For instance, $init(¡21, -5, 7¿)$ computes $¡\{val = 21, sum = 21, indices = \{1\}\}, \{val = -5, sum = -5, indices = \{2\}\}, \{val = 7, sum = 7, indices = \{3\}\}¿$.

Then, we can combine a segment ending at position p and a segment at position $p + 1$ to

gives a segment at position $p+1$ if this increase the local score :

$$\begin{aligned}&\texttt{trans}\, all\_max\_sum = \\ &\quad ((x,y)/(y.sum < (x.sum + y.val))) \\ &\quad => x, y + \{val = y.val, sum = x.sum + y.val, indices = x.indices@y.indices\}\ \texttt{;;}\end{aligned}$$

This transformation must be iterated until fixpoint. Then, the maximal segment sum can be extracted :

$$\begin{aligned}&\texttt{trans}\, max\_sum = \{ \\ &\quad x, y/x.sum > y.sum => x; \\ &\quad x, y/x.sum < y.sum => y; \\ &\}\ \texttt{;;}\end{aligned}$$

The whole process can be sumarized in a function :

$$\texttt{fun}\, mss(C) = max\_sum[\,\texttt{'fixrule}\,](all\_max\_sum[\,\texttt{'fixrule}\,](init(C)))\ \texttt{;;}$$

8.5 Tokenization

The tokenization problem can be stated as follows : it is required to process a sequence of letters to obtain the multiset of words constituting the sequence. A word is a sequence of letters without white space.

The solution, a two transformations long `MGS` program, relies on a nested collections structure. On the top level, we have a multiset and the elements of this multiset are sequences which finally must be without white space.

We first defines two new types :

$$\begin{aligned}&\texttt{collection}\, Word = \texttt{seq}\ \texttt{;;} \\ &\texttt{state}\, Split = \{before, after\}\ \texttt{;;}\end{aligned}$$

The type *Word* is just a distinguished sequence type used to representes the words[1]. The record *Split* will be used to record the two parts of a sequence splitted when a white space is detected. The rule :

$$\begin{aligned}&\texttt{trans}\, CutSeq\ = \quad (x/\ x\,!=\text{" "}) + \texttt{as}\, X,\ \ (y/\ y==\text{" "}),\ \ (z + \texttt{as}\, Z) \\ &\qquad\qquad\qquad\quad => \{before = X, after = Z\}\ \texttt{;;}\end{aligned}$$

applied on a sequence, gives a new sequence. If there is a white space " " in the sequence, the the pattern « $(x/\ x\,!=\text{" "}) + \texttt{as}\, X$ » filters, in a subsequence named X, all the non-white space letters until the first occurence of a white space binded to y. Then Z binds to the rest of the sequence. The result computed by $\{before = X, after = Z\}$ is a sequence containing

[1]Instead of letters, we use here strings (written between double quotes) to represent the elements of the words, because the current interpreter does not offer letters as a basic type.

only one element, a record of type *Split*. If there is no white space on the sequence, the rule does not apply and the transformation is the identity

Recall that a transformation acts by applying rules on subsequences and the results are gathered in a sequence. This is why the results of applying *CutSeq* is always a sequence, even if the entire sequence is matched by the rule[2].

The second transformation apply *CutSeq* on the elements of a multiset and extract the result of a split from the englobing sequence :

$$\begin{array}{ll} \texttt{trans}\ Cut = & \{ \\ & \quad x/Split(\texttt{hd}\ x)\ =>\ (\texttt{hd}\ x).before,\ \ (\texttt{hd}x).after,\ \ \texttt{bag} : (); \\ & \quad x\ =>\ CutSeq(x); \\ & \} \end{array}$$

The first rule of this transformation is applied if the first element of a sequence is a *Split*. In this case, the two fields of the *Split* are extracted and constitute the elements that are added into the multiset in place of the matched sequence. The second rule apply the transformation *CutSeq* to an element. It is important to give the two rules in this order. As a matter of fact, the second rule can always apply (because there is no guard, the pattern x matches any element in the multiset). But we want to apply this rule only if the element is not a split.

For example, see Fig. 8.2, the expression (the transformation *Cut* applied until fixpoint to a multiset of one element, this element being a sequence *Word*) :

$$Cut[\,\texttt{'fixpoint}\,](((\text{"}a\text{"},\text{"}b\text{"},\text{"}c\text{"},\text{" "},\text{"}d\text{"},\text{"}e\text{"},\text{" "},\text{"}f\text{"},\text{"}g\text{"},\text{"}h\text{"}, Word : ()), \texttt{bag} : ()))\ \texttt{;;}$$

evaluates to

$$\big(\text{"}a\text{"},\text{"}b\text{"},\text{"}c\text{"},():Word\big),\ \big(\text{"}d\text{"},\text{"}e\text{"},():Word\big),\ \big(\text{"}f\text{"},\text{"}g\text{"},\text{"}h\text{"},():Word\big),\ \ ():\texttt{bag}$$

that is, a bag of three elements, each element being a word without white space. See figure 8.2.

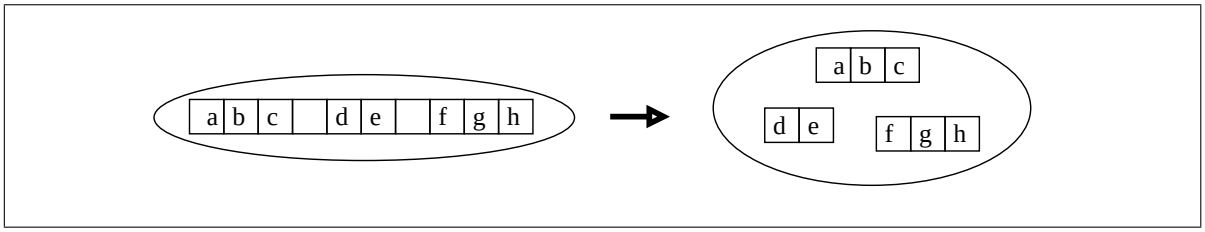


Fig. 8.2: Tokenisation of a sequence of letters

8.6 Token moving on a Ring

The problem is just to propagate a token on a ring. The idea is to use a rule

$$(x/x == 1),\ (y/y == 0) => 0, 1;$$

[2]We are devising mechanisms to ease the "dissolving" of a nested collection, in a manner analog of the dissolve operator used in P systems. Here we use a rule in the transformation *Cut*.

to say that the token "1" propagates in a medium of 0. However, the topology of a ring is not directly accessible as a collection kind (not yet). But it can be emulated by a sequence and by managing explicitly what occurs for the begin and the end of the sequence. Instead of written one rule, we have to write three rules. The rule for the first element looks like :

$$z/(z == 0) \quad \& \quad \tilde{}\texttt{left}\ z \quad \& \quad \ldots \quad => \quad 1;$$

where the condition $\tilde{}\texttt{left}\ z$ specifies that z is the first element in the sequence (it has no element to its left) and the condition $z == 0$ ensure that it is not occupied by a token. It remains to check that the last element of the sequence is occupied by a one.

For, we have to refer to the global collection on which the transformation is acting. This is possible, using simply an additional parameter of the transformation. When we apply the transformation, we arrange to pass the collection both as the argument and as the value of the additional parameter of the transformation (using a wrapper). The corresponding program is :

$$\begin{array}{l}
\texttt{trans}\ \mathit{Tore}[\mathit{self}] = \{ \\
\quad (x/x == 1),\ (y/y == 0) \quad => \quad 0, 1; \\
\quad y/(y == 1)\ \&\ \tilde{}\texttt{right}\ y\ \&\ (0 == \texttt{hd}(\mathit{self})) \quad => \quad 0; \\
\quad z/(z == 0)\ \&\ \tilde{}\texttt{left}\ z\ \&\ (1 == \texttt{last}(\mathit{self})) \quad => \quad 1; \\
\}\ ;; \\
\texttt{fun}\ \mathit{tore}(t) = \mathit{Tore}[\mathit{self} = t](t)\ ;;
\end{array}$$

The operators `hd` and `last` give the first and the last element in a sequence. The function *tore* is the wrapper of the transformation *Tore*. An n-times iteration of the transformation is then simply obtained by iterating the function *tore* n-times, which is realized with the same syntax as the iteration of a transformation : $\mathit{tore}[n](\ldots)$. The 6th first iterations starting from a ring with 5 element and just one token, give :

$$\begin{array}{l}
0, 0, 1, 0, 0 \\
0, 0, 0, 1, 0 \\
0, 0, 0, 0, 1 \\
1, 0, 0, 0, 0 \\
0, 1, 0, 0, 0 \\
0, 0, 1, 0, 0
\end{array}$$

This program gives an example of the smooth interplay between transformations and functions, and the use of additional arguments in a transformation.

Moving a token on a ring is not very interesting. Instead of moving one token, one can diffuse two morphogenes that, in addition, react together. This process is sketched in the next section. The previous idea is used to diffuse on a ring emulated on a sequence. The results of the `MGS` program are output in a `Mathematica` readable form, for the purpose of visualization. The result is plotted in figure 8.3. We do not give the corresponding `MGS` code because it simply combines the previous idea with the Turing diffusion-reaction process described below. For information, the `MGS` code takes 75 lines, including 35 lines dedicated to

format the output for `Mathematica` while an hand-coded `C` program takes 70 lines to only compute the diffusion-reaction process.

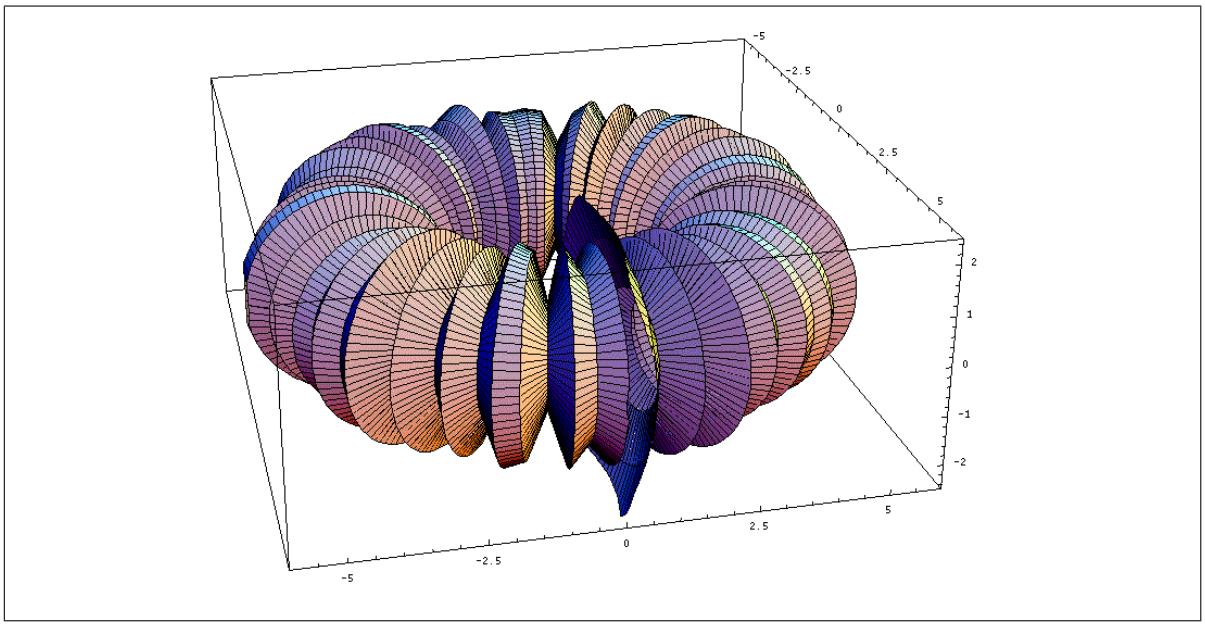


Fig. 8.3: Example of a Turing diffusion-reaction process on a ring. Each cell of the ring is rendered by a slice of the torus. The diameter of the slice is proportional to the b morphogene (cf. text of section 8.7). The results computed by the `MGS` program are written in a file later read by `Mathematica`. This file contains both the computed data and a `Mathematica` program used to compute the coordinate of the torus and to render the 3D objects. This figure plots a gif capture of the graphics rendered by `Mathematica` when reading the `MGS` produced file.

