Développement de la programmation par tuilage pour les systèmes multimédias interactifs

Simon Archipoff

THÈSE DE DOCTORAT DE L'UNIVERSITÉ DE BORDEAUX

École Doctorale Mathématique et Informatique

Développement de la programmation par tuilage pour les systèmes multimédias interactifs

Présentée par
Simon ARCHIPOFF

Pour obtenir le grade de
Docteur

Spécialité
Informatique

Soutenue le 27 novembre 2020 devant le jury composé de :

M. Marc Zeitoun	Président
M. Jean-Louis Giavitto	Rapporteur
M. Jean Bresson	Rapporteur
M. David Janin	Directeur de thèse
Mme. Catherine Dubois	Examinatrice
Mme. Myriam Desainte-Catherine	Examinatrice

université de BORDEAUX

LABORATOIRE BORDELAIS DE RECHERCHE EN INFORMATIQUE LaBRI

Résumé :

L'apparition des ordinateurs a été suivie de près par l'apparition d'applications multimédias. Pour concevoir ces applications, les utilisateurs d'ordinateurs ont souhaité se doter d'outils fiables, simples, et puissants leur permettant de se concentrer sur le cœur de leurs applications plutôt que sur les détails techniques. Ce travail de thèse s'inscrit dans cette démarche.

Nous allons aborder la programmation temporelle et spatiale, à travers les cas particuliers respectifs de la musique et du dessin. Usuellement, la démarche pour réaliser ce genre de travaux est de partir d'un cas d'usage, et de déduire quelles abstractions seraient adaptées à la modélisation des concepts manipulés. D'une certaine façon, nous avons suivi la démarche inverse. C'est-à-dire que nous sommes partis d'une abstraction et nous avons étudié si et comment elle se concrétisait dans le cas qui nous intéressait. Naturellement, nous savions déjà que cette abstraction semblait adaptée.

L'abstraction que nous allons traiter est algébrique, autrement dit, les programmes seront manipulés comme les termes d'une algèbre bien choisie, issue d'une théorie mathématique développée depuis les années cinquante, la théorie des monoïdes inversifs.

Cette thèse présente les résultats de ces travaux, plus exactement la manière dont les monoïdes inversifs peuvent être utilisés pour fournir à un développeur une interface pour concevoir des applications multimédias interactives.

Mots clés : multimédia, langage de programmation, programmation 3D, langage applicatif dédié, sémantique, animation temporelle, son, programmation fonctionnelle

Laboratoire Bordelais de Recherche en Informatique
(LABRI)-UMR 5800

Development of tiling programming for interactive multimedia systems

Abstract :

The spreading of computers was closely followed by the appearance multimedia applications. To design these applications, computer users wanted to acquire reliable tools, simple, and powerful, allowing them to focus on the heart of their applications rather than the technical details. This thesis is part of this process.

We will approach temporal and spatial programming through particular cases of music and drawing, respectively. Usually, the procedure for carrying out this type of work is to start from a use case, and deduce which abstractions would be adapted to the modeling of the concepts handled. In a sens, We took the opposite approach. We started from an abstraction and we studied if and how it suited the case that interested us. Naturally, we already knew that this abstraction seemed appropriate.

The abstraction that we are going to treat is algebraic, in other words, the programs will be handled like the terms of a well chosen algebra, resulting from a mathematical theory developed over the 50's, the theory of invert monoids.

This thesis presents the results of this work, more exactly the how inverting monoids can be used to provide to a developer an interface to design applications interactive multimedia.

Key words : multimedia, programming language, 3D programming, Domain Specific Language, semantics, temporal animation, sound, functional programming

Remerciements

Cette thèse n'est pas le fruit exclusif de mon travail. De nombreuses personnes ont contribué plus ou moins directement à la conduite de cette entreprise à son terme. Je souhaiterais donc témoigner ici de ma gratitude envers ces personnes.

Je tiens avant tout à remercier mon directeur de thèse, David Janin, pour sa patience, ses précieux conseils ainsi que sa bienveillance durant ces dernières années.

Je remercie le Scrime, ses membres et tout particulièrement Annick Mercier pour son accueil. Je remercie également Jean-Michaël Celerier pour son soutien et ses conseils, ainsi que Bernard Serpette.

Je tiens également à remercier les membres des équipes enseignantes dont j'ai eu la chance de faire partie. Je les remercie pour la confiance qu'ils m'ont accordée. Je veux citer tout particulièrement Philippe Narbel, David Renault, Guillaume Blin, Marc Zeitoun, Samuel Thibault, et Irène Durand. Je tiens à adresser mes remerciements particuliers à Lamine Lamali pour sa relecture attentive.

Je tiens aussi à remercier l'université de Bordeaux, le LaBRI, et l'EDMI. J'ai eu un parcours assez chaotique, et je suis extrêmement reconnaissant à l'institution de m'avoir offert la possibilité d'en arriver jusqu'ici. J'espère de tout cœur que les portes de l'université resteront grandes ouvertes à tous. Je remercie le personnel administratif de l'université de Bordeaux et du laboratoire. Toujours extrêmement efficace et agréable. Cette thèse doctorale a été l'expérience la plus enrichissante de ma vie. Pour la première fois, je me retrouvais seul avec mes forces et mes faiblesses face à la recherche. Heureusement, j'ai toujours pu trouver toute l'aide et la bienveillance dont j'ai pu avoir besoin. Finalement, cette expérience m'a profondément changé tant sur le plan professionnel que personnel.

Je remercie Yi-Ping pour son soutien et ses encouragements.

Enfin, je présente mes excuses à toutes celles et ceux que j'omets de nommer ici.

Table des matières

1

Introduction

Tout progrès scientifique et technologique s'accompagne de nouvelles applications artistiques. Nous pouvons par exemple citer le développement de l'horlogerie de précision au XVIII^{e} siècle. Cette innovation a conduit entre autres choses à la réalisation de scènes musicales automatiques. Cela va des boîtes à musique jusqu'aux automates humanoïdes jouant d'un instrument. Nous nous souviendrons par exemple de « la joueuse de tympanon »[24], automate datant de 1784, réalisé par l'horloger allemand Pierre Kintzing.

Ces réalisations, à la frontière entre les arts et la technologie, nécessitaient à l'époque des ressources importantes. Des matières premières et des composants coûteux, des journées entières de conception, d'assemblage et de calibrage, pour un résultat saisissant mais répétitif. Aujourd'hui, avec le développement de l'informatique, la réalisation de tels systèmes semble à la portée de tous.

A priori, un bon ordinateur doté d'un écran et d'une carte son doit suffire à la reproduction de ces scènes musicales. Mais a posteriori, nous remarquons que la programmation d'une machine est une chose ardue. Il faut doter l'ordinateur d'un outil ergonomique permettant de faire le pont entre les idées de l'artiste et leurs réalisations informatiques.

Une des finalités de ce projet est de proposer à des artistes – a priori non-informaticiens – un outil permettant de réaliser ce pont dans une démarche artistique. Il n'est donc pas raisonnable de confronter ces utilisateurs à la rudesse des langages de programmation standards, avec leur syntaxe cryptique, leur sémantique supposant une bonne compréhension de l'architecture sous-jacente, ou simplement les écueils des définitions divergentes, des bugs, etc.

Nous souhaitons donc simplifier le développement en utilisant un autre modèle de programmation, c'est-à-dire, en proposant à l'artiste qui veut

transcrire ses idées en code une interface adaptée aux médias manipulés, indépendante du langage sous-jacent.

1.1 Modélisation algébrique

De toute évidence, il est absolument impossible de permettre à l'artiste de manipuler directement les objets qu'il souhaite. Ces derniers ne sont pas tangibles. Il peut s'agir de « gestes », de « traits », de « coups de pinceaux »... Une approche possible est une représentation abstraite de « métaphores » utiles à l'artiste dans son processus de création. Ainsi, les connaissances que l'utilisateur a de ses abstractions vont l'aider à comprendre le comportement des objets sous-jacents. L'utilisateur pourra développer son application en manipulant ces abstractions. De plus, nous pourrons donner à ces abstractions plusieurs interprétations. Le code ainsi produit sera générique.

Cette thèse traite de cette problématique. Dans ce document nous rendons compte de travaux sur l'étude et le développement d'un modèle et d'un langage de programmation dédié qui permet la spécification et la réalisation de systèmes musicaux. Ce langage se révèle aussi applicable à la réalisation d'animations associées à ces systèmes musicaux.

La démarche habituelle lorsque l'on souhaite développer un langage de programmation est de partir d'un problème et d'en extraire les abstractions pertinentes pour le traiter. D'une certaine façon, nous avons effectué la démarche inverse. L'un des partis pris de ces travaux est d'adopter une modélisation algébrique. Le programmeur va partir d'objets de base pour en former des plus complexes à l'aide d'un opérateur de composition.

Un exemple proche du cas qui nous occupe est le montage d'un film sur support magnétique ou photochimique. Une fois les différents plans isolés et sélectionnés (en coupant physiquement la bande), le montage est assuré en recollant bout à bout les plans les uns aux autres comme souhaité. L'opérateur de composition est le collage, c'est-à-dire la mise en séquence de deux plans. On obtient alors une structure de monoïde [1]q. Il existe également un second monoïde, c'est celui de la composition parallèle, en l'occurrence, la superposition de deux séquences d'images, typiquement utilisée pour faire un fondu enchaîné.

Les objets que l'on manipule ici sont un peu plus riches. Ils contiennent non seulement le média, mais ils portent également en eux la manière dont ces médias peuvent être assemblés les uns avec les autres. Cela permet entre autres choses d'avoir un seul opérateur pour la mise en séquence et la mise

1. une structure associative avec un neutre

en parallèle [15]. De même, cela permet aussi de les équiper d'une notion d'inverse un peu plus faible que celle des groupes. Cela permet enfin de s'affranchir de la séquentialité d'un monoïde dans le cas général.

Plus précisément, les objets primitifs que nous proposons au programmeur de manipuler sont les éléments d'un monoïde inversif [26]. Les monoïdes inversifs ont été expérimentés avec succès pour la modélisation de phénomènes musicaux [8, 21]. Ce travail de thèse s'inscrit dans la poursuite de cette étude avec son extension aux cas interactifs, aux dessins, et aux animations.

Lorsqu'un peintre travaille, chacun de ses gestes tombe dans l'une des deux catégories suivantes. D'une part, les choses qu'il peut défaire, par exemple changer d'outil, déplacer sa main sur la toile, etc. C'est-à-dire toutes les actions qui sont réversibles. Et d'autre part, les chose qu'il ne peut pas défaire, comme déposer de la peinture sur la toile, l'étaler, etc. C'est-à-dire les actions qui augmentent l'entropie de l'œuvre.

Le processus de création peut ainsi être modélisé par une mise en séquence d'actions de ces deux familles. Autrement dit, ce processus peut être vu comme une composition de d'actions dans un espace où certaines sont réversibles, et d'autres pas. Les monoïdes inversifs sont une structure à mi-chemin entre les monoïdes et les groupes qui permettent de modéliser cela.

1.2 Modélisation

Au cours de cette thèse, nous allons nous concentrer sur trois domaines d'applications spécifiques : la musique, les dessins et les animations.

1.2.1 Musique

Le cas le plus simple est celui du compositeur, seul avec sa feuille. Il peut prendre toutes les libertés qu'il souhaite pour écrire sa musique. C'est-à-dire qu'il peut faire des sauts dans le temps, passer de l'introduction à la clôture, pour revenir au milieu, faire des références à des choses qu'il n'a pas encore écrite, etc. dans le processus de modélisation associé. C'est ce que nous appellerons la *programmation temporelle hors-temps*.

Un autre cas possible apparaît lorsque le musicien joue de la musique déjà écrite. Dans ce cas les caractéristiques intrinsèques du temps, (impossibilité de changer le passé, impossibilité de prédire le futur,...) sont à prendre en compte. Notre processus doit produire la musique au fur et à mesure de l'écoulement du temps, en fonction de ses entrées et d'après

le programme écrit. Le processus de modélisation temporelle associé est appelé la *programmation dans le temps*.

1.2.2 Dessins et animations

Dans le cas des dessins, un peintre qui travaille sur sa toile, tout est analogue au cas du compositeur de musique. Ce dernier peut se déplacer dans le temps pour placer des notes, le peintre se déplace sur sa toile pour placer des traits de pinceaux. Dans le processus de modélisation associé, nous parlerons de *programmation spatiale*.

Cependant, une différence demeure entre le compositeur et le peintre. Dans le cas de la musique, l'ordre dans lequel le compositeur place les note n'importe pas. À la fin, elles sont toujours jouées de la même manière. L'ajout des notes est commutatif. Dans le cas du peintre, ce n'est pas nécessairement vrai. S'il dessine avec un fusain, bien malin est celui qui parvient à dire quel trait recouvre quel autre. Mais s'il utilise de la peinture, alors chaque coup de pinceau recouvre ceux qui le précédaient. Dans le dessin, même en deux dimensions, apparaît déjà une troisième dimension, à savoir un ordre partiel entre les coups de pinceaux qui se recouvrent.

La musique peut être vue comme un ensemble de sons qui évolue au cours du temps. Si l'on remplace les notes que place le compositeur de musique dans le temps par des dessins, nous obtenons des animations. Autrement dit, les animations sont obtenues par combinaison de la modélisation temporelle hors-temps et de la modélisation spatiale.

1.2.3 Systèmes réactifs

Les systèmes musicaux interactifs sont des systèmes capables d'écouter de la musique et d'y appliquer des traitements en temps réel. Le modèle et le langage de programmation développés ici visent aussi à la spécification de ces systèmes. Nous traiterons aussi d'un sujet connexe pour lequel notre modèle se comporte bien : la définition de dessins en deux ou en trois dimensions et les animations qui en découlent.

Nous souhaitons définir des systèmes permettant d'augmenter la musique, que ce soit par transformation, par enrichissement, voir en produisant une animation associée. Par exemple, avec un système qui génère une séquence de dessins synchronisés avec la musique. Dans ce but, nous allons définir un langage de programmation permettant aux programmeurs de décrire efficacement ce type de procédés à une machine. À charge ensuite à cette machine de mettre ce procédé en œuvre.

Pour ce faire, le programmeur doit pouvoir spécifier la manière dont il conçoit son système. C'est-à-dire qu'il doit fournir à la machine un schéma de conception abstrait et de haut niveau. La machine a une représentation interne de ce qu'est un processus répondant à cette spécification. Cette représentation interne est extrêmement éloignée de l'abstraction humaine initiale. Elle est très concrète, et mécaniquement exécutable.

Tout l'enjeu des langages de programmation est d'être à la fois proche de la conception humaine d'un processus, tout en permettant une compilation ou une interprétation efficace vers la représentation machine de ce processus. Le programmeur humain va donc fournir les spécifications d'un système, et la machine, après une transformation de ces spécifications, va être en mesure de l'exécuter. Dans le cas d'un système interactif, le langage d'interaction va être le formalisme dans lequel l'humain va exprimer un scénario, et ce scénario sera automatiquement transformé – compilé – dans une forme exécutable par la machine.

Pour un modèle abstrait donné, nous cherchons une manière adéquate de le manipuler. Plutôt que de manipuler ses abstractions à travers un langage de programmation généraliste, nous pouvons définir un langage spécifique dont la syntaxe et la sémantique seront dédiées à la manipulation de notre modèle. Pour ce faire, le recours aux DSL (Domain Specific Language) [17], Langage Applicatif en français, est très précieux. Notre modèle étant algébrique, les valeurs construites en suivant cette abstraction vont naturellement ressembler à des expressions algébriques. Nous allons donc ajouter une surcouche de programmation qui permettra de refléter la sémantique de notre modèle plutôt que la notation traditionnelle associée à sa structure.

1.3 Programmation fonctionnelle

Le modèle que nous allons manipuler est algébrique. L'écriture de programmes se fera via la composition d'éléments de l'algèbre. À ce titre, nous allons tirer parti de la programmation équationnelle et, plus spécifiquement, la programmation fonctionnelle. Ces dernières permettrons au programmeur de spécifier son problème par un ensemble d'équations. La sémantique prêtée à ce modèle permet la définition et la transformation de nos abstractions à travers un système d'équations.

Nous allons donc utiliser Haskell pour illustrer ce travail de thèse. Ce langage, en tant que langage fonctionnel, est basé sur l'évaluation d'équations. Le langage Haskell se caractérise par un certain nombre de propriétés que nous allons maintenant passer en revue.

Paresse. La première de ces propriété est la paresse. Il s'agit d'une propriété de l'évaluation des termes des programmes. La majorité des langages de programmation courants a un mode d'évaluation strict, c'est-à-dire que les termes sont évalués dès qu'ils sont évaluables.

Avec la paresse, c'est tout l'inverse, les termes sont évalués le plus tard possible. C'est-à-dire qu'ils ne sont pas évalués avant d'être devenus strictement nécessaires pour avancer dans le calcul.

La paresse rend cette famille de langages particulièrement adaptée pour manipuler les structures de données infinies. Nous allons manipuler des flux de données infinis pour modéliser par exemple, les évolutions de valeurs de nos entrées au cours d'un temps non borné.

Le comportement d'un programme écrit dans un langage paresseux est par contre difficile à prévoir. En effet, imaginons un programme qui contiendrait un terme dont l'évaluation dépend de la lecture d'un fichier. Pour peu qu'il y ait des altérations de ce fichier au cours de l'exécution du programme, la sémantique du programme dépend du moment où ce terme est évalué.

La paresse rend difficile la maîtrise du comportement dynamique des programmes. Elle nécessite donc la propriété de pureté : si l'on ne peut pas prévoir quand sera évaluée une expression, il faut que la valeur associée à une expression ne dépende ni du moment ni de l'ordre de son évaluation. L'évaluation du langage doit être confluente. Cette propriété est nécessaire pour garantir le déterministe de toute évaluation.

Pureté. L'autre caractéristique d'Haskell qui découle de la précédente est la pureté [33]. Elle peut être caractérisée par l'indépendance de la sémantique avec la stratégie d'évaluation. Chaque objet manipulé est associé à une unique valeur qui ne change pas dans le temps.

Nous pouvons aussi la définir de manière négative : la pureté est l'absence d'effets de bord. C'est-à-dire que l'évaluation d'une fonction se fait sans aucun changement d'état. Dans un langage fonctionnel pur, l'ordre et le contexte de l'évaluation d'un programme est donc sans aucune incidence sur la sémantique.

Il en résulte que l'étude de la sémantique de deux fonctions permet à elle seule de prédire la sémantique de leur composition. C'est la propriété de compositionnalité. Elle rend le développement particulièrement aisé. Les propriétés d'un programme complexe se déduisent des propriétés de ses composants. Les langages autorisant des effets de bord ne vérifient pas cette propriété, car la sémantique d'un programme dépend alors de son contexte d'exécution. Cet aspect rend les langages fonctionnels purs particulièrement adaptés à l'implémentation de structures algébriques.

1.4 Structure de ce document

Dans le chapitre 2, nous allons parcourir les outils et concepts mathématiques dont ce travail de thèse fait usage. Il s'agit principalement des monoïdes inversifs, leurs morphismes, et des relations d'ordre induites. Nous y présentons aussi quelques constructions permettant de créer des monoïdes inversifs. Des constructions qui jouent un rôle centrale dans le travail présenté ici.

Dans le chapitre 3, nous allons présenter le modèle de la programmation tuilée. Progressivement, nous allons décrire les générateurs et les opérateurs de ce modèle tout en donnant une intuition de la sémantique ainsi qu'une sémantique formelle.

Dans le chapitre 4, nous allons nous intéresser à la programmation temporelle hors temps. Il s'agit de programmation temporelle dans le sens où le programmeur va créer une description symbolique d'un processus temporisé (musique, vidéo, etc.).

Dans le chapitre 5, nous allons explorer la programmation temporelle dans le temps. Les programmes que nous allons exécuter vont devoir produire leur sortie au fur et à mesure qu'ils reçoivent leur entrée. Nous allons décrire un moteur d'exécution qui produira un flux de données de manière réactive.

Dans le chapitre 6 nous allons aborder la programmation spatiale, en particulier le dessin. Il s'agira d'une autre instanciation du modèle présenté dans le chapitre 3. Le point de départ de notre étude du langage de programmation spatiale proposé ici sera sémantique de la tortue du langage LOGO, un langage de programmation qui a été proposé au milieu des années 1960 [29]. Dans ce langage, il s'agit de programmer une tortue qui porte un stylo. Nous allons étudier la sémantique de ce langage, l'abstraire, et l'implémenter à nouveau dans le modèle que nous développons.

En résumé, dans ce travail de thèse nous nous intéressons particulièrement à l'implémentation de langages de programmation d'application multimédia temporisée permettant de produire de la musique, des dessins et des animations.

2

Structures mathématiques

Dans cette partie nous allons faire un parcours des structures et concepts mathématiques dont nous allons nous servir tout au long de ce document. Nous en profitons en même temps pour introduire la syntaxe et la sémantique du langage Haskell dont nous nous servirons aussi, en donnant des implémentations de ces structures.

2.1 Ordres et ordres partiels

Définition 2.1.1 (Ordre partiel)**.** Un ensemble ordonné est un ensemble E muni d'une relation binaire $\leq$, appelé ordre partiel, qui satisfait les propriétés suivantes :

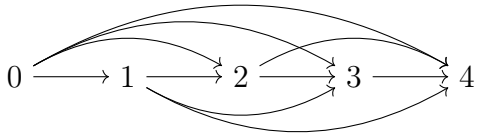
(1) $x \leq x$ (réflexivité),

(2) si $x \leq y$ et $y \leq x$ alors $x = y$ (anti-symétrie),

(3) si $x \leq y$ et $y \leq z$ alors $x \leq z$ (transitivité).

pour tout x,y, et $z \in E$.

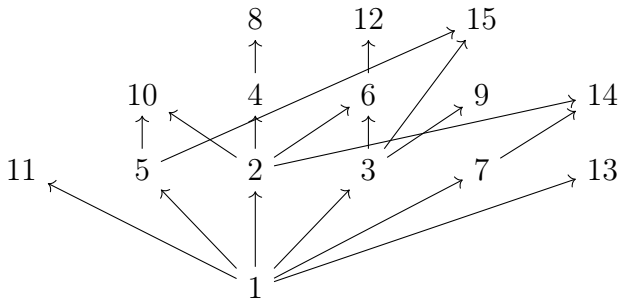
Exemple (L'ordre sur $\mathbb{N}$)**.** Le graphe suivant représente la relation $\leq$ sur les entiers de 0 à 4 :



Deux entiers a et b sont reliés par un arc $a \to b$ si et seulement si nous avons $a \leq b$. Ici, nous avons seulement représenter 5 entiers, et nous

avons déjà 10 arcs. Nous pourrons simplifier ce graphe en ne mettant les arcs qu'entre chaque paire de nombres consécutifs, et par transitivité de la relation $\leq$, nous considérerons dans le graphe la fermeture transitive des arcs, c'est-à-dire les chemins.

Exemple (La relation de divisibilité)**.** La relation de divisibilité sur les entiers naturels est un ordre partiel. Le graphe de la relation est représenté ici pour les nombres entre 1 et 15 :



Nous n'avons là encore pas dessiné dans ce graphe les arcs correspondant à la transitivité. Ainsi, si $a_1 \to a_2 \to \ldots \to a_n$ est un chemin de ce graphe, alors nous avons aussi $a_1 \leq a_n$. Et donc, a_1 divise a_n. Ce type de représentation des ensembles ordonnés est connu sous le nom de diagramme de Hasse.

Dans le langage Haskell, la classe des ordres partiels, c'est-à-dire des types de données munie d'une relation d'ordre est décrite par :

```
class PartialOrd a where
  (<=) :: a -> a -> Bool
  -- ∀ x. x <= x   (1)
  -- ∀ x y. x <= y ∧ y <= x ⟹ x = y    (2)
  -- ∀ x y z. x <= y ∧ y <= z ⟹ x <= z   (3)
```

Ici, nous utilisons une fonction de type $a \to a \to$ Bool pour décrire une relation R . C'est une fonction qui prend deux paramètres et qui retourne vrai ou faux suivant si les deux paramètres sont en relation par R ou pas.

Remarquons que le mot `class` n'a pas le même sens qu'en programmation orientée objet. Sa sémantique est plus proche de celle qu'on lui prête en mathématiques, on aurait pu parler de famille. Il introduit un ensemble de types qui ont certaines propriétés. Le mot `PartialOrd` est le nom donné à cet ensemble, et `a` est le nom prêté à un membre de cet ensemble. La ligne `Class PartialOrd a` déclare l'existence d'un ensemble de types pour

lesquels certaines propriétés sont respectées, et certaines constantes et fonctions définies. Le mot clé `where` introduit ces valeurs.

Pour qu'un type `a` soit instance de cette classe, il faut définir un opérateur infixe `<=` qui prend deux éléments de `a`, et retourne un booléen.

Le type d'une valeur est déclaré par l'opérateur `::`. C'est un opérateur infixe avec la valeur à gauche, et son type type à droite. Ainsi, une transcription en Haskell de l'expression $x \in X$ devient `x :: X`. La flèche `->` permet de construire les types des fonctions, avec le domaine à gauche, et le co-domaine à droite.

Il faut noter que le langage Haskell n'est pas assez expressif pour spécifier les propriétés logiques qui lient les paramètres d'une fonction à son résultat. Ces propriétés sont donc mises en commentaire à destination du programmeur. Il pourra supposer que ces propriétés sont vérifiées dès qu'il utilisera une instance de `PartialOrd`. S'il souhaite en déclarer une instance, ces propriétés doivent être respectées.

Exemple. Nous pouvons implémenter en Haskell l'ordre de divisibilité présenté dans l'exemple plus haut grâce au code suivant :

```
instance PartialOrd Integer where
  a <= b = 0 == (b `mod` a)
```

Cependant, il faut être prudent car cette relation d'ordre n'est pas celle qu'un programmeur s'attend à avoir lorsqu'il utilise l'opérateur `<=` sur des entiers. Si nous voulons définir une autre instance de `PartialOrd`, nous n'avons d'autre choix que de créer un nouveau type isomorphe à `Integer`, le type des entiers, et définir l'instance à `PartialOrd` que nous souhaitons.

Remarque (Curryfication)**.** Généralement, nous utilisons une forme curryfiée[1] des fonctions. C'est-à-dire qu'au lieu de définir des fonctions qui prennent un n-uplet, comme nous le ferions dans un langage comme `C`, chaque fonction ne prend qu'un paramètre, et retourne une fonction qui prend en paramètre le suivant, jusqu'à ce qu'il n'y en ai plus.

Par exemple, considérons une fonction `f` de type `a -> b -> c`. Le type de `f` peut être parenthésé `a -> (b -> c)`. Si on applique la fonction `f` à un argument `x :: a`, on obtient une fonction `f x :: b -> c`. Cette fonction peut ensuite être appliquée à un second paramètre `y :: b` pour produire la valeur `f x y :: c`. Ainsi, l'appel de fonction `f x y` est implicitement parenthésé `(f x) y`.

Les fonctions curryfiées sont isomorphes aux versions non curryfiées. En effet, nous pouvons passer de l'une à l'autre. Par exemple, pour les fonctions à deux arguments :

1. Le nom « curryfié » a été choisi en l'honneur de Haskell Curry.

```haskell
uncurry :: (a -> b -> c) -> ((a,b) -> c)
uncurry f = \ (x,y) -> f x y

curry :: ((a,b) -> c) -> (a -> b -> c)
curry f = \ x y -> f(x,y)
```

où `(a,b)` désigne le type des paires d'éléments de `a` et de `b`. Le `\` introduit les fonctions anonymes, cet opérateur a été choisi pour sa ressemblance avec le λ du λ-calcul.

Définition 2.1.2 (Ordre total)**.** Un ensemble totalement ordonné est un ensemble partiellement ordonné E qui vérifie en outre :

(4) $x \leq y$ ou $y \leq x$

pour tout $x, y \in E$

En Haskell, la classe des ensembles totalement ordonnés peut être codé ainsi :

```haskell
class PartialOrd a => Ord a
  -- ∀ x y. x <= y ∨ y <= x (4)
```

Cette classe ne déclare aucune constante ou fonction supplémentaire dépendante du type `a`. Elle permet juste de déclarer une propriété qui doit être satisfaite.

Dans la bibliothèque standard Haskell la notion d'ordre total est implémentée par la classe `Ord`. Elle est définie comme suit :

```haskell
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  compare a b = if a == b
                then EQ
                else if a <= b
                     then LT
                     else GT
  (<) :: a -> a -> Bool
  a < b = a <= && a != b
  (<=) :: a -> a -> Bool
  a <= b = compare a b == LT || a == b
  (>) :: a -> a -> Bool
  a > b = b < a
  (>=) :: a -> a -> Bool
```

```
  a >= b = b <= a
  max :: a -> a -> a
  max a b if a >= b then a else b
  min :: a -> a -> a
  min a b = if a >= b then b else a
  {-# MINIMAL (<=) | compare #-}
```

où `Ordering` est défini par :

```
data Ordering = LT | EQ | GT
```

La contrainte `Eq a =>` indique que pour qu'un type appartienne à la classe `Ord`, il doit déjà être équipé d'une notion d'égalité. Il s'agit de la classe `Eq` qui définit l'opérateur d'égalité `==`.

La classe `Ord` déclare bien plus de fonctions et d'opérateurs que le simple `<=`. Ces fonctions sont redondantes. Elles bénéficient toutes d'une implémentation par défaut utilisant les autres. En programmation orientée objet, nous les appellerions des méthodes abstraites. Il suffit de définir soit `<=` soit `compare` pour instancier la classe entièrement. Ces fonctions sont néanmoins dans la classe car cela autorise à les réimplémenter si nécessaire de manière plus performante.

Remarque (Ordre total et partiel)**.** Cette classe définit effectivement un ordre total car la fonction `compare` de type `a -> a -> Ordering` est elle-même totale. La bibliothèque standard Haskell ne propose pas de classe pour les ordres partiels.

Définition 2.1.3 (Majorant et minorant)**.** Soit $\langle E, \leq \rangle$ un ensemble ordonné et X une partie de E.

Un élément m est un *minorant* (resp. *majorant*) de $X \subseteq E$, ce que l'on note $m \leq X$ (resp. $X \leq m$) lorsque $m \leq x$ (resp. $x \leq m$) pour tout $x \in X$.

Définition 2.1.4 (Borne supérieure et borne inférieur)**.** Soit $\langle E, \leq \rangle$ un ensemble ordonné et X une partie de E.

Un élément m est la *borne inférieur* (resp. *supérieure*), aussi appelé *infimum* (resp. *supremum*) de X, ce que l'on note $m = \bigwedge X$ (resp. $m = \bigvee X$) lorsque $m \leq X$ (resp. $X \leq m$) et pour tout $m' \in E$, si $m' \leq X$ alors $m \leq m'$ (resp. $m' \leq m$). C'est-à-dire que m est le plus grand des minorants (resp. le plus petit des majorants) de X.

2.2 Structures associatives

Nous allons voir maintenant les quelques structures algébriques dont nous feront usage au cours de cette thèse.

2.2.1 Semi-groupes et monoïdes

Nous allons commencer par une des structures les plus simples qui soient, les semi-groupes.

Définition 2.2.1 (Semi-groupe)**.** Un *semi-groupe* est un ensemble S muni d'un opérateur binaire $\cdot$ de type $S \to S \to S$ qui est associatif, c'est-à-dire qui satisfait la propriété :

(5) $(x \cdot y) \cdot z = x \cdot (y \cdot z)$

pour tout $x, y, z \in S$

En Haskell, la classe des semi-groupes peut être spécifiée de la façon suivante[2] :

```haskell
class Semigroup s where
  (<>) :: s -> s -> s
  -- ∀ x y z. (x <> y) <> z == x <> (y <> z) (5)
```

Définition 2.2.2 (Monoïde)**.** Un *monoïde* est un semi-groupe S dans lequel il existe un élément e qui vérifie :

(6) $e \cdot s = s$

(7) $s \cdot e = s$

pour tout $s \in S$. On dit que l'élément e est un neutre à gauche et à droite.

En Haskell, cette définition devient :

```haskell
class Semigroup m => Monoid m where
  mempty :: m
  -- ∀ x. x <> mempty == x (6)
  -- ∀ x. mempty <> x == x (7)
```

Dès qu'il n'y a pas ambiguïté, nous identifions une structure avec l'ensemble de ses éléments, parfois appelé son support ou univers. Par exemple le monoïde $\langle M, +, 0 \rangle$ sera identifié par M.

