L'environnement de composition musicale Elody

Yann Orlarey, Dominique Fober, Stéphane Letz

L'environnement de composition musicale *Elody*

Yann Orlarey, Dominique Fober, Stéphane Letz
(orlarey, fober, letz)@grame.fr

Grame, 9 rue du Garet, BP 1185, 69202 Lyon Cedex 01, France

Résumé

Elody est un environnement pour la composition musicale permettant la description et la manipulation algorithmique de structures musicales et de procédés compositionnels. Son interface utilisateur est basée sur la *manipulation directe* d'objets musicaux et algorithmiques par le biais du glisser-déposer et de constructeurs visuels. *Elody* intègre également des fonctionnalités internet afin de faciliter l'échange et la collaboration entre utilisateurs. Tout l'environnement est écrit en Java et utilise les services de MidiShare pour les communications Midi et les fonctionnalités temps-réel.

1 Introduction

Elody est un environnement pour la composition musicale basé sur un langage de programmation visuel dérivé du λ-calcul, une interface utilisateur mettant l'accent sur la manipulation directe des objets musicaux et l'intégration de fonctionnalités Internet destinées à faciliter l'échange d'informations et la collaboration entre utilisateurs. L'environnement *Elody* a été conçu pour favoriser une attitude créative et expérimentale de l'utilisateur.

Le principe général d'utilisation d'*Elody* consiste à *construire* des objets musicaux, en partant de notes et de silences que l'on assemble suivant différentes modalités, par exemple en séquence et en parallèle. Tous ces modalités d'assemblage et du reste l'ensemble des fonctionnalités d'*Elody* sont représentés à l'écran par des *constructeurs visuels* comportant des cases arguments dans lesquelles l'utilisateur dépose les objets musicaux à assembler et une case résultat où il récupère le nouvel objet ainsi créé. Chaque action de l'utilisateur donne lieu à un résultat sonore et graphique immédiat.

Elody a été réalisé en tenant compte des développements récents d'Internet afin de faciliter sa diffusion mais également la collaboration entre utilisateurs. *Elody* est écrit en Java et fonctionne aussi bien comme application autonome que comme applet à l'intérieur d'une page HTML. Les documents *Elody* sont sauvegardés au format HTML ce qui permet :

1. de publier des objets musicaux réalisés avec *Elody* qui peuvent être directement visualisés dans une page web.
2. de manipuler des objets musicaux distants, réalisés par d'autre, en référençant les pages web qui les contiennent. Cela peut être aussi bien des objets *Elody* que des MIDI-Files.
3. d'ajouter un contenu musical à une page contenant une applet *player Elody*.
4. de bénéficier à terme de serveurs centralisant des objets *Elody* et dont tous les utilisateurs connectés pourront très simplement enrichir et réutiliser le contenu.

Nous avons également cherché à proposer un modèle de programmation utilisable par des non-informaticiens, et adapté à un travail de création et d'expérimentation. Ce modèle, appelé *programmation homogène*, intègre la *programmabilité* au sein même du langage musical, sans avoir recours à un langage de programmation extérieur. Il permet à l'utilisateur de créer, de manipuler et de composer ses *programmes* exactement comme les autres objets musicaux dont il a l'habitude. L'activité de programmation est ainsi vue comme une extension directe de l'activité de composition musicale, dont elle adopte les principes de structuration et les modes de représentation.

Ce modèle de programmation s'inspire très directement λ-calcul, une théorie axiomatique des fonctions ainsi qu'une caractérisation de la notion de processus effectif de calcul développée par le logicien et philosophe américain Alonzo Church dans le milieu des années 30 (voir [Barendregt 84] pour une présentation). Le λ-calcul est d'une grande simplicité et d'une grande beauté conceptuelle. Partant de deux concepts : l'*abstraction* et l'*application*, le λ-calcul permet de simuler toutes les notions habituelles de l'informatique : structures de données (nombres, booléens, listes etc.), structures de contrôle et même la récursivité. C'est en quelque sorte un langage de programmation réduit à son essence. Cette essence est très intéressante parce qu'elle peut être facilement introduite dans un langage purement descriptif, comme par exemple un langage musicale ou un langage graphique, pour le transformer en un véritable langage de programmation.