8.7 Morphogenesis Triggered by a Turing Diffusion-Reaction Process

Alan Turing proposed a model of chemical reaction coupled with a diffusion processus in cells to explain patterns formation. The system of differential equations [BL74] is :

$$\begin{aligned} da_r/dt &= 1/16(16 - a_r b_r) + (a_{r+1} - 2a_r + a_{r-1}) \\ db_r/dt &= 1/16(16 - b_r - \beta) + (b_{r+1} - 2b_r + b_{r-1}) \end{aligned}$$

where a and b are two chemical reactives that diffuse on a discrete segment of cells indexed by r. This model mixes a continuous phenomena (the chemical reaction in time) and a discrete diffusion process. In `MGS` we retrive these equations, three times, to handle the cell at the two ends of the segment (rule $evol\_left$ and $evol\_right$) and the cells with two neighboors (rule $evol$).

In addition, we complexify this processus by splitting one cell in two if the level of the morphogen b is greater than a given level (rule $Split$). This process does not correspond to any real biosystems, see however [HP96].

The corrresponding program starts by a transformation used to generate the initial se-

quence of cells.

```
trans init =
  x => {
    a = 3.5 + random(1.0) − 0.5,
    b = 4.0,
    beta = 12.0 + random(0.05 ∗ 2.0) − 0.05,
    size = 16
};

rsp := 1.0/16.0 ; ;
diff1 := 0.25 ; ;
diff2 := 0.0625 ; ;
NbCell := 18 ; ;

segment0 := init[1](iota(NbCell, () : seq)) ; ;
```

The *init* transformation is used to generate the initial sequence of cells *segment*0. Applied one times to a sequence of n arbitrary elements, it generates a sequence of records. The field a and b of the record corresponds to the morphogens. The field *beta* is an auxilliary variable of the diffusion-reaction process : it corresponds to a constant with some noise. The field *size* is used for the 3D output, see figure 8.4 and annex **??**. The expression $iota(NbCell, () : \texttt{seq})$ build a sequence made of the integers from 0 to $NbCell$.

The real computation takes place in the *Turing* transformation. One rule is used to split a cell that reach the adequate level of morphogen b and three other rules are used for the reaction-diffusion process. The functions *da* and *db* computes the increases in morphogen a

and b respectively.

```
fun da(a, b, la, ra) = rsp * (16.0 - a * b) + diff1 * (la + ra - 2.0 * a) ;;
fun db(a, b, beta, lb, rb) = rsp * (a * b - b - beta) + diff2 * (lb + rb - 2.0 * b) ;;

trans Turing = {
  Split =
    (x/x.b > 8) =>
      {a = x.a/2, b = x.b/2, beta = x.beta, size = x.size/2},
      {a = x.a/2, b = x.b/2, beta = x.beta, size = x.size - x.size/2};
  evol =
    (x/(left x)&(right x))+=>
      {a = x.a + da(x.a, x.b, (left x).a, (right x).a),
       b = Max(0.0, x.b + db(x.a, x.b, x.beta, (left x).b, (right x).b))};
  evol_right =
    (x/~left x)+=>
      {a = x.a + da(x.a, x.b, 0, (right x).a),
       b = Max(0.0, x.b + db(x.a, x.b, x.beta, 0, (right x).b))};
  evol_left =
    (x/~right x)+=>
      {a = x.a + da(x.a, x.b, (left x).a, 0),
       b = Max(0.0, x.b + db(x.a, x.b, x.beta, (left x).b, 0))};
} ;;
```

The rest of the code is used to trigger the computation and to output the results. The output is done in a dedicated language used to visualize 3D scenes. The result is plotted in figure 8.4. The functions *showBarre*, *pre_show* and *post_show* are detailed in annex **??**. This code is very short and easy to program, because the language used to produce the scene is very expressive.

```
fun showBarre(barre, t, tmax) = ... ;;
fun pre_show() = ... ;;
fun post_show(n, c) = ... ;;
fun evol(barre, t, tmax) =
(
  showBarre(barre, t, tmax);
  if (t < tmax)  then evol(Turing[iter = 1](barre), t + 1, tmax)
  else barre fi
);
fun evolve(n) = (pre_show(); evol(segment0, 0, n); post_show(n, NbCell)) ;;
```

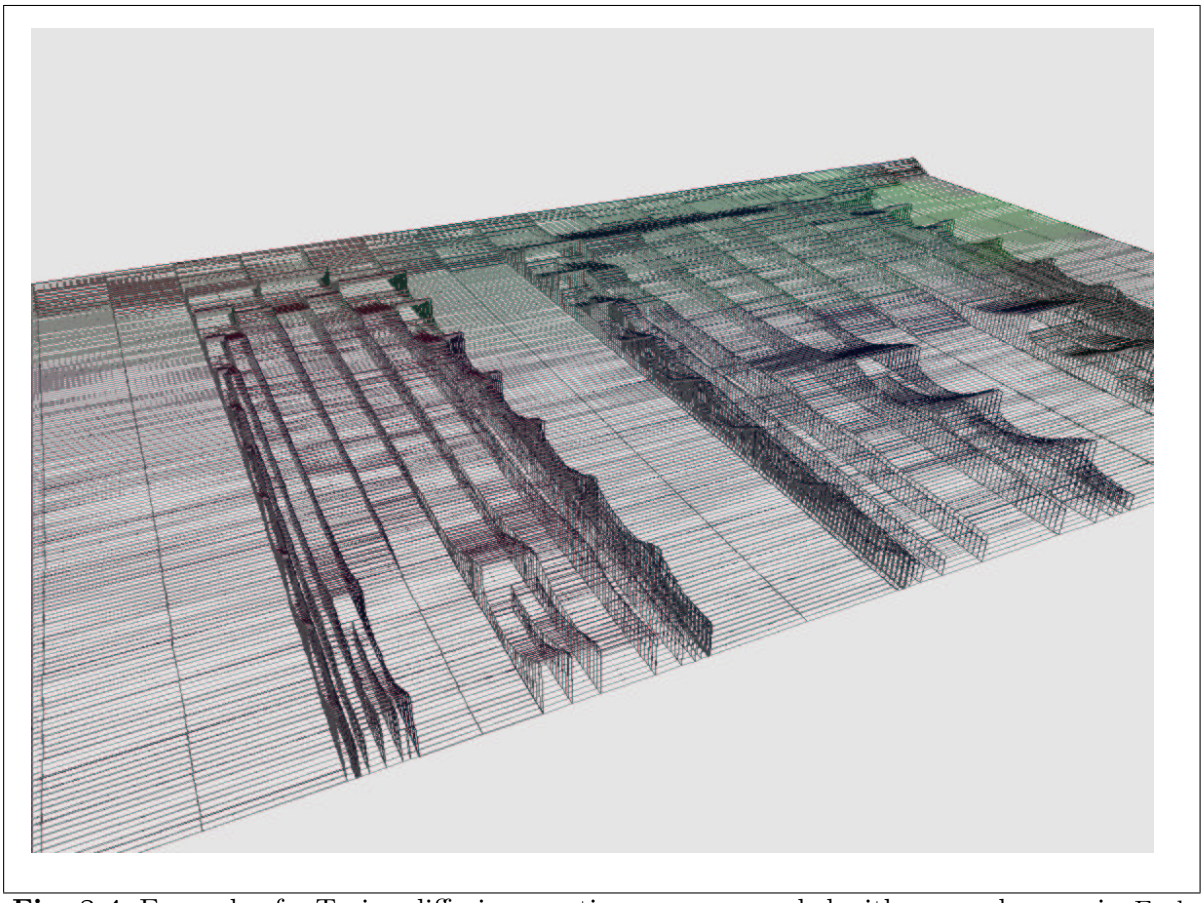


Fig. 8.4: Example of a Turing diffusion-reaction process coupled with a morphogenesis. Each cell is rendered by a block whose height is proportional to the b morphogene (cf. text). When a cell is splitted in two, the width of the two daughter cells is divided by two, such that cells with a common ancestor are in the same parallel line (the axis directed toward the reader, which represents the passing of time). This plot corresponds to 180 time step evolution of an initial sequence of 18 cells.

Chapitre 9

Les GBF

Ce chapitre est consacré à un nouveau type de collection permettant la représentation d'un espace *homogène*. De telles structures de données se rencontrent quand 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 peut alors se coder par la *présentation* d'un groupe. Cela constitue le fondement de la notion de *GBF* abbréviation de l'anglais « Group Based Field » ou *champ de données basé sur une structure de groupe* en français.

9.1 Introduction : tableau, champ 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 à 4 voisins (les voisins *nord*, *est*, *sud* et *ouest*), peut se représenter par un tableau ; les treillis rectangulaires à 8 voisins aussi, cf. figure 9.1. Chaque élément du tableau correspond à un point de l'espace. L'indice qui indexe un point du tableau peut être vue 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 dépace du point courant à son voisin sud (l'association d'une direction à une dimension est une convention arbitraire).

Avec un peu de précautions et de 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 à 6 voisins, est plus compliquée, et plus généralement, l'utilisation de tableaux pour représenter des espaces homogènes arbitraires présente de nombreux défauts [Mic96b] : 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. Un champ de données généralise la notion de tableau puisqu'on peut voir un tableau de dimension d comme une *fonction totale*[1] 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 généralise la notion de tableau en considérant des *fonctions partielles* sur $\mathbb{Z}^d$. L'intérêt des champs de données par rapport aux tableaux est de permettre la représentation de forme arbitraire : 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, cf. figure 9.2.

Représentation de topologies arbitraires. Si la notion de champ de données permet la représentation de région 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-dimensionelle 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 sur, on peut coder 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, cf. figure 9.3. Mais comme il a déjà été mentionné, le codage d'une topologie triangulaire (cf. figure 9.1) n'est de loin pas aussi simple.

Notion de GBF. Afin de pouvoir représenter des topologies arbitraires, en sus de géométries arbitraires, il faut étendre la notion de champ de données. Dans ce chapitre nous allons nous restreindre à des espaces *homogènes*. Un espace est dit homogène quand chaque point à 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 par `MGS` pour définir des collections topologiques permettant le représentation d'espaces homogènes et la généralisation de la structure de données de tableau. Ces collections sont nommées des GBF. Le suite du chapitre présente de manière plus concrète comment définir et calculer avec des GBF.

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 composées 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

[1] 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 $f : A \mapsto B$.

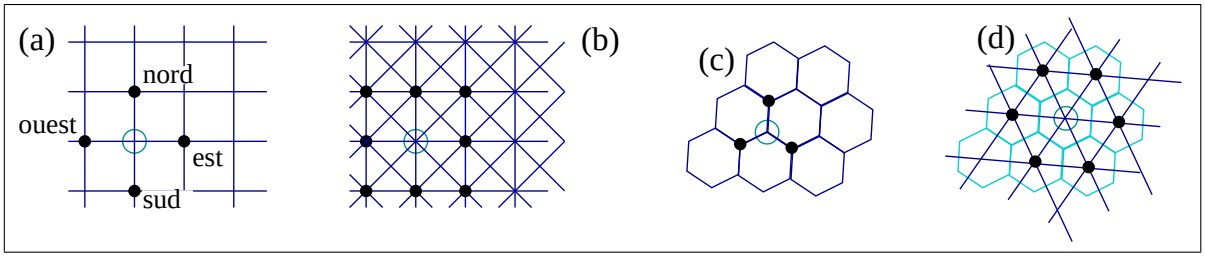


Fig. 9.1: Exemples de treillis discrétisant de manière homogène un espace plan. **(a)** : treillis rectangulaire à 4 voisins appelé *voisinage de Von Neuman*. **(b)** : treillis rectangulaire à 8 voisins appelé *Voisinage de Moore*. **(c)** : treillis triangulaire : chaque point possède trois voisins. **(d)** : treillis hexagonal : chaque point possède 6 voisins. Si on représente chaque point de l'espace par une cellule hexagonale, et que la connection entre deux points est représentée par le partage d'une face de la cellule, on obtient un pavage hexagonale (représenté en grisé).