2. En Haskell l'opérateur `.` est utilisé pour la composition fonctionnelle.

Exemple (Liste)**.** Les listes sont un exemple de monoïde. En Haskell, les listes peuvent être définies par le type inductif suivant :

```
data List a = Cons a (List a)
            | Nil
```

Ainsi, une liste d'élément de type `a` est soit la liste vide `Nil` soit le constructeur infixe `Cons` qui prend en paramètre un élément de type `a` et une liste d'élément de type `a`. Le constructeur `Cons` permet l'ajout en tête de liste, c'est-à-dire que la liste `[1,2,3]` est représentée par la valeur `Cons 1 (Cons 2 (Cons 3 Nil))`. Ce type a une structure de monoïde dont le neutre est la liste vide `Nil` et dont le l'opérateur de composition est la concaténation de liste. En Haskell, on peut ainsi définir les instances :

```
instance Semigroup (List a) where
  Nil <> l = l
  (Cons x xs) <> l = Cons x (xs <> l)

instance Monoid (List a) where
  mempty = Nil
```

Dans la partie à gauche du `=` dans la définition de `<>`, nous voyons un exemple de la reconnaissance de motif avec les constructeurs `Nil` et `Cons`, « pattern matching » en anglais. Elle se fait sur le premier paramètre. Sur la première ligne, nous voyons le cas où ce paramètre correspond à `Nil`. Sur la seconde, `Cons x xs`, correspond au cas où ce paramètre est décomposé en un premier élément `x` suivit d'une liste `xs`. Cette syntaxe permet tout à la fois de faire une analyse de cas sur les constructeurs utilisés dans les paramètre et de donner des noms aux paramètres de ces constructeurs.

Le type `List a` est un codage du monoïde libre engendré par `a`. En Haskell, ce type est primitif, c'est-à-dire qu'il est prédéfini. Le constructeur `Cons` est noté `::`, et le constructeur `Nil` est noté `[]`.

Remarque (Conversion vers un monoïde)**.** Tout semi-groupe sans neutre peut être converti en monoïde en ajoutant un élément neutre. Soit $\langle S, \cdot_s \rangle$ un semi-groupe, nous pouvons définir un opérateur $M = S \cup \{1_m\}$, et définir $\cdot_m$ ainsi :

$$a \cdot_m b = \begin{cases} a \cdot_s b & \text{si } a \in S \text{ et } b \in S \\ a & \text{si } a \in S \text{ et } b = 1_m \\ b & \text{si } b \in S \text{ et } a = 1_m \\ 1_m & \text{si } a = b = 1_m \end{cases}$$

En Haskell, cela peut être fait en utilisant le type `Maybe a`, définit ainsi :

```
data Maybe a = Just a
             | Nothing
```

Le constructeur `Just` permet d'encapsuler un élément du type `a`, et `Nothing` est la nouvelle valeur qui est introduite dans `Maybe a` par rapport à `a`. Les instances de semi-groupe et monoïde mentionnées ci-dessus se définissent alors :

```
instance Semigroup a => Semigroup (Maybe a) where
  (Just a) <> (Just b) = Just (a <> b)
  Nothing  <> (Just b) = Just b
  (Just a) <> Nothing  = Just a
  Nothing  <> Nothing  = Nothing

instance Semigroup a => Monoid (Maybe a) where
  mempty = Nothing
```

Pour le cas où `a` est un monoïde, il y a une seconde instance possible (il y a alors deux neutres) :

```
instance Monoid a => Monoid (Maybe a) where
  mempty = mempty
```

Définition 2.2.3 (Commutativité)**.** Un semi-groupe S est commutatif si :

(8) $x \cdot y = y \cdot x$

pour tout x et y de S.

Exemple. Soit E un ensemble. L'ensemble $\mathcal{P}(E)$ des parties E a une structure de monoïde commutatif avec l'ensemble vide comme neutre et $\cup$ comme composition. Cet opérateur est à la fois associatif et commutatif.

2.2.2 Structures qui ne sont *pas* des semi-groupes

Les semi-groupes et les monoïdes sont des structures tellement « naturelles » qu'il peut être délicat de trouver des structures qui n'en sont pas.

Exemple (Pierre-Papier-Ciseaux)**.** Comme exemple de structures qui ne sont pas un semi-groupe, nous avons la mécanique du jeu pierre-papier-ciseaux.

Si nous notons $J = \{P_i, P_a, C\}$ l'ensemble des coups possible du jeu (représentant respectivement pierre, papier, et ciseaux) et $\odot$ l'opérateur $(J \times J) \to J$ qui donne le coup gagnant. Voici la table de cet opérateur :

$\odot$	P_i	P_a	C
P_i	P_i	P_a	P_i
P_a	P_a	P_a	C
C	P_i	C	C

Comme nous pouvons le constater, cet opérateur n'est pas associatif. Nous avons en effet :

$$(P_a \odot C) \odot P_i = C \odot P_i = P_i$$
$$P_a \odot (C \odot P_i) = P_a \odot P_i = P_a$$

Remarquons en revanche que cette structure est commutative.

Exemple (Arbres binaires)**.** Les arbres binaires munis d'une composition bien choisie échouent aussi à définir un semi-groupe, car la propriété d'associativité n'est pas vérifiée. Nous pouvons implémenter ces arbres en Haskell grâce au type suivant :

```
data BinTree a =
    Leaf a
  | Bin (BinTree a) (BinTree a)
```

Nous pouvons munir ce type de la composition suivante :

```
a <> b = Bin a b
```

Cette composition binaire n'est ni associative, ni commutative. Il serait illégal de la mettre dans une instance de la classe `Semigroup`.

Exemple (Opérateur « moyenne »)**.** De même, sur les rationnels, nous pouvons considérer également $\langle \mathbb{Q}, \star \rangle$ où $x \star y = \frac{x+y}{2}$, avec $\frac{a}{b}$ la division sur $\mathbb{Q}$. Ce n'est pas un semi-groupe car l'opérateur $\star$ n'est pas associatif.

2.2.3 Semi-treillis

Il y a différentes manières de voir un semi-treillis : soit comme un ensemble ordonné, soit comme une structure algébrique.

Définition 2.2.4 (Semi-treillis vu comme semi-groupes)**.** Vu comme une structure algébrique, un semi-treillis $\langle T, \cdot \rangle$ est un semi-groupe qui est en plus commutatif et idempotent :

(9) $x \cdot y = y \cdot x$ (commutativité),

(10) $x \cdot x = x$ (idempotence).

pour tout $x, y \in T$.

Définition 2.2.5 (Semi-treillis vu comme un ensemble partiellement ordonné)**.** Vu comme un ordre, un inf-semi-treillis est un ensemble partiellement ordonné $\langle T, \leq \rangle$ où pour tout $x, y \in T$, l'ensemble $\{x, y\}$ a une borne inférieure. Cette borne inférieure est notée $x \wedge y$.

Lemme 2.2.6. *Il y a une « bijection » entre les semi-groupes idempotents et les inf-semi-treillis.*

Démonstration. Tout d'abord, montrons l'implication 2.2.4 $\Rightarrow$ 2.2.5. Soit $\langle T, \cdot \rangle$ un semi-treillis vérifiant la définition 2.2.4. Posons $x \leq y$ lorsque $x \cdot y = x$. Vérifions que $\leq$ est bien une relation d'ordre.

- Réflexivité : par la propriété d'idempotence $x \cdot x = x$, donc $x \leq x$
- Transitivité : supposons $x \leq y$ et $y \leq z$, donc (1) $x \cdot y = x$ et (2) $y \cdot z = y$. Il en résulte par (1) et (2) que $x \cdot (y \cdot z) = x$, et donc par associativité et (1) que $x \cdot z = x$, donc $x \leq z$.
- Anti-symétrie : supposons $x \leq y$ et $y \leq x$, c'est-à-dire $x \cdot y = x$ et $y \cdot x = y$, par commutativité, nous avons donc $x \cdot y = y \cdot x = x = y$.

Montrons que $x \cdot y$ est l'infimum de $\{x, y\}$:

— $x \cdot y \leq x$ est vrai car :

$$\begin{aligned} (x \cdot y) \cdot x &= x \cdot (y \cdot x) && \text{(assoc.)} \\ &= x \cdot (x \cdot y) && \text{(commu.)} \\ &= (x \cdot x) \cdot y && \text{(assoc.)} \\ &= x \cdot y && \text{(idem.)} \end{aligned}$$

$(x \cdot y) \cdot x = x \cdot y$ est précisément la définition de $x \cdot y \leq x$.

— La preuve de $x \cdot y \leq y$ est symétrique au cas ci-dessus.

— il s'agit de la plus grande borne inférieure car si $z \leq x$ et $z \leq y$, alors, par définition, $z \cdot x = z$ et $z \cdot y = z$, et donc, par combinaison, $(z \cdot x) \cdot y = z$, finalement, $z \cdot (x \cdot y) = z$, c'est-à-dire que $z \leq x \cdot y$.

Maintenant, montrons que : 2.2.5 $\Rightarrow$ 2.2.4. Soit un ensemble E muni d'une relation $\leq$ satisfaisant la définition 2.2.5. Considérons l'opérateur de composition $\wedge$ défini par $x \wedge y = \bigwedge\{x, y\}$ (l'infimum de $\{x, y\}$). Montrons que $\langle E, \wedge \rangle$ est un semi-groupe commutatif et idempotent.

- Cet opérateur est associatif, en effet, nous avons :

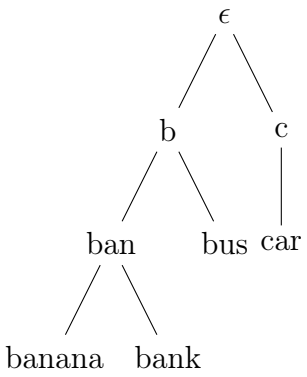
$$\bigwedge\{(\bigwedge\{x, y\}), z\} = \bigwedge\{x, y, z\} = \bigwedge\{x, \bigwedge\{y, z\}\}$$

et donc $(x \wedge y) \wedge z = x \wedge (y \wedge z)$.

- Il est idempotent, trivialement, $\bigwedge\{x, x\} = x$.
- Il est commutatif tout aussi trivialement $\bigwedge\{x, y\} = \bigwedge\{y, x\}$.

□

Exemple (Préfixe de mots)**.** Les mots ordonnés par la relation « être préfixe de ». Dans ce semi-treillis, $x \wedge y$ calcule le plus long mot qui est préfixe à la fois de x et de y. Par exemple : banana $\wedge$ bus = b.



Remarque (Interprétation sémantique)**.** Nous avons $x \cdot y \leq x$ et $x \cdot y \leq y$. La multiplication à gauche ou à droite induit une sorte de décroissance irréversible. Cette propriété sera utilisée pour modéliser l'entropie de certaines combinaisons de médias.

2.3 Structure dotée d'inverses

Après les structures associatives et commutatives, nous allons aborder les structures dotées d'une notion d'inverse.

2.3.1 Groupes

La notion la plus connue d'inverse provient de la théorie des groupes. Elle est rappelée ci-dessous.

Définition 2.3.1 (Inverse de groupe)**.** Soit $\langle M, \cdot, e\rangle$ un monoïde et $x \in M$ un élément. L'élément $y \in M$ est inverse (de groupe) de x lorsque :

$$x \cdot y = e = y \cdot x$$

Définition 2.3.2 (Groupe)**.** Un groupe $\langle G, \cdot, e\rangle$ est un monoïde tel que pour chaque élément $x \in G$ il existe un élément $y \in G$ qui vérifie :

(11) $x \cdot y = e$

(12) $y \cdot x = e$

Cet élément y s'appelle l'inverse, et est noté x^{-1}.

Lemme 2.3.3 (Unicité de l'inverse)**.** *Dans un groupe, l'inverse d'un élément est unique.*

Démonstration. Soient deux inverses y et y' à un élément x. Nous avons donc $x \cdot y = e$ et $x \cdot y' = e$.

$$\begin{aligned} x \cdot y &= e \\ (y' \cdot x) \cdot y &= y' && (x\cdot) \\ e \cdot y &= y' && (\textit{prop.}\ (12)) \\ y &= y' \end{aligned}$$

Donc les inverses sont uniques. □

Lemme 2.3.4. *Dans un groupe G, on a :*

$$(x^{-1})^{-1} = x$$

pour tout $x \in G$.

Démonstration.

$$\begin{aligned} x^{-1} \cdot (x^{-1})^{-1} &= e && (\text{prop } (11)) \\ x \cdot x^{-1} \cdot (x^{-1})^{-1} &= x \cdot e && (x\cdot) \\ (x^{-1})^{-1} &= x && (\text{prop } (11)) \end{aligned}$$

□

Remarque (Notation additive)**.** On peut aussi utiliser la notation additive pour les groupes. L'opérateur de composition interne est alors $+$, le neutre 0, et l'inverse x est noté $-x$. On note aussi $x-y = x+(-y)$. Attention, dans un semi-groupe inversif, non commutatif on aura : $-(x+y) = -y-x$ (qui peut être différent de $-x-y$, on ne fait pas l'hypothèse de la commutativité).

Nous allons souvent utiliser la notation multiplicative pour l'inverse, par abus de notation.

2.3.2 Semi-groupes inversifs

La structure de monoïde inversif est à mi-chemin entre la structure de groupe et la structure de monoïde. Dans un groupe, nous avons un opérateur qui permet à partir de chaque élément un « anti-élément ». Par exemple, $3 + (-3) = 0$. Nous appelons cet élément l'opposé ou l'inverse suivant si nous utilisons une terminologie additive ou multiplicative.

Dans les monoïdes inversifs, tout comme dans les groupes, chaque élément de l'ensemble a un inverse unique, mais dans un monoïde inversif, ces inverses sont trop « faibles »pour vérifier $x - x = 0$.

Définition 2.3.5 (Inverse de semi-groupe)**.** Soit $\langle S, \cdot \rangle$ un semi-groupe et $x \in S$. Un élément $y \in S$ est inverse (de semi-groupe) de x lorsque :

(13) $x \cdot y \cdot x = x$

(14) $y \cdot x \cdot y = y$

Définition 2.3.6 (Semi-groupe inversif)**.** Un *semi-groupe inversif* est une structure de semi-groupe $\langle S, \cdot \rangle$ où chaque élément x admet un unique inverse de semi-groupe. Cet inverse est noté x^{-1}.

Exemple (Promenade dans la neige)**.** Les déplacements d'une personne dans la neige et les traces qu'elle laisse sont un exemple de semi-groupe inversif. Cet exemple sera généralisé dans le chapitre 6.

Les éléments de ce semi-groupe inversifs modélisent les déplacements d'une personne dans la neige ainsi que les traces qu'elle laisse. La composition des déplacements est défini comme la mise en séquence des déplacements. L'inverse d'un déplacement est ce même déplacement, mais en marche arrière. C'est-à-dire partant de la fin et revenant au début.

De cette manière, si une personne fait un déplacement d'un point a à un point b, puis le déplacement inverse b vers a en prenant soin de remettre ses pieds dans les traces laissées à l'aller, elle reviendra à sa position initiale. Cependant, son déplacement laisse des traces. Ces traces font que la notion d'inverse de semi-groupe est « plus faible »que celle des groupes.

Un déplacement suivi de son inverse n'est pas équivalent à aucun déplacement. Il laissera toujours la trace du premier déplacement.

Lemme 2.3.7 (Semi-treillis)**.** *Tout semi-treillis est un semi-groupe inversif.*

Démonstration. En effet, si nous définissons l'inverse trivial $x^{-1} = x$, nous pouvons vérifier (par idempotence) : $x^{-1} \cdot x \cdot x^{-1} = x = x \cdot x^{-1} \cdot x$.
Cet inverse est l'unique inverse, en effet, supposons qu'il existe un élément y vérifiant : $x \cdot y \cdot x = x$ et $y \cdot x \cdot y = y$.

Par idempotence et commutativité, nous pouvons réécrire ces équations $x \cdot y = x$ et $y \cdot x = y$, c'est-à-dire $x = y$. Nous en concluons que x est l'unique inverse de x. □

Définition 2.3.8 (Monoïde inversif)**.** Un monoïde inversif est un semi-groupe inversif avec un élément neutre.

En Haskell, cette structure peut être formellement spécifiée ainsi. Il s'agit d'un semi-groupe auquel on rajoute une notion d'inverse :

```
class Semigroup s => InverseSemigroup s where
  inverse :: s -> s
  -- ∀ x. x <> inverse x <> x == x                    (13)
  -- ∀ x. inverse x <> x <> inverse x == inverse x    (14)
  -- Pour x fixé, inverse x est l'unique valeur vérifiant
  -- les deux axiomes précédants
```

Définition 2.3.9 (Idempotent)**.** Soit S un semi-groupe, un élément $x \in S$ est idempotent lorsque $x \cdot x = x$. On note alors $E(S)$ l'ensemble des idempotents de S.

Propriété 2.3.10. *Soit $\langle S, \cdot \rangle$ un semi-groupe inversif. Alors,*

$$E(S) = \{x \cdot x^{-1} | x \in S\}$$

De plus, pour tout $x, y \in E(S)$, nous avons :

(15) $x \cdot y \in E(S)$

(16) $x^{-1} = x$

(17) $x \cdot y = y \cdot x$

Autrement dit la partie $E(S)$ est un sous-semi-groupe idempotent et commutatif [26]. C'est-à-dire un semi-treillis (Lemme 2.2.6).

Remarque (Équivalence d'axiomes)**.** Dans la définition d'un semi-groupe inversif l'unicité de l'inverse est essentielle. Cependant, elle peut être remplacée par la commutation des idempotents. En effet, cette propriété (17) est équivalente à l'unicité de l'inverse [26].

Exemple (Bijections partielles)**.** Soit un ensemble E, et F l'ensemble $I(E)$ des bijections partielles de E dans E. Nous pouvons vérifier que $\langle F, \circ \rangle$ où $\circ$ est la composition fonctionnelle [3] est un monoïde inversif. Les idempotents sont l'ensemble des fonctions qui sont les restrictions du domaine et co-domaine de la fonction identité.

3. $(f_1 \circ f_2)\ x = f_1(f_2\ x)$ si $y = f_2\ x$ et $z = f_1\ y$ y sont tous deux définis.

Théorème 2.3.11 (Théorème de Wagner-Preston[26])**.** *Tout semi-groupe inversif est sous-semi-groupe d'un semi-groupe de bijection partielles.*

2.3.3 Ordre naturel sur semi-groupe inversif

Sur tous les semi-groupes inversifs, nous pouvons définir la relation d'ordre suivante [30].

Définition 2.3.12 (Ordre naturel)**.** Soit $\langle S, \cdot \rangle$ un semi-groupe inversif, soient $x, y \in S$, on pose $x \leq y$ lorsque l'une des quatre propriétés (équivalentes) suivantes est satisfaite :

(18) $x = x \cdot x^{-1} \cdot y$

(19) $x = y \cdot x^{-1} \cdot x$

(20) $x = e \cdot y$ pour $e \in E(S)$

(21) $x = y \cdot e$ pour $e \in E(S)$

Il s'agit de l'ordre *naturel*, au sens où cet ordre est défini de manière uniforme sur tous les semi-groupes inversifs. Intuitivement, cela signifie que x est plus petit que y si y peut être complété par un idempotent pour être égal à x. Intuitivement, que x contient plus de choses que y. Le sens de la relation d'ordre est très contre-intuitif, on peut l'interpréter par « x est plus petit que y dans le sens où x est plus dense, il coule, alors que y flotte ».

Lemme 2.3.13. *Dans un semi-groupe inversif :*

1. *Les propriétés (18), (19), (20) et (21) sont équivalentes.*
2. *La relation $\leq$ est bien une relation d'ordre.*

Démonstration. Montrons qu'il s'agit d'une relation d'ordre lorsque définit par (21), c'est-à-dire une relation transitive, réflexives, et antisymétrique. Considérons un semi-groupe inversif S et trois éléments x, y et z appartenant à S.

La réflexivité est triviale, en effet $x \leq x$ est équivalent au fait qu'il existe un élément y tel que $x = x \cdot x^{-1} \cdot y$, $y = x$ satisfait cette équation d'après l'axiome des semi-groupes inversifs.

La transitivité : supposons $x \leq y$ et $y \leq z$, montrons que $x \leq z$. Des hypothèses nous pouvons déduire qu'il existe $e_{xy} \in E(S)$ et $e_{yz} \in E(S)$ vérifiant $x = y \cdot e_{xy}$ et $y = z \cdot e_{yz}$. Montrons qu'il existe $e_{xz} \in E(S)$ vérifiant : $x = z \cdot e_{xz}$. Des hypothèses nous pouvons déduire que $y \cdot e_{xy} = z \cdot e_{xz}$ puis $z \cdot e_{xz} \cdot e_{xy} = z \cdot e_{xz}$, et enfin, $e_{xz} = e_{xy} \cdot e_{xz}$.

Pour l'antisymétrie, supposons $x \leq y$ et $y \leq x$, c'est-à-dire qu'il existe deux idempotents e_1 et e_2 vérifiant $x = y \cdot e_1$ et $y = x \cdot e_2$. Il en résulte que

$x = x \cdot e_1 \cdot e_2$ et $y = e_2 \cdot e_1 \cdot y$. Posons $e = e_2 \cdot e_1 = e_1 \cdot e_2$, nous avons $x = x \cdot e$ et $y = y \cdot e$. La même valeur e permet de compléter x pour obtenir y et y pour obtenir x, nous pouvons en déduire que $x = y$.

□

Exemple (Ordre naturel des bijections partielles)**.** La figure 2.1 représente l'ordre naturel sur les bijections partielles d'un ensemble de taille 2.

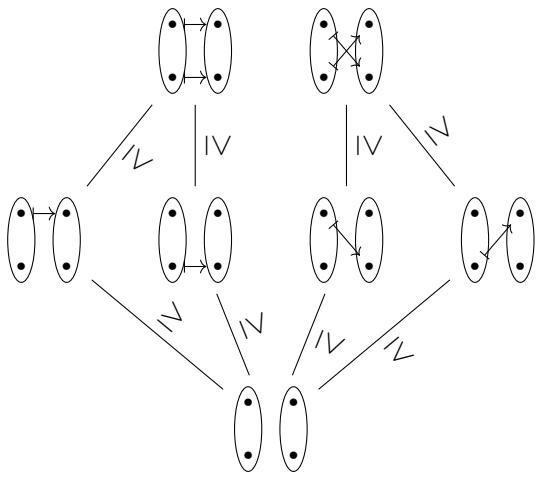


FIGURE 2.1 – Graphe de l'ordre naturel sur les bijections d'un ensemble de taille 2.

En haut, les éléments maximaux sont les bijections totales, et à gauche, nous voyons les 4 éléments idempotents, c'est-à-dire les identités partielles. Un élément est inférieur à un autre si et seulement s'il peut être « compléter » par un idempotent. Compléter ici signifie appliquer une restriction au domaine et co-domaine.

Exemple (Ordre naturel des promenades dans la neige)**.** Nous allons poursuivre avec l'exemple page 21 des promenades dans la neige. La composition de deux promenades dans la neige est la mise en séquence des deux promenades, avec leurs déplacements et les traces engendrées. L'ordre naturel sur les promenades dans la neige est équivalent à l'ordre suivant. Pour deux promenades p et q, nous avons $p \leq q$ si et seulement si :

1. Les déplacements de p et q sont égaux.
2. Si q peut être complété par un idempotent pour être égal à p.

C'est-à-dire que la trace de q est une partie de celle de p.

Les idempotents sont les traces laissées dans la neige associées à des déplacements nuls.

Dans cet ordre, les éléments maximaux sont les éléments correspondants aux déplacements les plus simples (en laissant une trace en ligne droite ou sans traces si le promeneur peut se téléporter).

Si l'on considère un élément quelconque, tous les éléments qui lui sont inférieurs ont le même déplacements, et la trace de ce maximum est une partie de la trace de chacun des éléments inférieurs.

À l'instar des ensembles totalement ordonnés pour les ensembles partiellement ordonnés, les groupes sont des monoïdes inversif avec des propriétés supplémentaires :

```
class (Monoid g, InverseSemigroup g) => Group g
   -- ∀ x. x <> inverse x == mempty
   -- ∀ x. inverse x <> x == mempty
```

2.4 Semi-groupe avec reset

Bien souvent, plus que l'existence d'un inverse, nous cherchons à pouvoir revenir à un état antérieur. Aussi, dans un souci d'efficacité de l'implémentation nous utiliserons souvent une structure un peu plus faible, celle de semi-groupe « avec reset » [5, 22, 15, 20, 19].

Définition 2.4.1 (Semi-groupe avec reset)**.** Soit S un semi-groupe, et soit $U \subseteq E(S)$. Le semi-groupe S est un semi-groupe avec reset lorsqu'il peut être équipé d'une fonction surjective et idempotente (c'est-à-dire une projection) reset de type $S \to U$ vérifiant [4] :

(22) $\text{reset}(x) \cdot x = x$

(23) $\text{reset}(x) \cdot \text{reset}(x) = \text{reset}(x)$

(24) $\text{reset}(x) \cdot \text{reset}(y) = \text{reset}(y) \cdot \text{reset}(x)$

(25) $\text{reset}(y) \cdot x = x \Rightarrow \text{reset}(y) \cdot \text{reset}(x) = \text{reset}(x)$

pour tout $x, y \in S$.

Remarque (Semi-treillis)**.** La satisfaction des axiomes (23) et (24) implique que U est un semi-treillis, c'est-à-dire un semi-groupe idempotent et commutatif.

4. L'application de la fonction reset est prioritaire sur l'opérateur $\cdot$.

Remarque (Reset et ordre)**.** L'axiome (25) implique que

$$\text{reset}(x) = \min\{e \in U | e \cdot x = x\}$$

Pour tout $x \in S$. Intuitivement, cela signifie que $\text{reset}(x)$ est l'idempotent dont la « partie irréversible » est exactement la même que celle de x.

L'intuition de cet opérateur reset appliqué à l'exemple de la personne qui marche dans la neige (exemple 2.3.2) serait la téléportation à la position initiale. C'est-à-dire que le reset d'une promenade dans la neige est la trace laissée par la promenade mais sans avoir quitté la position initiale.
Concernant l'implémentation en Haskell :

```haskell
class ResetableSemigroup s where
  reset :: s -> s
  -- ∀ x. reset x <> x == x y (22)
  -- ∀ x. reset x <> reset x == reset x (23)
  -- ∀ x y. reset x <> reset y == reset y <> reset x (24)
  -- ∀ x y. reset y <> x = x ==>
  --     reset y <> reset x == reset x (25)
```

Exemple (À partir d'un monoïde inversif)**.** Il est extrêmement aisé de construire un semi-groupe avec reset à partir d'un semi-groupe inversif. En effet, si $\langle S, \cdot \rangle$ est un semi-groupe inversif, alors on pose $U = E(S)$ et $\text{reset}(x) = xx^{-1}$. Dit autrement, le reset d'un élément est son idempotent à gauche associé dans le semi-groupe inversif.

Nous utiliserons souvent cette structure dans la suite du document. En effet, s'il est souvent difficile de donner une interprétation des inverses au sens des monoïdes inversifs, celle de l'idempotent des éléments est très naturelle. De plus, se contenter d'implémenter les idempotents permet des implémentations plus efficaces.

2.5 Morphisme

En algèbre, un morphisme est une fonction d'une instance d'une algèbre dans une autre qui en préserve la structure.

Définition 2.5.1 (Morphisme de semi-groupes)**.** Soient deux semi-groupes $\langle S, \cdot \rangle$ et $\langle T, \cdot \rangle$. Une fonction f de S dans T est un morphisme de semi-groupes si elle vérifie :

(26) $f(x \cdot y) = f(x) \cdot f(y)$

Définition 2.5.2 (Morphisme de monoïde)**.** Soient deux monoïdes quelconques $\langle M, \cdot, 1_M \rangle$ et $\langle N, \cdot, 1_N \rangle$, une fonction f de M dans N est un morphisme de monoïde si f est un morphisme de semi-groupe qui vérifie en plus :

(27) $f(1_M) = 1_N$

Lemme 2.5.3. *Soit $\langle S, \cdot \rangle$ et $\langle T, \cdot \rangle$ deux semi-groupes inversifs et soit f un morphisme de semi-groupes de S dans T. Alors : $f(x^{-1}) = (f(x))^{-1}$*

Démonstration. Par l'axiome des monoïdes inversifs, nous avons $x = x \cdot x^{-1} \cdot x$. Puisque f est un morphisme, nous avons $f(x) = f(x) \cdot f(x^{-1}) \cdot f(x)$. La valeur $f(x)$ appartient également à un monoïde inversif, donc nous avons :

$$f(x) \cdot f(x^{-1}) \cdot f(x) = f(x) = f(x) \cdot (f(x))^{-1} \cdot f(x)$$

Symétriquement, nous avons également :

$$f(x^{-1}) \cdot f(x) \cdot f(x^{-1}) = f(x^{-1}) = (f(x))^{-1} \cdot f(x) \cdot (f(x))^{-1}$$

Et par unicité de l'inverse, nous pouvons conclure que $f(x^{-1}) = (f(x))^{-1}$. □

Autrement dit, les morphismes de semi-groupes restreints aux monoïdes inversifs sont des morphismes de monoïdes inversifs.

Par extension, bien que les ensembles ordonnés ne soient pas des structures algébriques, on définit une notion de morphisme d'ordre.

Définition 2.5.4 (Morphisme d'ordre)**.** Soient un ensemble E muni d'une relation d'ordre $\leq$ et un ensemble F muni d'une relation d'ordre $\preceq$, une fonction $f : E \to F$ est un morphisme d'ordre si, pour tout x et y de E tel que $x \leq y$, on a $f(x) \preceq f(y)$.

Autrement dit, les morphismes d'ordre sont les fonctions croissantes.

2.6 Extension point-à-point

À moindre coût, nous pouvons élever une structure algébrique au niveau des fonctions, en faisant de notre structure le co-domaine de ces fonctions.

Définition 2.6.1 (Extension point-à-point)**.** À plusieurs reprises le long de ce document, nous allons utiliser la construction suivante : passer d'un ensemble S équipé d'une structure donnée à l'ensemble $E \to S$ en préservant

la structure de S. Par exemple, passer d'un semi-groupe S au semi-groupe $E \to S$.

Pour tous les opérateurs $\star$ de S, nous pouvons définir un opérateur $\star_\lambda$ sur $E \to S$ par $f_a \star_\lambda f_b = \lambda e.(f_a\ e) \star (f_b\ e)$.

Lemme 2.6.2. *Les propriétés satisfaites sur S sont héritées sur $E \to S$ (associativité, existence d'un neutre, commutativité, etc.).*

2.7 Action d'un monoïde sur un ensemble

Jusque-là, nous n'avons vu que des opérations qui se faisaient au sein d'un ensemble. Maintenant, nous allons ouvrir la porte aux opérations qui impliquent plusieurs ensembles. En particulier, l'action d'un monoïde sur un ensemble.

Définition 2.7.1 (Action à gauche d'un monoïde sur un ensemble)**.** Soit un monoïde $\langle M, \cdot, 1_M \rangle$ et un ensemble E. Une action à gauche du monoïde M sur E est un opérateur binaire $\star$ dont le domaine est $M \times E$ et le co-domaine est E, qui vérifie :

(28) $x \star (y \star e) = (x \cdot y) \star e$

(29) $1_M \star e = e$

pour tout $x, y \in S$ et pour tout $e \in E$.

Remarque (Notation de l'action)**.** L'axiome précédant justifie le fait que l'on utilise généralement le même symbole pour désigner l'opérateur de composition du monoïde et l'opérateur de son action sur l'ensemble. Les types des opérandes gauche et droite permettent de lever toute ambiguïté.

En Haskell, on peut définir la classe suivante pour implémenter les actions des monoïdes [5] sur les ensembles. Elle définit l'opérateur binaire `!*` que l'on note $\star$ plus haut. L'opérande de gauche et l'élément du monoïde, l'opérande de droite est celle de l'ensemble.

```
class Monoid m => ActionG m e where
  (!*) :: m -> e -> e
  -- ∀ x:e. mempty !* x == x
  -- ∀ a,b:m x:e. (a <> b) !* x  == a !* (b !* x)
```

Moins formellement, l'action d'un monoïde sur un ensemble permet de « modifier » les éléments de l'ensemble de façon cohérente avec la structure du

5. Nous pourrions aussi la définir en deux temps à partir de celle sur les semi-groupes.

monoïde. Ces modifications sont compositionnelles, c'est-à-dire qu'elles respectent les lois du monoïde.

Définition 2.7.2 (Action à droite d'un monoïde sur un ensemble)**.** Soit un monoïde $\langle M, \cdot, 1_M \rangle$ et un ensemble E. Une action à droite de M sur E est un opérateur $\star$ dont le domaine est $M \times E$. Les actions à droite vérifient les même propriété que les actions à gauche, à la permutation des opérandes de l'action près :

(30) $(e \star x) \star y) = e \star (x \cdot y)$

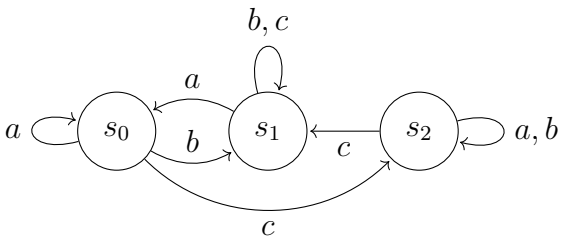
(31) $e \star 1_M = e$

pour tout $x, y \in S$ et pour tout $e \in E$.