2 Principes de l'interface Utilisateur

La figure 1 donne une vue d'ensemble de l'environnement Elody. Toutes les fonctionnalités proposées à l'utilisateur sont rendues accessibles par le biais de *constructeurs visuels* disposés spatialement dans les différentes fenêtres. Un *constructeur* est une façon particulière d'assembler des objets musicaux, par exemple en *séquence* ou en *parallèle*, pour en former de nouveaux. Tous les constructeurs fonctionnent de manière similaire. Ils sont représentés à l'écran par des cases arguments dans lesquelles l'utilisateur dépose les objets qu'il veut assembler et une case résultat dans laquelle il récupère l'objet résultant. Les actions de l'utilisateur sont donc essentiellement des glisser-déposer d'objets musicaux entre les différents constructeurs.

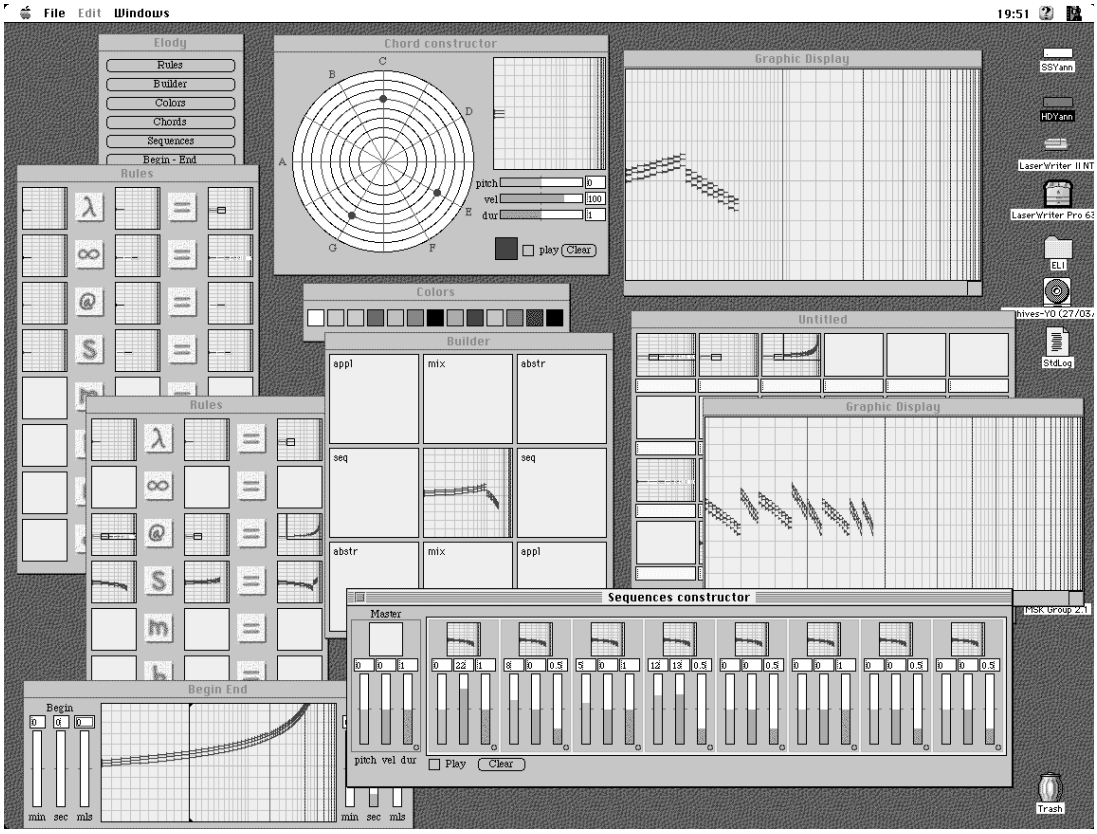


FIG. 1 – Vue d'ensemble de l'interface utilisateur

Le constructeur de séquences par exemple (figure 2) comporte deux cases arguments, de part et d'autre du signe S, dans lesquelles on dépose les objets que l'on veut assembler en séquence, et une case résultat, à droite du signe = dans laquelle on récupère la séquence résultante. Les constructeurs fonctionnent également *à l'envers*. Si l'on glisse un objet de type séquence dans la case résultat du constructeur de séquences, cet objet est *déconstruit* et l'on récupère ses constituants dans les deux cases arguments.