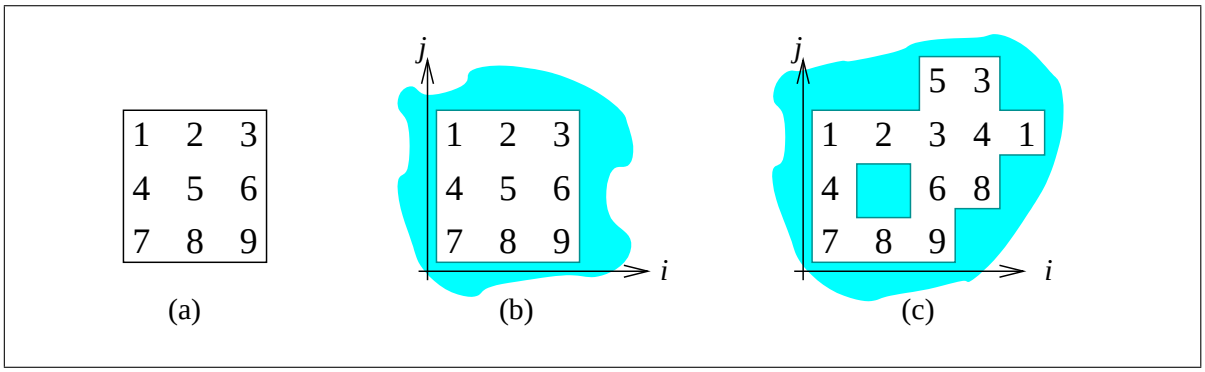


Fig. 9.2: Notion de tableau et de champ de données. **(a)** : un tableau. **(b)** : le même tableau vu comme un champ de donnée : c'est une fonction qui associe à chaque indice (i, j) une valeur. Cette fonction est partielle : la plupart des indices (il y en a une infinité) ne sont associés à aucune valeur. Le domaine de définition du champ de données est symboliquement représenté par une région blanche sur le fond coloré. **(c)** : un champ de données permet de représenter simplement des régions arbitraires qui généralisent les régions rectangulaires des tableaux.

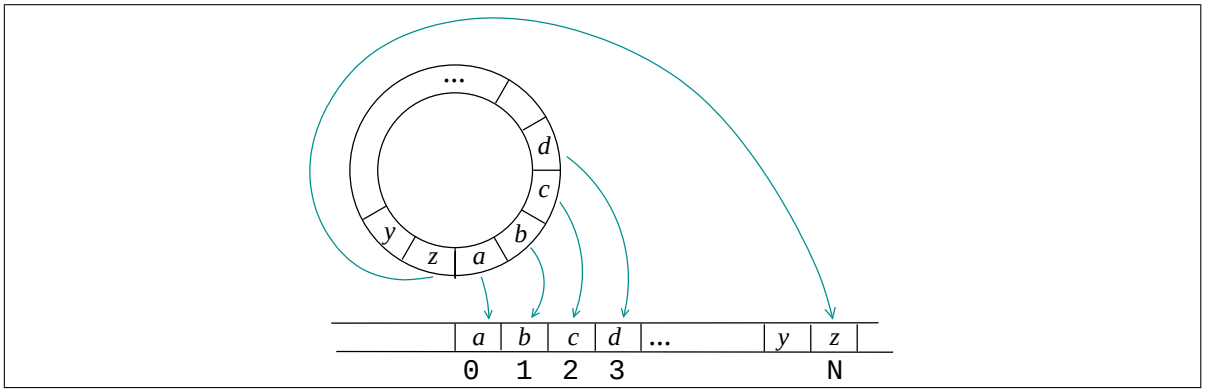


Fig. 9.3: Codage d'une topologie d'anneau sur un champ de données de dimension 1. Chaque point d'indice i pour $0 \leq i \leq N$ dans le champ de données correspond à un élément de l'anneau. L'opération qui permet de calculer l'indice j du voisin de droite d'un élément d'indice i est alors défini par : $j = i + 1 \mod N$.

d'un groupe de déplacements[2].

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 élémentaires qui définissent le voisinage homogène.

Le cadre que nous venons de décrire constitue le fondement de la notion de *GBF* abbréviation 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	$(i, \ldots, k) \to (i \pm 1, \ldots, k)$, ...

champ de données

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

GBF

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

9.2 Un premier 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 egal, entre les délimiteurs

[2]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.

`<` 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* reliants une direction et sa direction inverse :

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

Les equations 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'equation

`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. Au passage, notons qu'une équation `w = 0` dans la présentation d'un GBF, peut s'écrire plus simplement juste `w`. Toute équation `v = w` peut donc se réécrire `v - w`. De plus, toujours dans la présentation d'un GBF, on peut omettre le signe de multiplication entre un entier et un générateur. Avec ces conventions, la définition de `A5` devient :

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

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 préssentation 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 certains indices. L'ensemble des positions qui possèdent une valeur s'appelle le *domaine de définition du GBF*.

9.3 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 9.4 illustre la correspondance entre graphes et groupes.

Mots et chemins. Un mot w du groupe correspond à une somme de générateurs, par exemple $a + b + b - a$. Un mot correspond aussi à un *chemin* dans le graphe. La composition des chemins correspond à l'addition des mots (dans tout ce chapitre, la loi de groupe est notée additivement).

Tout mot w correspond aussi à un élément du groupe et donc un sommet dans le graphe. Ce sommet est le sommet que l'on atteint en partant de l'origine (le sommet identifié par l'élément neutre du groupe) et en suivant le chemin w.

Equations et cycles. Un chemin fermé, un *cycle*, est un mot égal à l'élément neutre (notée ici 0). Toute équation $v = w$ peut se réécrire $v - w = 0$ et correspond donc à un cycle dans le graphe.

Equations caractéristiques Certaines équations $w = 0$ sont vrais pour tout les groupes : ce sont les équations où w est de la forme $w' - w'$ pour un certain w'. Ces équations s'interprètent en terme de « rebrousser le chemin w' ». D'autres équations ne sont valides que pour un groupe spécifique et définissent la topologie « globale » du graphe.

Solution d'une équation et connexité. La connexité du graphe revient à l'existence d'une solution v à l'équation $P + v = Q$: P et Q dénotent des sommets du graphe, donc des éléments du groupe, et v est un chemin (donc aussi un élément du groupe).

9.4 GBF abéliens

Le graphe de Cayley que nous avons représenté à la figure 9.4 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`.

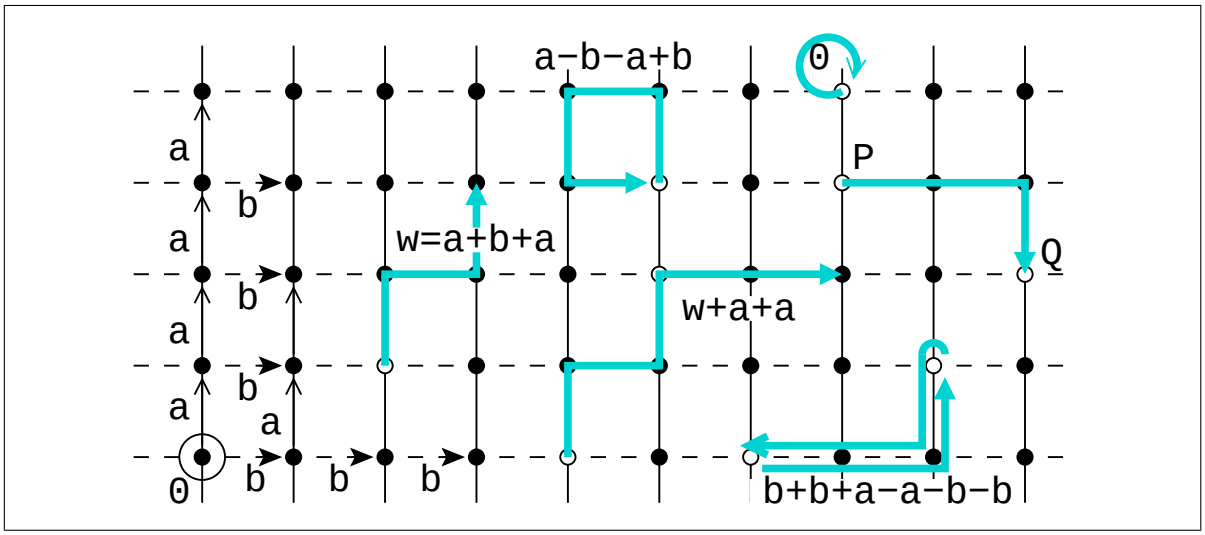


Fig. 9.4: Les graphes de Cayley permettent de faire un pont entre la théorie des graphes et les concepts de la théorie des groupes. Etant donné une présentation d'un groupe, on peut lui associer une représentation par un graphe appelé graphe de Cayley. Dans ce graphe, un sommet correspond à élément du groupe et un arc étiqueté à un générateur (les arcs sont orientés). 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 visualise 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.

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 9.2, au renomage des générateurs près : `nord` devient[3] `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 »). Par contre, 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

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

dans le fils droit et ensuite dans le fils gauche. Autrement dit, un arbre binaire correspond à un *GBF non-abélien*. Nous n'en dirons pas plus dans ce document sur les GBF non-abéliens.

9.5 Calculer avec des chemins

Reprenons notre exemple favori

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

cette définition introduit un nouveau type `G` de collection. Mais incidement, elle introduit aussi un nouveau type atomique correspondant aux *mots de la présentation du groupe*. Ce type n'est pas nommé mais, une fois le GBF défini, on peut former des expressions correspondant à des mots. Les mots de la présentation `G` sont appelés les *chemins* de `G` et correspondent à des sommes formelles de générateurs.

Les chemins associés à un groupe 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.

Un chemin permet de dénoter une *position* dans le GBF : c'est la position atteinte en partant de l'origine et en suivant ce chemin.

Les constantes. Les chemins 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>
```

notez bien qu'il n'y a pas d'espace entre les caractères `|`, `>` et le nom du générateur. 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 chemins. En général, un mot est une somme de générateurs. Les opérations d'addition et de soustractions sont surchargées pour permettre de calculer simplement des chemins :

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

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 chemin 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 chemins. Par exemple :

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

est une expression dont le résultat est :

```
2*|nord>
```

En effet, `|nord> = - <nord|` et `|est> + <est|` est le chemin vide, correspondant à l'élément neutre du groupe et indiquant qu'aucun déplacement n'a été effectué. Le chemin vide peut être le résultat d'un calcul, auquel cas il s'affiche ainsi : `|0>`. Cependant on ne peut pas écrire directement ce chemin dans une expression[4]. Si on a besoin du chemin vide, il faut l'obtenir comme résultat d'une calcul, comme par exemple :

```
zeroG := <est| + |est>;;
```

On peut ensuite utiliser cette valeur, par exemple l'expression

`zeroG -` *chemin*

l'inverse de *chemin*, pour n'importe quel chemin dans `G`.

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

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

Par exemple :

```
3*(2*|nord> - <est|*5)
```

est une expression qui a pour valeur `6*|nord> + 15*|est>`. La multiplication d'un chemin c par un entier négatif n correspond à multiplier l'inverse de c par la valeur absolue de n :

```
-8 * <nord|
```

retourne la valeur `8*|nord>`. On peut donc obtenir l'inverse d'un chemin quelconque en multipliant ce chemin par `-1`.

[4]La raison est qu'on a besoin d'un nom de générateur pour retrouver le GBF dans lequel ce chemin a un sens. La constante `|0>` dénote de manière ambigüe un chemin qui existe dans tous les GBF.

Opérations d'égalité sur les chemins. Outre les combinaisons linéaires, on peut aussi tester l'égalité de deux chemins. Par exemple :

```
|nord> + |nord> == 2*|nord>
```

retourne la valeur vraie. Reprenons le type `A5`, et testons l'égalité de `|a>` avec `6*|a>`. Ces deux chemins représentent le même élément dans le groupe associé à `A5`, puisque `5*|a> = |0>`. Cependant :

```
6*|a> == |a>
```

retourne la valeur fausse. En effet, *un chemin n'est pas une position* (i.e. un élément du groupe, un sommet dans le graphe de Cayley) mais correspond à un trajet. Plusieurs chemins, en partant de l'origine (le sommet correspondant à l'élément neutre dans le graphe) peuvent aboutir au même sommet. C'est le cas dans l'exemple précédent pour `|a>` et `6*|a>`. Ces deux chemins désignent donc le même élément dans le groupe, mais en tant que chemin, ils sont distinguées.

L'égalité `p == q` de deux chemins `p =` $a_1.g_1 + \cdots + a_d.g_d$ et `q =` $a'_1.g_1 + \cdots + a'_d.g_d$ se contente de tester l'égalité des coefficients $a_1 = a'_1$ et $a_2 = a'_2$ et ... et $a_d = a'_d$. Autrement dit, on teste que chaque chemin représente le même déplacement dans chaque direction. En particulier, on ne vérifie pas que l'on effectue chaque mouvement dans le même ordre ; par exemple

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

est une expression qui renvoie la valeur vraie bien que l'ordre des deux mouvements élémentaires soit différent, car on s'est déplacé autant vers le nord et vers l'est.

Si on veut tester que les deux chemins réfèrent au même élément du groupe, il faut utiliser la primitive `same` :