L'implémentation en Haskell la même que celle de l'action à gauche, à la position des paramètres près :

```haskell
class Monoid m => ActionD m e where
  (*!) :: e -> m -> e
  -- ∀ x:e. x *! mempty == x
  -- ∀ a,b:m x:e. x *! (a <> b)  == (x *! a) *! b
```

Exemple (Graphe d'état d'automate)**.** Considérons l'automate suivant sur l'alphabet $A = \{a, b, c\}$, et les états $S = \{s_0, s_1, s_2\}$:



Cet automate définit une action à droite $\cdot$ du monoïde A^* sur S. Par exemple, nous pouvons lire :

$$s_0 \cdot ccba = (s_0 \cdot c) \cdot ccba = s_2 \cdot ccba = s_0$$

Utiliser une action à gauche rendrait le calcul peu intuitif, car il faudrait renverser l'ordre des lettres :

$$bacc \cdot s_0 = bac \cdot (c \cdot s_0) = s_2$$

Dans ce document nous allons nous intéresser à des actions particulières, celle des actions qui se comportent comme des morphismes. C'est-à-dire que nous allons considérer les actions d'un monoïde sur un monoïde, qui respecte la structure de ce dernier monoïde.

Définition 2.7.3 (Action par auto-morphisme)**.** Soient deux monoïdes $\langle M, \cdot, 1_M \rangle$ et $\langle N, +, 0_N \rangle$. Une action de M sur N est appelée par auto-morphisme si elle vérifie pour tout $x \in M$ et tout $a, b \in N$

$$x \star (a + b) = x \star a + x \star b$$

Comme il ne s'agit que d'ajouter une propriété, nous pouvons le représenter en Haskell à l'aide d'une nouvelle classe de type vide :

```
class ActionG m e => ActionAutoMorphism m e where
  -- ∀ a:m x,x':e. a !* (x <> x') == (a !* x) <> (a !* x')
```

Nous allons maintenant voir quelques constructions impliquant les théories que l'on a vu jusqu'à présent.

2.8 Produit Semi-Direct (PSD)

Une autre manière de former de nouveaux ensembles dotés de structure particulière est de faire des paires d'ensembles. La construction la plus immédiate est le produit direct :

Définition 2.8.1 (Produit direct)**.** Soient $\langle A, \circ \rangle$ et $\langle B, \cdot \rangle$ deux monoïdes. Nous pouvons construire le monoïde $\langle A \times B, \otimes \rangle$ où

$$(a, b) \otimes (a', b') = (a \circ a', b \cdot b')$$

Concernant l'implémentation :

```
newtype DP a b = DP (a,b)

instance (Semigroup a, Semigroup b) =>
  Semigroup (DP a b) where
    (DP (a,b)) <> (DP (a',b')) = DP (a <> a', b <> b')

instance (Monoid a, Monoid b) => Monoid (DP a b) where
  mempty = DP (mempty,mempty)
```

Nous allons maintenant voir le produit semi-direct. Cette construction est particulièrement intéressante. En utilisant une action dotée de propriétés particulière nous en feront un usage intensif dans ce document.

2.8.1 PSD à gauche

Nous allons voir deux versions du produit semi-direct, tout d'abord, celui définit à partir d'une action à gauche :

Définition 2.8.2 (Produit semi-direct)**.** Soient un groupe $\langle G, \times, 1_G \rangle$, un semi-groupe $\langle S, + \rangle$, et une action $\star$ de G sur S par automorphisme. Alors le produit semi-direct est défini comme $G \ltimes S = \langle G \times S, \cdot \rangle$, où :

$$(g, s) \cdot (g', s') = (g \times g', s + (g \star s'))$$

Théorème 2.8.3 (Associativité du PSD avec l'action à gauche)**.** *Soit $G \ltimes S$ un produit semi-direct vérifiant la définition 2.8.2. $\langle G \ltimes S, \cdot \rangle$ est un semi-groupe.*

Démonstration. La propriété d'associativité est prouvée sous l'hypothèse que l'action G sur S agisse par automorphisme, c'est-à-dire qu'elle respecte : $g \star (s_1 + s_2) = g \star s_1 + g \star s_2$. Comme nous pouvons le voir sur ces développements :

$$((g_1, s_1)(g_2, s_2))(g_3, s_3) = (g_1 \times g_2 \times g_3, s_1 + g_1 \star s_2 + g_1 \star g_2 \star s_3)$$
$$(g_1, s_1)((g_2, s_2)(g_3, s_3)) = (g_1 \times g_2 \times g_3, s_1 + g_1 \star (s_2 + g_2 \star s_3))$$

Remarquons que nous n'avons pas besoin que G soit un groupe pour prouver l'associativité, son associativité suffit.

□

Concernant l'implémentation, il s'agit d'une simple paire :

```
newtype SDP a b = SDP (a,b)

instance (Monoid a, Semigroup b, ActionG a b) =>
  Semigroup (SDP a b) where
  (SDP (a,b)) <> (SDP (a',b')) = SDP (a <> a',b <> (a !* b'))

instance (Monoid a, Monoid b, ActionG a b) =>
  Monoid (SDP a b) where
  mempty = SDP (mempty,mempty)
```

2.8.2 PSD à droite

Comme nous l'avons vu dans l'exemple page 29 sur les automates, parfois l'action à droite est plus naturelle à utiliser que l'action à gauche. Nous

avons défini le produit semi-direct d'un groupe sur un semi-groupe en partant d'une action à gauche, mais fort heureusement cette construction est isomorphe avec le produit semi-direct défini à partir d'une action à droite.

Définition 2.8.4 (Produit semi-direct avec action à droite)**.** Soient un groupe $\langle G, \times, 1_G \rangle$, un semi-groupe $\langle S, + \rangle$ et $\star$ une action à droite par automorphisme de G sur S. Le produit semi-direct avec action à droite est un semi-groupe défini par $G \rtimes S = \langle G \times S, \cdot \rangle$, où :

$$(g, s) \cdot (g', s') = (g' \times g, s + s' \star g)$$

Nous attirons l'attention du lecteur sur le membre gauche de la composition, où g' et g sont permutés par rapport au produit semi-direct $G \ltimes S$.

Théorème 2.8.5 (Associativité du PSD avec l'action à droite)**.** *Soit $G \rtimes S$ un produit semi-direct vérifiant la définition 2.8.4. $\langle G \rtimes S, \cdot \rangle$ est un semi-groupe.*

Démonstration. L'associativité de l'opérateur $\cdot$ est prouvé par le développement suivant :

$$\begin{aligned}
((g_1, s_1)(g_2, s_2))(g_3, s_3) &= (g_2 \times g_1, s_1 + s_2 \star g_1)(g_3, s_3) \\
&= (g_3 \times g_2 \times g_1, s_1 + s_2 \star g_1 + s_3 \star (g_2 \times g_1)) \\
&= (g_3 \times g_2 \times g_1, s_1 + (s_2 + s_3 \star g_2) \star g_1) \\
&= (g_1, s_1)((g_3 \times g_2, s_2 + s_3 \star g_2)) \\
&= (g_1, s_1)((g_2, s_2)(g_3, s_3))
\end{aligned}$$

Une fois encore, nous n'avons besoin que de l'hypothèse que G est un semi-groupe, et que $\star$ est une action par automorphisme. □

2.8.3 Isomorphisme des PSD à gauche et à droite

Les deux versions du produit semi-direct sont équivalente :

Théorème 2.8.6 (Produit semi-direct, action à gauche et action à droite)**.** *Soient un groupe $\langle G, \times, 1_G \rangle$, un semi-groupe $\langle S, + \rangle$, $\star_g$ une action à gauche par automorphisme de G sur S et $\star_d$ une action à droite par automorphisme de G sur S.*

Les produits semi-directs $G \ltimes S$ (avec l'action à gauche) et $G \rtimes S$ (avec l'action à droite) sont isomorphes.

Démonstration. Nous allons définir $G \rtimes S = \langle G \times S, \cdot_d \rangle$ à partir de $G \ltimes S = \langle G \times S, \cdot_g \rangle$. Tout d'abord, définissons l'action à droite $\star_d$ de G sur S par :

$$s \star_d g = g^{-1} \star_g s$$

pour tout $s \in S$ et $g \in G$. Montrons qu'il s'agit bien d'une action à droite, soient $g, g' \in G$ et $s \in S$.

$$s \star_d 1_G = {1_G}^{-1} \star_g s = s$$

$$\begin{aligned} s \star_d (g \times g') &= (g \times g')^{-1} \star_g s \\ &= {g'}^{-1} \times g^{-1} \star_g s \\ &= {g'}^{-1} \star_g g^{-1} \star_g s \\ &= {g'}^{-1} \star_g (s \star_d g) \\ &= (s \star_d g) \star_d g' \end{aligned}$$

Définissons maintenant la fonction f de type $G \ltimes S \to G \rtimes S$ par :

$$f(g, s) = (g^{-1}, s)$$

Très trivialement, nous remarquons qu'au type près, $f^{-1} = f$, il s'agit donc d'une bijection.

Soient (g, s) et (g', s') deux éléments de $G \rtimes S$, montrons que la fonction f est un morphisme.

$$\begin{aligned} f((g, s) \cdot_d (g', s')) &= f(gg', s + g \star_g s') \\ &= ((gg')^{-1}, s + g \star_g s') \\ &= ({g'}^{-1} g^{-1}, s + s' \star_d g^{-1}) \\ &= (g^{-1}, s) \cdot_d ({g'}^{-1}, s') \\ &= f(g, s) \cdot_g f(g', s') \end{aligned}$$

L'autre sens est très symétrique. Soient (g, s) et (g', s') deux éléments de $G \ltimes S$, montrons que f^{-1} est également un morphisme.

$$\begin{aligned} f^{-1}((g, s) \cdot (g', s')) &= f^{-1}(g'g, s + s' \star_d g) \\ &= ((g'g)^{-1}, s + s' \star_d g) \\ &= (g^{-1} {g'}^{-1}, s + g^{-1} \star_g s') \\ &= (g^{-1}, s) \cdot_g (g^{-1}, s') \\ &= f^{-1}(g, s) \cdot_g f^{-1}(g', s') \end{aligned}$$

Ce qui permet de conclure que $G \ltimes S$ et $G \rtimes S$ sont isomorphes. □

2.8.4 PSD sur semi-groupe inversif

Nous pouvons préserver la propriété d'être un semi-groupe inversif, et nous pouvons également en créer un à partir d'un semi-treillis.

Théorème 2.8.7 (Produit semi-direct d'un groupe et d'un semi-groupe inversif)**.** *Soient un groupe $\langle G, \times, 1\rangle$ et $\langle S, +\rangle$ un semi-groupe inversif. Soit $\star$ une action par automorphisme de G sur S. Le produit $G \ltimes S$ est un semi-groupe inversif.*

Démonstration. Montrons que $G \ltimes S$ est un semi-groupe inversif. C'est-à-dire qu'il est associatif et qu'il vérifie les axiomes des semi-groupes inversifs ((13), (14), et l'unicité des inverses).

Nous avons un semi-groupe. Nous pouvons vérifier que

$$(g, s)^{-1} = (g^{-1}, g^{-1} \star -s)$$

est bien un inverse au sens des semi-groupes inversifs. En effet, on a :

$$\begin{aligned}(g, s)(g, s)^{-1}(g, s) &= (g, s)(g^{-1}, g^{-1} \times -s)(g, s)\\ &= (g \times g^{-1}, s + g \times (g^{-1} \times -s))(g, s)\\ &= (1, s + 1 \times -s)(g, s)\\ &= (1 \times g, s - s + 1 \times s)\\ &= (g, s + -s + s)\\ &= (g, s)\end{aligned}$$

La preuve du second axiome des semi-groupe inversifs est très similaire, mais nous avons besoin de l'hypothèse que l'action est morphique pour conclure :

$$\begin{aligned}(g, s)^{-1}(g, s)(g, s)^{-1} &= (g^{-1}, g^{-1} \star -s)(g, s)(g^{-1}, g^{-1} \star -s)\\ &= (g^{-1}, g^{-1} \star -s + g^{-1} \star s + g^{-1} \star -s)\\ &= (g^{-1}, g^{-1} \star (-s + s - s))\\ &= (g^{-1}, g^{-1} \star -s) = (g, s)^{-1}\end{aligned}$$

Enfin, nous montrons la commutativité des idempotents, équivalente à l'unicité des inverses. Soient x et y deux idempotents de $(G \ltimes S)$. Posons $x = (g_x, s_x)$ et $y = (g_y, s_y)$. Puisque x est idempotent, nous avons $xx = x$, c'est-à-dire $(g_x \times g_x, s_x + g_x \star s_x) = (g_x, s_x)$. Nous en déduisons que $g_x = 1$ et $s_x \in E(S)$, nous pouvons effectuer la même démarche avec y.

$$xy = (g_x \times g_y, s_x + g_x \star s_y) = (1, s_x + s_y) = (1, s_y + s_x) = yx$$

La propriété de commutation des idempotents est héritée de S. Nous pouvons en conclure que $(G \ltimes S)$ est un semi-groupe inversif. □

Nous allons utiliser ce théorème à de nombreuses reprises au cours de ce document pour construire des monoïdes inversifs dans cette thèse.
Nous pouvons implémenter ces semi-groupes ainsi en Haskell :

```
instance (Group a,
          InverseSemigroup b,
          ActionAutoMorphism a b) =>
  InverseSemigroup (SDP a b) where
  inverse (SDP (a,b)) = SDP (inverse a,
                             inverse a !* inverse b)
```

Par ailleurs, comme nous l'avons vu dans l'exemple 21, tout semi-treillis est également un semi-groupe inversif. Dans ce semi-groupe inversif, l'inverse d'un élément est égal à l'élément lui-même. On a donc :

Corollaire 2.8.1 (Produit semi-direct d'un groupe et d'un semi-treillis)**.** *Soit T un semi-trellis avec le « meet » comme produit ($\wedge$). Soient $\langle G, \times \rangle$ un groupe, et $\star$ une action de G sur T qui agit par automorphisme (cf. définition 2.7.3).*
Alors le produit semi-direct $(G \ltimes T)$ est un monoïde inversif. Et l'inverse de (g, s) est $(g^{-1}, g^{-1} \times s)$.

2.8.5 PSD sur semi-groupe avec reset

Nous également nous contenter d'un semi-groupe avec reset.

Théorème 2.8.8 (Produit semi-direct d'un groupe et d'un semi-groupe avec reset)**.** *Soient $\langle G, \times, 1 \rangle$ un groupe et $\langle S, + \rangle$ un semi-groupe resetable (présenté en section 2.4). Soit $\star$ une action par automorphisme de G sur S. Le produit $G \ltimes S$ est un semi-groupe resetable. Le reset d'un élément de ce semi-groupe est :*

$$\mathrm{reset}\,((g, s)) = (1, \mathrm{reset}\,(s))$$

Démonstration. Montrons que $G \ltimes S$ est un semi-groupe resetable. C'est-à-dire qu'il est associatif et qu'il vérifie les axiomes des semi-groupes resetable

(axiomes (22),(23), (24), et (25)).
L'associativité est prouvée plus haut.
L'axiome (22) est $\text{reset}(x) \cdot x = x$, Soit $x = (g_x, s_x)$,

$$\begin{aligned}\text{reset}(x) \cdot x &= (1, \text{reset}(s_x)) \cdot (g_x, s_x) \\ &= (1 \times g_x, 1 \star \text{reset}(s_x) + s_x) \\ &= (g_x, \text{reset}(s_x) + s_x) = (g_x, s_x) = x\end{aligned}$$

L'axiome (23) est l'idempotence des reset [6]. Posons $x = (g_x, s_x)$:

$$\begin{aligned}\text{reset}(x) \cdot \text{reset}(x) &= (1, \text{reset}(s_x)) \cdot (1, \text{reset}(s_x)) \\ &= (1 \times 1, \text{reset}(s_x) + 1 \star \text{reset}(s_x)) \\ &= (1, \text{reset}(s_x) = \text{reset}(x))\end{aligned}$$

Nous pouvons montrer la commutativité des idempotents, c'est-à-dire que l'axiome (24) est satisfait. Soient x et y deux éléments de $(G \ltimes S)$. Posons $x = (g_x, s_x)$ et $y = (g_y, s_y)$.

$$\begin{aligned}\text{reset}(x) \cdot \text{reset}(y) &= (1, \text{reset}(s_x)) \cdot (1, \text{reset}(s_y)) \\ &= (1 \times 1, \text{reset}(s_x) + 1 \star \text{reset}(s_y)) \\ &= (1, \text{reset}(s_x) + \text{reset}(s_y)) \\ &= (1, \text{reset}(s_y) + \text{reset}(s_x)) \\ &= \text{reset}(y) \cdot \text{reset}(x)\end{aligned}$$

Enfin, l'axiome (25), $\text{reset}(y) \cdot x = x \Rightarrow \text{reset}(y) \cdot \text{reset}(x) = \text{reset}(x)$. Soient $x = (g_x, s_x)$ et $y = (g_y, s_y)$ vérifiant la prémisse de cette dernière implication, c'est-à-dire que $\text{reset}(y) \cdot x = x$ et donc que

$$(1, \text{reset}(s_y)) \cdot (g_x, s_x) = (g_x, \text{reset}(s_y) + s_x) = (g_x, s_x)$$

dans cette équation, nous remarquons que $\text{reset}(s_y) + s_x = s_x$. En appliquant l'axiome (25) à s_y et s_x, nous obtenons $\text{reset}(s_y) + \text{reset}(s_x) = \text{reset}(s_x)$.

Vérifions que la conclusion est vérifiée, $\text{reset}(y) \cdot \text{reset}(x) = (1, \text{reset}(s_y) + \text{reset}(s_x))$, ce qui est égal à $(1, s_x) = \text{reset}(x)$. □

Nous pouvons implémenter ces semi-groupes ainsi en Haskell :

6. $\text{reset}(x) \cdot \text{reset}(x) = \text{reset}(x)$

```
instance (Monoid a,
          ResetableSemigroup b,
          ActionAutoMorphism a b) =>
  ResetableSemigroup (SDP a b) where
  reset (SDP (_,b)) = SDP (mempty,reset b)
```

Les travaux réalisés pendant cette thèse tournent autour de la structure de monoïde inversif et son affaiblissement, les monoïdes avec reset. Nous allons montrer comment ces structures se prêtent aux applications multimédia.

3

Le modèle des tuiles

Dans ce chapitre, nous allons présenter le modèle des tuiles. Nous allons donner une intuition de la sémantique des tuiles ainsi qu'une sémantique abstraite que nous pourront instancier pour différents médias.

3.1 Médias

Les tuiles que nous souhaitons définir vont nous permettre de décrire un média. Nous pouvons définir un média comme une fonction partielle dont le domaine est un espace fixé et le co-domaine est un espace de valeurs. Par exemple cela pourra un ensemble de couleurs sur une surface en deux dimensions, des notes de musique dans un temps musical, les échantillons d'un signal audio dans un temps discretisé, des mouvements dans le temps, pour une danse... Une tuile va alors décrire non seulement un média, mais aussi la façon dont il pourra être composé avec d'autres médias.

3.1.1 Espace de positions

Tout d'abord, étudions l'espace qui va porter les valeurs de notre média. Sur cet espace nous auront besoin d'une notion de position et d'une notion de déplacement. Voyons tout d'abord l'espace des positions, il s'agit d'un ensemble équipé d'une position particulière, une origine. Nous n'avons pas besoin de faire davantage d'hypothèse à son sujet.

En ce qui concerne les déplacements, nous pouvons les voir comme des fonctions des positions dans les positions. Ces fonctions prennent en paramètre une position arbitraire et retourne la position obtenue après le déplacement. Nous les supposerons bijectives afin d'être réversibles.

Plus que l'espace, c'est la notion de déplacement que nous allons utiliser pour définir les tuiles. Nous avons besoin que cet ensemble de déplacement soit doté d'une structure de groupe. La composition pourra être vue comme la mise en séquence de deux déplacements, le neutre sera le déplacement nul, et l'inverse sera celui au sens des groupes, c'est-à-dire le déplacement qui permet de revenir à la position initiale. Nous allons utiliser une notation additive pour la composition des déplacements, avec l'opérateur $+$, ainsi que $-$, l'opérateur pour la négation.

Exemple (Groupe de déplacement temporel)**.** Considérons le cas où l'espace est le temps, par exemple pour définir de la musique. Alors les positions sont des instants dans le temps, les déplacements sont des délais positifs ou négatifs. Équipés de la somme et du délais nul, il s'agit bien d'un groupe.

Remarque (Position et état)**.** Nous parlons ici de position, mais dans la suite, cette notion sera largement généralisée. L'espace de positions auquel on s'intéresse finira par ressembler à un ensemble d'états possibles, et ce que nous appelleront déplacement sera alors plus proche d'une notion de transition. Ces transitions devront cependant rester réversibles.

3.1.2 Domaine de valeur

Les valeurs sont définies comme ponctuelle dans l'espace, ou « instantanée ». Par abus de langage nous allons parler d'évènements. Il est possible que deux valeurs se retrouvent à la même position, nous allons devoir les fusionner.

Afin d'avoir un inverse sur les tuiles, c'est-à-dire qu'elles aient une structure de semi-groupe inversif, nous avons besoin que l'ensemble des valeurs ait une structure particulière. À minima ces valeurs doivent avoir une structure de semi-treillis $\langle V, \cup \rangle$ (présenté en section 2.2.3) c'est-à-dire un semi-groupe idempotent, vérifiant $x \cup x = x$ et commutatif c'est-à-dire vérifiant $x \cup y = y \cup x$, pour tout $x, y \in V$.

Nous allons utiliser la relation d'ordre naturel sur ce domaine de valeur. Au lieu de l'habituel ordre donné par l'inclusion $a \leq b \Leftrightarrow a \subseteq b$, nous allons utiliser $a \leq b \Leftrightarrow a = a \cup b$. C'est exactement l'inverse de l'ordre donné par l'inclusion, l'ensemble vide est le plus grand élément.

Une autre possibilité que nous pourrons utiliser, est celle que cet ensemble de valeurs soit doté d'une structure plus riche, à savoir un semi-groupe inversif. Ceci est présenté en section 2.3.2, il ne s'agit que d'une généralisation, puisque tout semi-treillis est un semi-groupe inversif.

Exemple (Musique tonale)**.** Considérons la composition de musique tonale. À chaque instant dans le temps, il y a un ensemble de notes en train d'être

jouées. Nos valeurs étant supposées instantanée, chaque instant du temps porte pour valeur l'ensemble des notes en train d'être jouées. Nous pouvons faire le choix que deux notes identiques jouées au même instant sont équivalente à une seule. C'est la propriété d'idempotence. Les notes de musique jouées à un instant donné ont une structure de semi-treillis. C'est ce cas qui motive l'utilisation de l'opérateur $\cup$ pour la composition des valeurs.

Remarque. Dans la perspective d'applications, il est légitime d'objecter que la loi d'idempotence est parfois non souhaitable. Par exemple, dans le contexte de la composition musicale, nous pouvons souhaiter que jouer deux fois la même musique en parallèle soit équivalent à la jouer plus fort. Cependant, la loi d'idempotence est sans conséquence. Au besoin, elle force juste à ce que des copies discernables soit rendues explicites, par exemple en passant d'évènements portant une valeur v à un évènement portant une paire (v, n) avec $n \in \mathbb{N}$ où la seconde composante serait un marqueur utilisé pour distinguer les copies d'un même évènement v.

3.2 Modèle sémantique

Nous pouvons maintenant définir formellement ces tuiles. Soient $\langle V, \cup \rangle$ un semi-treillis de valeurs et $\langle P, 0_p \rangle$ un ensemble de positions avec une origine 0_p.

Soit $C = P \to V$, les fonctions partielles des positions dans les valeurs. Le contenu média d'une tuile est décrit par une fonction partielle de type C. Nous souhaitons que ces contenus soient dotés de la même structure que les valeurs V, nous définissons donc l'opérateur de fusion de la manière suivante :

$$(f \cup g)(x) = \begin{cases} f(x) \cup g(x) & \text{si } f(x) \text{ et } g(x) \text{ sont définies} \\ f(x) & \text{si seulement } f(x) \text{ est définie} \\ g(x) & \text{si seulement } g(x) \text{ est définie} \\ \bot & \text{sinon} \end{cases}$$

pour tout $x \in P$.

Cette construction est la composition de deux constructions décrite au chapitre 2. Tout d'abord la conversion d'un semi-groupe vers un monoïde (cf. la remarque page 15) et l'extension point à point (cf. la définition 2.6.1). Nous pouvons vérifier que cette opération est commutative et idempotente. Nous avons maintenant un semi-treillis $\langle P \to V, \cup, \varnothing \rangle$, où $\varnothing$ désigne la fonction partielle vide. Nous allons noter δ_v la fonction de Dirac définie seulement en 0_p et qui vaut v.

Soit $D \subseteq S(P)$, un sous-groupe (clos par composition et inverse) du groupe symétrique de P, c'est-à-dire l'ensemble des bijections de P dans P. L'ensemble D représente l'ensemble des déplacements réversibles dans l'espace de positions P. Cet ensemble est doté d'une structure de groupe que nous allons noter $\langle D, \circ, 1_D \rangle$, où $\circ$ désigne la composition fonctionnelle, c'est-à-dire la composition de déplacements. La constante 1_D représente la fonction identité, c'est-à-dire le déplacement nul. Nous allons noter d^{-1} l'inverse de $d \in D$.

La sémantique d'une tuile sera représentée par une paire de $(D \times C)$, c'est-à-dire un déplacement associé à un contenu d'évènements placés à des positions.

Nous allons maintenant passer en revue les fonctions et opérateurs permettant créer et de manipuler les tuiles

3.3 Tuile élémentaire

Les tuiles sont définies de manière incrémentale, en partant de tuiles primitives construites à partir de valeurs et de déplacements, jusqu'aux tuiles plus complexes obtenues par compositions de tuiles plus simples. Nous nous intéressons en particulier au cas des tuiles finies engendrées à partir de tuiles primitives.

Nous pourrons penser à des notes de musique placées dans le temps. C'est cette intuition qui est représentée dans les schémas qui représentent le temps s'écoulant de gauche à droite. La sémantique de chaque combinateur est fidèle à l'intuition donnée par les représentations graphiques que nous allons en donner.

Pour ce qui est de l'implémentation en Haskell, nous allons déclarer un type abstrait :

```
data Tile d v
```

qui décrit les tuiles de valeurs v sur un groupe de déplacements d entre des positions. Ce codage peut varier, mais quotienté par la sémantique il sera isomorphe à $(D \times C)$. Ces tuiles seront une instance de la classe `Semigroup`, qui déclare l'opérateur de composition `<>`. L'opérateur `+` est utilisé pour la classe `Num`, qui décrit peu ou proue un corps. Utiliser un opérateur `<>` évite les collisions, et des confusions vis-à-vis de la structure manipulée. La sémantique des tuiles est représentée par la fonction $[\![]\!]$ de type Tile $d\ v \rightarrow (D \times C)$.

3.3.1 Tuile évènementielle

Une tuile évènementielle peut être vue comme une manière de convertir une valeur dans le monde des tuiles.

La primitive permettant de créer une tuile contenant un simple évènement à partir d'une valeur est :

```
event :: Semigroup v =>  v -> Tile d v
```

Cette fonction créé une tuile primitive ne portant qu'un évènement. Cette tuile ne porte aucune notion de déplacement, pour cette raison, le type d paramètre du type de retour est polymorphe.

L'illustration que nous allons adopter pour une telle tuile est celle donnée par la figure 3.1. Le symbole $\curlyvee$ représente le repère d'entrée de la tuile, c'est-à-dire l'origine du déplacement induit par la tuile. Le symbole $\downarrow$, quant à lui représente le repère de sortie. C'est-à-dire l'image du repère d'entrée par le déplacement. Dans la figure 3.1, les positions des deux repères sont confondues, ce qui correspond à un déplacement nul associé à cette tuile. Notons que l'évènement e est placé à l'origine, dans le repère $\curlyvee$.

FIGURE 3.1 – Un évènement portant une valeur v.

Nous pouvons maintenant former la tuile $\text{event}\,(x)$, qui représente un seul évènement placé dans P, et qui engendre un déplacement nul. La sémantique de $\text{event}\,(x)$ est formellement définie par :

$$[\![\,\text{event}\,(x)]\!] = (1_D, \delta_x)$$

pour toute valeur $x \in V$.

3.3.2 Tuile de déplacement

Les autres tuiles primitives sont les déplacements. Ils jouent un rôle fondamental dans notre approche puisqu'ils nous permettent de voir les déplacements comme des cas particuliers de tuiles.

Le constructeur que nous allons utiliser sera la fonction suivante. Elle permet de créer une tuile à partir d'un déplacement.

```
move :: Group d => d -> Tile d v
```

La fonction move est bien nommée, elle construit le déplacement spécifié. La représentation d'un déplacement est donnée par la figure 3.2. La marque $\curlyvee$ représente le début du déplacement et la marque $\downarrow$ sa fin.

(a) Un déplacement d. (b) Un déplacement d^{-1}.

FIGURE 3.2 – Deux déplacements.

La sémantique du constructeur move est la suivante :

$$[\![\operatorname{move}(d)]\!] = (d^{-1}, \varnothing)$$

C'est-à-dire la tuile avec un contenu d'évènements vide, qui engendre le déplacement d. La raison pour laquelle la sémantique est exprimée en fonction de l'inverse du déplacement, plutôt que le déplacement lui-même est un peu technique et arbitraire, mais elle devrait apparaître plus clairement dans la suite. La valeur stockée permet de passer d'une valeur de C exprimée dans le repère de sortie d'une tuile vers son repère d'entrée.

3.4 Composition tuilée

L'opérateur principal défini sur les tuiles est la *somme tuilée*. Il s'agit simplement de combiner deux tuiles en suivant les étapes suivantes :

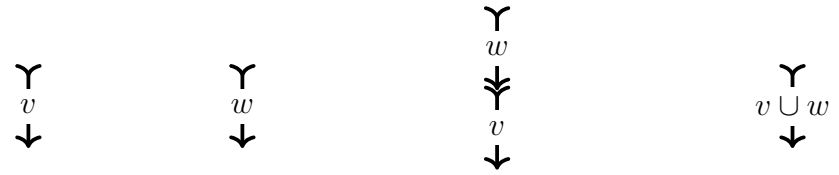
1. Combiner les deux tuiles en synchronisant le repère d'entrée ($\curlyvee$) de la *seconde* tuile sur le repère de sortie ($\downarrow$) de la *première*.
2. Fusionner les contenus.

Nous allons présenter les différents cas qui peuvent se présenter lors d'une composition.

3.4.1 Tuiles évènementielles

Tout d'abord, la composition de deux tuiles évènementielles, elle est illustrée sur la figure 3.3.

Puisque qu'aucune opérande n'induit un déplacement, le déplacement de la résultante est nul, les évènements sont simplement fusionnés.

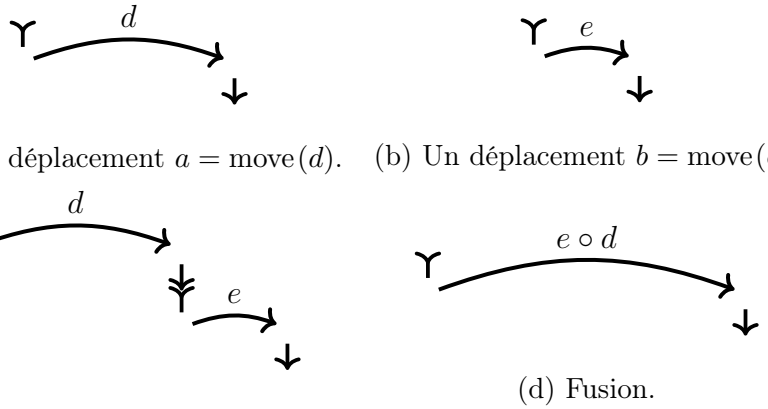


(a) Une tuile a. (b) Une tuile b. (c) Synchronisation de a et b. (d) Fusion des valeurs.

FIGURE 3.3 – Somme de deux tuiles évènementielles.

3.4.2 Tuiles de déplacements

Pour ce qui est des tuiles déplacements, elles se composent de la même manière : Synchronisation du repère d'entrée de la seconde opérande sur le repère de sortie de la première, et fusion. Ceci est illustré sur la figure 3.4.



(a) Un déplacement $a = \text{move}(d)$. (b) Un déplacement $b = \text{move}(e)$.

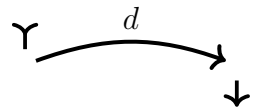
(c) Synchronisation. (d) Fusion.

FIGURE 3.4 – Somme de deux déplacements.

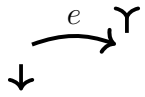
Observons que dans cet exemple, la somme se comporte exactement comme un opérateur de composition séquentiel. A priori, cette somme n'est pas commutative.

La composition est exactement la même lorsqu'un déplacement est de signe négatif, comme illustré sur la figure 3.5.