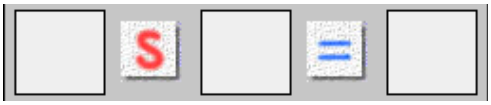
Un constructeur n'effectue aucune opération réelle, mais indique plutôt une promesse d'opération. Le résultat d'un constructeur peut être vu comme un arbre dont le noeud racine indique l'opération qu'il faudra effectuer et les branches les ingrédients à utiliser. *Elody* se comporte comme une machine à calculer qui, lorsque l'on taperait $2 + 3$ afficherait bien 5 mais garderait en mémoire le fait qu'il s'agit de $2 + 3$. Un objet *Elody* garde ainsi toujours trace de son histoire, de tous ses ingrédients et de la façon dont ils ont été assemblés, ce que l'on nommera sa *description en intention*. Les opérations réelles ne sont effectuées que pour jouer et afficher l'objet. C'est ce que l'on appelle l'évaluation, qui transforme une description en intention en une description en extension équivalente mais où tous les calculs ont été réalisés et qui peut être directement jouée et affichée. L'utilisateur voit et écoute des objets évalués (le 5) mais en réalité ne manipule que des descriptions en intention, non-évaluées (le $2 + 3$).

Grce à cela il est possible de généraliser un objet en rendant variable aussi bien les ingrédients que les constructeurs utilisés dans sa fabrication. C'est le mécanisme de programmation proposé par *Elody*. En rendant variable les ingrédients d'un objet on obtient une abstraction, un programme qui, quand on l'appliquera à d'autres ingrédients, sera capable de refaire toutes les opérations ayant conduit à l'objet à partir de ces nouveaux éléments.

Les principes d'interface utilisateur adoptés nécessitent de représenter un grand nombre d'objets musicaux dans un espace limité. La représentation choisie est de type piano mécanique. L'axe vertical représente les hauteurs en demi-tons, l'axe horizontal le temps. L'échelle temporelle n'est pas linéaire, mais suit une loi arc-tangente afin de toujours pouvoir représenter la totalité d'un objet quelle que soit sa durée. Les techniques d'évaluation paresseuse employées permettent de gérer des objets infinis, l'affichage s'arrête alors automatiquement lorsque la taille des objets devient inférieure à la résolution graphique (voir figure 3).

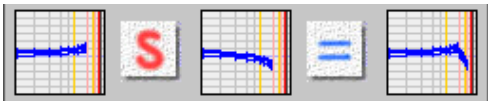
3 Exemples de constructeurs

Les expressions musicales élémentaires sont les *notes* et les *silences* qui comportent une durée et une couleur. La couleur est en quelque sorte un nom de famille. Ce paramètre est utilisé pour la programmation et nous verrons plus loin son rôle exact. Les notes comportent également une hauteur, une vélocité et un canal. La fenêtre Colors (figure 4) contient une palette de notes, toutes de hauteur 60, dans les différentes couleurs disponibles.



2.1: Le constructeur de séquences avant utilisation

2.2: Placement du premier argument



2.3: Placement du deuxième argument et résultat

FIG. 2 – Utilisation du constructeur de séquences

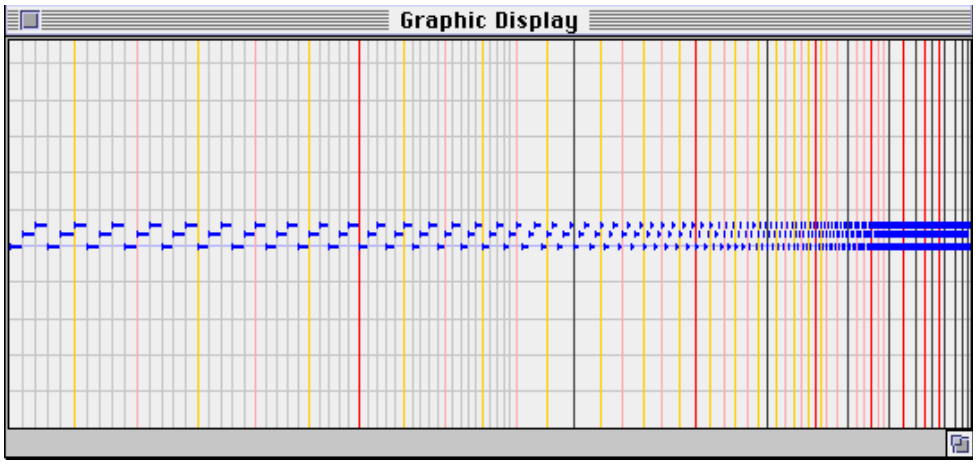


FIG. 3 – Do, Mi, Sol répétés à l'infini

Le constructeur d'accords, figure 5, permet de placer les notes dans la tessiture et de former des accords. Les cercles concentriques représentes les octaves et les rayons les degrés dans l'octave. Des réglages permettent d'ajuster la hauteur de l'accord, sa vélocité et sa durée.