```
same(6*|a>, |a>)
```

retourne la valeur vraie. La fonction `same(c,d)` teste que les chemins `c` et `d` réfèrent à la même position (voir le paragraphe 9.7 pour une présentation du principe de l'algorithme de la fonction `same`).

Morphisme de chemin. Le GBF `G` peut se représenter simplement par un treillis de points dans le plan euclidien $\mathbb{R}^2$. Supposons que l'on veuille traduire les chemins de `G` en coordonnées cartésiennes correspondant au sommet associé au chemin. Un chemin de la forme x`*|nord> +` y`*|est>` se traduirait par exemple en un enregistrement {`x =` x`, y =` y}. Cet enregistrement peut se calculer grâce à la fonction `pmorphism` qui prend trois arguments :

```
pmorphism(fa, fg, c)
```

les deux premiers arguments sont des fonctions et le dernier est un chemin :

– La fonction `fg` prend deux arguments dont le premier est un entier et le second un générateur. Elle retourne une valeur d'un type V.
– La fonction `fa` combine deux valeurs de type V et retourne une valeur de type V.

Le calcul réalisé est le suivant :

$$\texttt{pmorphism}(\oplus, \otimes,\ a_1.g_1 + \cdots + a_d.g_d) \quad = \quad \bigoplus_{i=1}^{i=d} a_i \otimes g_i$$

(l'opérateur $\otimes$ correspond à `fg` et l'opération $\oplus$ à la fonction `fa` ; ces notations infixes sont utilisées pour faire ressortir la structure de produit scalaire du calcul réalisé). L'application partielle `pmorphism(fa, fg)` est une fonction φ qui envoie un chemin sur un élément de V. Dans le cas où V est un groupe, et si les fonctions `fa` et `fg` possèdent quelques propriétés supplémentaires, alors φ est un morphisme de groupe. C'est le cas dans l'exemple précédent si on interprète les enregistrements avec les champs `x` et `y` comme des éléments du groupe $(\mathbb{R}^2, +)$. Ceci explique le nom choisi pour la fonction `pmorphism` (le "p" initial est pour l'anglais "path").

L'exemple précédent peut donc se programmer avec les fonctions suivantes :

```
fun fg(n, x) = if x == |nord> then {x=0.0, y=n*1.0} else {x=n*1.0, y=0.0} fi;
fun fa(v1, v2) = {x = v1.x + v2.x,   y = v1.y + v2.y};;
```

(multiplier un entier `n` par le flottant `1.0` est une façon de le convertir en flottant). Ainsi, l'expression :

```
pmorphism(fa, fg, x*|nord> + y*|est>)
```

va calculer

```
fa(fg(x, |nord>), fg(y, |est>))
= fa({x=0.0, y=y}, {x=x, y=0.0})
= {x=x, y=y}
```

9.6 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 operateurs qui permettent de manipuler des GBF. Ces opérateurs sont polytypiques dans le sens où ils s'appliquent à tous les GBF abéliens.

9.6.1 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 = < ... >;;
```

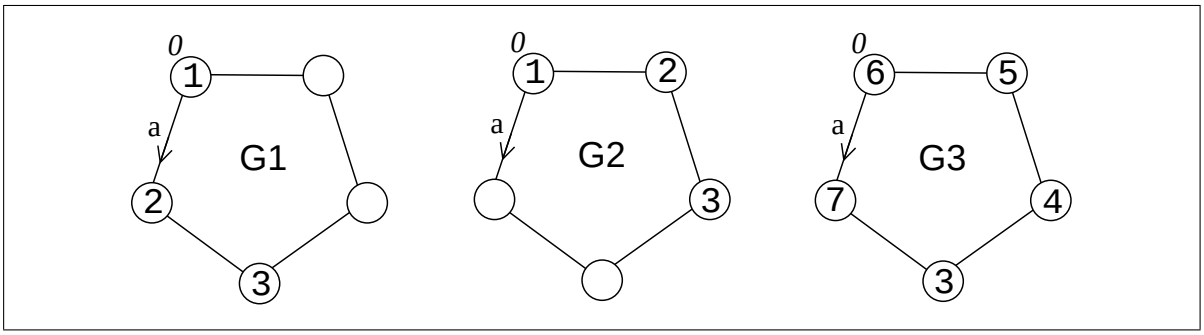


Fig. 9.5: Trois instances de GBF de type `A5` construit avec l'opérateur `following`.

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 et le domaine de définition d'un GBF vide 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 vraie.

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` :

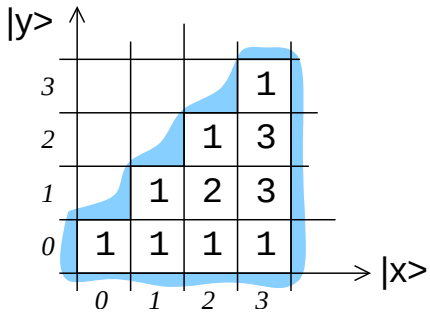
```
gbf A5 = < a; 5a >;;

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 à la figure 9.5. On voit que les éléments $s_1, s_2, ...$ 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 chemin $(i-1)$`*|a>` 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.

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 :

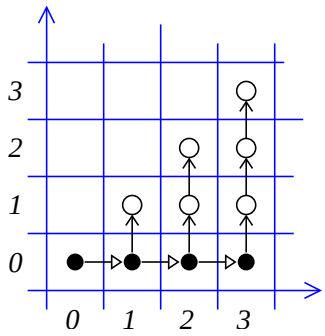
[5] étiqueté par l'élément neutre `0` du groupe et repéré par le chemin vide `|0>`

```
gbf grid2 = < x, y >;;
Pascal := ( (1, ():seq)  ::
            (1, 1)       ::
            (1, 2, 1)    ::
            (1, 3, 3, 1) ::
            ():seq )
            following |x> |y>;;
```



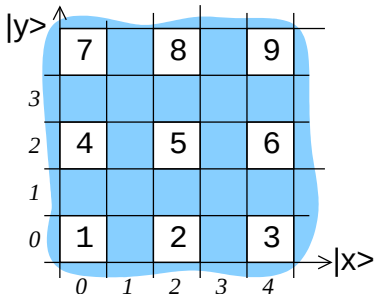
Dans cet exemple, le premier argument est une séquence de séquences s_i. La direction `|x>` est utilisée pour se déplacer entre les s_i et la direction `|y>` est utilisée pour se déplacer entre les éléments $s_{i,j}$ de s_i. Les positions visitées successivement sont donc les suivantes :

```
0,
0+1*|x>,      0+1*|x>+1*|y>,
0+2*|x>,      0+2*|x>+1*|y>,    0+2*|x>+2*|y>,
0+3*|x>,      0+3*|x>+1*|y>,    0+3*|x>+2*|y>,    0+3*|x>+3*|y>
```



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. Par contre, la fonction `build_gbf` admet des chemins 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>));;
```



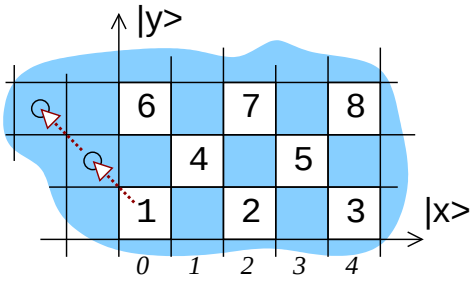
.

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]. Cela permet de construire des GBF « avec des trous ». Par exemple, un GBF « en échiquier » peut se construire par

[6]Cela contraste avec un collection monoïdale dans laquelle un élément peut avoir comme valeur `<undef>`. Dans un GBF, cela correspond a une position qui n'a pas de valeur. Ces deux points de vue sont différents, voir par exemple le paragraphe **??**.

l'expression :

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



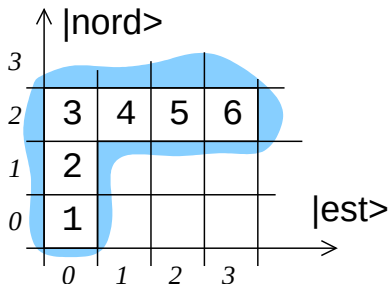
En effet, chaque fois que l'on se déplace suivant la direction `|y> - |x>`, on se déplace suivant la direction diagonale indiquée sur le schéma. La valeur de la position `|y> - |x>` est la première valeur de la seconde sous-séquence, et c'est donc `<undef>`. La position correspondante (indiqué par un 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 filtre. 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 7.1). Cette double notation ne peut pas être maintenu 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

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` en `|0>` puis se déplacer vers le `|nord>`, mettre la valeur `2` et se déplacer vers le `|nord>`, mettre la valeur `3` puis se déplacer dans la direction `|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`)`

Et comme la virgule, cette construction admet des arguments mixtes : deux éléments sont associés pour faire un GBF et un élément et un GBF sont aussi acceptés pour faire un GBF. On remarquera cependant que si g et g' sont deux GBF, l'expression

g `|nord>` g'

construit un GBF qui compte un seul élément supplémentaire par rapport à g' (au plus) : l'élément de position `|0>` qui a pour valeur le GBF g. Ceci constraste avec les collections monoïdales dans lesquelles la virgule fusionne les deux collections arguments.

Exemple : une marche au hasard. Dans cet exemple, on veut visualiser le résultat d'une marche au hasard dans le `gbf G = < x, y >`. La fonction `marche` prend une liste de valeurs en argument. Elle dépose la première valeur de cette liste sur le sommet 0. Puis on se déplace sur un voisin au hasard et on recommence.

```
fun marche(l) = if empty(l) then G:()
                else let r = random(4)
                     in if r == 0 then hd(l) |x> g(tl(l))
                        else if r == 1 then hd(l) |y> g(tl(l))
                             else if r == 2 then hd(l) <x| g(tl(l))
                                  else hd(l) <y| g(tl(l))   fi fi fi fi;;
```

La figure 9.6 illustre le GBF construit par l'expression `marche(iota(100, ():seq))`.

9.6.2 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 de g en la position `p`. 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 :

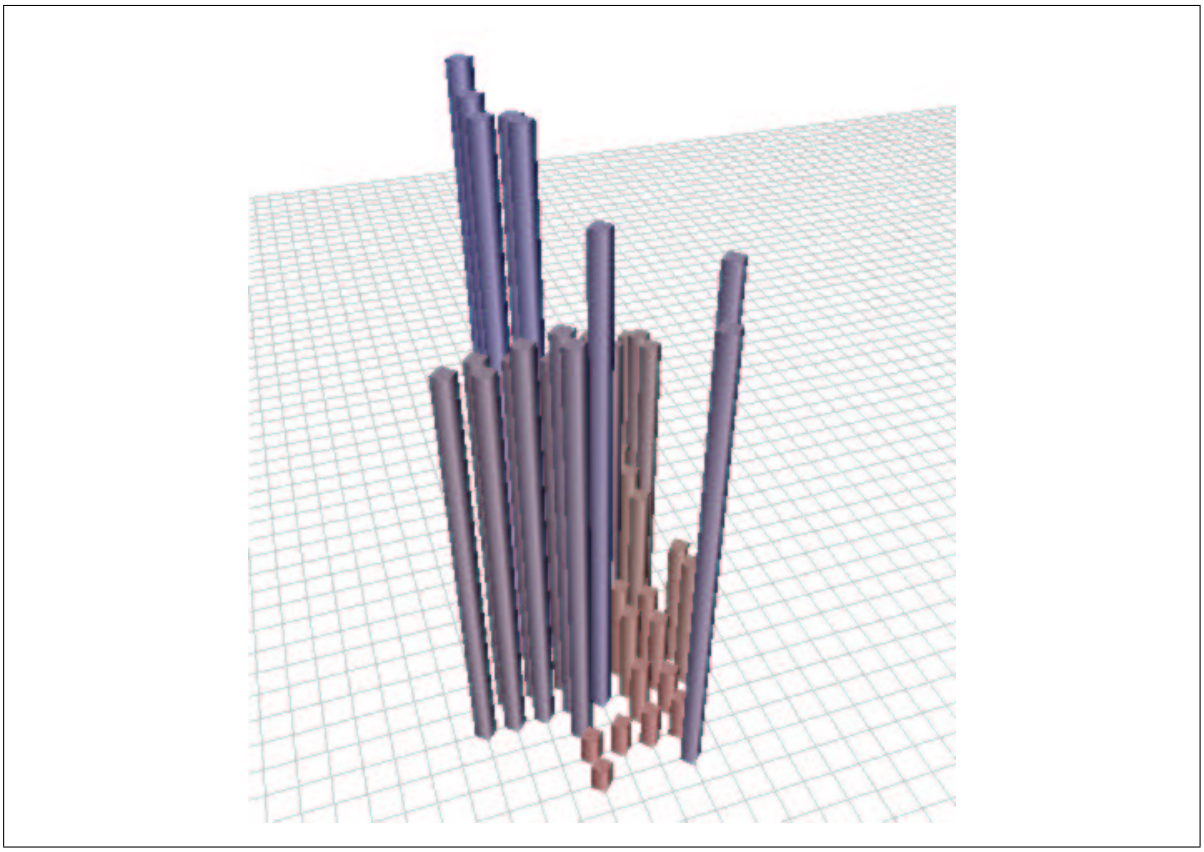


Fig. 9.6: Exemple d'un GBF construit par une marche au hasard. La fonction `marche` (voir texte) permet de construire un GBF en donnant une valeur le long d'un chemin au hasard : en chaque point du chemin, on tire au sort la direction à suivre parmi les générateurs du GBF et leurs inverses. La hauteur de la colonne en position `p` est proportionnelle à la longueur du chemin suivi pour arriver à `p`. Le chemin se recoupant de nombreuse fois, certaines colonne ont disparue, remplacé par des colonnes plus tardives.