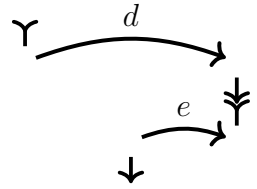
Dans la figure 3.5b, le déplacement e^{-1} va dans le même sens que e, mais nous avons permuté les repères d'entrée et de sortie. De cette manière le déplacement est toujours exprimé du repère d'entrée vers le repère de sortie.



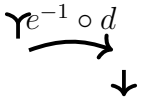
(a) Un déplacement a.



(b) Un déplacement $b = \text{move}(e^{-1})$.



(c) Synchronisation.



(d) Fusion.

FIGURE 3.5 – Somme d'un déplacement positif et d'un déplacement négatif.

Nous allons noter $|t|$ le déplacement global associé à la tuile a. Formellement, nous avons $|t| = d$ avec $[\![t]\!] = (d^{-1}, c)$. Dans la composition tuilée l'égalité suivante est toujours vérifée :

$$\text{move}(|t_1 + t_2|) = \text{move}(|t_1|) + \text{move}(|t_2|)$$

avec $|t| = d^{-1}$ lorsque $\langle t \rangle = (d, c)$. Exprimée dans D cela donne :

$$|t_1 + t_2| = |t_2| \circ |t_1|$$

.

Remarque. La fonction move est un morphisme des déplacements dans le temps dans les tuiles.

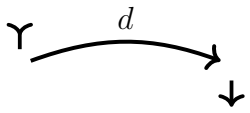
3.4.3 Tuiles de déplacement et évènementielles

Voyons maintenant les compositions mêlant des déplacements et des évènements, c'est illustré sur la figure 3.6.

Dans le cas de la figure 3.6c, la tuile b est synchronisée avec le repère de sortie de la tuile a, ainsi, l'évènement v que cette tuile porte se retrouve synchronisé avec le repère de sortie de a. À l'inverse, dans la figure 3.6d, la valeur v se retrouve synchronisé avec le repère d'entrée de a.

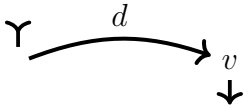
3.4.4 Tuiles composées

Pour des tuiles plus complexes, la somme de deux tuiles peut être illustrée par la figure 3.7. Dans cette fusion de tuiles tous les évènements qui se retrouvent à la même position sont fusionnés.

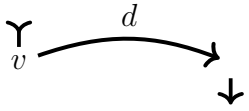


(a) Un déplacement a.

(b) Une tuile b.

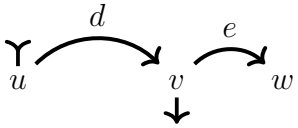


(c) Tuile $a + b$.

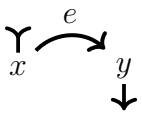


(d) Tuile $b + a$.

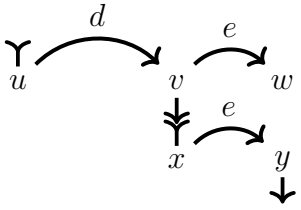
FIGURE 3.6 – Composition d'un déplacement et d'un évènement.



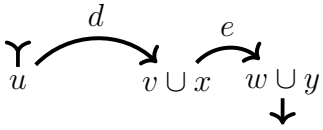
(a) Tuile a



(b) Tuile b



(c) Tuiles a et b synchronisées et non fusionnées



(d) La somme des tuiles $a + b$ fusionnée.

FIGURE 3.7 – La procédure de calcul de la somme $a + b$ de deux tuiles a et b.

Observons que dans le cas des tuiles primitives induites par un groupe de délais (déplacements dans le temps), les marques d'entrées et de sorties peuvent être interprétée comme les évènements de début et de fin associés à ce délai, et cette intuition est compatible avec la notion de somme tuilée décrite par la figure 3.7. Cependant, dans le cas général décrit sur la figure 3.8, une telle intuition sème la confusion. En effet, la marques d'entrée

d'une tuile peut se situer après la marque de sortie d'une tuile, comme c'est le cas dans la tuile b. Il en résulte que ces marques sont mieux comprises en étant interprétées comme des *marques de synchronisation* décrivant comme les tuiles seront positionnées l'une par rapport à l'autre dans une somme tuilée.

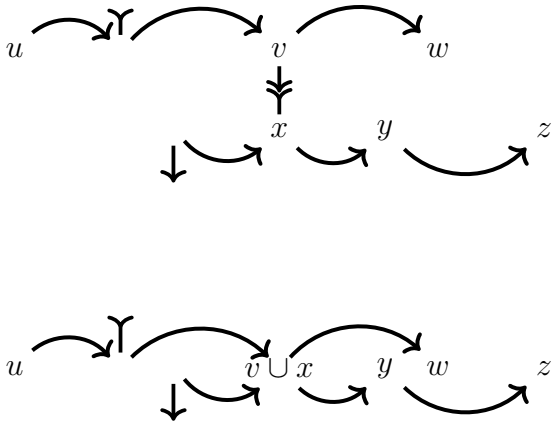


FIGURE 3.8 – La somme de deux tuiles générales.

3.4.5 Sémantique de la composition

Formellement, la sémantique de la composition de deux tuiles est définie par :

$$[\![t_1 + t_2]\!] = (d_2 \circ d_1, c_1 \cup (c_2 \circ d_1)) \tag{3.1}$$

avec $[\![t_1]\!] = (d_1, c_1)$ et $[\![t_2]\!] = (d_2, c_2)$

Pour rappel, les fonctions d_1 et d_2 représentent les déplacements du repère de sortie au repère d'entrée de leur tuile respective. Nous voulons calculer le déplacement global de la tuile $t_1 + t_2$, c'est-à-dire le déplacement du repère de sortie de t_2 au repère d'entrée de t_1, en appliquant successivement ces deux fonctions, on obtient $d_2 \circ d_1$.

Le contenu c_1 est placé par rapport au repère d'entrée de t_1, est donc correctement placé dans $t_1 + t_2$. Le contenu c_2 est placé par rapport au repère d'entrée de t_2. Après la composition, le repère d'entrée de t_2 sera oublié, il faut donc replacer c_2 par rapport au repère d'entrée de t_1. Nous obtenons cette nouvelle position par composition avec d_1.

Remarque (PSD)**.** La formule de la composition des sémantiques (equation 3.1) est probablement le cœur de ce travail de thèse. Cette équation est un produit semi-direct avec l'action à droite (définition 2.8.4).

3.5 Négation

Le lecteur peut être surpris par la présence de ces déplacements « négatifs ». Considérons une tuile dont le déplacement global est négatif, par exemple la seconde opérande de la figure 3.8. Nous avons deux manières de générer une telle tuile. La première est de construire le contenu de la tuile de manière séquentielle à l'aide de déplacements positifs, et de composer ensuite cette tuile avec deux déplacements positifs ou négatifs afin de placer les marques d'entrée et de sortie aux positions souhaitées. La seconde est de construire la négation de la tuile souhaitées.

Des propriétés algébriques apparaissent avec la définition de la négation d'une tuile, définie par le renversement des repères d'entrée et de sortie de la tuile. Dans le cas simple des tuiles primitives, la négation peut être illustrée par la figure 3.9.

(a) Une tuile a. (b) la tuile $-a$.

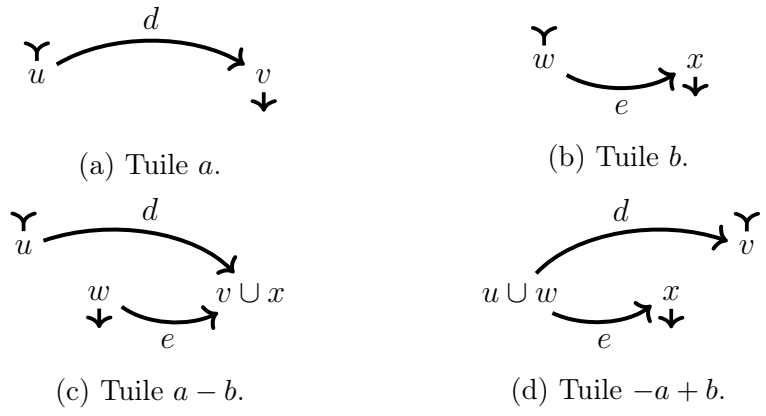
FIGURE 3.9 – Une tuile a et sa négation $-a$.

La grandeur du déplacement $|-a|$ de la négation d'une tuile a est simplement définie comme la négation $-|a|$ de la grandeur de son déplacement. Nous devons insister sur le fait que le contenu de la tuile n'est pas affecté par la négation de la tuile, cela ne fait que déplacer les repères d'entrée et de sortie sans affecter quoi que ce soit d'autre. En particulier, ni les évènements, ni les déplacements n'ont été renversés.

Une manière d'implémenter cette opération en Haskell est la suivante, il s'agit d'ajouter à gauche et à droite de la tuile l'inverse du déplacement de la tuile :

```
negate a = let nad = move (inverse (getMove a))
           in nad <> a <> nad
```

Le véritable intérêt de la négation apparaît en définissant la différence $a - b$ de deux tuiles par $a + (-b)$. Cette opération et son opération duale $-a + b$ sur les tuiles primitives est illustrée sur la figure 3.10.



(a) Tuile a. (b) Tuile b.

(c) Tuile $a - b$. (d) Tuile $-a + b$.

FIGURE 3.10 – Différence $a - b$ et $-a + b$ de deux tuiles a et b.

Nous pouvons facilement vérifier que $-(a + b) = -b + -a$, c'est-à-dire que la négation agit sur les tuiles comme un anti-morphisme. En ce qui concerne les grandeurs des déplacements, nous avons $|a - b| = |a| - |b|$, et $|-a + b| = |b| - |a|$.

Comme il devrait devenir plus clair dans la suite, dans le cas général, $a - a \neq 0$. Au contraire, l'algèbre des tuiles implémente une sorte de principe d'entropie sur les contenu de sorte qu'aucun évènement ayant été produit ne puisse disparaître... En fait, ces tuiles seront une instance de la classe `InverseSemigroup` présentée à la section 2.3.2.

Remarque. Alors que la somme de deux tuiles primitives peut être interprétée comme une composition séquentielle, leur différence introduit du parallélisme. Dans la différence $-a + b$, les deux tuiles primitives sont synchronisées sur leur repère entrée. De façon duale, dans la différence $a - b$, les tuiles primitives sont synchronisées sur leurs repère de sortie. Ceci montre qu'avec la négation, la somme est un opérateur de composition ni séquentiel ni parallèle, il est les deux à la fois. Cette propriété est l'une des caractéristiques importante de la notion de programmation temporelle tuilée que nous souhaitons développer ici.

Le fait que les évènements forment un semi-treillis garanti que les tuiles idempotentes commutent les unes avec les autres. Donc, suivant un argument classique de la théorie des semi-groupes inversifs, pour toute tuile a, la tuile $-a$ est, à équivalence opérationnelle près, l'*unique* inverse de la tuile a.

Dit autrement, nos tuiles équipée de la somme et de la négation forment un monoïde inversif avec 0, le déplacement nul, comme élément neutre.

Pour ce qui est de la sémantique, la négation est définie ainsi :

$$[\![-t]\!] = (d^{-1}, c \circ d^{-1})$$

avec $[\![t]\!] = (d, c)$.

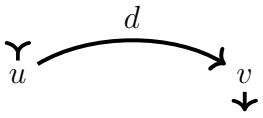
Le déplacement de la négation de (d, c) est naturellement d^{-1}, pour ce qui est du contenu, il est placé par rapport à l'entrée de la tuile, cette dernière se retrouve à la position de la sortie dans la négation, il faut donc replacer le contenu par rapport à la sortie, d'où le $c \circ d^{-1}$.

3.6 Opérateurs dérivés

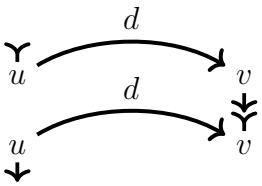
Les opérateurs définis par un semi-groupe inversif, couplé avec leur propriété, permettent de définir des opérateurs dérivés qui nous seront utiles dans la suite.

3.6.1 Reset et coreset

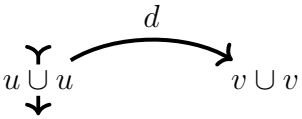
Le reset et le coreset sont deux opérations unaires qui permettent toute les deux de ramener le déplacement de la tuile à 0 sans en changer le contenu. La différence entre les deux réside dans le fait que les marques de synchronisation des tuiles ne sont pas au même endroit. Le reset place la marque de synchronisation de fin à la même place que celle de début. Cette opération est illustrée par la figure 3.11.



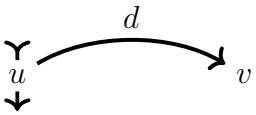
(a) Une tuile a



(b) La tuile $\text{reset}(a) = a + -a$ en cours de composition.



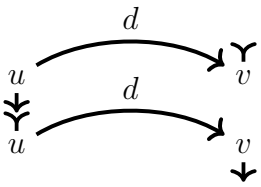
(c) Composition des évènements.



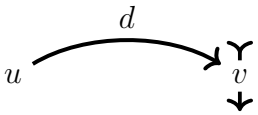
(d) Application de la loi d'idempotence

FIGURE 3.11 – La construction du reset d'une tuile a.

Le coreset fait « l'opposé », c'est-à-dire qu'il place la marque de début à la même place que celle de fin. La tuile obtenue est représentée sur la figure 3.12.



(a) La tuile coreset a en cours de composition.



(b) Application de la loi d'idempotence.

FIGURE 3.12 – Le coreset de la tuile a.

L'implémentation peut se faire directement à l'aide de la négation et de la composition :

```
reset :: InverseSemigroup s => s -> s
reset a = a <> inverse a
coreset :: InverseSemigroup s => s -> s
coreset a = inverse a <> a
```

La sémantique de ces opérateurs peut être exprimée ainsi :

$$[\![\operatorname{reset}(t)]\!] = (1_D, c)$$
$$[\![\operatorname{coreset}(t)]\!] = (1_D, c \circ d^{-1})$$

avec $[\![t]\!] = (d, c)$.
Nous pouvons d'ailleurs retrouver ce résultat grâce à la propriété $\operatorname{reset}(t) = t - t$. Nous obtenons le développement suivant :

$$\begin{aligned}[\![\operatorname{reset}(t)]\!] = [\![t - t]\!] &= (d, c) + (d^{-1}, c \circ d^{-1}) \\ &= (d^{-1} \circ d, c \cup (c \circ d^{-1}) \circ d) \\ &= (1_D, c \cup c \circ 1_D) \\ &= (1_D, c)\end{aligned}$$

3.6.2 Fork et join

Le reset d'une tuile permet donc de synchroniser les marqueurs de début de deux tuiles. C'est-à-dire que la composition avec le reset d'une tuile est équivalent à une mise en parallèle avec une synchronisation sur les débuts. Au contraire, le coreset permet de synchroniser le marqueur de fin d'une tuile avec une autre tuile, et la composition avec une telle tuile revient à une mise en parallèle avec la synchronisation sur la fin. Cela permet par exemple d'implémenter respectivement une sorte de « fork » et de « join ».

```
fork :: InverseSemigroup s => s -> s -> s
fork a b = reset a <> b
join :: InverseSemigroup s => s -> s -> s
join a b = a <> coreset b
```

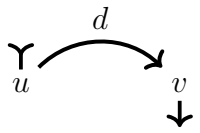
La structure de monoïde est en elle-même suffisante pour composer des médias arbitraire à l'aide de ce modèle. Cependant, les opérations qui dérivent du fait qu'il s'agisse en réalité d'un monoïde inversif permet de rajouter de la structure en explicitant des synchronisations entre différentes composantes du média. D'un point de vue sémantique, on aura :

$$[\![\operatorname{fork}(t_1, t_2)]\!] = (d_2, c_1 \cup c_2)$$
$$[\![\operatorname{join}(t_1, t_2)]\!] = (d_1, c_1 \cup c_2 \circ {d_2}^{-1} \circ d_1)$$

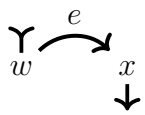
avec $[\![t_1]\!] = (d_1, c_1)$ et $[\![t_2]\!] = (d_2, c_2)$.

3.7 Ordre naturel

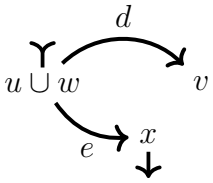
L'implémentation en Haskell de l'ordre naturel sur les tuiles peut se faire de la manière suivante :



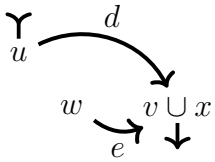
(a) Tuile a



(b) Tuile b



(c) la tuile fork(a, b).



(d) la tuile join(a, b).

FIGURE 3.13 – Deux tuiles a et b, leur fork (à gauche) et leur join (à droite).

```
instance (Group d, Semigroup v) => PartialOrd (Tile d v) where
  a <= b =
    a == a <> inverse a <> b
    -- or, a == reset a <> b
```

Cet ordre partiel est stable sous la composition, c'est-à-dire que si $a \leq b$, alors $c + a \leq c + b$ et $a + c \leq b + c$. Réfléchissons une minute à la différence entre une tuile a et la sa tuile associée move$(|a|)$, la seconde est obtenue de la première en omettant tous ses évènements, ne conservant que son déplacement. C'est la tuile la plus simple que nous pouvons associer à la tuile a avec le même déplacement. Ceci implique que, bien que ce soit légèrement contre intuitif, plus complexe est une tuile, plus bas elle est dans l'ordre naturel.

L'interprétation sémantique de l'ordre naturel reflète cette idée que nous évoquions plus tôt, l'augmentation irréversible de l'entropie au gré des compositions.

$$[\![t_1 \leq t_2]\!] \Leftrightarrow d_1 = d_2 \text{ et } c_1 \leq c_2$$

avec $[\![t_1]\!] = (d_1, c_1)$ et $[\![t_2]\!] = (d_2, c_2)$. C'est-à-dire que $a \leq b$ si a et b ont le même déplacement global, et si le contenu de b est inclus dans celui de a.

3.8 Tuile vers déplacement

Similaire à la fonction move, qui associe un déplacement d a la tuile engendrant ce déplacement, nous allons définir la fonction toMove. Cette

dernière retourne le déplacement ayant la même grandeur que la tuile en paramètre. Autrement dit, elle supprime le contenu de la tuile.

```
toMove :: Group d => Tile d v -> Tile d v
-- toMove a == move |a|
```

Comme nous l'attendons, la fonction toMove est un morphisme du monoïde inversif de l'ensemble des tuiles dans lui-même, c'est-à-dire :

$$\begin{aligned}\text{toMove}(a+b) &= \text{toMove}(a) + \text{toMove}(b)\\ \text{toMove}(-a) &= -(\text{toMove}(a))\end{aligned}$$

Pour toute tuile a et b. De plus, il s'agit d'une projection car nous avons $\text{toMove} = \text{toMove} \cdot \text{toMove}$. Autre propriété intéressante, observons la relation entre une tuile a et le déplacement $|a|$ que nous définissons par $d = \text{toMove}(a)$. Plus tôt, nous avons défini l'ordre naturel sur les semi-groupes inversifs (2.3.3), la tuile $\text{toMove}(a)$ est le maximum des tuiles plus grande que a. La sémantique de toMove est la suivante :

$$\begin{aligned}[\![\,\text{toMove}(t)]\!] &= (d, \varnothing)\\ \text{avec } [\![t]\!] &= (d, c)\end{aligned}$$

Remarque (Implémentation alternative de reset et coreset)**.** Une autre manière de l'implémenter, tirant profit de l'idempotence de v est la suivante, à l'aide de la fonction toMove.

```
reset :: (Group d, Semigroup v) => Tile d v -> Tile d v
reset a = a <> inverse (toMove a)
coreset :: (Group d, Semigroup v) => Tile d v -> Tile d v
coreset a = inverse (toMove a) <> a
```

3.9 Multiplication et division tuilées

Plus haut nous avons défini une algèbre additive pour les tuiles. Lorsque l'espace des déplacements est aussi doté d'une multiplication. C'est-à-dire qu'il s'agit d'un anneau, une structure $\langle S, +, \cdot, 0_S, 1_S\rangle$ tel que $\langle S, \cdot, 1_s\rangle$ est un monoïde et $\cdot$ est commutatif et distribue sur $+$, autrement dit que pour tout a, b et c, nous vérifions $a \cdot (b + c) = (a \cdot b) + (a \cdot b)$. Dans ce cas, nous pouvons définir un opérateur d'étirement stretch qui permet d'utiliser cette multiplication sur les tuiles :

```
stretch :: Num d => d -> Tile d v -> Tile d v
-- stretch 1 t == t
-- stretch d (a + b) == stretch d a + stretch d b
-- |stretch d a| == d * |a|
-- stretch is a morphism between d and Tile d v
```

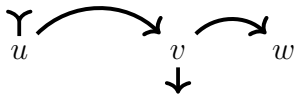
Remarque. Pour le cas `stretch 0`, tous les évènements se retrouvent composés en un point.

3.9.1 Multiplication

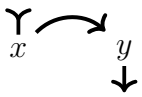
En combinant les étirements avec des opérateurs de composition parallèles plus explicites dérivant de l'algèbre additive des tuiles, nous parvenons à la définition d'une algèbre multiplicative des tuiles.

```
mult a b = fork (stretch (getMove b) a)
                (stretch (getMove a) b)
```

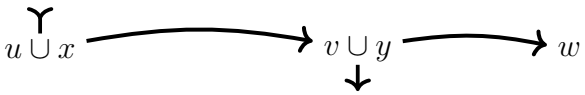
Nous définissons le produit des tuiles a et b en deux étapes. Tout d'abord, en définissant deux tuiles $a' = \text{stretch}(|b|, a)$ et $b' = \text{stretch}(|a|, b)$. Ces deux tuiles ont une longueur égale à $|a| \times |b|$. Maintenant, nous pouvons composer $\text{fork}(a', b') = \text{reset}(a') + b'$. Remarquons également que comme a' et b' ont même longueur, alors nous avons : $\text{fork}(a', b') = \text{join}(a', b')$, un coup d'œil à la figure 3.13 permet de s'en convaincre. Ce produit nous donne une nouvelle définition d'un produit parallèle. La figure 3.14 illustre cette opération.



(a) Tuile a



(b) Tuile b



(c) Chaque tuile est étiré du déplacement de l'autre, et synchronisée.

FIGURE 3.14 – Le produit de deux tuiles a et b

Remarque (Composition parallèle dans Euterpea)**.** Un problème rencontré dans Euterpea [14] est que des choix arbitraires doivent être fait pour la

définition de la mise en parallèle (point de synchronisation, durée résultante de la composition, troncature du plus long ? etc.). Le produit défini ci-dessus généralise tous ceux qui semblent raisonnable en rendant égale les durées des deux tuiles que l'on souhaite mettre en parallèle.

3.9.2 Propriétés algébriques

Tout d'abord, ce produit est associatif. Cela vient de l'associativité de la multiplication sur l'espace sous-jacent, et de l'associativité de la composition tuilée utilisée dans fork.

Ce produit donne aux tuiles une structure de monoïde, avec la tuile vide de longueur 1 pour neutre.

Le produit est également commutatif. Cela vient de la commutativité de la multiplication sur l'espace sous-jacent, et de la commutativité des idempotents reset(a) du fork.

Notons cependant que cette multiplication ne distribue pas sur l'addition. Nous constatons la présence d'idempotents. À l'instar des idempotents de l'addition, les idempotents des tuiles multiplicatives sont celles dont la longueur est le neutre pour l'opération impliquée sur les longueurs, c'est-à-dire 1 en l'occurrence.

3.9.3 Sémantique

Pour ce qui est de la sémantique, la construction est similaire. À la place d'un groupe de déplacement, nous avons besoin d'un anneau commutatif. Il également faut une seconde action à droite avec sémantique du « stretch ». Nous allons donc considérer l'anneau de déplacement suivant, où nous notons $+$ l'opération du groupe additif et $\times$ l'opération du monoïde commutatif multiplicatif $\langle D, +, \times, 0_+, 1_\times \rangle$. Nous définissons également l'action multiplicative morphique à droite de D sur C, que nous noterons également $\times$. Cette action est la sémantique de l'étirement. La sémantique du produit est alors [1] :

$$t_1 \times t_2 = (d_1 \times d_2, \lambda d.c_1(d/d_2) \times \lambda d.c_2(d/d_1)$$

avec $[\![t_1]\!] = (d_1, c_1)$ et $[\![t_2]\!] = (d_2, c_2)$.

Cette équation dérive directement du code Haskell, après simplification de la composition par le reset d'une tuile induite par l'opérateur fork.

1. Rappelons que $\times$ et $\cup$ sont deux opérateurs commutatifs.

3.9.4 Division

Si l'espace des déplacements est un corps, c'est-à-dire nous avons non seulement un groupe $\langle S, +, 0_S \rangle$ mais $\langle S \setminus \{0_s\}, \cdot, 1_S \rangle$ en est également un. Pour tout élément de $x \in S \setminus \{0_S\}$, il existe un élément $1/x \in S$ vérifiant $x \cdot 1/x = 1_S$. Utilisant cet élément nous pouvons implémenter une division sur les tuiles. La division par une tuile dont le déplacement est 0_S n'est pas définie. Nous nous proposons de noter / cet opérateur de division tuilé.

```
div a b = let ib = 1/(getMove b)
          in mult (stretch ib a) (stretch ib b)
```

Pour la sémantique, cela donne :

$$t_1/t_2 = (d_1 \times 1/d_2, c_1 \times 1/d_2 \cup c_2 \times 1/d_2 \times d_1)$$

avec $[\![t_1]\!] = (d_1, c_1)$ et $[\![t_2]\!] = (d_2, c_2)$.

3.9.5 Semi-groupe inversif multiplicatif

Ce nouveau groupe multiplicatif a lui aussi une structure de monoïde inversif. Contrairement au monoïde inversif additif, celui ci est également commutatif. Cette commutativité est héritée de celle du semi-treillis.

Jusque-là nous avons étudié la multiplication et la division, intéressons-nous maintenant aux opérateurs dérivés.

Le reset, dans le cas additif est défini comme une tuile composée avec sa négation. Pour le cas multiplicatif, il s'agit de la tuile composée avec son inverse.

$$\text{reset}_\times(t) = t \times t^{-1} = t/t$$

Soit une tuile t, quelle est la tuile t/t ? Son déplacement est égal au neutre du groupe multiplicatif, noté $1_\times$. Le contenu de cette tuile est celui de t étiré d'un facteur $|t|^{-1}$. Autrement dit, le reset multiplicatif d'une tuile ramène son déplacement à $1_\times$.

Pour ce qui concerne le coreset, il est défini sur une tuile t de manière additive par $-t + t$. Du côté multiplicatif, cela devient $t^{-1} \times t$, et par commutativité $t \times t^{-1} = t/t$. Nous avons donc :

$$\text{coreset}_\times = \text{reset}_\times$$

Le fork additif de t et u est $\text{reset}(t) + u$. Cet opérateur synchronise les opérandes sur leur début, et le déplacement de la résultante est celui de la seconde opérande. Pour le cas multiplicatif sur des tuiles t et u, le résultat

est $(t/t) \times u$. C'est-à-dire que cet opérateur va étirer t au déplacement de u et les synchroniser.

Pour la même raison que le reset est égal au coreset, le join est égal au fork.

3.9.6 Ordre naturel multiplicatif

La notion d'ordre naturel est définie modulo un opérateur de composition. Nous allons étudier l'ordre naturel multiplicatif. Remarquons tout d'abord que cet ordre n'est pas défini pour les tuiles dont le déplacement est 0_S.

Théorème 3.9.1. *Soient $\leq_+$ l'ordre naturel additif et $\leq_\times$ l'ordre naturel multiplicatif. Sur l'ensemble des tuiles dont le déplacement n'est pas nul, nous avons :*

$$\leq_+ = \leq_\times$$

C'est-à-dire que l'ordre naturel additif et l'ordre naturel multiplicatif sont les mêmes.

Démonstration. Supposons que le membre gauche de l'équivalence suivante est vérifiée, évaluons la partie droite :

$$a \leq_\times b \Leftrightarrow a = a/a \times b$$

Tout d'abord, remarquons que $a/a = (1_\times, c_a \times d_a^{-1})$, il s'agit du reset multiplicatif. Si nous multiplions cela par b, nous obtenons : $(d_b, c_a \times d_a^{-1} \times d_b \cup c_b)$. Si $a \leq_\times b$ nous avons $d_a = d_b$ et $c_a = c_a \times d_a^{-1} \times d_b \cup c_b$. Cette dernière équation peut se simplifier si $d_a = d_b$ en $c_a = c_a \cup c_b$, c'est-à-dire $c_a \leq c_b$.Autrement dit, $(d_a, c_a) \leq_\times (d_b, c_b)$ si et seulement si $d_a = d_b$ et $c_a \leq c_b$. Ce qui est précisément l'implication sémantique de $\leq_+$ (cf. section 3.7). □

3.9.7 Multiplication et unités

Une propriété du produit est à prendre en compte lors de son usage en pratique, il s'agit de sa sensibilité à l'unité choisie. Par exemple, plaçons-nous dans le cas où l'espace tuilé est le temps. Le comportement du produit dépend de l'unité choisie pour l'échelle temporelle. Considérons une tuile a dont la durée est de 20 minutes, et considérons une autre tuile b quelconque. Est-ce que l'on a $a \times b > b$ ou bien $a \times b < a$? Cela dépend de l'unité choisie pour l'échelle temporelle. Si l'unité et la minute, alors le produit sera 20 fois plus long que b. Si c'est l'heure, le produit sera 3 fois plus court

que b. Donc, le changement de l'échelle de temps sous-jacente à un programme peut changer son comportement. Notre but étant de décharger le programmeur de ces problématiques, un tel opérateur peut potentiellement les réintroduire.

D'un point de vue physique, nous remarquons que l'équation aux unités de l'opérateur stretch est différent de son type en Haskell. En effet, si l'on note d le type de l'échelle temporelle, v le type des valeurs, et T le type des tuiles, l'équation aux unités donne à stretch un type $d \to T\ d\ e \to \mathrm{T}\ d^2\ e$, alors que son type Haskell est `d -> T d v -> T d v`. Ce constat explique la sémantique étrange de cet opérateur. Le produit de tuile sera donc à utiliser avec précaution.

En revanche, cela peut être très utiles si l'on se restreint au cas où une opérande a un déplacement de 1, le neutre du monoïde multiplicatif. C'est typiquement le cas de toute tuile de la forme t/t. Nous pouvons ensuite la multiplier par une tuile du déplacement désiré pour l'étirer. Il s'agit là du fork multiplicatif.

3.10 Conclusion

Nous sommes partis des travaux initiés par David Janin [15, 8, 16, 9] avant le début de ce travail de thèse. L'intuition et la sémantique livrée dans ce chapitre est le fruit du cheminement conceptuel réalisé au cours de ce travail de thèse. L'exhibition de ces structures en partant du modèle n'a pas été triviale.

Nous avons présenté ici le modèle à travers l'interface permettant de le manipuler, ainsi que quelques propriétés de ce modèle. Ce modèle est la base des travaux qui seront exposés dans les chapitres suivants. Nous allons considérer le cas d'une structure totalement ordonnée : le temps. Dans un premier temps, pour la programmation « hors temps », et ensuite pour la programmation réactive.

4

Implémentation des tuiles temporelles

Les tuiles syntaxiquement décrites au chapitre 3 ne sont pas isomorphe à leur sémantique. Elles sont formées par produits de tuiles primitives (déplacements et évènements), de manière syntaxique et ressemblent à des « zigzags ». Généralement, nous sommes intéressés par le contenu de la tuile dans l'ordre chronologique. La question qui se pose alors est de trouver une forme normale qui permet d'implémenter la sémantique. C'est-à-dire, d'avoir une correspondance entre l'implémentation et la sémantique.

Dans ce chapitre nous allons restreindre notre étude au cas où l'espace sur lequel nous allons placer des valeurs est le temps. Sur cet espace nous pouvons former un groupe de déplacement, que nous allons appeler des « délais » et noter delay. De plus, nous allons supposer que cet espace est totalement ordonné. En utilisant cette nouvelle propriété en plus de celles présentées dans le chapitre 3, nous allons pouvoir normaliser les tuiles et, en s'appuyant sur cette forme normale, proposer une implémentation efficace. En particulier, deux tuiles de même sémantique auront la même implémentation.

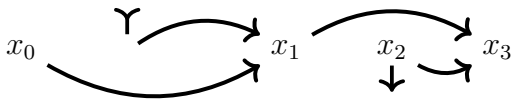
Dans ce chapitre, l'addition est étendue point-à-point aux « déplacements » temporels. On identifie aussi la durée d au déplacement temporel $\lambda t.t - d$.