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

permet de faire la somme de tous les éléments du GBF g. 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.

9.6.3 Les opérateurs `map_indexed` et la représentation d'une position

Il est parfois utile de connaître la position d'une valeur lors de son traitement dans un `map`. C'est possible grâce aux opérateurs `map_indexed_p` et `map_indexed_g`. Contrairement à `map`, la fonction argument de `map_indexed_`x prend deux arguments qui sont la valeur d'une position et cette position elle-même. Par exemple :

```
map_indexed_x(\v,p.{val=v, pos=p}, g)
```

va construire un GBF semblable à g mais où toutes les valeurs vont être remplacées par un enregistrement comportant un champ `val` et un champ `pos`. À une position `p` donné, le champ `val` contient la valeur de g à cette position, et le champ `pos` a pour valeur `p`.

Le problème qui se pose est de décider *comment représenter une position* ? Une solution possible est d'utiliser un chemin, puisqu'un chemin dénote aussi une position. La fonction `map_indexed_p` réalise cela, en passant pour argument `p`, un chemin dénotant la position courante. Hélas, comme on l'a vu, plusieurs chemins peuvent dénoter la même position (Cf. le paragraphe 9.5 page 77 concernant l'égalité de chemin). Aussi, la position `p` passée en argument est un des chemins possibles, sans autre précision.

L'autre version `map_indexed_g` représente une position par une séquence de nombres entiers. Ces séquences, dont la longueur dépend uniquement du type du GBF, représentent un « *chemin normalisé* » et on est donc assuré que pour une position donnée, on obtient bien toujours la même séquence (voir la section 9.7 pour plus de détails sur cette séquence).

9.6.4 Fusion et restriction de GBF

On peut fusionner asymétriquement deux GBF de même type. Le domaine de définition du GBF résultat est l'union des domaine de définitions des arguments. L'expression

```
merge(g, h)
```

calcule un GBF du même type que `g` et `h` et dont la valeur en une position p est la valeur de `g` en p si p appartient au domaine de définition de `g` et la valeur de `h` en p sinon.

La primitive `restrict` permet de restreindre le domaine de définition d'un GBF :

```
restrict(g, h)
```

a pour résultat un GBF dont la valeur en une position p est la valeur de `g` en p à la condition que la valeur de `h` en p soit interprétée comme la valeur vraie (le résultat retourné, et les arguments partagent le même type). Si `h` n'est pas défini en p, alors le résultat ne sera pas défini en p non plus.

Exemple : partition d'un GBF en deux sous-GBF. Pour un GBF `g`, on peut calculer un GBF `h` de même domaine de définition et dont les valeurs sont `0` ou `1` au hasard, grâce à l'expression

```
h := map[\x.random(2)](g)
```

Ce GBF de booléens permet de partitionner le GBF `g` en deux GBF `g1` et `g2` :

```
g1 := restrict(g, h);;
g2 := restrict(g, map[\x.~x](h));;
```

et on peut reconstruire le GBF `g` à partir de `g1` et `g2` :

```
g == merge(g1, g2)
```

est une expression qui retourne la valeur vraie.

9.6.5 Translation

L'opérateur de translation, présenté dans ce paragraphe, et les opérateur de morphisme et de quotient présentés dans les deux paragraphes suivants permettent de manipuler l'association entre une position et une valeur.

La primitive `translate` permet de translater les valeurs d'un GBF. Par exemple

```
gbf H = < a >;;
h := iota(10, ():seq) following |a>;;
th := translate(h, 27*|a>);;
```

Les éléments de `h` sont localisés aux positions `|0>, |a>, 2*|a>, ..., 9*|a>`. Les valeurs de `th` correspondent au valeurs de `h` translatés par le chemin `27*|a>`. Elles sont localisées aux positions `27*|a>, 28*|a>, ..., 36*|a>`. De manière générale, la fonction `translate(h,` c`)` calcule un GBF où la valeur de la position p est la valeur de `h` au point $p - c$.

9.6.6 Endomorphisme

Si on considère une GBF comme une fonction f qui associe une valeur à une position, alors `translate(`f`,`c`)` est la fonction

$$f' = f \circ tr \qquad \texttt{avec} \qquad tr = \backslash p.(p - c)$$

Autrement dit, $f'(p) = f(tr(p))$. Les liens entre l'argument c (i.e. le « vecteur de translation ») et tr sont simples : tr représente une translation d'un vecteur inverse de c. La forme $f' = f \circ tr$ n'est pas très commode, car elle fait apparaître une fonction tr reliée à l'inverse de c. Il est plus naturelle de la réécrire :

$$f' \circ t = f \qquad \texttt{avec} \qquad t = tr^{-1} = \backslash p.(p + c)$$

car cette forme fait apparaître directement la fonction t spécifiée par le programmeur. L'idée de la fonction `morphism` est de généraliser ce schéma, en adoptant une classe plus grande de fonctions t : les morphismes de positions. Mais avant de voir comment définir un morphisme de position, voyons à quoi correspond exactement le calcul du GBF f'.

Si on représente un GBF f par une liste d'associations (position, valeur), le calcul de f' définie par $f' \circ t = f$, où t est une fonction quelconque, est simple :

$$\left.\begin{array}{lcl} p_1 & \mapsto & v_1 \\ & \dots & \\ p_n & \mapsto & v_n \end{array}\right\} = f \longrightarrow f' = \begin{cases} t(p_1) & \mapsto & v_1 \\ & \dots & \\ t(p_n) & \mapsto & v_n \end{cases}$$

Cependant, il se peut for bien que la fonction t ne soit pas *injective*. Autrement dit, il se peut qu'il existe deux positions p et p' telles que $p'' = t(p) = t(p')$. La fonction f' n'est alors pas bien définie puisqu'on devrait avoir à la fois $f'(p'') = f(p)$ et $f'(p''') = f(p')$. On dira que les positions p et p' collisionent en p'' par t. En cas de collision, on calcule la valeur de f'en p'' grâce à une fonction binaire h qui combine les valeurs des positions qui collisionnent. Par exemple, ici, $f'(p'') = h(f(p), f(p'))$ en supposant que p et p' soient les seules positions en collision sur p''. Pour que cette valeur soit indépendante de l'ordre dans lequel on considère les collisions, il faut que la fonction h soit associative et commutative. Verifier que cette fonction possède ces propriétés relève de la responsabilité du programmeur.

Pour définir un morphisme de position t, il suffit de donner l'image de chaque générateur. Cela se fait par une séquence de couples (un couple est une séquence contenant 2 éléments). Le premier élément du couple est un générateur et le second un chemin. Par exemple, si on suppose la déclaration suivante :

```
gbf H = < x, y >
```

alors la séquence :

```
(|x>, |y>)::(|y>, |x>)::seq:()
```

représente la fonction t :

```
|x>  ↦  |y>
|y>  ↦  |x>
```

Cette fonction t est étendue par linéarité à tout chemin

$$\alpha*|x> + \beta*|y> \quad \longmapsto \quad \alpha*t(|x>) + \beta*t(|y>) = \alpha*|y> + \beta*|x>$$

Par exemple, si f est le GBF représenté en partie gauche de la figure 9.7, alors f' est le GBF illustré à droite. On voit qu'avec ce morphisme de position, on obtient un analogue de la transposition. La figure 9.8 illustre un autre exemple : le morphisme t utilisé permet de « disperser » les valeurs d'un GBF.

Le morphisme de position, spécifié par l'image de chaque générateur, n'est pas nécessairement bien défini. Par exemple, soit le GBF

```
gbf H = < x, y, z; x = z >;;
```

qui introduit un alias `z` pour `x`. La séquence

$t =$ `(|x>, |x>)::(|y>, |y>)::(|z>, 2*|z>)::seq:()`

ne définit pas une fonction t de manière cohérente. En effet, prenons le chemin réduit à `|x>`. Son image par t est `|x>` lui-même. Mais on a `|x>` $=$ `|z>` et donc

$t($`|x>`$) = t($`|z>`$) =$ `2*|z>` $=$ `2*|x>` $\neq$ `|x>`

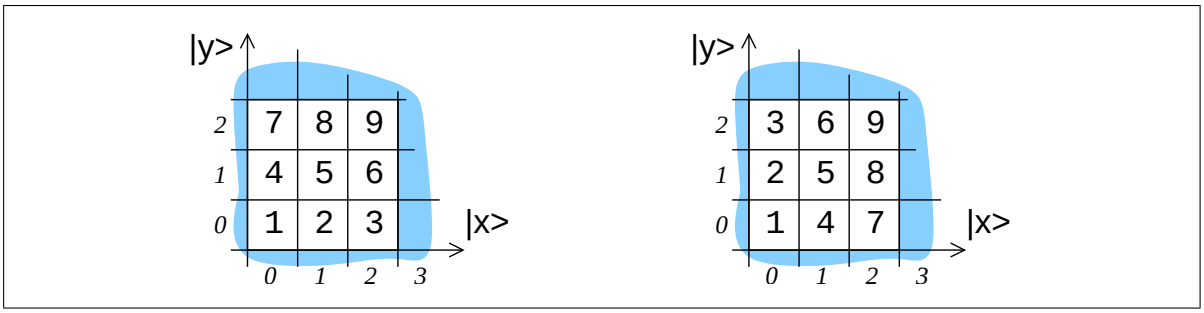


Fig. 9.7: La fonction `morphism` : si f est le GBF de gauche, alors f' = `morphism(h, f, (|x>, |y>)::(|y>, |x>)::seq:())` est le GBF de droite. Dans cet exemple, l'opération réalisée généralise la transposition des matrices. La fonction de collision `h` n'est pas réellement utilisée ici car le morphisme de position est injectif (voir texte).

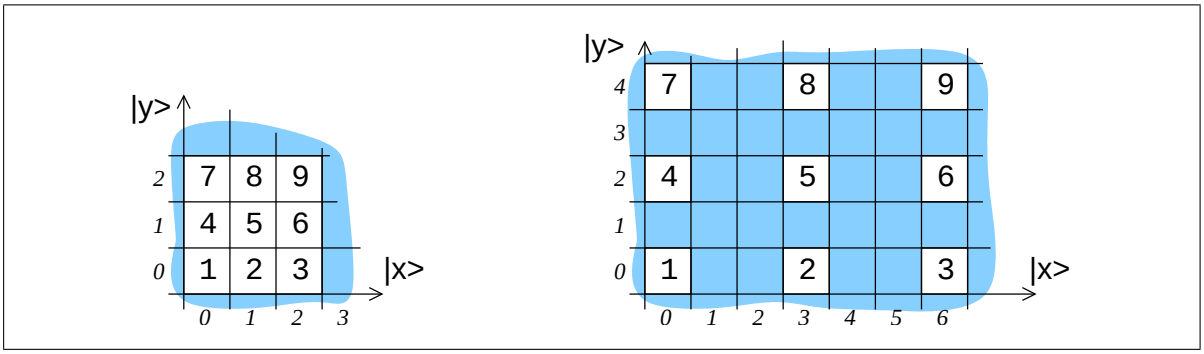


Fig. 9.8: La fonction `morphism` est utilisée pour disperser les valeurs du GBF de gauche g : le facteur de dispersion suivant `|x>` est de 3 et il est de 2 dans la dimension `|y>`. Le GBF de droite s'obtient comme résultat de l'expression `morphism(h, g, (|x>, 3*|x>)::(|y>, 2*|y>)::seq:())`. Cette opération étend les opérations de type *scatter* bien connu dans la manipulation parallèle des tableaux. Là encore, la fonction de collision `h` n'est pas réellement utilisée car le morphisme de position est injectif.