4.1 Primitives d'accès

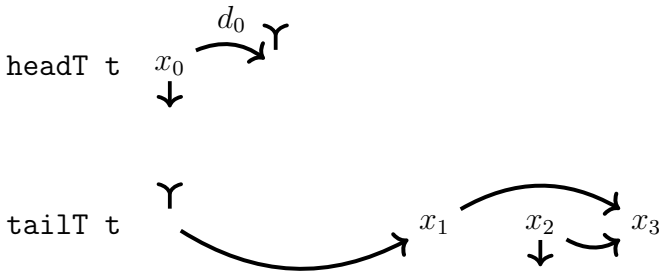
Lorsque nous voulons calculer le rendu de la tuile, par exemple, jouer les notes de musique qui la compose dans l'ordre, nous avons besoin de savoir quel est le premier évènement qui doit être joué dans la tuile. Les fonctions suivantes permettent de séparer la tuile à jouer en deux, juste après le premier évènement. La fonction `headT` retourne la tuile composée

du premier délai et du premier évènement, et `tailT` retourne le reste de la tuile, comme illustré sur la figure 4.1.

```
headT :: Tile d v -> Tile d v
tailT :: Tile d v -> Tile d v
```



(a) Tuile t.



(b) Décomposition de t en headT et tailT.

FIGURE 4.1 – Les primitives `headT` et `tailT`.

Pour toute tuile t, la propriété suivante est vérifiée :

$$t = \text{headT}\,(t) + \text{tailT}\,(t)$$

Ces fonctions sont nommées d'après les fonctions `head` et `tail` définies sur les listes à cause de leur ressemblance sémantique, en effet, toute liste non vide l vérifie :

$$l = \text{head}(l) : \text{tail}(l)$$

Une manipulation plus fine des tuiles peut être faite à l'aide de fonctions pour récupérer les évènements et les durées d'une tuile.

```
firstDE :: Tile d v -> (d, Maybe v)
```

La fonction `firstDE` retourne le délai avant le premier évènement ainsi que ce premier évènement, ou bien `Nothing` si la tuile ne porte aucun évènement. Cette interface utilise le type `Maybe d`, qui est le type des valeurs optionnelles. Il est défini par :

```haskell
data Maybe a = Just a
             | Nothing
```

Ce type est fréquemment utilisé dans le type de retour de fonctions partielles, afin de les rendre totales. Si la valeur est définie la fonction retourne `Just x`, et `Nothing` sinon. En C++, ce type s'appelle `std::optional`.

```haskell
headT t = case (firstDE t) of
  (d,Nothing) -> delay d
  (d,Just e) -> delay d + event e
```

Maintenant, nous allons nous intéresser à deux aspects de cette normalisation. Le premier est la structure de la forme normale, et ce qu'il nous dit pour une implémentation efficace algorithmiquement.

4.2 Forme normale

Des travaux antérieurs à cette thèse [8] ont montré que toute tuile avait une forme normale de la forme suivante :

$$t = \text{delay}(d_0) + \text{event}(e_0) + \sum_{i=1}^{n}[\text{delay}(d_i) + \text{event}(e_i)] + \text{delay}(d_{n+1})$$

avec n le nombre de positions qui portent des évènements. Les déplacements d_i sont strictement positive pour $0 < i < n + 1$. C'est-à-dire que seuls d_0 et d_{n+1} peuvent être négatifs ou nuls. La figure 4.2 illustre cette forme normale pour la tuile de la figure 4.1a avec $n = 3$.

Pour le cas où $n = 0$, il n'y a pas d'évènement, la forme normale est un simple délai.

Cette forme normale explicite l'ordre des évènements, ainsi que la distance dans le temps les séparant. Cette forme normale explicite également les positions des repères d'entrée et de sortie par rapport (respectivement) au premier évènement e_0 et au dernier e_n, via d_0 et d_n.

Les primitives headT et tailT permettent de calculer cette forme normale. En effet, cette dernière équation peut se réécrire :

$$t = \text{headT}(t) + \sum_{i=1}^{n}(\text{headT} \circ \text{tailT}^{i+1})(t) + \text{headT}^{n+1}(t)$$

où f^k est la fonction f itérée k fois. En effet, la fonction $\text{headT} \circ \text{tailT}^k$ calcule $d_k + e_k$.

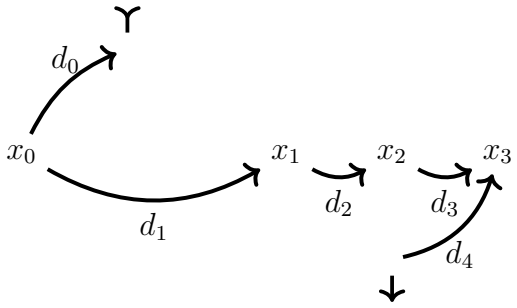


FIGURE 4.2 – Tuile t normalisée.

Remarque. Dans cette forme normale, la durée de la tuile est exprimée par rapport au premier et au dernier évènement. Dans cette forme normale, la durée d'une tuile infinie n'est pas définie. La durée d'une tuile est toujours la somme algébrique des délais du repaire d'entrée au repère de sortie. Pour la tuile de la figure 4.2, nous avons bien $|t| = -d_0 + d_1 + d_2 + d_3 - d_4$.

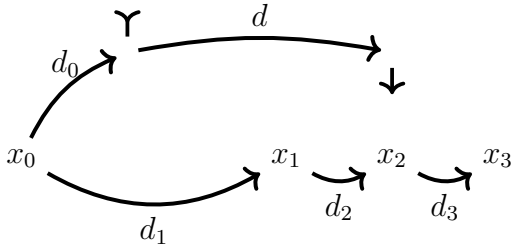


FIGURE 4.3 – Tuile avec la durée explicitée.

Nous allons utiliser une légère variation de cette forme normale, représentée sur la figure 4.3. La durée de la tuile est explicitée, et le contenu ne porte pas le repère de sortie. Cette représentation offre deux avantages : la durée est plus simple à calculer, et elle permet une forme normale pour les tuiles dont la durées est finie mais le contenu est potentiellement infini.

4.3 Tuiles idempotentes et délais

Nous pouvons donc représenter les tuiles temporelles par une paire dont une des composantes est la durée de la tuile, et l'autre est son contenu. Autrement dit, cela revient à représenter une tuile $t = e+ \operatorname{delay}(d)$ par (d, e) où d est le délai de durée $|t|$ et dont e est la tuile idempotente $e = \operatorname{reset}(t)$.

Remarque. Dualement, une autre manière de décomposer les tuiles est par l'équation $t = d + e$ où $d = \text{delay}(|t|)$ et $e = \text{coreset}(t)$. Cette manière n'est pas équivalente car l'évènement indexé par rapport au repère d'entrée est alors le tout dernier de la tuile, et non le premier, comme vu juste avant.

Considérant la réécriture $t = e + \text{delay}(d)$ et en notant cette tuile (d, e) la composition devient :

$$(d_1, e_1) + (d_2, e_2) = (d_1 + d_2, e_1 + (\text{delay}(d_1) + e_2 - \text{delay}(d_1)))$$

Dans la seconde composante, il y a des parenthèses inutiles destinée à mettre en valeur l'expression $d_1 + e_2 - d_1$. Cette dernière est parfaitement équivalente à e_2' où tous les évènements de e_2 seraient décalées d'une durée d_1. En résumé, la durée de la composition de deux tuiles et la composition de leur durées. Et le contenu de leur composition est le contenu de la première tuile, composé ou fusionné avec le contenu de la seconde tuile décalée de la durée de la première tuile.

De cette équation on constate qu'une implémentation des tuiles peut se faire grâce à une implémentation des tuiles idempotentes, et donc, d'un contenu temporisé. Concrètement les tuiles idempotentes sont des ensembles d'évènements placés dans le temps, c'est-à-dire des évènements associées à un instant de l'échelle temporelle, et le délai permet de représenter le déplacement qu'engendre la tuile.

4.4 Définition du type QList

Plus concrètement, les contenus de tuiles peuvent être représentés par ce que nous appelons des « QList ». Cette transformation est illustrée sur la figure 4.4.

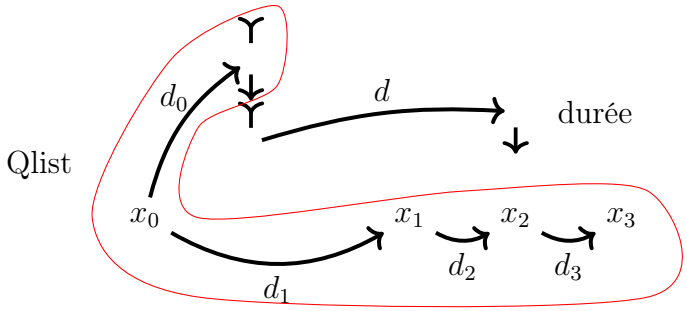


FIGURE 4.4 – Décomposition de t en une durée et une qlist.

Ces Qlist sont des listes dans lesquelles chaque élément est associé à une date. Ce type va nous permettre d'encoder le flux d'évènements temporisés.

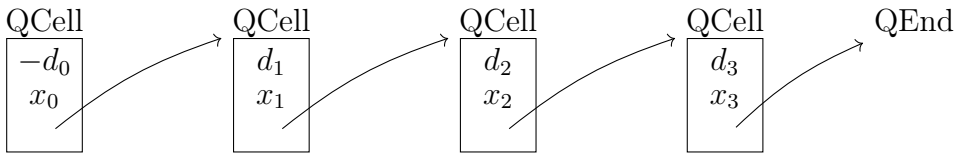


FIGURE 4.5 – Qlist encodant la tuile de la figure 4.4

Il s'agit d'une définition standard de liste dont les cellules sont étiquetés et enrichis d'informations temporelles :

```
data QList d v =
  QCell d v (QList d v)
  | QEnd
```

Aux constructeurs du type `QList d v` nous prêtons la sémantique suivante :

`d` le type de l'échelle temporelle, un ensemble de délais,

`v` le type des évènements

`QEnd` encode un flux temporisé vide,

`QCell d v ql` représente un délai `d` précédant un évènement `v`, `ql` est le flux temporisés dont tous les évènements sont strictement postérieurs à `v`.

Dans l'implémentation que nous proposons ici, les étiquettes associées aux évènements ne sont pas directement les dates absolues, mais les délais qui les séparent des évènements qui les précèdent immédiatement. Autrement dit, il s'agit de date relative au « début » de la cellule. Tous les délais, sauf éventuellement le tout premier, seront strictement positifs.

Naturellement, ce choix d'implémentation est équivalent à l'alternative qui est d'utiliser des dates absolues. En effet, nous pouvons passer des dates absolues aux délais en calculant les différences des dates entre une cellule et son successeur. Et dans l'autre sens, nous pouvons retrouver les dates absolues d'une cellule en sommant tous les délais du début de la liste jusqu'à elle.

4.4.1 Shift

Nous allons commencer par définir le shift car nous en aurons besoin pour implémenter la composition de qlist.

```
shift :: Group d => d -> QList d v -> QList d v
shift _ QEnd = QEnd
shift s (QCell d v ql) = QCell (s <> d) v ql
```

Dans le cas où le constructeur est `QCell` il suffit d'ajouter au délai précédant l'évènement la durée par laquelle nous voulons décaler l'instance de `QList`.

4.4.2 Composition

La composition que nous voulons définir ici est la fusion de deux listes temporisées. Elle est réalisée par la fonction `merge` sur les `QList` :

```
merge :: (Group d, Ord d, Semigroup v) =>
    QList d v -> QList d v -> QList d v
merge QEnd y = y
merge x QEnd = x
merge q1@(QCell d1 v1 ql1) q2@(QCell d2 v2 ql2) =
  case compare d1 d2 of
    EQ -> QCell d1 (v1 <> v2) (merge ql1 ql2)
    LT -> QCell d1 v1 (merge ql1 (shift (inverse d1) q2))
    GT -> QCell d2 v2 (merge (shift (inverse d2) q1) ql2)
```

rappelons que, pour cette fonction, les évènements doivent être équipés d'une composition pour la mise en parallèle, d'où l'instance `Semigroup` requise pour le type `v`. Le temps doit être aussi équipé d'une relation d'ordre total. Afin de pouvoir réaliser la fusion, il faut enfin pouvoir « avancer dans la file dont le premier évènement est le plus distant. Il faut donc une notion d'inverse pour soustraire un temps écoulé.
L'instance de monoïde est maintenant naturelle :

```
instance (Ord d, Semigroup v, Group d) =>
  Monoid (QList d v) where
  mempty = QEnd
  mappend = merge
```

Contrairement au monoïde des mots, que nous pourrions qualifier « d'horizontal », ce monoïde est en quelque sorte « vertical » puisqu'il trie les évènements par ordre d'arrivée.

Remarque. Exactement comme nous l'avons introduit en section 3.3.1, nous faisons l'hypothèse que l'ensemble des évènements $\langle E, \diamond \rangle$ forme un semi-treillis, nous pouvons constater que le type `QList` a aussi une structure de semi-treillis. C'est-à-dire un semi-groupe idempotent.

Théorème 4.4.1. *Si l'ensemble des évènements forment un semi-treillis, alors les listes temporisées composées par* `merge` *forment un semi-treillis.*

Démonstration. Ces propriétés sont héritées de la composition sur les évènements et du fait que la fusion ne garde pas d'information sur la provenance d'un évènement (première ou seconde opérande). □

En ce qui concerne l'ordre associé à ce treillis, il s'agit de l'ordre naturel des tuiles idempotentes. C'est-à-dire que `QEnd` est l'élément maximal, et que $u \leq v$ signifie que v peut être complétée par une liste temporisée e pour former u.

Remarque. L'ordre est dit « naturel » car il est défini au sein même du semi-groupe. Par contre, sa sémantique est contre intuitive. La plus petite liste temporisée est l'élément maximal, et plus une liste est grosse, plus elle est basse dans l'ordre. Une bonne image est « plus un élément est complexe, plus il est lourd, plus il coule ».

Théorème 4.4.2. *La fonction* $\text{shift}(d)$ *est un morphisme de semi-groupe des qlist dans les qlist, pour toute durée* d *(donc son action sera par automorphisme).*

Démonstration. Montrons qu'il s'agit d'un morphisme de semi-groupe, autrement dit que cette fonction vérifie la propriété :

$$\text{shift}(d, \text{merge}(x, y)) = \text{merge}(\text{shift}(d, x), \text{shift}(d, y))$$

La fonction shift applique un délai supplémentaire d au premier évènement de la `QList`. Et `merge` fusionne les deux `QList` sans changer les positions des évènements des deux tuiles par rapport à l'origine. Cette propriété de morphisme découle du fait que $+d$ est un morphisme sur l'échelle temporelle. □

Corollaire 4.4.1. *La fonction* $\text{shift}(d)$ *est un morphisme d'ordre des qlists dans les qlists, autrement dit, il préserve l'ordre naturel.*

4.4.3 Primitives d'accès

Ces `QList` peuvent être déconstruites en utilisant deux primitives, les fonctions `headQList` et `tailQList`. Cette première fonction permet de récupérer le premier évènement présente dans la `QList`, ainsi le délai la précédant. La figure 4.6 illustre la sémantique de ces fonctions :

```
headQList :: QList d v -> Maybe (d, v)
headQList QEnd = Nothing
headQList (QCell d v _) = Just (d,v)
```

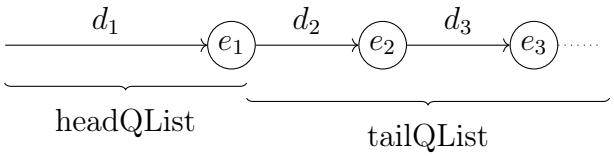


FIGURE 4.6 – Sémantique de `headQList` et `tailQList`

Cette seconde fonction permet de récupérer tout le reste de la `QList` :

```
tailQList :: (Group d, Ord d, Semigroup v) =>
  QList d v -> Maybe (QList d v)
tailQList QEnd = Nothing
tailQList (QCell _ _ l) = Just l
```

Ces deux fonctions vérifient la propriété suivante :

```
propertyHeadTail ql =
  case (headQList ql,tailQList ql) of
    (Nothing, Nothing) -> ql == QEnd
    (Just (d,v), Just qt) ->
       ql == merge (QCell d v QEnd) (shift d qt)
```

où `===` dénote l'équivalence modulo normalisation décrite à la section 4.2.

4.4.4 Préfixe et suffixe

Deux autres fonctions nous seront utiles pour la suite, il s'agit des fonctions permettant de couper une qlist en deux, après une durée d :

```
prefixQList :: (Semigroup v, Group d, Ord d) =>
  d -> QList d v -> QList d v
prefixQList d ql = case headTailQList ql of
  Nothing -> QEnd
  Just ((t',v),r) ->
    if d > t'
    then QCell t' v (prefixQList (inverse t' <> d) r)
    else QEnd
```

```
suffixQList :: (Semigroup v, Group d, Ord d) =>
  d -> QList d v -> QList d v
suffixQList d ql = case headTailQList ql of
  Nothing -> QEnd
```

```
  Just ((d',v),r) -> if d' >= d
                     then shift (inverse d) ql
                     else suffixQList (inverse d' <> d) r
```

```
headTailQList :: QList d v -> Maybe ((d, v), QList d v)
headTailQList QEnd = Nothing
headTailQList (QCell d v l) =
  Just ((d,v),l)
```

La fonction `headTailQList` combine les appels à `headQList` et `tailQList`. Dans le cas où la qlist contient un évènement antérieur au délai `d`, alors la qlist résultante commencera par cet évènement. Dans l'alternative, cet évènement est trop tardif, la résultante sera vide.
Ces deux fonctions vérifient la propriété suivante :

```
propertyPrefixSuffix d x =
      x == merge (prefixQList d x)
                 (shift d (suffixQList d x))
  && prefixQList 0 x == mempty
  && suffixQList 0 x == x
```

4.5 Retrouver les tuiles

Sous l'hypothèse de l'idempotence de la loi de composition sur les évènements, nous avons réalisé une implémentation des listes temporisées. C'est-à-dire une implémentation des tuiles idempotentes. Il nous suffit maintenant de donner une durée aux `QList` pour retrouver les tuiles :

```
data Tile d e =
  Tile { duration :: d
       , content  :: QList d e
       }
```

L'instance de `Semigroup` vient naturellement. Les durées sont ajoutées, les contenus sont fusionnés, mais le contenu temporisé de la seconde tuile doit être décalé de la durée de la première.

```
instance (Group d, Semigroup v, Ord d) =>
  Semigroup (Tile d v) where
  (Tile d1 c1) <> (Tile d2 c2) =
    Tile (d1 <> d2) (c1 <> (shift d1 c2))
```

L'instance de monoïde est triviale :

```
instance (Group d, Ord d, Semigroup v) =>
  Monoid (Tile d v) where
    mempty = Tile mempty mempty
```

On obtient ainsi une implémentation concrète et normalisée des tuiles temporisées.

Remarque (Produit semi-direct)**.** La définition de la composition des tuiles est le produit semi-direct (introduit en section 2.8.2). Il s'agit du produit semi-direct des déplacements temporels et des listes temporisées. L'opérateur shift est l'action par automorphisme d'ordre (introduit en 2.7) du semi-groupe des durées sur l'ensemble des listes temporisées.

Remarque (reset)**.** La définition de `reset` par `x <> inverse x` peut être simplifiée si l'implémentation permet un accès direct à la partie idempotente de la tuile, ce qui est le cas ici. En effet, `x <> inverse x` est la tuile idempotente dont le contenu est le même que celui de `x`.
Ceci permet d'implémenter le reset ainsi :

```
reset (Tile d ql) = Tile mempty ql
```

C'est-à-dire l'on omet la partie « déplacement » de la tuile, pour ne garder que le contenu.

Cela suggère également que le fait d'avoir un groupe n'est pas indispensable. En effet, le reset est une opération plus intéressante que l'inverse pour la programmation. Nous nous retrouvons avec un semi-groupe avec reset (voir 2.4). Une échelle temporelle non réversible pourrait être utilisée à condition de changer l'implémentation du merge. En effet, nous utilisons la structure de groupe des durées pour implémenter le merge efficacement.

4.6 Fonctionnelle d'ordre supérieur

Explorant l'expressivité d'une construction, on en vient tôt ou tard à se poser la question de l'ordre supérieur. Qu'est-ce que cela donne si les valeurs sont des fonctions ?

Nous allons commencer par chercher une éventuelle implémentation d'un opérateur d'application de fonction sur le type `Q = QList d`. Cet opérateur aura le type `Q (Q a -> Q b) -> Q a -> Q b`.

La sémantique de cette application peut être décrite comme suit. La valeur de type `QList` fonctionnelle f_q est synchronisée avec la `QList` argument

a_q. On applique chaque fonction f portée par f_q au suffixe de p_q démarrant à l'instant f. Ce résultat se retrouve dans la qlist final à partir du point f.

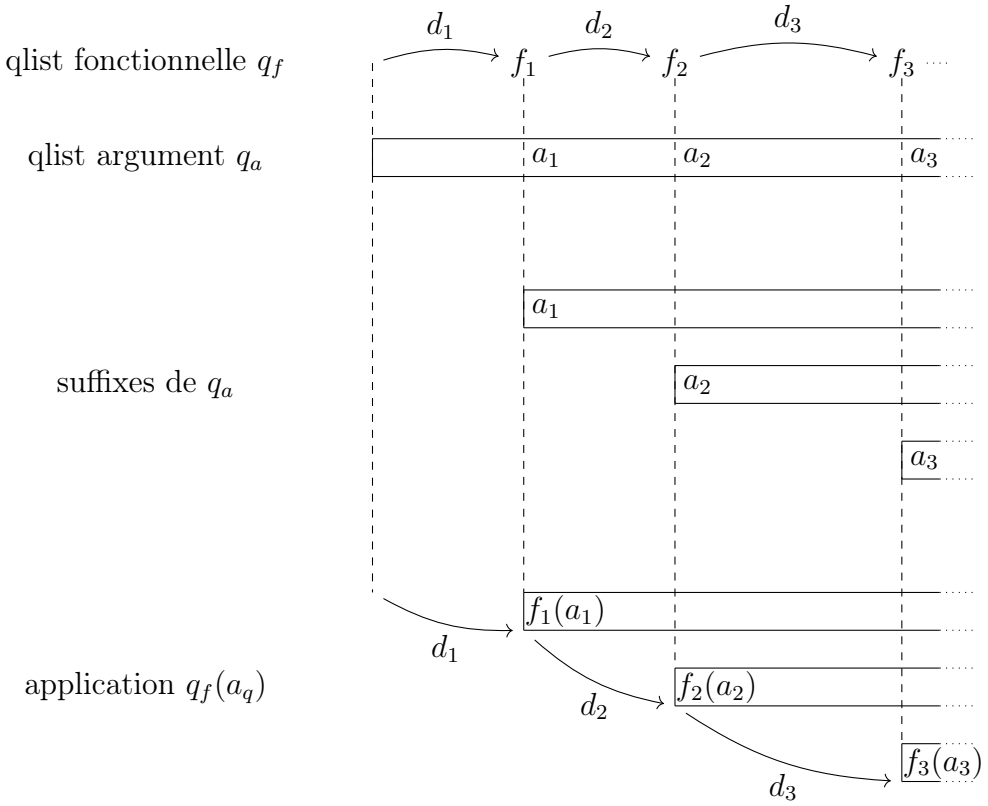


FIGURE 4.7 – Application d'une liste temporisée fonctionnelle

La figure 4.7 décrit l'application d'une liste temporisée fonctionnelle (partie supérieure de la figure) à une liste temporisée (partie centrale de la figure). Le résultat est dans la partie inférieur de la figure. La synchronisation et le découpage du paramètre sont mises en exergue, de même que la synchronisation de la fusion résultante. Il est important de noter qu'il ne s'agit pas là du cas le plus général. En effet, ici, les fonctions ne produisent pas de valeur antérieure à l'instant « zéro » de la liste temporisée, dans le cas général, rien ne l'interdit.

Remarque. Chaque fonction portée par la qlist fonctionnelle se voit passer un argument potentiellement infini. Si ces fonctions retournent un résultat de taille non borné, il est possible que des problèmes de temps de calcul arrivent.

Nous pouvons implémenter l'application d'une qlist fonctionnelle sur une qlist argument ainsi :

```
applyQList :: (Group d, Ord d,
               Semigroup a, Semigroup b) =>
  QList d (QList d a -> QList d b) -> QList d a -> QList d b
applyQList qf qa = case headTailQList qf of
  Nothing -> QEnd
  Just ((d,qf),rqf) ->
    let qa' = suffixQList d qa
    in shift d (merge (qf  qa')
                      (applyQList rqf qa'))
```

Nous pouvons maintenant utiliser ces fonctions pour écrire les versions qui travaillent sur les tuiles au lieu des `QList`. Étant donné que les tuiles portent strictement plus d'informations que les listes temporisées, nous allons devoir en perdre pour utiliser ces fonctions. Nous commençons par convertir les fonctions temporisées sur les tuiles en fonctions temporisées sur les listes temporisées :

```
temporalFunQListToTile  :: Monoid d =>
 (Tile d a -> Tile d b) -> (QList d a -> QList d b)
temporalFunQListToTile (TemporalFun d f) =
  \ql -> content (f (Tile mempty ql))
```

La tuile passée à la fonction `f` a une durée qui ne vient de nulle part. La valeur `mempty`, c'est-à-dire le « 0 » de l'échelle temporelle est arbitraire. Nous aurions aussi pu mettre `undefined`. Il s'agit d'une valeur spéciale qui provoque une erreur à l'évaluation. Elle contamine les calculs à la manière d'un `nan` dans l'arithmétique des `float`.

Il ne reste plus qu'à calculer la liste temporisée des fonctions des listes temporisées dans les listes temporisées :

```
applyTile :: (Ord d, Group d,
              Semigroup a, Semigroup b) =>
  Tile d (Tile d a -> Tile d b) -> Tile d a -> Tile d b
applyTile tf ta =
  let qf = content (fmap temporalFunQListToTile tf)
      qa = content ta
  in Tile mempty (applyQList qf qa)
```

Remarque. Il peut sembler artificiel de définir ces fonctions sur les tuiles, alors qu'il est infiniment plus naturel de le faire sur les listes temporisées. Mais précisément, les tuiles n'offrent aucun pouvoir expressif de plus que les

listes temporisées. En revanche, elles permettent d'ajouter de la structure aux objets que l'on manipule. Il est en fait très naturel de partir de listes temporisées, de les convertir en tuiles pour travailler avec, et enfin de les reconvertir en listes temporisées pour le rendu.

4.7 Tests unitaires automatiques

L'un des avantages de l'algèbre est qu'il est extrêmement aisé de vérifier les propriétés. Ici, nous ne parlerons de pas de preuves formelles, mais de tests unitaires. La bibliothèque QuickCheck[11] d'Haskell est un outil formidable pour réaliser ces tests.

La partie un peu technique est l'écriture des instances de `Arbitrary`. Comme son nom le suggère, il s'agit de décrire une instance arbitraire. Cette interface fait usage de foncteurs applicatifs [1]. Pour paraphraser ce code, on peut dire qu'une tuile arbitraire a une durée arbitraire et une liste temporisée arbitraire.

```
instance (Arbitrary d, Group d, Ord d, Eq d,
          Arbitrary v, Semigroup v, Eq v) =>
  Arbitrary (Tile d v) where
  arbitrary = liftA2 Tile arbitrary arbitrary
```

Pour générer une qlist arbitraire, c'est un peu plus compliqué car on veut laisser au système le contrôle de la taille. La fonction `f` créé une qlist arbitraire de la taille passée en paramètre :

```
instance (Arbitrary a, Semigroup a , Eq a,
         Arbitrary d, Group d, Ord d, Eq d) =>
  Arbitrary (QList d a) where
  arbitrary =
    let f 0 = pure mempty
        f 1 = liftA3 QCell arbitrary arbitrary  (f 0)
        f n = liftA2 (<>) (f (n `div` 2))
                          (f $ (n`mod`2) + (n `div` 2))
    in sized $ f
```

Nous définissons un type sur lequel les prédicats seront effectivement testés. Le type pris est assez explicite. Le type `Sum a` encode le groupe additif sur `a`. Le type `Set` est celui des ensembles, c'est-à-dire un monoïde

1. Les foncteurs applicatifs ont un opérateur de type `T (a -> b) -> T a -> T b`, c'est-à-dire qu'ils définissent une notion d'application de fonction.

idempotent. Le type `Small a` est défini dans quickcheck, il sert à indiquer au système que l'on veut garder les valeurs de `a` petites. Les bugs étant généralement présents dans les cas limites, nous ne voulons pas les rater en testant des valeurs trop grandes.

```
type T = Tile (Sum (Small Int)) (Set (Small Int))
```

Ici, nous écrivons prédicats implémentants les axiomes des semi-groupes inversifs, présentés en section 2.3.2. C'est-à-dire l'associativité, les propriétés de l'inverse, et la commutativité des idempotents (équivalente à l'unicité de l'inverse).

```
assoc_tile :: T -> T -> T -> Bool
assoc_tile x y z = (x <> y) <> z == x <> (y <> z)
```

```
inv_ax_tile :: T -> Bool
inv_ax_tile x =
  let y = inverse x
  in x <> y <> x == x
  && y <> x <> y == y
```

```
idem_com_tile :: T -> T -> Bool
idem_com_tile x y =
  let ex = reset x
      ey = reset y
  in ex <> ey == ey <> ex
```

Grâce aux annotations de types, le système sera capable de retrouver comment générer des instances arbitraire de notre type `T`, et pourra soit vérifier que nos prédicats sont vérifiés, soit exhiber une instance qui ne les vérifie pas

```
testTile = do
  quickCheck $ counterexample "assoc prop." assoc_tile
  quickCheck $ counterexample "inverse prop."  inv_ax_tile
  quickCheck $ counterexample "idem. comm." idem_com_tile
```

Si jamais un bug se trouve là, par exemple si l'ensemble des évènements n'est plus commutatif ou idempotent, comme dans le type :

```
type Tbad = Tile (Sum (Small Int)) [Small Int]
```

À l'exécution le système va trouver un contre-exemple, qui une fois nettoyé des constructeurs `Sum` et `Small` donne[2] :

```
*** Failed! Falsified (after 3 tests):
inverse properties
Tile {duration =  1}, content = QCell 2 [-2] QEnd}
*** Failed! Falsified (after 2 tests):
idempotent commute
Tile {duration = -1, content = QCell 0 [-1] QEnd}
Tile {duration =  1, content = QCell 0 [1] QEnd}
```

Cette méthode permet de vérifier la satisfaction des axiomes par l'implémentation à moindre coût et avec une confiance raisonnable. Les prédicats de vérification des axiomes sont les axiomes écrits en Haskell. Cela permet de vérifier que les instances satisfont les propriété des classes, la non régression, la compatibilité ascendante et descendante, etc.

La limite de cette approche réside dans le fait que les tests ne sont pas exhaustifs. Le générateur d'instance aléatoire va générer des exemples de plus en plus gros à partir des générateurs fournis par le programmeur. Si un problème apparaît uniquement sur une valeur relativement complexe ou grosse, il y a des chances que les tests ne les détectent pas.

4.8 Autres approches

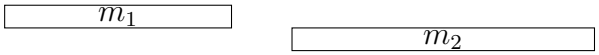
De nombreux autres travaux ont été réalisé dans ce domaine, nous tenons citer les suivant qui sont tout particulièrement proche des nôtres.

4.8.1 Polymorphic Temporal Media (PTM)

Les Polymorphic Temporal Media de Paul Hudak [13] sont des médias temporels. Par temporel nous entendons des médias qui s'exécutent dans le temps. L'ensemble de ces médias, est équipé de deux lois de compositions interne, :+: la composition séquentielle et :=: la composition parallèle (voir la figure 4.8). En dépit de sa simplicité, ce modèle est utilisé dans les bibliothèques Haskore [18] et Euterpea [14], toutes deux écrites en Haskell). Ce paradigme est aussi utilisé dans libaudiostream [28], il est également très proche des graphes série-parallèle [12].

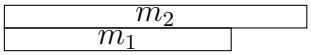
Ce modèle supporte les médias infinis, par exemple le média $m = m_1 :+: m$ est valide et jouable si m_1 a une durée finie. Concernant l'implémentation,

2. Nous omettons également de parler d'un mécanisme permettant de réduire la taille d'un contre exemple.



(a) m_1 et m_2

(b) La composition séquentielle $m_1 :+: m_2$



(c) La composition parallèle $m_1 :=: m_2$

FIGURE 4.8 – Les compositions des PTM

la sémantique « évaluation paresseuse »de Haskell permet au programmeur d'économiser beaucoup de travail et rend le code extrêmement simple.

4.8.2 Tiled Polymorphic Temporal Media (TPTM)

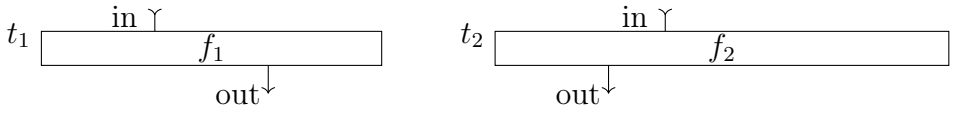
Les Tiled Polymorphic Temporal Media définit par Hudak et Janin[15] offrent une alternative aux PTM en utilisant une seule composition interne notée %. Avec les TPTM, les informations de synchronisation sont embarquées avec le média. Ils offrent une solution au manque de modularité des PTM par la décorrélation du début et de la fin des médias de leur point de synchronisation. Plus précisément, une tuile est un flux enrichi de deux marqueurs temporels appelés entrée ($\curlyvee$) et sortie ($\downarrow$), comme le montre la figure 4.9.

FIGURE 4.9 – Une tuile

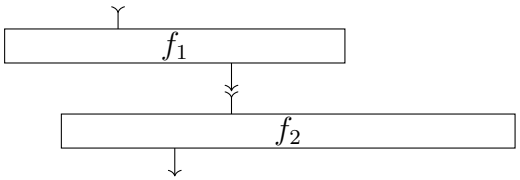
L'implémentation des TPTM n'est pas triviale. Nous avons à satisfaire les contraintes du polymorphisme (qui implique de faire des hypothèses minimales sur le média tuilé). Nous voulons supporter les structures de donnée infinies, et nous voulons que nos tuiles soient jouables en temps réel. Une première implémentation a été faite par Paul Hudak et David Janin sur Euterpea [15] en encapsulant le type `Music a` d'Euterpea dans un type `Tile a` :

```
data Tile a = Tile { inT :: Duration
                   ; outT :: Duration
                   ; musT :: Music a}
```

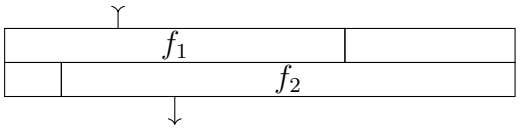
où `Duration` implémente l'ensemble des nombres rationnels. Simple et élégante, cette approche permet d'étudier la programmation par tuilage. Cependant, cette implémentation souffre d'un manque de polymorphisme car les primitives des tuiles sont définies avec les primitives du type `Music`. Le produit est défini comme décrit sur la figure 4.10.



(a) Deux flux enrichis de repères d'entrée et de sortie



(b) Lors de la composition de t_1 avec t_2, le repère de sortie de t_1 est aligné sur le repère d'entrée de t_2



(c) f_1 et f_2 sont complétés avec des silences

FIGURE 4.10 – Les tuiles sont des flux enrichis

4.8.3 Elody

Le langage Elody [32] est un langage de programmation musicale. Il s'agit d'un lambda calcul où les objets primitifs sont des objets musicaux, les notes, les silences. Les opérateurs sont la lambda abstraction, l'application, la mise en parallèle et la mise en séquence. Il existe aussi une version réactive [27].

4.9 Conclusion et perspectives

Dans ce chapitre nous avons achevé de présenter notre modèle de programmation tuilé temporisée. En particulier, la normalisation est permise par le fait que l'espace soit totalement ordonné. Une perspective de recherche serait l'étude de cette normalisation dans le cas ou l'espace n'est pas ordonné, par exemple un espace en deux dimensions. Sans aborder le problème de la forme normale, les dessins dans un espace en deux dimensions sont l'objet du chapitre 6.

Nous avons également montré en quoi le produit semi-direct était pertinent pour l'utilisation de monoïde inversif pour la programmation temporelle. Nous avons également introduit les opérateurs dont nous allons nous servir pour la programmation réactive. Il nous manque cependant quelques briques théoriques à venir. Le chapitre suivant y est consacré.

5

Programmation réactive dans le temps

Au cours du chapitre sur la programmation temporelle hors temps, nous avons introduit un modèle algébrique des flux temporisés. Avec ce modèle, nous pouvons définir un média temporisé par composition de médias plus simples et application de transformations. Ici, nous nous intéressons à la possibilité d'utiliser tout ou partie de ce modèle pour la programmation dans le temps.

Ce chapitre traite principalement de l'implémentation d'un moteur réactif dans un langage de programmation fonctionnel pur. Nous souhaitons appliquer une transformation du flux d'entrée vers un flux de sortie de manière paresseuse. Dans ce chapitre, le média n'est pas encore connu intégralement lorsque l'on en calcule le rendu, mais il n'est pas pour autant complètement indéfini. L'entrée n'est connue que jusqu'à l'instant présent. Nous pouvons donc calculer que partie de la sortie qui dépend de l'entrée connue jusque-là. A fortiori, nous ne pouvons pas calculer une partie de la sortie qui dépend d'un instant postérieur de l'entrée. Une représentation d'une entrée partiellement connue, un exemple est représentée sur la figure 5.1.

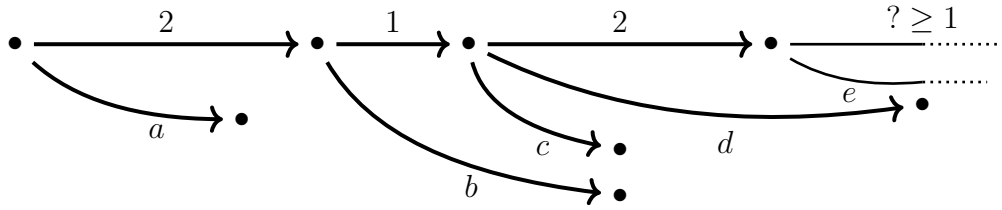


FIGURE 5.1 – Exemple d'entrée partiellement connue

Le programme à proprement parler que notre moteur va exécuter est une fonction des qlists dans les qlists. Une telle fonction va prendre comme

paramètre une qlist, et produire une qlist. Pour faire cela, nous mettons en œuvre une notion de durée explicitement inconnues. Nous mettons également en œuvre une notion d'application « gelée », dans le but de retarder l'application d'une fonction jusqu'à ce que son paramètre soit suffisamment définit pour que le calcul soit possible. L'architecture générale du moteur est décrite par la figure 5.2.

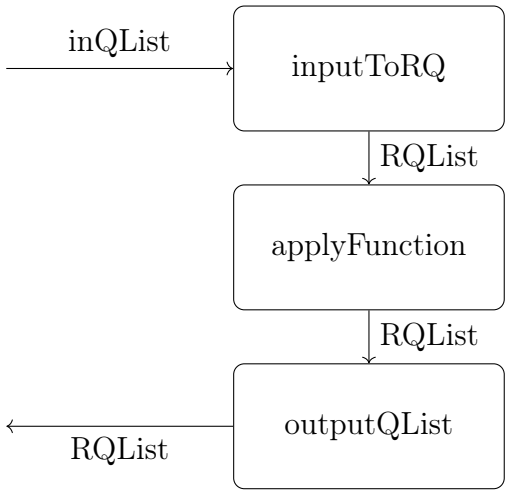


FIGURE 5.2 – Flux de donnée dans moteur réactif

En entrée, le moteur réactif prend une liste d'évènements placés dans le temps, *timed event list*. Cette liste d'évènement est converti en une *inQ-List*, il s'agit d'une représentation interne qui permet la représentation de données partiellement connues. C'est sur cette représentation interne que le traitement des données va avoir lieux, par exemple avec les tuiles d'ordre supérieur décrites à la section 4.6. Le résultat est évalué dans *outputQList*, avant d'être finalement reconverti vers une liste d'évènement temporisée.

5.1 Implémentation

À l'interface d'entrée de ce moteur réactif, nous allons considérer des valeurs temporisées v qui sont émises entre deux marques. Ces marques sont `On` et `Off`. C'est très proche de l'encodage de la musique en `MIDI`, c'est-à-dire un flux de `NoteOn` et `NoteOff` qui marquent le début et la fin de chaque note. À chaque note correspond une paire d'évènements `On` et `Off`. Une mélodie de plusieurs notes qui se chevauchent partiellement est donc représentée par une fusion de mots bien parenthésés, un pour chaque note. Le type des évènements est le suivant :

```
data Event v = On v
             | Off v
```

L'étage d'entrée de notre machine réactive va transformer cette entrée en une qlist. Il s'agit là du type des valeurs qui seront manipulées par le programme que nous allons exécuter :

```
data Atom d v = Atom d v
```

Ce type représente l'émission d'une valeur v sur une durée d.

5.1.1 Listes temporisées réactives

Nous allons utiliser un type proche de celui que nous avons défini au chapitre précédant, les listes temporisées réactives. Nous ajoutons aux `QList` un constructeur `RQRec` qui permet de retarder l'application d'une fonction afin de pouvoir mettre à jour son paramètre *après* l'application [1]. C'est-à-dire ajouter de la paresse à la paresse. Il s'agit d'un codage explicite des « chunks » d'Haskell, des expressions en attente d'évaluation.

```
data RQList d iv v where
  RQEnd :: RQList d iv v
  RQCell :: [Atom d v] -> d -> RQlist d iv v -> RQList d iv v
  RQRec :: Updatable p d iv =>
    (p -> RQList d iv v) -> p -> RQList d iv v
```

Les paramètres du constructeur `RQRec` sont une fonction `f` d'un type `p` dans les qlist réactives ainsi qu'une valeur « `a` » de type `p`. Ce constructeur permet de « geler » l'application `f a` jusqu'à ce que cette valeur soit devenue indispensable. Le but est de retarder cette application aussi longtemps qu'il y a des valeurs indéfinies dans le paramètre `a` de type `p`. La classe `Updatable p d iv` présentée en section 5.1.4 est celle des types `p` qui contiennent des valeurs `iv` temporisée sur `d` qui peuvent être mises à jour.

Le constructeur `RQCell` prend lui en premier paramètre non pas un simple évènement, mais une liste d'`Atom`, c'est-à-dire une liste de valeurs associée à des durées. La valeur de type `d` est la durée (strictement positive) qui sépare les valeurs du reste de la qlist.

1. C'est une chose triviale en programmation impérative, mais impossible en programmation fonctionnelle pure sans artefacts

5.1.2 Tuiles et RQList

Nous souhaitons programmer avec les tuiles, mais notre moteur réactif travaille sur les qlist. Voyons comment passer de l'une à l'autre.

Concernant les tuiles, nous allons les implémenter d'une manière similaire à celle du chapitre précédent. La seule différente est que nous devons ajouter un offset éventuellement négatif au premier élément. En effet, nous ne l'autorisons pas dans la `RQList`.

```
data Tile d iv v = Tile { duration :: d,
                          offset :: d,
                          content :: RQList d iv v
                        }
```

```
tileToRQList :: (Num d, Ord d) =>
    Tile d iv v -> RQList d iv v
tileToRQList (Tile d ad q) = case compare 0 ad of
  LT -> RQCell [] ad q
  EQ -> Q
  GT -> dropQ (-ad) q
```

où la fonction `dropQ` retourne la partie de la qlist en paramètre postérieure à une durée donnée elle aussi en paramètre. Remarquons que si jamais une tuile contient des évènements antérieurs à son marqueur d'entrée, ceux-ci sont simplement ignorés, nous considérons le média coupé à 0. Notons qu'il est possible que cette fonction ne soit que partielle. Dans le cas général, les objets de type `RQList` portent des durées partiellement définies, qui peuvent donc être incomparables.

Afin d'injecter une fonction des tuiles dans les tuiles dans le moteur, nous devons la convertir vers une fonction des qlists dans les qlists. La programmation ressemble au « Continuation Passing Style ».

```
tileFunToRQListFun :: (Num d, Monoid d, Ord d) =>
  (Tile d iv v -> Tile d iv v') ->
    (RQList d iv v -> RQList d iv v')
tileFunToRQListFun f q = case q of
  RQEnd -> tileToRQList (f (Tile mempty mempty RQEnd))
  RQCell la d ql -> RQRec (tileToRQList.f) (Tile d mempty ql)
  RQRec g p -> RQRec ((tileFunToRQListFun f).g) p
```

— Le cas `RQEnd` est assez simple, on construit une tuile à partir de la qlist vide, et on y applique `f`.

— Le cas `RQCell` est plus compliqué car les paramètres du constructeurs ne sont pas forcement entièrement définis, il faut retarder l'évaluation. On construit une tuile à partir de la cellule de la qlist, et on gèle l'évaluation de `f` et la conversion vers les qlist. Remarquons que `Tile` doit être une instance de `Updatable`.

— Le cas `RQRec` est finalement plus simple. L'application `g p` retourne une qlist à laquelle nous voulons appliquer `f`.

5.1.3 InQList

Nous présentons maintenant le premier composant du cœur du moteur réactif : la définition d'une entrée explicite construite au gré de la réception des évènements « start » et « stop ». Il s'agit d'une qlist partiellement définie. Cette entrée est construite à la volée grâce au type de donnée suivant :

```
data InQList d iv where
  InQList :: [(d,iv)] -> d -> InQList d iv ->  InQList d iv
  InQUndef :: InQList d iv
  InQEnd ::  InQList d iv
```

Le constructeur `InQUndef` permet d'expliciter que la suite d'une entrée est toujours inconnue.
Voilà la transformation des qlist d'entrée aux qlist réactives, utilisée en interne :

```
inputToRQ :: InQList d iv -> RQList d iv iv
inputToRQ (InQList ial d iq) =
  RQCell ial d (RQRec inputToQ iq)
inputToRQ InQEnd = RQEnd
inputToRQ InQUndef = error "Causality error"
```

Nous pouvons observer qu'un appel à `inputToRQ InQUndef` créé une erreur d'exécution. En effet, dans notre implémentation, un tel appel est le symptôme d'un problème de causalité, c'est-à-dire que le programme a eut besoin de données qui n'étaient pas encore disponibles. Dans le cas où le constructeur est `IQList`, cela signifie qu'un nouvel ensemble de valeurs a commencé à arriver (`ial`), et un délai `d` commence à s'écouler avant les suivantes. Ces valeurs suivantes arriveront strictement plus tard.

5.1.4 Valeurs updatable

Nous voulons faire en sorte que l'arrivée d'un nouvel ensemble de valeurs temporelle génère une nouvelle entrée qui va remplacer l'obsolète `InQUndef`. Cette entrée est de la forme :

$$\text{InQList } [(X_{i,j}, v_{i,j})_{j \in J_i}] \ Y_i \text{ InQUndef}$$

où :

i est le rang de la salve de valeurs temporisées,

j est le rang de la valeur au sein de la salve,

$v_{i,j}$ est la j-ième valeur de la i-ième salve,

$X_{i,j}$ est la durée j-ième valeur de la i-ième salve,

Y_i est le délai qui sépare la salve de rang i de celle à venir, de rang $i+1$.

Rappelons que nous utilisons un langage pur, c'est-à-dire que les effets de bord ne sont pas permis, et donc, la mise à jour des valeurs ne sont pas possible. Dans cette partie nous décrivons le mécanisme d'émulation de la mise à jour des valeurs. C'est ce que la construction suivante va permettre. Le type `UpdateData` contient les informations nécessaires à la mise à jour des valeurs.

```
type UpdateData d iv = (d->d, InQList d iv -> InQList d iv)
```

Le type `UpdateData` encode les informations nécessaires pour mettre à jour une valeur. Deux choses devront être mises à jour :

— les durées,

— les nouvelles valeurs arrivée dans la `InQList` d'entrée.

Pour les durées, nous allons avoir besoin de dire comment le temps a évolué depuis la dernière mise à jour, c'est l'utilité de la fonction de type `d->d`. Concernant les valeurs dans la `INQList`, il s'agit de la seconde fonction. Le cas d'usage normal est une fonction qui va prendre en paramètre `InQUndef` et qui va retourner `InQList nv d InQUndef` où `nv` sont les nouvelles valeurs reçues. Ce type sera utilisé dans la déclaration de classe suivante :

```
class Updatable p d iv where
  update :: UpdateData d iv -> p -> p
```

Cette classe représente le fait qu'il est possible de faire des mises à jour sur les valeurs de type `d` et les valeurs de type `InQList d iv` encapsulés à l'intérieur d'un type `p`. La fonction `update` prend deux paramètres, une

instance de `UpdateData d iv`, et un paramètre `p`. Le membre gauche de ce premier paramètre contient une fonction des durées dans les durées qui sera appliquée sur toutes les durées non échues dans la valeur `p`. Et son membre droit permet la mise à jour de `InQUndef`, c'est-à-dire l'injection dans la valeur de type `p` des valeurs nouvellement reçues.

Par exemple, voilà l'instance pour les `InQList`, où nous devrons à la fois mettre à jour des durées et injecter de nouvelles valeurs :

```
instance Updatable (InQList d iv) d iv where
  update  (_,_) InQEnd = InQEnd
  update  (f,nv) (InQList al d q) =
    InQList (fmap (\(d1,v1) -> (f d1,v1)) al)
            (f d)
            (update (f,nv) q)
  update  (_,nv) InQUndef = nv InQUndef
```

- La première clause, traitant le cas `InQEnd` est trivial, dans ce cas la qlist est entièrement définie, il n'y a rien à faire.
- La seconde clause est celle permettant la mise à jour des durées. Ici, `al` est une liste de paire `(d,iv)`. La fonction `fmap` applique la fonction passée en paramètre à chaque paire, utilisant la fonction `f` pour mettre à jour les durées de chaque valeur, et la durée jusqu'aux prochaines valeurs.
- La troisième et dernière clause sera appelée lorsque des nouvelles valeurs arrivent. La fin de la tuile était non définie, et la fonction `nv` va retourner le constructeur `InQList` avec les nouvelles valeurs.

Remarque. La toute dernière clause est la raison pour laquelle le type des valeurs d'entrée `iv` est omniprésent. En effet, c'est à cet endroit que sont « injectée » les valeurs fraîchement arrivées. Une valeur de type `InQList` est encapsulée dans un constructeur `RQRec` par la fonction `inputToQ`. Si nous voulons pouvoir effectuer un « update » sur cette `RQList`, il faut pouvoir prouver que dans le constructeur `RQRec`, le paramètre de type `p` vérifie `Updatable p d iv`. Il faut donc que ces valeurs apparaissent dans le type de `RQList`.

```
instance Updatable (RQList d iv v) d iv where
  update _ RQEnd = RQEnd
  update (f,nq) (RQCell al d q) =
    RQCell (update (f,nq) al) (f d) (update (f,nq) q)
  update (f,nq) (RQRec g p) = RQRec g (update (f,nq) p)
```

Nous pouvons définir l'instance `Updatable` sur tous les types dont nous aurions besoin pour écrire des fonctions de transformations de l'entrée. Par exemple, pour écrire des fonctions qui prennent plus d'un paramètre, nous pouvons définir les instances sur les n-uplets :

```
instance (Updatable p1 d iv,
          Updatable p2 d iv) =>
  Updatable (p1,p2) d iv where
    update u (p1,p2) = (update u p1,
                        update u p2)

instance (Updatable p1 d iv,
          Updatable p2 d iv,
          Updatable p3 d iv) =>
  Updatable (p1,p2,p3) d iv where
    update u (p1,p2,p3) = (update u p1,
                           update u p2,
                           update u p3)
```

Quelques types de base, pour lesquels il n'y a rien à faire, mais d'un point de vue génie logiciel, il est précieux de pouvoir les traiter comme les autres.

```
instance Updatable Integer d iv where
  update _ = id

instance Updatable Bool d iv where
  update _ = id

instance Updatable Int d iv where
  update _ = id
...
```

Les paires :

```
instance (Updatable a d iv, Updatable b d iv) =>
  Updatable (a,b) where
    update u (a,b)  = (update u a, update u b)
```

Tous les foncteurs (les listes, maybe...) :

```
instance (Functor f, Updatable p d iv) =>
  Updatable (f p) where
    update u f = fmap (update u) f
```

Remarque. Cette instance n'est peut-être pas celle que nous souhaitons sur certains foncteurs, par exemple `Either a b`. En effet, ce dernier type est fonctoriel sur la valeur de type `b`. Utilisant un paramètre de ce type, nous pourrions souhaiter qu'à la fois `a` et `b` soient `Updatable`, avec l'instance :

```
instance (Updatable a d iv, Updatable b d iv) =>
  Updatable Either a b where
    update u (Left a)  = Left (update u a)
    update u (Right b)  = Right (update u b)
```

5.2 Durées partiellement définies

Comme mentionné, les durées dont nous avons besoin doivent supporter des durées inconnues, ainsi que des opérations arithmétiques sur ces durées. Nous proposons de l'implémenter comme des fonctions affines sur des variables de deux types :

```
data ID v = ValueID v
          | DelayID Integer
```

Le constructeur `ValueID` représente la durée d'une valeur temporisée, et `DelayID` représente l'attente entre deux réceptions de valeurs temporisées. Les durées des valeurs sont indexées par les valeurs elles-mêmes `v`, et les délais sont indexés par un entier.

Remarque. Nous pouvons indexer les durées des valeurs par les valeurs même puisque comme nous avons fait l'hypothèse d'idempotence des valeurs, deux valeurs identiques ne peuvent se trouver en concurrence. À chaque instant il y a au maximum une seule instance de chaque valeur.

Les durées sont modélisées par des fonctions affines à plusieurs variables. Concrètement, nous les implémentons ainsi :

```
data Affine d id = Affine d [(d,id)]
```

Ce type de donnée encode des expressions de la forme :

$$o + c_1 \times x_1 + c_2 \times x_2 + \ldots + c_n \times x_n$$

où o est une constante, les c_i sont des coefficients et les x_i des variables.

Exemple. Par exemple, une expression de la forme :

$$2 + 3d_5 + 1d_a$$

où d_5 est la durée du $5^{\text{ème}}$ délai et d_a est la durée de la valeur 'a', la valeur de type Affine sera :

```
Affine 2 [(3,DelayID 5),(1,ValueID 'a')]
```

L'addition se définit ainsi :

```
plusAF :: (Num d, Eq d,Ord i) =>
    Affine d i -> Affine d i -> Affine d i
plusAF (Affine d1 l1) (Affine d2 l2) =
  Affine (d1 + d2) (mergeL (+) l1 l2)
```

Où la fonction mergeL a le type :

```
mergeL :: Eq id => (a -> a -> a) -> (a,id) -> (a,id) -> (a,id)
```

et fusionne les deux listes en utilisant la fonction donnée en paramètre pour le cas où une étiquette est présente dans les deux listes.

Sur ces fonctions affines nous définissons en plus un produit partiel, à savoir la multiplication par une constante :

```
multAF :: (Eq d, Num d) =>
  Affine d i -> Affine d i -> Affine d i
multAF (Affine 0 []) _ = (Affine 0 [])
multAF _ (Affine 0 []) = (Affine 0 [])
multAF (Affine d1 []) (Affine d2 l2) =
    Affine (d1*d2) (fmap (\(d,i) -> (d1*d,i) l2)
multAF (Affine d1 l1) (Affine d2 []) =
    Affine (d1*d2) (fmap (\(d,i) -> (d2*d,i) l1)
multAF _ _ = error "Affine product undefined"
```

Afin de pouvoir faire des tests sur les durées même si celles si ne sont pas complètement définies, les variables dans la fonction affines ne représentent pas la totalité de la durée, mais plutôt leur durée restante. C'est-à-dire que les durées sont mesurées à partir de l'instant présent. Ces mises à jour se font de deux façons.

Lorsqu'un temps δ_t s'est écoulé, nous remplaçons toute les variables X par $\delta_t + X$.

```
shiftAffine :: (Eq d, Num d,Ord i) =>
    d -> Affine d i -> Affine d i
shiftAffine _ (Affine dd []) = Affine dd []
```

```
shiftAffine d (Affine dd l) =
  Affine (foldl (+) dd (fmap (\(di,i) -> di * d)) l
```

Lorsqu'une durée d'entrée arrive à échéance, il suffit de la supprimer de la fonction affine, c'est-à-dire mettre la variable à 0.

```
setToZeroAffine :: (Num d,Ord i) =>
    i -> Affine d i -> Affine d i
setToZeroAffine i (Affine dd ld) =
  Affine dd (filter (\(_,x) -> x /= i) ld)
```

Nous filtrons la liste des variables pour enlever celles qui sont dans la listes l de variables échues.

Ce faisant, toutes les durées inconnues sont nécessairement positives et cela autorise un traitement relativement simple des fonctions affines.

5.3 Moteur réactif

La conversion des entrées datées (avec*timestamp*) dans le types `InQList` est gérée par le cœur du noyau que nous allons exposer ci-après. Il est basé sur la notion d'état réactif. Un tel état est initialisé par une fonction f sur les tuiles (interprété comme une fonction sur les listes temporisées réactives). Et cet état sera manipulé par une boucle qui converti le flux temporisé d'évènements d'entrée en un flux temporisé de sortie en utilisant la définition donnée par la fonction f. Le type encodant les états réactifs est défini par :

```
type Dur d = Affine d ID
data QState d iv v =
  QState { rank :: Int,
           date :: d,
           outputQList :: RQList (Dur d) iv v,
           currentOutputValue :: [(Dur d, v)]
         }
```

```
initQState f =
  let g q = toQList (f (Tile 0 0 (inputToQ q)))
  in QState 0 0 (QRec g InQUndef) []
```

Le premier paramètre de `QState` est le rang de la dernière salve de valeurs temporelles reçues. Le second paramètre est le timestamp courant,

le troisième est la sortie, cette valeur sera évaluée au fur et à mesure que l'entrée arrivera. Et le dernier paramètre est la liste des valeurs de sortie qui sont actives et qui attendent leur clôture.

Le type de la fonction principale du moteur réactif est le suivant :

```
updateState ::
  (Num d, Ord d,
   Ord iv,
   Ord v,Semigroup v) =>
     QState d iv v ->
     [(d,iv)] ->
     (QState d iv v, [(d,v)], WakeUpOrder d)
```

- Le premier paramètre est l'état de la machine, de type `QState`,
- le second paramètre est une liste de valeurs associés à leur durée,
- la valeur de retour est composée :
  - du nouvel état,
  - des valeurs de sortie associées à leur durée,
  - d'une requête à la boucle pour une date de réveil.

Le type `WakeUpOrder` est isomorphe à `Maybe`. Il permet à `updateState` de demander à son environnement d'exécution de le réveiller après une durée donnée écoulée. Cela est utile si elle doit produire un évènement à une date ultérieure connue.

```
data WakeUpOrder d = WakeUp d
                   | NoWakeUp
```

Nous pouvons décrire l'algorithme global de la boucle réactive de la manière suivante. Ces étapes sont effectuées lors de chaque réception d'évènements et chaque réveil programmé :

1. mise à jour de la `RQlist` de la sortie et des valeurs en cours de production en fonction du temps écoulé depuis la dernière mise à jour,
2. création de la nouvelle salve d'évènement (évènements `On`) et remplacement de la valeur `InQUndef`,
3. affectation définitive des variables dont les `Off` correspondant ont été reçus,
4. éventuellement, effectuer l'application de fonction du constructeur `RQRec`, suppression des `Atom` de la sortie programmée avec les insertions correspondance dans la liste de sortie,

5. produire les évènements `On` et `Off` correspondants, en les retirant de la liste de sortie,
6. et, enfin, si besoin, programmer un nouveau réveil.

En d'autre mots, la fonction sur les tuiles f a été transformée en un programme réactif en temps réel qui agit sur les évènements.

Remarque. La production en sortie d'un évènement à une date antérieure à la date courante provoque une erreur. Les fonctions misent en œuvre doivent être temporellement causales. Cette vérification n'est que dynamique.

5.4 Implémentation et expérimentations

Un prototype a été réalisé, et a été utilisé par un musicien professionnel. Un compte rendu de cette expérience est décrit dans la publication [7]. Par ailleurs, des enregistrements de ces productions sont disponibles sur la page web du projet [2].

5.5 Autres approches

En plus des extensions réactives que nous avons évoqué au chapitre précédant, nous pouvons citer quelques projets de recherche :

FAUST (Functional AUdio STream)

Le langage FAUST [31], projet du Grame, est un langage fonctionnel dédié à la synthèse sonore et le traitement audio en temps réel. Il s'agit d'un langage temps réel : tout programme effectue ses calculs en temps bornés, quels que soient ces entrées.

OSSIA

OSSIA est un logiciel et un projet de recherche [10] proche de celui que nous menons à bien des égards [3].

Il s'agit d'un logiciel visant à permettre la spécification et l'exécution de scénarios interactif. La principale différence avec les travaux exposés

2. `https://poset.labri.fr/interpolations/`
3. le doctorant qui écrit ces lignes a partagé pendant trois ans le bureau du principal contributeur de ce projet.

dans ce document est le modèle sous-jacent. Alors que nous avons une approche algébrique du modèle, dans OSSIA il s'agit d'un graphe manipulé à l'aide d'une interface utilisateur graphique. La fenêtre principale représente l'écoulement du temps de la gauche vers la droite. Les arêtes représentent des délais temporels, et les nœuds des points de synchronisations. Les nœuds peuvent agir comme une structure de contrôle de l'exécution et inhiber une partie du graphe. L'aspect important ici est que la durée que représente une arête n'est pas forcément connue de manière statique. La fin d'une durée peut être marquée par la réception d'un évènement. Cependant, ces durées dynamiques ne sont pas directement mesurables et réutilisables.

5.6 Conclusion et perspectives

Dans ce chapitre nous avons posé notre approche de la programmation temporelle interactive. Un angle mort majeur de cette thèse est la gestion de la causalité. En effet, il nous semble très important d'être à même de gérer cette contrainte de manière statique, en amont, plutôt que de manière dynamique, trop tard.

Par ailleurs, un autre aspect peu exploré pour l'instant est la caractéristique des valeurs. Nous utilisons soit des valeurs instantanées, c'est-à-dire des évènements, soit des valeurs continues, qui évoluent au cours du temps. Autrement dit, les tuiles que nous définissons fonctionnent soit de manière asynchrone, soit de manière synchrone, mais nous n'avons pas de moyen élégant d'avoir les deux comportements simultanément.

6

Programmation spatiale

Dans ce chapitre, nous allons aborder ce que nous appellons la programmation spatiale, c'est-à-dire la programmation dont l'exécution produit des dessins. Notre approche peut être illustrée de manière plutôt simple en analysant la sémantique de programmes tortue LOGO qui permettent de définir des dessins en programmant les mouvements d'une tortue. Il se trouve que ce langage de dessins offre une vue pertinente sur la sémantique mathématiquement bien définie que nous proposons.

6.1 Un langage graphique historique

Le langage LOGO est un langage générique de la fin des années 1960 défini dans un but pédagogique. Il incorpore notamment un langage de contrôle d'un robot capable de peindre sur le sol. Il a été popularisé par la suite sous le nom de la tortue LOGO.

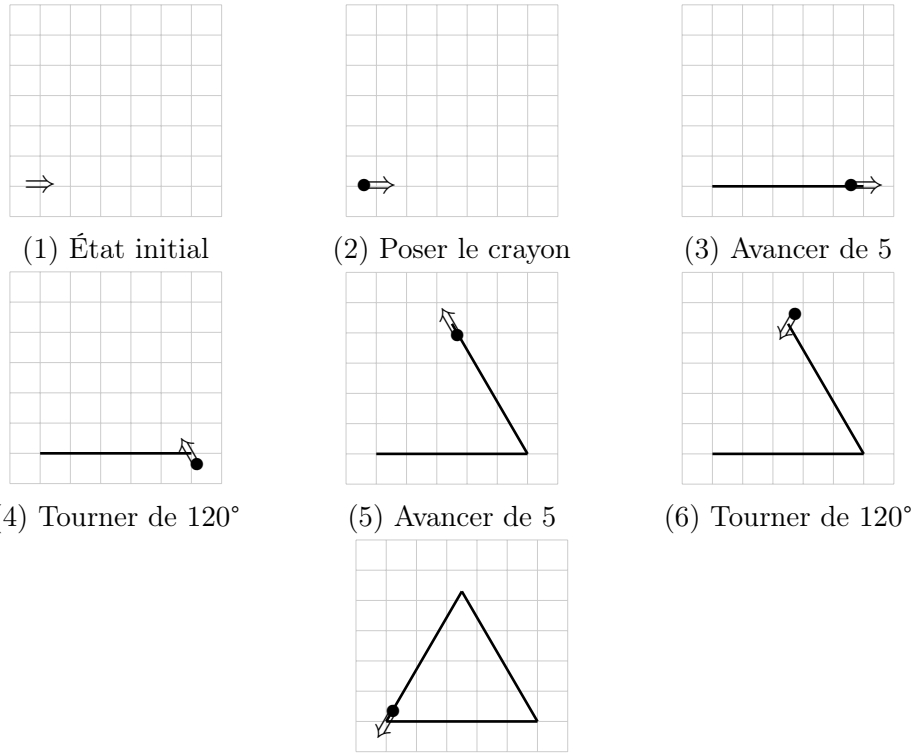
Dans ce langage applicatif, il est possible de programmer les déplacements d'une tortue équipée d'un crayon. Les traces de ce crayon étant la sémantique du programme. La tortue LOGO est le concept le plus populaire de ce langage. Tellement populaire qu'il a été incorporé dans de nombreux autres langages. Il s'agit d'un DSL pour décrire des dessins en deux dimensions. Le modèle sous-jacent à ce langage peut être décrit ainsi :

- Une feuille infinie
- Une tortue avec des stylos de couleur, qui peut :
    - avancer sur une distance donnée, éventuellement négative pour reculer,
    - tourner sur elle-même d'un angle donné,

- poser un stylo de couleur sur la feuille,
- lever son stylo de la feuille,

- Les déplacements du stylo posé sur la feuille laissent une trace de la couleur du stylo,
- l'ensemble des traces laissées par la tortue, ainsi que son déplacement global définissent la sémantique d'un programme.