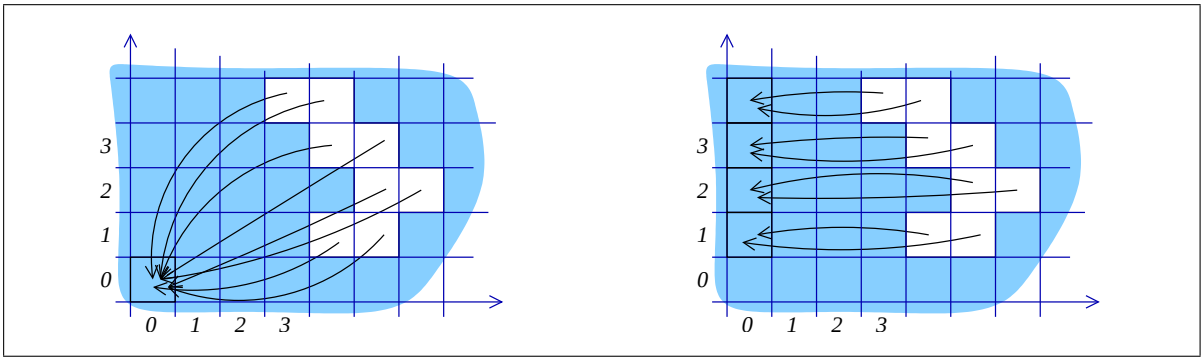


Fig. 9.9: Fonction de collision dans l'opérateur `morphism`. *Figure de gauche.* Toutes les positions du domaine de définition de f sont envoyées à la position `|0>` par le morphisme t = `seq:()`. Le premier argument de la fonction `morphism(h, g, t)` permet de combiner les valeurs qui se collisionent. Le résultat est un GBF dont le domaine de définition est réduit à `|0>` et dont la valeur en `|0>` est la combinaison par `h`, dans un ordre arbitraire, de toutes les valeurs de `g`. *Figure de droite.* Dans cet exemple, le morphisme utilisé est une projection « par ligne » envoyant tous les chemins α`*|x> + `β`*|y>` en β`*|y>`. Cela peut se faire grâce au morphisme t = `(|y>, |y>)::seq:()`.

Donc toutes les séquences d'associations ne correspondent pas à un morphisme bien défini. L'utilisation d'une séquence cohérente est de la responsabilité du programmeur.

Notons enfin que si un générateur ne figure pas dans la séquence spécifiant le morphisme, son image est supposée être le chemin vide.

Exemple : réduction des valeurs d'un GBF. En faisant se collisioner toutes les valeurs d'un GBF g, on peut faire la réductions des éléments de g sans utiliser l'opérateur `fold`. Ainsi, le résultat de l'expression

```
morphism((\x, y.x+y), g, seq:())
```

est un GBF de même type que `g`, dont le domaine de définition est réduit à `|0>` et dont la valeur en `|0>` est la somme de tous les élément de `g`. En effet, l'image de tout générateur est spécifié comme le chemin vide, et donc le morphisme de position utilisé correspond à la fonction constante qui renvoit toujours `|0>`. Cf. l'exemple de gauche de la figure 9.9.

9.6.7 Quotient.

L'exemple précédent montre que l'on peut réaliser une réduction de tous les éléments grâce à la fonction `morphism`. Cette réduction est vue comme une opération de projection, la fonction de collision permettant de calculer le résultat des valeurs se projetant sur la même position. Comme en APL, il est possible de définir une opération de projection suivant une des dimensions. Par exemple :

```
morphism((\x, y.x+y), g, (|y>, |y>)::seq:())
```

réalise une projection suivant `|x>` (les coordonnées suivant `|x>` collapsent en zéro), cf. l'exemple figuré en partie droite de la figure 9.9.

Cependant, l'opérateur `morphism` ne permet pas de faire des projections suivant des directions arbitraires. Une généralisation de cette opération de projection consiste à utiliser un sous-groupe comme « direction de projection » et est relié à l'idée de *groupe quotient*, cf. la section 9.7.

Dans un groupe abélien G, si on se donne un sous-groupe H, on peut partitionner G en des « sous-ensembles parallèles » à H. La figure 9.10 montre des partitions de G induites par divers sous-groupes H. Les éléments d'une partition peuvent s'écrire $x + H = \{x + h, h \in H\}$ et l'ensemble des x permettant de partitionner ainsi tout G forme lui-même un groupe H'. Symboliquement, on peut écrire : $G = H \oplus H'$. Par exemple, cf. figure 9.10, si H est engendré par y, alors H' correspond au sous-groupe[7] $y = 0$; si H est engendré par $x + y$, alors H' correspond au sous-groupe $x + y = 0$

[7] A proprement parlé, un sous-groupe H' n'est pas défini par une équation $w = 0$, mais dans un groupe G, un mot engendre un sous-groupe $H = \{w^n, n \in \mathbb{Z}\}$ et dans un groupe abélien, on peut définir le sous-groupe G/H. La construction de ce groupe, le quotient de G par H, est décrite dans le paragraphe 9.7. Intuitivement, si on assimile un sous-groupe H à n générateurs à un sous-espace de dimension n passant par l'origine, alors G/H est le « sous-espace complémentaire ». La partition du groupe G correspond aux cosets de G/H (les cosets sont aussi appelés classes et on distngue les classes à gauche et les classes à droite, mais ces deux notions sont identiques dans un groupe abélien).

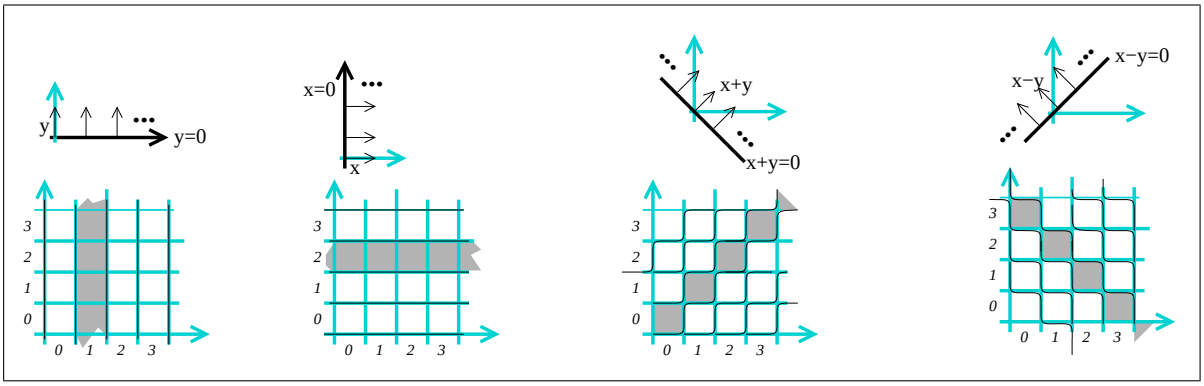


Fig. 9.10: Quatre exemple d'axe d'un quotient. *Première schéma.* Le chemin `|y>` génère un sous-groupe H, correspondant à toutes les positions que l'on peut atteindre en partant de l'origine et en ne faisant que des mouvement `|y>`. On peut *paver* l'espace G tout entier par des copies de H translatées d'une quantité x. On appelle *coset* de H et on note note $x + H$ la translation d'un pavé. Un coset particulier est représenté en grisé dans le schéma. L'idée d'un quotient `quotient(`f`,` g`,` H`)` est de combiner par la fonction f toutes les valeurs d'un coset de H. Le résultat est associé à certains éléments de g qui forment un sous-groupe H'. Pour la première figure, le sous-groupe H est engendré par un générateur et correspond donc à un découpage suivant une dimension. Le sous-groupe H' correspond au sous-groupe engendré par les autres générateurs (ici juste `|x>`). *Deuxième schéma.* On a le cas symétrique du précédent en échangeant le role de `|x>` et `|y>`. *Troisième schéma.* Le partitionnement du GBF g est déterminé par le sous-groupe engendré par `|x>+|y>`. Ce partitionnement correspond à des translation de la « première diagonale ». Le sous-groupe H' correspond à la « deuxième diagonale ». *Quatrième schéma.* Le chemin `|x>-|y>` engendre la seconde diagonale.

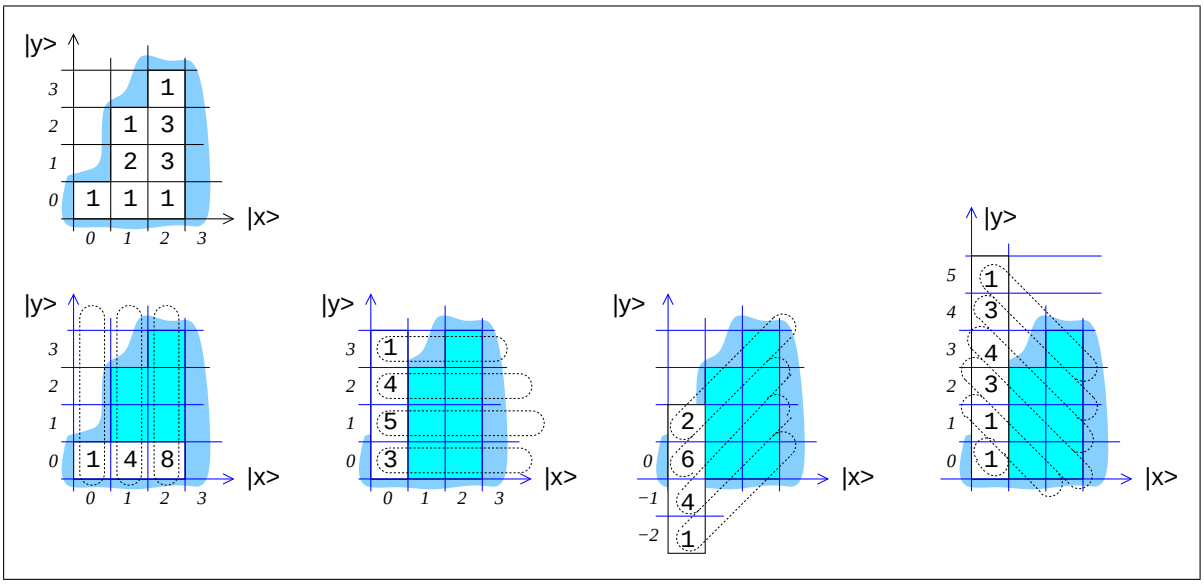


Fig. 9.11: Quatre exemple d'axe d'un quotient. Les quatre figures illustrent le résultat des opérations de quotient suivant les 4 axes de la figure 9.10. Le calcul se fait par une expression `quotient((\x,y. x+y),` g`,` ℓ`)` où ℓ vaut successivement : `|x>,seq:()`, puis `|y>,seq:()`, puis `(|x>+|y>),seq:()` et enfin `(|x>-|y>),seq:()`.

L'opérateur `quotient` permet d'indiquer les générateurs du sous-groupe H définissant la partition (simplement en donnant la séquence des chemins constituant la séquence des générateurs), et la fonction permettant de combiner les valeurs des positions d'un élément de la partitions, voir la figure 9.11.

Exemple : relaxation rouge-noir. Dans certains schémas de résolution numérique, il faut calculer une valeur en chaque point d'une grille par un algorithme en deux phases. Pendant la phase rouge, la nouvelle valeur d'un point sur deux est calculée, et pendant la phase noire, la nouvelle valeur des autres points est calculée. Supposons que l'on veuille calculer pour chaque ligne le maximum de tous les points rouges, et le maximum de tous les points noirs : cela peut se faire en une seule opération grâce à `quotient` :

```
gbf G = <x, y> ;;
quotient((\x, y.max(x,y)), g, 2*|x>::seq:())
```

Le résultat est illustré à la figure 9.12.