La figure 6.1 donne un exemple de l'exécution d'un tel programme.



(1) État initial (2) Poser le crayon (3) Avancer de 5

(4) Tourner de 120° (5) Avancer de 5 (6) Tourner de 120°

(7) Avancer de 5

FIGURE 6.1 – Exécution tortue LOGO

Ce modèle est très inspiré de notre monde et quiconque ayant une tortue coopérative pourra simplement trouver une feuille infiniment grande pour le mettre en œuvre. La sémantique de ce langage est en quelque sorte opérationnelle. Le programmeur doit décrire ce que la machine doit faire pour produire ce qu'il souhaite, plutôt que ce qu'il souhaite obtenir directement (sémantique dénotationnelle). Autrement dit, le programmeur décrit le processus de réalisation du dessin, et non pas, comme dans une approche dénotationnelle, le dessin tel qu'il devra être à la fin.

En l'état, la complexité du dessin qu'il est possible de produire — complexité en termes de nombre de traits — est bornée par la taille du programme. Aussi, ce langage est généralement enrichi, comme c'est le cas dans LOGO, de variables, d'affectations, de structures de contrôle, de tests, de conditionnelles et de boucles.

Comme nous nous en doutons à l'exposition de ce modèle, s'il est aisé de dessiner un triangle ou un carré, il est impossible de dessiner un cercle. Le mieux que nous puissions faire est de l'approcher par un polygone dont les côtés sont tellement petits qu'ils donnent l'illusion de définir un cercle.

Nous pouvons observer que les « programmes de tortue » agissent à la fois sur l'espace des positions possibles du crayon et sur les figures qui sont dessinées. Ensemble, les positions et les dessins forment un état qui est finalement transformé par les programmes

Les actions sur les positions sont appelé *déplacements*, les actions sur les figures sont appelés *dessins*, ensembles ils forment le modèle sémantique des « programmes de tortue » que l'on cherche à définir.

6.2 Syntaxe

Le langage de la tortue LOGO, dans sa forme minimale que nous allons étudier ici, consiste en trois primitives et un opérateur de composition. Nous allons implémenter les instructions en Haskell :

`TogglePen` lève le stylo s'il est posé sur la feuille, le pose sinon,

`Walk d` avance d'une distance d (qui peut être négative),

`Turn a` pivote d'un angle a.

`P1 , P2` composition `P1` et `P2` de manière séquentielle.

Remarque. Nous préférons substituer aux instructions `PenUp` et `PenDown` présentes dans les implémentations standards du langage LOGO une seule instruction `TogglePen`. La différence tient au fait que `TogglePen` est réversible, contrairement à `PenUp` et `PenDown`. Cela nous permettra plus tard de former un groupe.

Le code suivant implémente les types nécessaires à la représentation d'un programme :

```
data Instruction = TogglePen
                 | Turn Angle
                 | Walk Length
type Program = [Instruction]
```

Très simplement, le type `Instruction` a trois constructeurs différents dont la sémantique intuitive est décrite plus haut. La composition séquentielle de ces instructions est implémentée par l'utilisation d'une liste, ce qui est fait dans le type `Program`. Plus formellement, les programmes sont implémentés par le monoïde libre formé sur l'alphabet des instructions, cet alphabet est virtuellement infini à cause de la présence de paramètres dans $\mathbb{R}$ pour `Walk` et $\mathbb{R}/2\pi\mathbb{R}$ pour `Turn`.

Exemple. Dessiner un triangle. En supposant que dans l'état initial la tortue a le crayon baissé, le programme suivant dessine un triangle équilatéral de côté 1.

```
prog_triangle= [ Walk 1, Turn (2*pi/3),
                 Walk 1, Turn (2*pi/3),
                 Walk 1]
```

6.3 Travail préparatoire à la sémantique

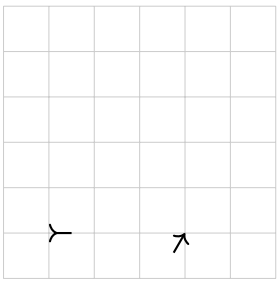
Définis par des listes d'actions de base, les « programmes de tortue » forment un monoïde où la composition séquentielle des programmes est modélisée par la concaténation de listes. Visant la définition d'un modèle sémantique compositionnel pour ces programmes, nous cherchons donc un monoïde sémantique.

La tortue effectue des déplacements en changeant de position, et des dessins en créant de nouveaux traits de crayon. Ces deux actions constituent les éléments de notre sémantique. La figure 6.2 illustre ces sémantiques. Les conventions graphiques sont les mêmes que celles utilisées au chapitre 3. Nous gardons les même symboles pour désigner les marques d'entrée et de sortie de la tuile. Les déplacements sont constitués de translations et de rotations dans le plan. La composition est là encore effectuée tout d'abord en synchronisant la marque d'entrée de l'opérande droite sur la marque de sortie de l'opérande gauche, et en fusionnant les dessins.

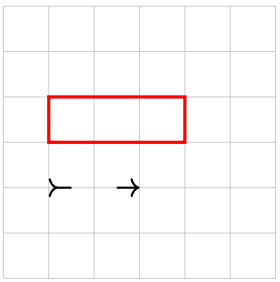
6.3.1 Déplacements

Formellement, l'état de la tortue, que nous appellerons « position », est défini par un triplé composé d'un booléen (vrai quand le crayon est en position basse, et faux sinon), un vecteur à deux dimensions (les coordonnées du crayon dans le plan), et un angle (l'orientation de la tortue).

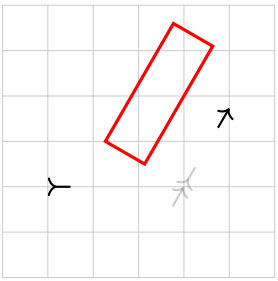
Le type `Move` est le type des déplacements, c'est une fonction des états de la tortue dans les états de la tortue, c'est-à-dire qu'une valeur calcule



(a) Tuile a, un déplacement et une rotation



(b) Tuile b, un déplacement et un dessin



(c) Tuile $a + b$

FIGURE 6.2 – Composition de deux sémantiques

un état final étant donné un état initial. Autrement dit, une fonction qui transforme les états. Il s'agit de la composition fonctionnelle renversée afin que le premier mouvement effectué soit à gauche, et le second à droite. Nous allons anticiper sur la suite et l'implémenter dès maintenant comme le groupe symétrique sur les positions :

```
type Point = (Length,Length)
data Position = Position { position  :: Point
                         , direction :: Angle
                         , penDown   :: Bool }
type Move = SymGroup Position
```

Le type `SymGroup` implémente les bijections des positions dans les position. C'est l'objet de la section suivante.

6.3.2 Codage du groupe symétrique

Un groupe connu est l'ensemble des bijections d'un ensemble E dans lui-même munis de la composition. Ce groupe s'appelle le groupe symétrique

de E et on le note $S(E)$. Nous ferons usage de ce groupe dans la suite. Dans le cas où E est fini, il s'agit du groupe des permutations de E.

Tout comme dans de nombreux langages de programmations, il est difficile de faire des calculs sur les fonctions. Nous pouvons néanmoins implémenter les bijections de manière générique, utilisant le type `->` d'Haskell de la manière suivante :

```
data SymGroup a = SymGroup  {apply  :: (a -> a),
                             ifun :: (a -> a)}
-- with the property (in pseudo Haskell)
-- f (SymGroup g ig) = g  . ig == id
--                   && ig . g  == id
instance Semigroup (SymGroup a) where
  (SymGroup g ig) <> (SymGroup g' ig') =
    SymGroup (g' . g) (ig . ig')
instance Monoid (SymGroup a) where
  mempty = SymGroup id id
instance InverseSemigroup (SymGroup a) where
  inverse (SymGroup g ig) = SymGroup ig g
instance Group (SymGroup a)
```

Dans ce codage du groupe symétrique on code la paire de fonctions (f, f^{-1}) par `SymGroup f if`, si `f` code f et `if` code f^{-1}. Dans la composition, on a l'ordre standard de la composition f puis g est codé $g \cdot f = \lambda x.g(f(x)$. La fonction inverse est un anti-morphisme, il faut donc renverser la composition $(g \cdot f)^{-1} = f^{-1} \cdot g^{-1}$. Pour l'inverse, il suffit de permuter la fonction et son inverse.

À charge pour le programmeur d'implémenter les générateurs qu'il souhaite. Ainsi, l'inverse est pré-codé. Nous obtenenons le sous-groupe finiment engendré.

6.3.3 Dessin

Une figure est définie par un ensemble de segments, c'est-à-dire un ensemble de paire de points[1]. Pour une question de simplicité, nous allons représenter cet ensemble par une liste. Ces figures vont avoir une structure de monoïde. En effet, nous auront besoin d'une figure vide pour l'initialisation, et d'une composition pour l'ajout d'un segment ou d'une autre figure.

1. non ordonnés au sein de la paire

```
type Line = (Point,Point)
data Figure = Figure [Line]
instance Semigroup Figure where
  (Figure a) <> (Figure b) =
    Figure (a <> b)
instance Monoid Figure where
  mempty = Figure []
```

Nous remarquons que les figures que nous venons de définir ont une position absolue dans l'espace. C'est suffisant pour une première implémentation, mais puisque nous cherchons à caractériser la sémantique des programmes, il nous faut pouvoir tracer un dessin à partir d'une position arbitraire. Nous ajoutons donc une légère sur-couche aux figures pour pouvoir injecter une position initiale.

Le type `Draw` est habité par des fonctions des états de la tortue dans les dessins (les ensembles de segments, modélisés ici par une liste de paires de points), les fonctions instances de `Draw` calculent une figure étant donné un état initial. Nous en profitons pour déclarer la sémantique du programme vide, qui est formée à partir de la fonction identité pour les déplacements, et la fonction constante qui renvoie une liste vide de lignes pour le dessin, ainsi que l'état initial de la tortue :

```
data Draw = Draw (Position -> Figure)
instance Semigroup Draw where
  (Draw a) <> (Draw b) = Draw (\p -> a p <> b p)
instance Monoid Draw where
  mempty = Draw (\_ -> mempty)
```

Il s'agit de l'extension point à point pour un monoïde.

6.4 Sémantique

La sémantique d'un programme sera encodée par une paire de type `Semantics` formée d'un déplacement (le déplacement global de la tortue) et d'une figure (un ensemble de segments).

```
data Semantics = Semantics (Move,Draw)
emptySemantics :: Semantics
emptySemantics = Semantics (mempty, mempty)
semGetMove (Semantics (m,_)) = m
semGetDraw (Semantics (_,d)) = d
```

L'état du programme est quant à lui également représenté par une paire contenant l'état de la tortue, et l'état de la feuille :

```
type State = (Position, Figure)
initState :: State
initState = (initPosition, mempty)

initPosition :: Position
initPosition = Position { position = (0,0)
                        , direction = 0
                        , penDown = True }
```

Ces valeurs sont tout à fait arbitraires. La tortue commence aux coordonnées $(0, 0)$ de la feuille, regardant dans la direction 0, et le crayon posé.

6.4.1 Sémantique des instructions

La fonction `moveI` va calculer le changement d'état de la tortue induit par l'évaluation de chacune des instructions. Puisque nous implémentons le groupe symétrique, nous donnons également les réciproque des fonctions :

```
moveI :: Instruction -> Move
moveI TogglePen = SymGroup togglepen togglepen
  where togglepen =
          \s -> s { penDown = not (penDown s)}
moveI (Turn a)  = SymGroup (turn a) (turn (-a))
  where turn a = \s ->
          s { direction = direction s + a}
moveI (Walk d)  = SymGroup (walk d) (walk (-d))
 where walk d = \s ->
         let (x,y) = position s
             a     = direction s
         in s { position = (x + d * cos a, y + d * sin a)}
```

De la même manière, la fonction `drawI`, va calculer le dessin engendré par chacune des instructions. C'est-à-dire une liste vide de segments dans tous les cas, sauf s'il s'agit d'avancer (ou reculer) avec le crayon posé sur la feuille :

```
drawI :: Instruction -> Draw
drawI TogglePen = mempty
drawI (Turn _)  = mempty
```

```
drawI i@(Walk d) = Draw (\s ->
  if penDown s
  then Figure [(position s,
                position.(apply.moveI) i $ s)]
  else mempty)
```

La fonction `instructionToSemantics` calcule simplement la sémantique d'une instruction, réutilisant les deux fonctions précédemment définies.

```
instructionToSemantics :: Instruction -> Semantics
instructionToSemantics i = Semantics (moveI i, drawI i)
```

Nous en avons fini pour les instructions, voyons maintenant comment les composer.

6.4.2 Sémantiques des programmes

C'est maintenant que l'on en arrive à la partie la plus intéressante du code, l'opérateur `<>` va calculer la sémantique de deux programmes composé séquentiellement.

```
instance Semigroup Semantics where
  (Semantics (m1,d1)) <> (Semantics (m2, Draw d2)) =
    Semantics (m1 <> m2, d1 <> Draw (d2 . apply m1))
```

Si nous avons deux sémantiques `(m1,d1)` et `(m2,d2)`, le mouvement résultant de leur composition consistera à d'abord effectuer le mouvement `m1`, et ensuite celui de `m2`[2]. Pour ce qui concerne les dessins, la tortue va effectuer `d1` à partir de sa position courante, puis elle va se déplacer du mouvement engendré par le premier programme (`m1`), puis elle dessinera `d2`.

Nous pouvons calculer la sémantique d'un programme de la manière suivante :

```
semantics :: Program -> Semantics
semantics p = mconcat (map instructionToSemantics p)
```

Cette fonction combine dans l'ordre (grâce à `mconcat`) la sémantique de chacune des instructions (calculée grâce au `map`).

La fonction `mconcat` est une fonction qui évalue une composition dans un monoïde dont les termes sont les éléments d'une liste.

2. La composition de `SymGroup` est renversée.

```
mconcat :: Monoid m => [m] -> m
mconcat [] = mempty
mconcat (x:xs) = x <> mconcat xs
```

6.4.3 Évaluation des programmes

Nous pouvons maintenant définir deux nouvelles fonctions. La première est `applySemantics`, elle va appliquer la sémantique d'un programme à l'état d'un programme. La seconde permettra de calculer effectivement une figure à partir d'un programme.

```
applySemantics :: Semantics -> State -> State
applySemantics (Semantics (m, Draw d)) (p,f) =
  (apply m p, d p <> f)
```

Dans les paramètres de la fonction, `m` est la fonction de déplacements engendrée par la sémantique, et `p` la position initiale dans l'état. La position résultante est donc `m p`. La figure `f` est la figure déjà présente faite, et `d` est de type `Drawing = Position -> Figure`, donc, `d p` est le dessin décrit par la sémantique à partir de la position `p` de la tortue. Grâce à cette fonction, nous pouvons maintenant calculer une figure à partir d'un programme. La sémantique des dessins d'un programme tortue est définie comme étant la seconde projection de la sémantique complète du programme.

```
runProgram :: Program -> Figure
runProgram p =
  let ps = semantics p
  in  snd (applySemantics ps initState)
```

Observons que la sémantique des dessins des « programmes de tortue » est légèrement plus complexe à définir que la sémantique des déplacements parce qu'elle dépend de la position qui peut être modifiée par les déplacements. Au demeurant, cela correspond à la notion bien connue en algèbre appelée de produit semi-direct introduit en section 2.8.2. Cette sémantique est compositionnelle dans le sens où pour tout programme `p1` et `p2`, nous avons :

```
property_morphism1 = semantics mempty == mempty
property_morphism2 p1 p2 =
  semantics (p1 <> p2) == semantics p1 <> semantics p2
```

En effet, la fonction `semantics` est l'unique morphisme du monoïde (libre) des « programmes de tortue » dans la sémantique tortue induite par `(move i, draw i)` sur les actions élémentaires.

La généralisation de ces constructions présentées plus bas peut sembler inutilement théorique, cependant, elle présente le mérite, comme nous allons le voir, de se généraliser très bien dans une grande variété de contextes de programmation. Nous allons donc réécrire ce code en prenant soin d'exhiber les structures mathématiques cachées derrière les types et fonctions que nous utilisons.

6.5 Monoïde inversif

Jusque-là, nous n'avons pas utilisé le fait que les déplacements sont réversibles. C'est-à-dire qu'il ne s'agit pas seulement d'un monoïde, mais d'un groupe. Il apparaît alors que ces propriétés ouvrent la voie à une transformation des « programmes de tortue », appelée `inverse` qui a, à l'équivalence sémantique près, des propriétés intéressantes.

Procéduralement, l'inverse d'un programme est le même programme, mais exécuté de la fin au début. La figure réalisée sera exactement la même, mais la position initiale de la tortue d'un programme `p` sera la position finale de la tortue du programme `inverse p`, il en va de même pour les positions respectivement finales et initiales. La fonction suivante permet de calculer les inverses de chaque instruction du langage :

```
invI :: Instruction -> Instruction
invI TogglePen = TogglePen
invI (Turn a) = Turn (-a)
invI (Walk d) = Walk (-d)
```

Maintenant, nous pouvons faire des programmes une instance de semi-groupes inversifs. Le programme inverse de `p` est `p` exécuté en partant de la fin et remontant à son début, tout en exécutant l'inverse de chacune des instructions.

```
instance InverseSemigroup Program where
  inverse p = reverse (map invI p)
```

Chacun peut vérifier que pour tout programme `p`, le programme `inverse p` est l'inverse sémantique du programme `p` :

```
propertyInverse1 p =
  semantics p == semantics (p <> inverse p <> p)
```

et

```
propertyInverse2 p =
  semantics (inverse p) ==
    semantics (inverse p <> p <> inverse p)
```

Du coté sémantique, nous avons :

```
inverseSemantics (m,d) = (inverse m, (inverse m) !* d)
```

Par le fait que les déplacements forment un groupe, et que les dessins ont une structure de semi-treillis, nous en déduisons la commutativité des idempotents, et donc il s'agit bien là de l'inverse au sens des semi-groupes inversifs.

Remarque (Produit semi-direct)**.** Nous avons vu que les monoïdes (ou même simplement les semi-groupes) jouent un rôle central dans la définition de la sémantique de la tortue. De plus, la plupart des définitions sémantiques sont génériques. Les fonctions comme `moveI`, `drawI` et `semantics` sont définies en terme d'action basique de type `Instruction`.

Ceci suggère que la sémantique tortue repose sur des structures encore plus générales et qui méritent donc d'être rendues explicites. Dans cette section, nous étudions plus en profondeur les constructions génériques qui ont été — bien qu'implicitement — appliquées et les propriétés mathématiques que ces constructions satisfont.
Pour revenir à la sémantique tortue, nous déclarons simplement :

```
instance ActionG Move Draw where
  m !* (Draw d)  = Draw (d . apply m)
```

Et nous pouvons donc redéfinir l'instance de monoïde ainsi :

```
instance Semigroup Semantics where
  (Semantics (m1,d1)) <> (Semantics (m2,d2)) =
    Semantics (m1 <> m2, d1 <> (m1 !* d2))
```

En fait, nous aurions tout aussi bien pu définir le type par `SDP Move Draw`, `SDP` étant défini dans l'introduction en section 2.8.2. C'est-à-dire le monoïde formé par le produit semi-direct de `Move` et `Draw`. In fine, nous avons abstrait la sémantique des « programmes de tortue » ainsi :

— Un monoïde M représente les déplacements de la tortue,

— Un semi-treillis D représente les dessins faits,

— Une action de M sur D implémente les transformations des dessins induits par les déplacements de la tortue,

— Les paires $(M \times D)$ encodent les sémantiques des programmes,

— Le produit semi-direct de $(M \times D)$ calcule la, composition séquentielle de deux programmes.

6.6 Monoïde avec reset

Comme nous l'avons évoqué dans le chapitre d'introduction aux théories utilisées en section 2.4, il est malaisé de donner une sémantique intuitive aux inverses dans le cas général. Et il se trouve que dans bien des cas, nous nous servons des inverses principalement pour créer des idempotents, qui sont les éléments qui nous intéressent particulièrement.

Un autre avantage de l'utilisation de monoïdes avec reset est que nous n'avons plus besoin que les déplacements aient une structure de groupe. En fait, il suffit qu'il s'agisse d'un monoïde. Le semi-groupe inversif décrit ci-dessus induit une nouvelle transformation de programme appelée l'opération de reset, qui est également intéressante. Cette opération est définissable même dans le cas où l'on considère un monoïde de déplacements non réversible, et, comme nous allons le voir, il se comporte comme un opérateur `fork`, permettant des évaluations concurrentes. Plus précisément, nous définissons `resetP` sur les programmes tortue par :

```
resetP p = p <> inverse p
```

En exécutant le programme `p <> inverse p`, la tortue exécute les instructions spécifiées par `p` en produisant une figure, et ensuite exécute l'inverse de ses action dans l'ordre inverse, reproduisant l'exacte même figure, tout en revenant à sa position initiale. En d'autre mots, le programme `resetP p` décrit un double traçage de la figure, un dans un sens par `p`, et dans l'autre sens par `inverse p`. Au niveau sémantique, cette redondance peut être évitée en faisant comme suit.

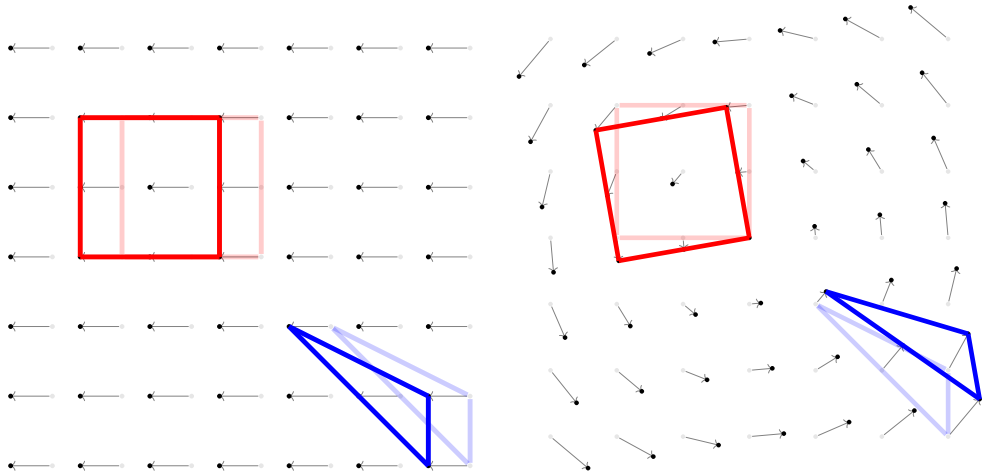
Étant donné la sémantique `(m,d) = semantics p` d'un programme `p`, qui dit que la tortue doit effectuer le mouvement `m` tout en dessinant `d`, nous pouvons vérifier que `(mempty,d) = semantics (resetP p)`. Autrement dit, le programme `resetP p` va exécuter le programme `p` (c'est-à-dire dessiner `d`) et restaurer la position initiale de la tortue. Nous pouvons définir l'instance suivante :

```
instance ResetableSemigroup Semantics where
  reset (Semantics (_,d)) = Semantics (mempty, d)
```

Ceci nous apprend au moins deux choses. La première, c'est que la fonction `resetP` agit comme une sorte d'opérateur `fork` qui permet l'évaluation parallèle, la seconde, c'est qu'implémenter le reset au niveau sémantique est plus efficace, et peut être défini même dans le cas de déplacements non réversibles.

6.7 Remarque sur les déplacements basiques de la tortue

Nous sommes partis de la sémantique des programmes de la tortue LOGO, les déplacements permis par ce langage sont les translations et les rotations (illustrés figure 6.3).



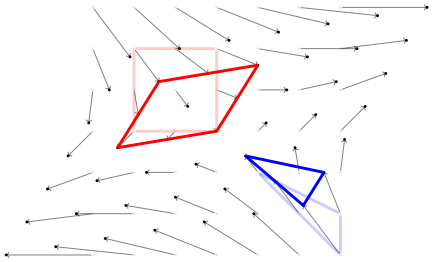
(a) Translation de 0.6 unité. (b) Rotation de 10°.

FIGURE 6.3 – Translation et rotation de l'ensemble des positions.

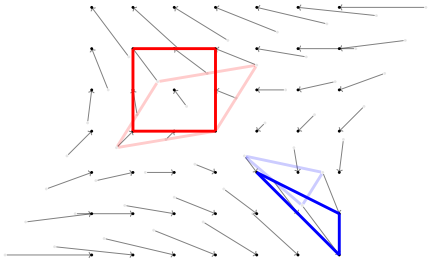
Nous allons maintenant considérer les déplacements dans un sens plus large que celui admis jusque-là. Par déplacements nous entendons, comme au chapitre 3, toute transformation réversible de l'espace des positions. Nous pouvons considérer différentes classes de transformation de l'espace :

- Les isométries arbitraires : rotations, réflexions, ...
- Les transformations linéaires arbitraires, ainsi que leurs inverses, illustré sur les figures 6.4a et 6.4b. Une simple homothétie entre également dans cette ensemble.

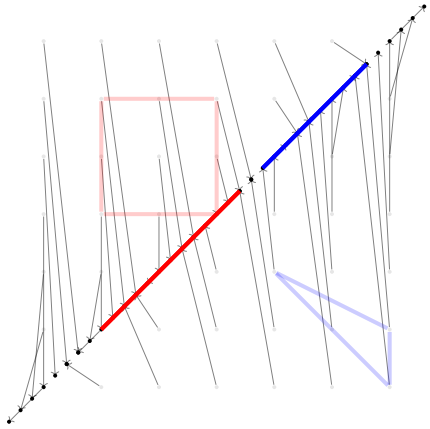
- Les transformations affines arbitraires, illustré sur la figure 6.4d.
- Les transformations non réversibles comme la projection de l'espace 2D entier sur une ligne. Une telle transformation ne pas être autorisée si nous tenons à ce que l'ensemble des déplacements forme un groupe. Une illustration est proposée sur la figure 6.4c.
- Des transformations non-linéaires.



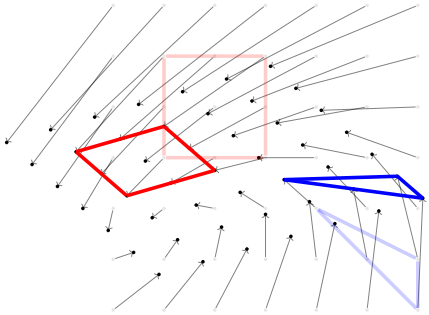
(a) Transformation linéaire A.



(b) Transformation linéaire A^{-1}



(c) Transformation linéaire non réversible.



(d) Transformation affine très arbitraire.

FIGURE 6.4 – Des transformations linéaires.

6.8 Sémantique monadique induite

Jusque-là, notre langage de programmation de tortue est essentiellement basée sur des listes construites à partir d'un alphabet paramétré (les actions de base), doté de propriétés mathématiques riches, mais pas encore clairement exploitables.

Par exemple, les conditionnelles, boucles, et affectation de variables font défaut pour dessiner des figures complexes. De plus, dessiner concrètement

des figures nécessite de faire des actions d'entrée/sortie, tout ceci nous conduit à encapsuler la sémantique de notre langage dans des monades.

6.8.1 Sémantique de la monade de dessin

Implémenté par-dessus Haskell, notre proposition de langage hérite de toute l'infrastructure logicielle proposée par Haskell. Cependant, combiner les fonctions Haskell et les programme tortue n'est pas très pratique. En effet, le produit monoïdal <> ressemble peu à de la programmation.

Par ailleurs, la discussion ci-dessus sur les monoïdes avec reset (voir section 6.6) et l'extension possible aux déplacements non réversibles suggère que nos programmes devraient être définis directement au niveau de la sémantique comme une paire composée d'un déplacement et d'un dessin. Il faut cependant prendre garde à l'utilisation mémoire pour ne pas devoir stocker un état inutilement lourd.

L'approche monadique (et la *do-notation*) propose un mécanisme simple et efficace pour ce problème. Il se trouve que la bibliothèque *Prelude* d'Haskell offre un moyen facile d'encapsuler un monoïde dans une monade.

L'instance suivante, définie dans la bibliothèque de base d'Haskell peut être utilisée dans le cas qui nous occupe.

```
instance Monoid a => Monad ((,) a) where
  return b = (mempty, b)
  (m1,b) >>= f = let (m2,b') = f b
                 in (m1 <> m2, b)
```

Nous allons instancier la paire de cette manière :

```
type MSemantics a = (Semantics, a)
```

Ceci nous permet de (re)définir les déplacements de base comme des actions monadiques :

```
nop    :: MSemantics ()
toggle :: MSemantics ()
nop = load mempty
toggle = load [TogglePen]
--walk :: Length -> MSemantics ()
walk d = load [Walk d]
turn :: Angle -> MSemantics ()
turn d = load [Turn d]
```

La fonction de chargement `load` définie par :

```
load :: Program -> MSemantics ()
load p = (semantics p, ())

runMSemantics :: MSemantics a -> Figure
runMSemantics (Semantics (_,Draw d), _) = d initPosition
```

Grâce à cette construction, notre tortue hérite de la *do-notation*.

Exemple. Un code dessinant un triangle :

```
prog = do walk 1
          turn 120
          walk 1
          turn 120
          walk 1
triangle = runMSemantics prog
```

6.8.2 Sémantique de la monade IO de dessin

Visualiser les dessins sur l'écran nécessite une fonction supplémentaire dont le code dépendra de la bibliothèque graphique utilisée. Nous allons donc supposer que nous avons une fonction `render :: Figure -> IO ()`.

Nous en déduisons que la sémantique des « programmes de tortue » LOGO peut être vue comme des actions d'entrée/sortie qui prennent en paramètre une position et qui produisent un dessin tout en renvoyant la position courante. Donc, le type des déplacements dans l'espace peut être implémenté par :

```
data IOSemantics s a =
  IOSemantics {iosemGetFun :: (s -> IO (s, a))}
```

Nous remarquons que l'instance sur `IOSemantics` est très similaire à celle sur les paires. La seule différence est le fait que la paire est encapsulée dans une monade `IO` :

```
instance Monad (IOSemantics s) where
  return a = IOSemantics $ \s -> return (s,a)
  (IOSemantics m) >>= f = IOSemantics $ \s ->
    do (ns,a) <- m s
       (iosemGetFun $ f a) ns
```

Maintenant, grâce à la fonction `render`, nous pouvons élever le monoïde sémantique de la tortue dans l'`IOSemantics`. C'est-à-dire de manipuler le monoïde sémantique de la tortue à travers la monade.

```
liftSemantics :: Semantics -> IOSemantics Position ()
liftSemantics (Semantics (m,d)) = IOSemantics $ \p ->
  do render (drawGetFun d p)
     return (apply m p, ())
```

ou bien la monade sémantique dans la sémantique `IO` :

```
liftMSemantics :: MSemantics a -> IOSemantics Position a
liftMSemantics  (Semantics (m,d),a) = IOSemantics $ \p ->
  do render (drawGetFun d p)
     return (apply m p, a)
```

Plusieurs égalités pourraient être écrites dans le but de valider ces définitions, par exemple, nous pouvons vérifier que nous avons :

```
liftMSemantics (load p) == liftS (semantics p)
```

L'important est que le monoïde, la monade, ou la monade IO sont trois approches pour définir des sémantiques équivalentes.

Remarque. Le rendu instruction par instruction peut être utile suivant le contexte de programmation, par exemple le *live-coding*, où le programme est écrit en même temps qu'il est exécuté. Cependant, pour un rendu en deux dimensions ou en trois dimensions d'images complexes, il peut être plus avantageux d'accumuler dans un état les commandes de dessin de sorte de pouvoir calculer tout le rendu en une seule fois.

6.9 Animation

Pour l'instant, notre tortue est capable d'effectuer des figures dans un espace à deux dimensions. Nous pouvons maintenant proposer une extension pour réaliser des animations en deux dimensions. Les outils algébriques que nous avons proposés jusque-là peuvent être utilisés. Au niveau sémantique, l'extension de notre tortue à la dimension temporelle est assez facile. Il s'agit d'une construction identique à celle présentée au chapitre 3. Pour l'implémenter, nous définissons le type :

```
data Animation = Animation (Time -> Semantics)
```

Les valeurs de ce type sont des fonctions qui associent une tuile à chaque instant du temps. L'idée est que cette tuile évolue avec le temps qui passe. Autrement dit, il s'agit de faire varier le déplacement et/ou le dessin engendré par une tuile avec le temps.

Nous pouvons définir une fonction `runAnimation` permettant de calculer le rendu d'une de ces animations. L'interprétation prêtée à un élément `a::Animation` est la suivante : le déplacement effectué et la figure dessinée à un instant `t` sont définis par `a t`. Dès lors, nous pouvons écrire une fonction de rendu qui va permettre de calculer l'animation.

```
runAnimation :: Animation -> Position -> Time  -> Figure
runAnimation (Animation a) o = \t -> runSemantics (a t) o
```

Puisque `Semantics` est un monoïde, les animations peuvent elles aussi être vues comme un monoïde grâce à l'extension point-à-point (présentée en section 2.6.1).

```
instance Semigroup Animation where
  (Animation a) <> (Animation b) =
    Animation $ \p -> a p <> b p

instance Monoid Animation where
  mempty = Animation $ \_ -> mempty
```

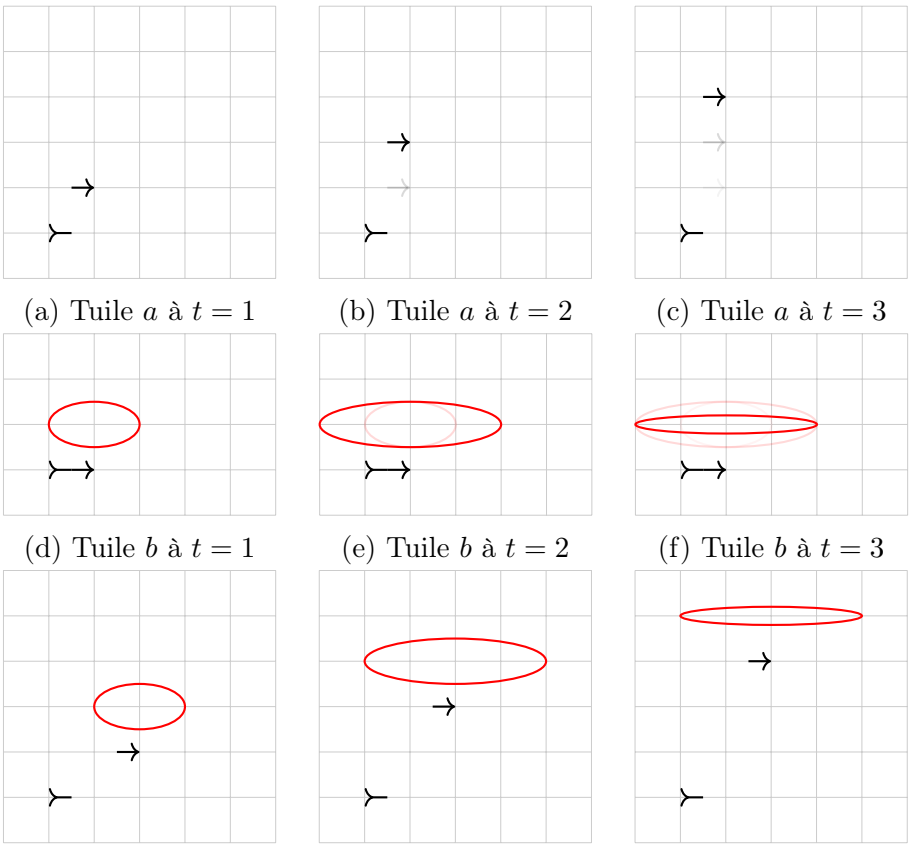
Cette instance de monoïde permet de composer des tuiles évoluant avec le temps. Une illustration de cette composition est donnée par la figure 6.5.

```
instance InverseSemigroup Animation where
  inverse (Animation a) =
    Animation $ \t -> inverse (a t)
```

Cette construction hérite de l'instance sur le type `Semantics`. La superposition temporisée de `a` et `a'` au même instant et à la même position peut être définie par `reset a <> a'`.

6.10 Déplacements temporels

En terme sémantique, l'animation du type défini plus haut autorise d'ores et déjà la définition de toute animation en deux dimensions. Cependant, en termes de programmation temporelle, ce n'est pas très pratique. Nous pouvons souhaiter un ensemble de fonctions qui permet d'effectuer



(a) Tuile a à $t = 1$ (b) Tuile a à $t = 2$ (c) Tuile a à $t = 3$

(d) Tuile b à $t = 1$ (e) Tuile b à $t = 2$ (f) Tuile b à $t = 3$

(g) Tuile $a + b$ à $t = 1$ (h) Tuile $a + b$ à $t = 2$ (i) Tuile $a + b$ à $t = 3$

FIGURE 6.5 – Composition de deux tuiles animées.

des opérations simples sur tous les programmes temporisés de manière générique. Par exemple, démarrer une animation à un instant donné, la terminer à un instant donné, ou la jouer uniquement entre deux instants. Nous pouvons également souhaiter pouvoir modifier la vitesse d'une animation. Et, plus important encore, nous pouvons vouloir composer des animations sur le temps.

Ces observations nous conduisent à la définition d'une notion de déplacement temporel, ainsi que son type, et monoïde associé. Nous allons également utiliser l'implémentation des bijections des positions temporelles dans les positions temporelles permise par l'exemple donnée en page 99.

```
type TMove = SymGroup Time
```

Extension sémantique temporelle

L'extension temporelle de notre tortue sémantique est obtenue en appairant les déplacements temporels et les animations.

```
data TExtension = TExt (TMove, Animation)
```

Le monoïde des déplacements temporels agit sur les animations par l'action :

```
instance ActionG TMove Animation where
  d !* (Animation a) =
    Animation $  a . apply d
```

Ici, dans le cas où l'application est stoppée par un déplacement temporel, l'animation ne produit plus que la figure vide.

Maintenant, grâce à la construction du produit semi-direct, nous avons :

```
instance Semigroup TExtension where
  (TExt (d,a)) <> (TExt (d',a')) =
    TExt (d <> d', a <> d !* a')

instance Monoid TExtension where
  mempty = TExt (mempty, mempty)
```

Exactement comme nous l'avons évoqué plus haut à propos du type `Semantics`, l'instance de monoïde du type `TExtension` est parfaitement isomorphe à celle de `SDP TMove Animation`. Le reset est également étendu à l'extension temporisée (et aux animations sous-jacentes) avec l'instance suivante :

```
instance ResetableSemigroup TExtension where
  reset (TExt (_,d)) = TExt (mempty, d)
```

Remarque (Extension temporelle et produit semi-direct)**.** Ce même résultat est obtenu par le théorème 2.8.7. En effet, le produit semi-direct d'un groupe et d'un semi-groupe inversif est un semi-groupe inversif. Autrement dit, nous pouvons également l'instancier ainsi :

```
type TExtension = SDP TMove Animation
```

Les instances de monoïde et de semi-groupe avec reset sont celles du produit semi-direct `SDP`.

Cette composition permet de superposer deux animations encapsulées au même instant, à la même vitesse, et à la même position. Notons que cette implémentation est exactement la même que sur le type `Semantics`. Nous pouvons par ailleurs vérifier que nous satisfaisons les axiomes des semi-groupes inversifs.

Exécuter une sémantique temporelle à partir d'un instant donné `t` et étant donné une position dans l'espace `p` peut être fait grâce à la fonction :

```
runTimedSemantics :: TExtension -> Time -> Position -> Figure
runTimedSemantics (TExt (_,Animation a)) t p =
  runSemantics (a t) p
```

qui explicite le fait qu'un déplacement temporel `d` dans la sémantique temporelle de `(d,a)` affecte seulement les animations qui seront combinées à droite de `(d,a)`. Ceci généralise à la dimension temporelle le fait que les déplacements spatiaux effectués par un programme tortue en deux dimensions `p` affectent les dessins réalisés par les programmes combinés à droite de `p`, comme nous l'illustrons dans les exemples suivants.

Exemple. Toute animation `a` et tout déplacement temporel `d` peuvent être encapsulées simplement dans une sémantique temporelle en prenant :

```
liftTS a = TExt (mempty, a)
liftTM d = TExt (d, mempty)
```

La fonction `liftTM` permet d'encapsuler n'importe quelle transformation temporelle dans le monoïde temporel. La fonction `liftTS` encapsule une animation dans le monoïde temporel. Nous pouvons vérifier que les fonctions `liftTS` et `liftTM` sont des morphismes. Nous pouvons définir quelques fonctions qui agissent sur le temps, celles-ci, par exemple, permettent de définir toute les transformations affines du temps dans le temps :

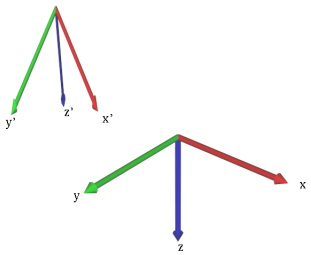
```
delay d   = SymGroup (+ d) (\t -> t - d)
stretch f = SymGroup (* f) (/ f)
```

Comme premier exemple d'application, retarder une animation `a` de `d` unités de temps peut être réalisé grâce à `liftTM (delay d) <> liftTS a`. Un second exemple d'application, modifier la vitesse d'un facteur `f` d'une animation, peut être fait à l'aide de la fonction : `liftTM (stretch f) <> liftTS a`.

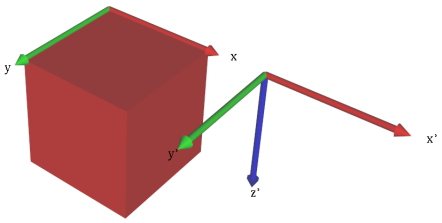
6.11 Extension à la troisième dimension

Notre tortue est maintenant spatio-temporelle, elle dessine des figures en deux dimensions dans le temps. Maintenant nous allons étendre l'espace de dessin pour pouvoir effectuer des figures en trois dimensions composées de surfaces. Encore une fois, les outils algébriques développés plus haut sont réutilisables dans ce but. La seule différence notable est que nous allons partir de dessins primitifs simples, plutôt que les générer comme nous l'avons fait dans le cas en deux dimensions.

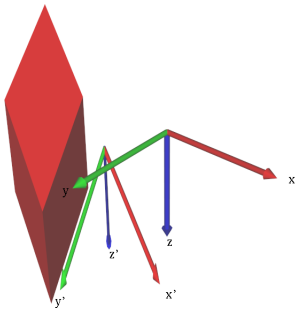
Sur la figure 6.6 nous pouvons voir la composition de tuile en 3D. Les vecteurs x, y, et z représentent de repère d'entrée des tuiles, et les vecteurs x', y', et z' leur repères de sortie. Exactement comme pour le cas en deux dimensions, la composition se fait par synchronisation du repère d'entrée de la seconde opérande sur le repère de sortie de la première opérande.



(a) Tuile a, un déplacement et une déformation.



(b) Tuile b, un cube et une translation.



(c) Tuile $a + b$.

FIGURE 6.6 – Tuiles 3D et leur composition.

Les points sont représentés par des vecteurs de taille 3, encodant les coordonnées cartésiennes :

```
type Point3D = Vect3 Length
```

Nous nous limitons aux transformations linéaires de cet espace. Ces fonctions affines peuvent être encodées par la paire suivante :

```
data Affine3D = Affine3D { affineCoef :: Matrix3x3 Length
                         , affineOffset :: Vect3 Length }
```

Dans la composition des transformations affines nous pouvons voir apparaître le produit semi-direct des matrices sur les vecteurs :

```
instance Semigroup Affine3D where
  (Affine3D c1 o1) <> (Affine3D c2 o2) =
    Affine3D (c <> c') (o1 <> c1 !* o2)

instance Monoid Affine3D where
  mempty = Affine3D mempty -- identity matrix
                    mempty -- null vector
```

L'inverse est donc défini comme nous l'avons vu en section 2.8.2 :

```
instance InverseSemigroup Affine3D where
  inverse (Affine3D c o) =
    Affine3D (inverse c) ((inverse c) !* (inverse o))
```

Les mouvements de la tortue en trois dimensions sont définis par des transformations affines arbitraires de l'espace 3D sous-jacent. Ceci nous autorise à définir des mouvements complexes en composants de plus simples.

Maintenant, nous avons besoin d'un type pour représenter les dessins en trois dimensions :

```
data Drawing3D
```

Avec une action à gauche des transformations affine sur ces dessins, nous pouvons définir la composition tuilée :

```
data Semantics3D = Semantics3D (Affine3D, Drawing3D)

instance Semigroup Semantics3D where
  (Semantics3D (m1,d1)) <> (Semantics3D (m2,d2)) =
   Semantics3D (m1 <> m2, d1 <> m1 !* d2)

instance Monoid Semantics3D where
  mempty = Semantics3D (mempty, mempty)
```

6.12 Résumé de la construction

Cette construction formée à partir du produit semi-direct d'un groupe et d'un monoïde inversif est très régulière. Elle permet de produire des monoïdes inversifs de plus en plus complexes.

Finalement, nous pouvons voir le groupe comme un contexte duquel va dépendre la valeur associée dans le monoïde inversif.

Nous pouvons donc créer un espace dont le temps s'écoule différemment suivant l'endroit où l'on se trouve. Et, réciproquement, un temps qui déforme l'espace différemment suivant l'instant.

Nous remarquons aussi qu'il n'y a pas de limite à la richesse de l'espace dont les transformations vont former le groupe. En particulier, cela peut être des informations sur le contenu tuilé, l'élément du monoïde inversif. Cependant, cet échange d'information n'est pas possible avec le produit semi-direct, en effet, il n'y a transfert d'information que du groupe vers le monoïde inversif.

6.13 Prototype

Nous avons réalisé une implémentation des concepts explorés ici. Elle est décrite en détails dans l'article [5]. Il s'agit d'une bibliothèque permettant de programmer des dessins animés en trois dimensions, et qui permet de se déplacer à l'intérieur à grâce aux interfaces d'entrée de l'ordinateur, joystick, souris, clavier... Le code source est disponible ici : `https://github.com/djanin/Octopus`. Un exemple de scène généré est représenté sur la figure 6.7.

FIGURE 6.7 – Scène 3D avec Octopus

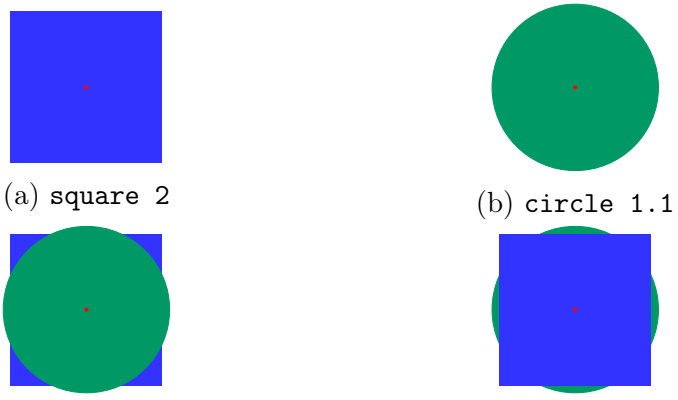
Il faut noter que la génération dans ce programme utilise de l'extrusion plutôt que des primitives géométriques, à l'image de ce qui se fait en deux dimensions (extrusion de la pointe du crayon sur le plan).

6.14 Autres approches

De nombreuses autres approches méritent d'être citées. Parmi celles dont la comparaison avec notre approche sont les plus pertinentes nous pouvons citer les suivantes.

Diagrams

La bibliothèque diagrams [34] est écrite en Haskell et permet de réaliser des diagrammes en deux dimensions. L'abstraction permettant la manipulation des données est la suivante : Les diagrammes de base sont des formes simples (cercle, ligne brisée, texte, etc.). Chaque diagramme a un point particulier appelé « origine », et les diagrammes sont composés en les superposant et en synchronisant leurs origines respectives. L'ensemble des diagrammes forme un monoïde. En effet, il y a un diagramme vide, et la composition (par superposition) est associative. La bibliothèque fournit également un ensemble de primitives pour déplacer et faire pivoter l'origine, sans quoi les figures composables ne seraient que des empilements de figures simples. Une illustration de cette mécanique est donnée sur la figure 6.8.



(a) `square 2` (b) `circle 1.1`

(c) `square 2 <> circle 1.1` (d) `circle 1.1 <> square 2`

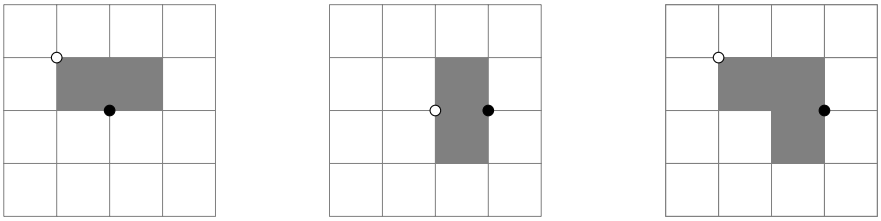
FIGURE 6.8 – Exemple de composition dans diagrams.

Comparé avec notre approche, dans diagrams, les éléments sont tous idempotents, mais ils ne commutent pas puisqu'ils se superposent. Dans les

travaux présentés dans cette thèse, les idempotents, c'est-à-dire les resets de tuiles commutent et sont commutatifs. En contrepartie, on ne peut pas faire de superposition.

Monoïdes inversifs et dessins

Le monoïde inversif libre peut être caractérisé de la manière suivante. Il s'agit des chemins dans le graphe de Cayley d'un groupe libre. Des chemins aux dessins, il n'y a qu'un pas. Les travaux de M. Latteux, D. Robilliard et D. Simplot [25] partent de cet idée. Ils considèrent le groupe de Cayley quotienté par $ab = ba$, c'est-à-dire une grille. Il s'agit là encore de réaliser des dessins en deux dimensions par compositions de dessins plus simples. Mais ici, contrairement à diagrams et à l'instar de nos travaux, le modèle algébrique sous-jacent est un monoïde inversif. De plus la sémantique associée est très proche de celle que nous avons adoptée. Une illustration du modèle est donnée sur la figure 6.9. Ce que nous appelons les repères d'entrée et de sortie sont matérialisées ici par les disques blancs et noirs respectivement. Ces travaux abordent ce modèle sous l'angle de la théorie des langages.



(a) d_1 (b) d_2 (c) $d_1 + d_2$

FIGURE 6.9 – Composition de dessins formés de pixels

6.15 Conclusion

Dans ce chapitre, dans un souci de simplicité, nous avons fixé le type permettant de représenter le temps, le type de la mesure dans l'espace, le repère de l'espace (orthogonal et orthonormé). Certains de ces choix limitent l'expressivité autorisée par le modèle. Mais nous pouvons abstraire tous ces types. La structure qui équipe notre construction dépend des structures qui équipent tous ces types.

La principale nouveauté de notre approche est de rendre explicite l'usage des produits semi-directs. Ceci permet en particulier de combiner au sein

d'une structure de semi-groupe inversif à la fois les déplacements (spatiaux et temporels) ainsi que les dessins.

À notre connaissance, la notion de produit semi-direct, bien que dérivant de la notion d'action de monoïde qui est déjà utilisée dans la bibliothèque *Diagrams* [34], n'a jamais été considérée dans la programmation de média temporisés.

7

Conclusion et perspectives

Dans cette thèse, nous avons proposé une approche formelle pour définir (et implémenter) un modèle sémantique de médias temporisés et spatialisés basé sur les concepts théoriques des semi-groupes.

7.1 Contributions

Nous avons apporté plusieurs contributions dont les principales sont :

- Une meilleure compréhension générale de la sémantique programmation tuilée, présentée dans les chapitres 3 et 4.
- Une compréhension de la programmation interactive par tuilage suffisante pour réaliser un prototype fonctionnel, ainsi qu'une proposition d'implémentation entièrement pure (au sens de la programmation fonctionnelle), dans le chapitre 5.
- L'utilisation des monoïdes inversifs pour la programmation spatiale (le dessin), présentée dans le chapitre 6.
- Plusieurs applications pratiques du théorème présenté en section 2.8.1 sur les treillis et les monoïdes inversifs.
- Une construction élégante et uniforme pour différents médias temporisés.

7.2 Publications

Ce travail de thèse a abouti à plusieurs publications :

— « An Efficient Implementation of Tiled Polymorphic Temporal Media [1] » où est présenté le passage de l'encodage syntaxique des tuiles à un encodage sémantique.

— « Pour un raffinement spatio-temporel tuilé [2] » où est explorée une propriété intéressante (quoi que banale avec le recul), le fait que le raffinement des valeurs portées par les tuiles soit monadique.

— « Structured reactive programming with polymorphic temporal tiles [3] » où est décrit l'extension réactive de la programmation temporelle tuilée.

— « Vers une programmation réactive structurée [4] » où est explorée plus profondément la sémantique de différentes classes de programmes réactifs.

— « Interpolations : écriture de contraintes réactives pour improvisations pianistiques [7] » qui décrit une expérience réalisée avec un prototype d'implémentation du langage réactif décrit et un pianiste, Edwin Buger.

— « Unified Media Programming : An Algebraic Approach [5] » où est présenté le travail sur la généralisation au spatial de nos travaux précédents.

— « Des promesses, des actions, par flots, en OCaml [6] » où est présenté une autre approche pour la programmation réactive.

7.3 Perspectives

Pour la suite des travaux, plusieurs voient restent à explorer, parmi lesquelles nous pouvons citer les suivantes.

Utilisation monadique des promesses

Présentement, l'implémentation des « valeurs gelées » est ad-hoc, elle a simplement l'avantage de la pureté, c'est-à-dire qu'il s'agit de code sans effet de bord, donc aisément testable, etc. Cependant, dans le cas qui nous occupe, il se trouve que les valeurs à venir dont nous avons besoin sont précisément des promesses. En tant que monade, ses propriétés sémantiques sont simples et d'utilisation aisée. Il reste néanmoins à trouver le bon modèle pour les durées partiellement connues. Des travaux de formalisation ont déjà été menés dans ce sens, voir [23, 6].

Axiomatisation des tuiles réactives

L'utilisation de promesses proprement axiomatisées permettra également une axiomatisation complète des tuiles réactives. Les avantages d'une telle axiomatisation sont multiples : cela permettrait de découvrir et prouver des propriétés du modèle, rendrait plus aisée l'écriture de tests d'une implémentation, et ces tests seront plus convaincants vis-à-vis de la correction du code.

Implémentation unifiées

Pour l'heure, les implémentations les plus avancées de ces travaux sont deux prototypes réalisés en Haskell. Il serait naturel de proposer une implémentation distribuable des concepts développés lors de ce travail de thèse. En particulier, une application permettant la programmation multimédia. Même s'il s'agit principalement d'un problème de génie logiciel, il y a néanmoins quelques problèmes qui devront être résolus avant. Par exemple, étudier l'opportunité d'une syntaxe ad-hoc, lister les primitives pertinentes pour chacun des médias, etc.

Programme synchrone et asynchrone

Il y a une dualité entre les sémantiques synchrones et asynchrones. La question est de savoir à quel rythme tourne l'application : à son rythme propre, ou à celui de son entrée ? Dans cette thèse, au chapitre 5, nous avons considéré une approche asynchrone. Nous travaillons sur des systèmes MIDI. C'est la réception des évènements qui rythme l'application. Dans la programmation multimédia, chaque média à son rythme propre.

Causalité

Aujourd'hui, l'évaluation d'une fonction non causale dans notre prototype t-calculus provoque une erreur. La faute de causalité est détectée au dernier moment, lorsque l'application tente d'effectuer un calcul en utilisant une valeur non définie. Cela n'est pas satisfaisant. A fortiori puisqu'au cours de ces travaux nous avons toujours cherché à utiliser au maximum le système de type d'Haskell afin d'avoir les garanties statiques les plus fortes possibles.

Parmi les travaux à venir, il y a la découverte d'un modèle adéquat et élégant permettant d'abstraire cette propriété de causalité au niveau de l'algèbre, afin de pouvoir la vérifier de manière statique.

Expressivité des valeurs

Dans ce document, nous n'avons pas exploré les possibilités concernant les valeurs tuilées. Les compétences musicales faisant défaut au doctorant écrivant ces lignes pour le faire. Comme nous l'avons vu à la section 6.14, un monoïde inversif qui nous intéresse est l'ensemble des chemins dans le graphe de Cayley d'un groupe libre. C'est précisément ce que l'on obtient en se déplaçant d'intervalles donnés dans une gamme. Il en va de même pour les modes.

Pour la partie spatiale, les valeurs tuilées étaient simplement des couleurs. Mais nous pouvons ajouter des informations. Par exemple, nous pouvons ajouter des données sur la qualité du rendu nécessaire. Nous pouvons également y associer des transformations temporelles, afin de créer un environnement où le temps ne s'écoule pas de la même manière partout.

Glossaire

action à droite d'un monoïde sur un ensemble . 29
action à gauche d'un monoïde sur un ensemble . 28

CPS Continuation Passing Style. 84

DSL Domain Specific Language. 5, 95

groupe . 19

Idempotent . 22

monoïde . 14
monoïde inversif . 22
morphisme . 26

ordre naturel . 23
ordre partiel . 9
ordre total . 12

produit direct . 30
produit semi-direct . 31
PSD Produit Semi-Direct. vii, viii, 30–32, 35
PTM Polymorphic Temporal Media. ix, 76, 77

semi-groupe . 14
semi-groupe avec reset . 25
semi-groupe inversif . 21
semi-treillis . 17, 18

TPTM Tiled Polymorphic Temporal Media. ix, 77

Bibliographie

[1] Simon Archipoff. "An Efficient Implementation of Tiled Polymorphic Temporal Media". In : *ICFP FARM*. Proceedings of the 3rd ACM SIGPLAN International Workshop on Functional Art, Music, Modelling and Design. ACM SIGPAN. Vancouver, Canada, sept. 2015. DOI : 10.1145/2808083.2808089. URL : https://hal.archives-ouvertes.fr/hal-01214101.

[2] Simon Archipoff et David Janin. "Pour un raffinement spatio-temporel tuilé". In : *JFLA 2016 : Vingt-septièmes Journées Francophones des Langages Applicatifs*. Saint-Malo, France, jan. 2016. URL : https://hal.archives-ouvertes.fr/hal-01247424.

[3] Simon Archipoff et David Janin. "Structured reactive programming with polymorphic temporal tiles". In : *ACM International Workshop on Functional Art, Music, Modelling, and Design (FARM)*. Nara, Japan, 2016. DOI : 10.1145/2975980.2975984. URL : https://hal.archives-ouvertes.fr/hal-01350525.

[4] Simon Archipoff et David Janin. "Vers une programmation réactive structurée". In : *Journées d'Informatique Musicale (JIM)*. Albi, France, mars 2016. URL : https://hal.archives-ouvertes.fr/hal-01326557.

[5] Simon Archipoff et David Janin. "Unified Media Programming : An Algebraic Approach". In : *5th ACM SIGPLAN International Workshop on Functional Art, Music, Modeling and Design (FARM)*. Oxford, United Kingdom, 2017. DOI : 10.1145/3122938.3122943. URL : https://hal.archives-ouvertes.fr/hal-01571133.

[6] Simon Archipoff, David Janin et Bernard P. Serpette. “Des promesses, des actions, par flots, en OCaml”. In : *JFLA 2020 - 31ème Journées Francophones des Langages Applicatifs*. Sous la dir. de Zaynah Dargaye et Yann Regis-Gianas. Gruissan, France, jan. 2020. URL : https://hal.archives-ouvertes.fr/hal-02389651.

[7] Simon Archipoff et al. “Interpolations : écriture de contraintes réactives pour improvisations pianistiques (démo)”. In : *Journées d'Informatique Musicale (JIM)*. Albi, France, mars 2016. URL : https://hal.archives-ouvertes.fr/hal-01326559.

[8] T. Bazin et D. Janin. “Flux média tuilés polymorphes : une sémantique opérationnelle en Haskell”. In : *Journées Francophones des Langages Applicatifs (JFLA)*. 2015.

[9] Florent Berthaut, David Janin et Benjamin Martin. “Advanced Synchronization of Audio or Symbolic Musical Patterns : An Algebraic Approach”. In : *International Journal of Semantic Computing* 6.4 (2012), p. 409-427. DOI : 10.1142/S1793351X12400132. URL : https://hal.archives-ouvertes.fr/hal-00794196.

[10] Jean-Michael Celerier. “Authoring interactive media : a logical & temporal approach”. Theses. Université de Bordeaux, mars 2018. URL : https://tel.archives-ouvertes.fr/tel-01947309.

[11] Koen Claessen et John Hughes. “QuickCheck : a lightweight tool for random testing of Haskell programs”. In : *Acm sigplan notices* 46.4 (2011), p. 53-64.

[12] R.J. Duffin et Carnegie inst of tech Pittsburgh PA. *Topology of series-parallel networks*. Defense Technical Information Center, 1964.

[13] P. Hudak. “An Algebraic Theory of Polymorphic Temporal Media”. In : *Proceedings of PADL'04 : 6th International Workshop on Practical Aspects of Declarative Languages*. Springer Verlag LNCS 3057, juin 2004, p. 1-15.

[14] P. Hudak. *The Haskell School of Music : From signals to Symphonies*. Yale University, Department of Computer Science. 2013.

[15] P. Hudak et D. Janin. “Tiled Polymorphic Temporal Media”. In : *ACM Workshop on Functional Art, Music, Modeling and Design (FARM)*. Gothenburg, Sweden : ACM Press, 2014, p. 49-60. DOI : 10.1145/2633638.2633649. URL : http://hal.archives-ouvertes.fr/hal-00955113.

[16] P. Hudak et D. Janin. “From out-of-time design to in-time production of temporal media”. In : (fév. 2015). URL : http://www.labri.fr/perso/janin/Papiers/OutOfAndInTime.pdf.

[17] Paul Hudak. “Building Domain-specific Embedded Languages”. In : *ACM Comput. Surv.* 28.4es (déc. 1996). ISSN : 0360-0300. DOI : 10.1145/242224.242477. URL : http://doi.acm.org/10.1145/242224.242477.

[18] Paul Hudak et al. “Haskore Music Notation – An Algebra of Music”. In : *Journal of Functional Programming* 6.3 (mai 1996), p. 465-483.

[19] David Janin. *On quasi-inverse monoids (and premorphisms)*. Rapp. tech. Fév. 2012. URL : https://hal.archives-ouvertes.fr/hal-00673123.

[20] David Janin. “Quasi-recognizable vs MSO definable languages of one-dimensional overlapping tiles”. In : *MFCS*. Sous la dir. de Vladimiro Sassone Branislav Rovan et Peter Widmayer. T. 7464. LNCS. Bratislava, Slovakia : Springer, août 2012, p. 516-528. DOI : 10.1007/978-3-642-32589-2_46. URL : https://hal.archives-ouvertes.fr/hal-00671917.

[21] David Janin. “Vers une modélisation combinatoire des structures rythmiques simples de la musique”. In : *Revue Francophone d'Informatique Musicale* 2 (oct. 2012). Rapport de recherche LaBRI RR-1455-11, 40 pages, juillet 2011, V2 octobre 2011, V3 mai 2012. URL : https://hal.archives-ouvertes.fr/hal-00608295.

[22] David Janin. “A robust algebraic framework for high-level music writing and programming”. In : *Technologies for Music Notation and Representation (TENOR)*. Cambridge, United Kingdom, mai 2016. URL : https://hal.archives-ouvertes.fr/hal-01246584.

[23] David Janin. “An equational modeling of asynchronous concurrent programming”. In : *21st International Symposium on Trends in Functional Programming*. Sous la dir. de Springer. T. 12222. Lecture Notes in Computer Science. Aleksander Byrski and John Hugues. Krakow, Poland, fév. 2020. URL : https://hal.archives-ouvertes.fr/hal-02865894.

[24] *La joueuse de tympanon*. https://www.arts-et-metiers.net/musee/automate-joueuse-de-tympanon. Dernier accès : 12/08/2020. 1784.

[25] Michel Latteux, Denis Robilliard et David Simplot. “Figures composees de pixels et monoıde inversif”. In : *Bull. Belg. Math. Soc* 4 (1997), p. 89-111.

[26] M. V. Lawson. *Inverse Semigroups : The theory of partial symmetries*. World Scientific, 1998.

[27] S. Letz, Y. Orlarey et D. Fober. “Real-time Composition in Elody”. In : *International Computer Music Conference (ICMC)*. ICMA, 2000, p. 336-339.

[28] Stéphane Letz. *Spécification de l'extension LibAudioStream*. Rapp. tech. Mars 2014. URL : https://hal.archives-ouvertes.fr/hal-00965269.

[29] *Logo Fundation : Logo History*. https://el.media.mit.edu/logo-foundation/what_is_logo/history.html. Dernier accès le 10/08/2020.

[30] K. S. S. Nambooripad. “The natural partial order on a regular semigroup”. In : *Proc. Edinburgh Math. Soc.* 23 (1980), 249–260.

[31] Y. Orlarey, D. Fober et S. Letz. “Syntactical and semantical aspects of Faust”. In : *Soft Computing* 8.9 (2004), p. 623-632. ISSN : 1433-7479. DOI : 10.1007/s00500-004-0388-1. URL : https://doi.org/10.1007/s00500-004-0388-1.

[32] Yann Orlarey, Dominique Fober et Stéphane Letz. “L'environnement de composition musicale Elody”. In : (1997).

[33] Amr Sabry. “What is a Purely Functional Language ?” In : *Journal of Functional Programming* 8 (1998), p. 1-22.

[34] B. A. Yorgey. “Monoids : Theme and Variations (Functional Pearl)”. In : *Proceedings of the 2012 Haskell Symposium*. ACM, 2012, p. 105-116.