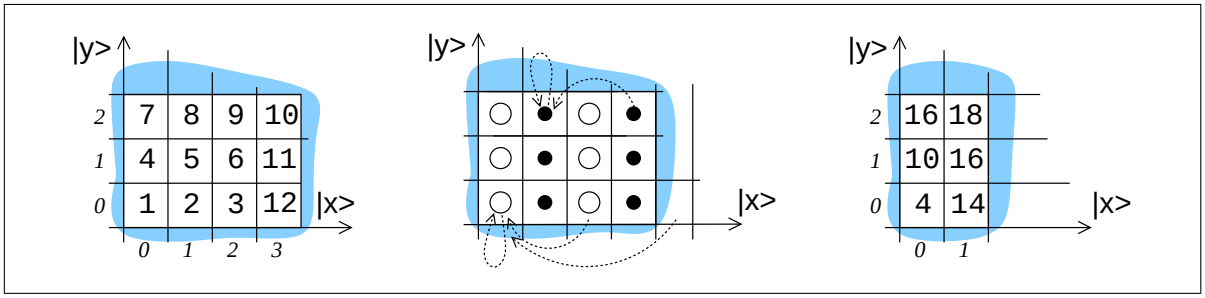


Fig. 9.12: Exemple de quotient appliqué à la partition rouge/noire d'une grille. *Schéma de gauche.* GBF `g` initial. *Schema du milieu.* Partitionnement induit par le sous-groupe engendré par `2*|x>`. *Schema de droite.* GBF résultat de l'expression `quotient((\x, y. x+y), g, 2*|x>::seq:())`.

Exemple : réduction des valeurs d'un GBF. En prenant un sous-groupe H correspondant au groupe tout entier, le complémentaire de H est réduit au *groupe trivial* (qui ne contient que `|0>`). La valeur du quotient en `|0>` est donc la réduction de toutes les valeurs du GBF. Cela constitue donc une troisième manière de faire un `fold` :

```
gbf G = < g1, g2, ..., gn; ...> ;;
quotient((\x, y.x+y), g, (g1, g2, ..., gn))
```

cette dernière expression calcule la somme des valeurs du GBF `g` et le résultat est la valeur de `|0>`.

9.7 Compléments techniques : représentation normalisée d'une position dans un GBF abélien et groupe quotient

Nous donnons dans cete section quelques indications sur la représentation normalisé d'une position dans un groupe abélien. Les considérations ci-dessous permettent d'implémenter la fonction `same` et

de calculer la séquence identifiant une position dans la fonction `map_indexed_g`. 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, $\mathbb{Z}/n$ représentent 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 *module de torsion*. Un élément d'un module de torsion $\mathbb{Z}/n$ peut se représenter de manière unique par un nombre 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 dimensions supérieures à 1 sont simplement des produits cartésiens de $\mathbb{Z}$-modules de dimensions 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).

Le théorème fondamentale des groupes abéliens finiment engendrés. Le théorème fondamentale des groupes abéliens finiment engendrés indique que pour tout groupe G engendré par un nombre fini de générateurs et de relations, tels que par exemple les GBF définis par une présentation, ce groupe G 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 chemin c dans G (i.e. à tout mot de générateurs), on peut associer l'uplet d'entiers représentant de manière unique l'élément correspondant à p dans le $\mathbb{Z}$-module isomorphe à G. Le second argument de la fonction `map_indexed_g` correspond à cet uplet d'entiers. Pour connaître le coefficient de torsion de la ième composante, on peut utiliser la fonction `torsion` :

```
torsion(g)
```

retourne la séquence des coefficients de torsion du $\mathbb{Z}$-module canoniquement associé au groupe du GBF `g`. L'argument `g` peut aussi être un chemin. Dans la séquence résultat, les entiers doivent s'interpréter comme suit :

i^{eme} élément	*interprétation*
0	la i^{eme} dimension est isomorphe à $\mathbb{Z}$
1	cette dimension ne doit pas être prise en compte la coordonnée correspondante aura toujours pour valeur 0
n	la i^{eme} dimension est isomorphe à $\mathbb{Z}/n$

Par exemple :

```
gbf G = <x, y, z; x = z >;;
torsion(G:());; ↩
1, 0, 0
```

`G` est un GBF définit par 3 générateurs. Le générateurs `x` est juste un alias pour `z` (ou vice-versa). Par exemple, les positions désignées par les chemins `3*|x> + |y>` et `|y> + 3*|z>` sont bien évidemment les mêmes. Il est intuitivement claire que chaque position peut se repérer uniquement par deux coordonnées prenant leur valeur dans $\mathbb{Z}$ (par exemple suivant la dimension `|y>` et la dimension `|z>`). Ceci éclaire le résultat de la fonction `torsion` : la forme normalisée d'une position de `G` correspond à un triplet dont la première coordonnée doit être ignorée, et dont les second et troisième composants prennent leurs valeurs dans $\mathbb{Z}$.

Le cas des groupes abéliens libres. Un groupe abélien libre est défini uniquement par la donné 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.

Groupe quotient. Le quotient G/H d'un groupe G par un sous-groupe H est l'ensemble des classes d'équivalences de la relation d'équivalence $\simeq$:

$$x \simeq y \quad \Leftrightarrow \quad x - y \in H$$

La classe d'équivalence de x est le coset $x + H = \{x + h, h \in H\}$. Afin de munir G/H d'une structure de groupe, il est nécessaire que H vérifie la condition :

$$\forall x \in G, \quad x + H = H + x$$

Si le sous-groupe H vérifie cette condition, on dit que c'est un *sous-groupe normal* (ou *distingué*). Tout sous-groupe est un sous-groupe normal dans un groupe abélien, puisque les éléments commutent. Nous pouvons alors définir une structure de groupe sur G/H : H est l'élément neutre, et si u et v sont deux représentants des classes H_1 et H_2, alors $H_1 + H_2$ est définie comme la classe d'équivalence de $u + v$. Cette définition ne dépend pas des représentations u et v qui sont choisies.

Cette opération de quotient est reliée à la construction d'un groupe par une présentation finie. Par exemple, si on considère les deux groupes associés $G1$ et $G2$ aux GBF :

```
gbf G1 = < a >;;
gbf G2 = < a ; 5a >;;
```

alors le groupe $G2$ est isomorphe au quotient de $G1$ par le sous-groupe engendré par `5a`. Et plus généralement, si $\mathcal{P}$ est une présentation spécifiant une liste d'équations ℓ, alors le groupe défini par la présentation $\mathcal{Q} = \mathcal{P} + w$, où $+w$ indique que la liste d'équations ℓ a été augmentée par une équation de la forme $w = 0$, est le quotient de $\mathcal{P}$ par le sous-groupe engendré par w.

En `MGS` l'opération `quotient` calcule les classes d'équivalence en choisissant un représentant arbitraire p dans le groupe de départ (on plonge donc le groupe quotient dans le GBF de départ). Le choix du représentant p dépend de l'implémentation, mais la classe de 0 est représentée par 0. La valeur associée au représentant p est la combinaison par la fonction de collision de toutes les valeurs du GBF de départ appartenant aux positions de la même classe d'équivalence. Le résultat est un GBF de même type que le GBF de départ, ce qui évite de créer un nouveau type de GBF.

9.8 Filtrage dans un GBF

9.9 Exemples

9.9.1 Propagation d'une tumeur : le modèle d'Eden

9.9.2 Les marguerites

9.9.3 Les poissons et les requins

Chapitre 10

Compléments : sous-typage et chemin linéaire de filtrage

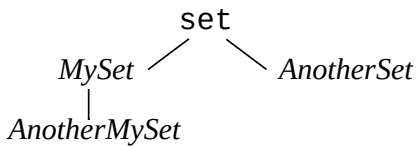
Nous présentons ici deux notions qui complètent ce qui a été vu précédemment sur les types de collections et sur le filtrage. Le *sous-typage* permet d'affiner les types collections définissables en `MGS`. La notion de *chemin de filtrage* permet quant à elle de généraliser les règles d'une transformation en interprétant une séquence comme la sous-collection filtrée dans une topologie arbitraire. Les notions présentées s'appliquent à tout type de collection

10.1 Sous-type d'une collection

Il est souvent nécessaire de distinguer plusieurs collections de la même sorte (par exemple, des multi-ensembles imbriqués dans un autre multi-ensemble). Plusieurs solution existent pour permettre cette distinction. Une solution en `MGS` est de distinguer entre les collections d'une même sorte par des *types*. Le type d'une collection doit être vu comme une étiquette qui ne change pas la structure de la collection. Les types sont organisés selon une relation de sous-typage. La relation de sous-typage sur les collections hiérarchise les types de collections en un ensemble partiellement ordonné. Ce sont les éléments maximaux de cette hiérarchie qui constituent les différentes sortes de collections (`seq`, `bag`, `gbf grid`, etc). Chaque sorte de collection est la racine de l'*arbre* de ses sous-types.

Voici un exemple de déclarations de types collection :

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



Ces trois déclarations définissent la hiérarchie de trois types figurée en partie droite. Le type *AnotherMySet* est un sous-type de *MySet* lui-même sous-type de `set` . Le type *AnotherSet* est aussi un sous-type de `set` mais il n'est pas comparable à *MySet*.

Un type introduit lors d'une déclaration de type peut être utilisé dans un filtre (Cf. sec-

tion 6.1, page 35) ou comme prédicat utilisé pour tester si une valeur est d'un type donné. Un type de collection monoïdale peut aussi être utilisé dans la construction d'une collection par l'énumération de ses éléments :

```
1,  1+1,  2+1,  2*2,  MySet:()
```

est une expression s'évaluant en un ensemble de quatre entiers : 1, 2, 3 et 4. La sorte de collection est un ensemble , et son type est `MySet`. En fait, l'expression `Myset:()` dénote le `MySet` vide et « , » est l'opérateur de join surchargé (Cf. section 7.2, page 46).

Cette construction est valable pour toutes les sortes de collections. Par exemple :

```
gbf grid = ...  ;;
collection Mgrid = grid;;
```

défini un sous-type de `grid`. On peut créer un `Mgrid` vide de la manière usuelle :

```
Mgrid:()
```

Si cela est nécessaire, il est possible de créer une collection d'un type donné, à partir d'une collection A et du type d'une collection B, grâce à la primitive `setkind(`B`,` A`)`. Le résultat de cette expression est une collection similaire à A mais dont le type est celui de la collection B. Par exemple :

```
collection Mseq = seq;;
A := 1, 2, 3;;
Mseq(A), Mseq(setkind(Mseq:(), A))
```

renvoie la séquence *faux, vrai*.

Le type d'une collection est pris en compte pour de nombreuses opérations sur les collections. Par exemple, l'*ajout* de deux collections de type A et B donne une collection de type C correspondant à l'ancêtre commun de A et de B (sur l'exemple précédent, `set` est l'ancêtre commun de `MySet` et de `AnotherSet`). Autre exemple, `MySet` est l'ancêtre commun de *AnotherMySet* et de lui-même. Plus généralement, si une opération $\oplus$ entre deux collections A et B demande des collections de même type, alors :

$$type(A \oplus B) = ancêtre\_commun(A, B)$$

Nota Bene. Une déclaration `collection ... = ...` introduit un sous-type d'un type collection[1] pré-existant. Les racines des arbres de sous-typage des collections correspondent :

1. aux types collections de base (comme `seq` ou `bag`), ou bien
2. aux types collections définis par l'utilisateur, comme par exemple ceux introduits par une déclaration `gbf ... = ...`.

Parfois, afin de distinguer les sortes de collections de leurs sous-types, on parle de type pour désigner les sortes de collections et de *couleur* pour désigner les sous-types. Ainsi une déclaration `collection ...` introduit une nouvelle couleur d'un type collection.

[1] Il existe des sous-types pour les enregistrements par exemple, mais on ne peut actuellement pas dériver de nouveaux sous-types d'enregistrements par une déclaration `collection`.

10.2 Exemple d'utilisation : représentation de formules logiques par des collections

Nous allons illustrer la notion de sous-types de collection en utilisant ce mécanisme pour représenter simplement des termes logiques.

Problématique

En logique des prédicats du premier ordre on manipule des termes représentant des formules logiques dans le but de déterminer des formules équivalentes qui soient plus simples ou répondant à certains critères. Pour cela on transforme les termes selon des règles comme $\neg(\neg A) \rightsquigarrow A$ ou encore $A \wedge (B \vee C) \rightsquigarrow (A \wedge B) \vee (A \wedge C)$. `MGS` est particulièrement bien adapté à ce genre d'exercices puisque le programme réalisant ces manipulations consiste en une transformation composée des règles de réécriture des termes. Le résultat est obtenu en appliquant cette transformation jusqu'à atteindre un point fixe.

Nous allons décrire ici un codage des termes logiques et un programme calculant la forme normale disjonctive d'une formule logique.

Choix des structures de données

L'expression du programme sous forme de règles le rend particulièrement simple à lire mais ceci n'est pas le seul avantage qu'apporte `MGS`. En choisissant la bonne structure de données une partie du travail est donnée « gratuitement ». En effet les propriétés d'associativité, de commutativité et d'idempotence du *et* et du *ou* logiques (notés $\wedge$ et $\vee$) n'ont pas à être programmées si l'on utilise des ensembles pour représenter les formules et l'union ensembliste pour représenter ces deux opérateurs.

Ainsi une règle telle que $A \wedge A \rightsquigarrow A$ devient superflue puisque $A \wedge A$ sera représentée par $A \cup A$ qui a la même valeur que A. Rappelons qu'en `MGS`, si `A` et `B` sont deux ensembles, `A,B` calcule leur union.

Code commenté

Le $\wedge$ et le $\vee$ sont deux types d'ensembles, le $\neg$ est une collection singleton par construction et le $\Rightarrow$ est une séquence à deux éléments par construction. Nous avons représenté également le $\neg$ et le $\Rightarrow$ par des collections afin d'utiliser des itérateurs sur les termes de façon transparente. Le $\Rightarrow$ n'étant pas commutatif, une séquence est utilisée plutôt qu'un ensemble. Seuls les atomes ne sont pas représentés par des collections mais par des chaînes de caractères.

```
collection Et :  set ;;
collection Ou :  set ;;
collection Non = seq ;;
collection Imp = seq ;;
```

Les fonctions d'accès suivantes permettent la transparence des collections dans l'implémentation du $\neg$ et du $\Rightarrow$. Ceci permet à l'algorithme d'être indépendant du choix de représentation de la négation et de l'implication.

```
fun get_pre(x)=hd(x);;
fun get_post(x)=hd(tl(x));;
fun get_non(x)=hd(x);;
```

Contient_Et, *Contient_Ou* et *Contient_Non* indiquent qu'un terme contient un $\wedge$, un $\vee$ ou bien un $\neg$.

```
fun Contient_col (pred,x) =
    ~(empty(x)) & ( pred(hd(x)) | Contient_col(c,tl(x)) );;
fun Contient_Et(x) = Contient_col (Et,x) ;;
fun Contient_Ou(x) = Contient_col (Ou,x) ;;
fun Contient_Non(x)= Contient_col (Non,x);;
```

distrib_aux réalise la distribution d'un $\wedge$ sur un $\vee$.

```
fun aux2(X, E) = hd(X)::(hd(E)::tl(X))::seq:()

fun distrib_aux (E) =
    if (Ou(hd(E))) then hd(E)::tl(E)::seq:()
                   else aux2(distrib_aux(tl(E)), E)
    fi;;
```

FND est la transformation principale du programme et permet d'obtenir la forme normale disjonctive simplifiée d'une formule.

```
   // On définit l'alias |=> pour une flèche qui n'applatit pas
arrow |=> = ={flat=0}=>;;

trans FND = {

   // a ⇒ b devient ¬a ∨ b
   x / Imp(x)
   |=> get_post(x) ::( (get_pre(x::Non:()) ::Ou:()) ) ;;

   // ¬(¬a) devient a
   x / Non(x) & Contient_Non(x)
   |=> get_non(get_non(x)) ;;

   // distribution du non sur le ou : ¬(a ∨ b ∨ ...) devient ¬a ∧ ¬b ∧ ...
   x / Non(x) & Ou (get_non(x))
   |=> fold [mycons, neg , Et:() ] (get_non(x)) ;;

   // distribution du non sur le et : ¬(a ∧ b ∧ ...) devient ¬a ∨ ¬b ∨ ...
   x / Non(x) & Et (get_non(x))
   |=> fold [mycons, neg , Ou:() ] (get_non(x)) ;;

   // distribution du et sur le ou (forme normale disjonctive) : a ∧ (b ∨ c ∨ ...) devient (a ∧ b) ∨ (a ∧ c) ∨ ...
```

```
x / Et(x) & Contient_Ou(x)
|=> fold [mycons,( fun (l)=hd(distrib_aux(x))::l ),Ou:()] (tl(distrib_aux(x)));;

// associativite du ou : a ∨ (b ∨ c) devient a ∨ b ∨ c (codé par un ensemble {a, b, c} de type "ou")
x / Ou(x) & Contient_Ou(x)
|=> flatten_ou (x);;

// associativité du et : a ∧ (b ∧ c) devient a ∧ b ∧ c (codé par un ensemble {a, b, c} de type "et")
x / Et(x) & Contient_Et(x)
|=> flatten_et (x);;

// suppression des ou à un seul élément
x / Ou(x) & (size(x)==1)
|=> hd(x);;

// suppression des et à un seul élément
x / Et(x) & (size(x)==1)
|=> hd(x);;

// application récursive de la transformation aux sous termes
x /  atom(x)
|=> FND['fixpoint](x)
} ;;
```

La dernière règle de *FND* permet d'appliquer la transformation à tous les sous-termes du terme considéré.

Utilisation

Une transformation s'appliquant à une collection, on peut appliquer *FND* à un multi-ensemble de formules par exemple. Ici la transformation est appliquée aux formules $a \wedge \neg\neg a \wedge (b \vee c)$ et $a \vee b \vee c \vee (d \wedge e \wedge f)$:

```
FND ['fixpoint] (
    (     ("a",( ("a",Non:()),Non:()),( "b","c",Ou:()), Et:()),
          ("a","b","c",("d","e","f",Et()),Ou:()),
          Bag:()
    ) );;
```

L'utilisation de cette tranformation montre que `'fixpoint` et `'fixrule` ne sont pas équivalents. En effet, on peut toujours appliquer la dernière règle, même lorsqu'un point fixe a été atteint.

10.3 Filtrage linéaire et rôle particulier de la séquence

Un motif comme `(x, y, z) as S` en partie gauche d'une règle filtre une sous-collection `S` de trois éléments. Si la règle s'applique sur un ensemble, on a donc un sous-ensemble de trois éléments, si la règle s'applique à une séquence, on a une séquence de trois éléments, si la règle s'applique à un GBF, on a une partie connexe de 3 éléments, i.e. il existe un générateur u

et un générateur v telle que *position*(`y`) = *position*(`x`) + u et *position*(`z`) = *position*(`y`) + v.

Mais d'un autre point de vue, la spécification d'un motif repose essentiellement sur la définition d'une *chemin linéaire* à suivre dans une collection. Par exemple le motif :

```
x, (y / y > 0)
```

peut s'interpréter comme : à partir du point `x`, passer par un point `y` qui doit être voisin de `x` et dont la valeur est supérieure à 10. Dans une séquence, il y a un seul voisin possible et si la valeur de celui-ci ne satisfait pas la condition, un tel chemin n'existe pas. Dans un ensemble, il y aurait plusieurs voisins dont certains pourrait avoir une valeur supérieur à 10. Il y aurait donc plusieurs chemins éventuels dans un ensemble qui répondrait au chemin ci-dessus. Idem pour un GBF. L'application d'une règle consiste à emprunter un tel chemin. Cette idée de chemin est encore plus claire dans le cas des GBF :

```
((x |est>*) as X) |north> ((y |north>*) as Y)
```

est un chemin de longueur indéterminé qui est spécifié par les « instructions » suivantes : d'abord aller vers l'est (partie de chemin identifiée par `X`) un nombre indéterminé de fois, puis aller vers le nord (partie de chemin identifiée par `Y`) un nombre indéterminé de fois.

Si on regarde un motif du point de vue des opérations de filtrage successives à réaliser, un motif spécifie donc en plus d'une sous-collection, un certain séquencement des éléments de la sous-collection : dans l'exemple précédent, il faut d'abord filtrer `X` puis `Y`. L'ordre des éléments filtrés est imposé par la syntaxe gauche/droite de la grammaire des motifs.

Ce point de vue, qui superpose une structure de séquence S_g sur la collection filtrée, justifie le comportement particulier suivant :

> Si la partie droite d'une règle s'évalue en une séquence S_d, alors les éléments de S_d sont utilisés dans l'ordre pour remplacer les éléments de S_g.

il faut cependant que la propriété `flat` de la règle soit à *vrai* et, si la règle s'applique à un GBF, il faut aussi que la longueur de S_d soit la même que celle de S_g.

L'intérêt de cette règle est visible surtout dans le cas des GBF : on n'a pas besoin de reconstruire un GBF en partie droite de chaque règle, mais uniquement une séquence qui viendra remplacer le chemin défini par le filtre. Par exemple

```
x, y, z => 1, 2, 3
```

remplacera `x` par `1`, `y` par `2` et enfin `z` par `3`. Si cette règle s'applique sur un GBF, chaque virgule en partie gauche correspond à un choix d'un générateur du groupe. Mais grâce à cette interprétation des séquences, chaque virgule en partie droite peut aussi s'interpréter comme un générateur, celui de la virgule correspondante en partie gauche.

10.4 Les entrées/sorties et le visualisateur `imoview`

Bibliographie

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

[BL74] J. Bard and I. Lauder. How well does turing's theory of morphogenesis work ? *Journal of Theoretical Biology*, 45 :501–531, 1974.

[BM86] J. P. Banatre and Daniel Le Metayer. 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, 18 September 1995.

[col95] collectif. *Encyclopædia Universalis*, volume 10, chapter groupes (Mathématiques), page 987. 1995.

[FM97] P. Fradet and D. Le Métayer. Shape types. In *Proc. of Principles of Programming Languages*, Paris, France, Jan. 1997. ACM Press.

[FS96] Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, Programs from outer space). In *Conference Record of POPL '96 : The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 284–294, St. Petersburg Beach, Florida, 21–24 January 1996.

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

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

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

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

[GS90] C. A. Gunter and D. S. Scott. *Handbook of Theoretical Computer Science*, volume 2, chapter Semantic Domains, pages 633–674. Elsevier Science, 1990.

[HP96] Mark Hammel and Przemyslaw Prusinkiewicz. Visualization of developmental processes by extrusion in space-time. In Wayne A. Davis and Richard Bartels, editors, *Graphics Interface '96*, pages 246–258. Canadian Information Processing Society, Canadian Human-Computer Communications Society, May 1996. ISBN 0-9695338-5-3.

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

[MFP91] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *5th ACM Conference on Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144, Cambridge, MA, August 26–30, 1991. Springer, Berlin.

[Mic96a] O. Michel. Design and implementation of 81/2, a declarative data-parallel language. *Computer Languages*, 22(2/3) :165–179, 1996. special issue on Parallel Logic Programming.

[Mic96b] O. Michel. *Représentations dynamiques de l'espace dans un langage déclaratif de simulation.* PhD thesis, Université de Paris-Sud, centre d'Orsay, December 1996. N°4596, (in french).

[Mos90] P. D. Mosses. *Handbook of Theoretical Computer Science*, volume 2, chapter Denotational Semantics, pages 575–631. Elsevier Science, 1990.

[NO94] Susumu Nishimura and Atsushi Ohori. A calculus for exploiting data parallelism on recursively defined data (Preliminary Report). In *International Workshop TPPP '94 Proceedings (LNCS 907)*, pages 413–432. Springer-Verlag, November 94.

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

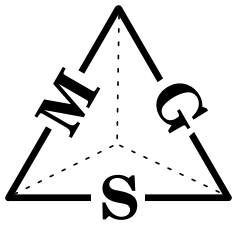
[Rém92] Didier Rémy. Syntactic theories and the algebra of record terms. Technical Report 1869, INRIA-Rocquencourt, BP 105, F-78 153 Le Chesnay Cedex, 1992.

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

[Vic88] Steven J. Vickers. *Topology Via Logic*, volume 5 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1988.

[VN66] J. Von Neumann. *Theory of Self-Reproducing Automata.* Univ. of Illinois Press, 1966.

Index


M
G
S