La fenêtre rules, figure 6, comporte les principaux constructeurs élémentaires :

1. le constructeur λ permet de construire des abstractions en déposant dans la case centrale l'objet que l'on veut généraliser et dans la case de gauche l'ingrédient que l'on veut rendre variable dans cet objet. Le résultat est une abstraction, une fonction que l'on va pouvoir appliquer à un argument.
2. le constructeur ∞ permet de construire des objets récursifs. D'un point de vue technique il combine l'application d'un combinateur de point fixe et une abstraction. On dépose dans la case centrale l'objet que l'on veut rendre récursif et dans la case de gauche l'ingrédient qui va servir de point de récursion. Le résultat est un objet récursif où le point de récursion est remplacé par la totalité de l'objet.
3. le constructeur @ permet de construire des applications. On dépose dans la case de gauche la fonction que l'on veut utiliser et dans la case centrale l'argument que l'on veut passer à cette fonction.
4. le constructeur S permet de monter en séquence deux objets. On dépose dans la case de gauche le premier objet et dans la case centrale le deuxième objet.
5. le constructeur M permet de mixer (de monter en parallèle) deux objets. On dépose dans la case de gauche le premier objet et dans la case centrale le deuxième objet.
6. le constructeur B permet de prendre le début d'un objet en fonction de la durée d'un deuxième objet. On dépose dans la case de gauche l'objet dont on veut prendre le début et dans la case centrale un objet dont la durée va définir celle du début que l'on souhaite prendre.
7. le constructeur R permet de prendre le reste d'un objet en fonction de la durée d'un deuxième objet. C'est l'opération complémentaire de B. On dépose dans la case de gauche l'objet dont on veut prendre le reste et dans la case centrale un objet dont la durée va définir celle du début dont on souhaite amputer l'objet.
8. le constructeur D permet de dilater (ou de compresser) un objet à la durée d'un autre objet. On dépose dans la case de gauche l'objet dont on veut ajuster la durée et dans la case centrale l'objet définissant la durée souhaitée.

FIG. 4 – La palette des notes suivant les différentes couleurs disponibles

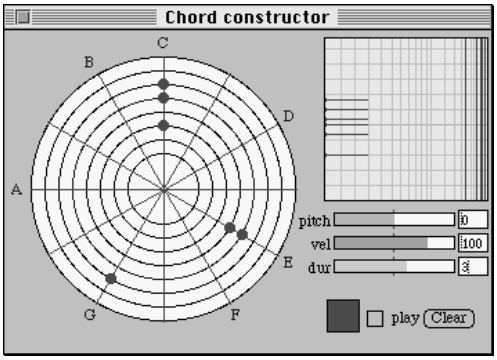


FIG. 5 – Le constructeur d'accords

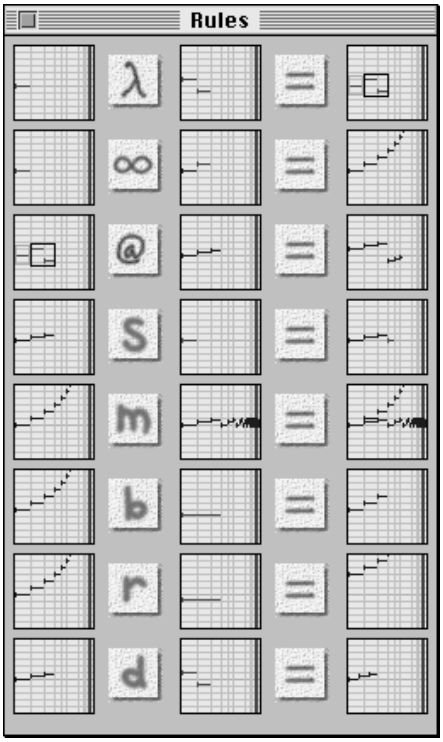


FIG. 6 – La fenêtre Rules et ses constructeurs

Des constructeurs plus sophistiqués sont également proposés, qui combinent des constructeurs élémentaires, comme le séquenceur d'objets figure 7. Son principe de fonctionnement est similaire à celui des anciens séquenceurs analogiques. Il comporte 8 pas dans lesquels on peut déposer des objets. Des réglages permettent d'ajuster la transposition, la vélocité et la durée individuels de chaque pas. Il fonctionne en temps-réel. Lorsque l'option Play est cochée, les pas sont joués en boucle, un point rouge indique le pas en cours et l'utilisateur peut dynamiquement changer les réglages ou déposer de nouveaux objets.

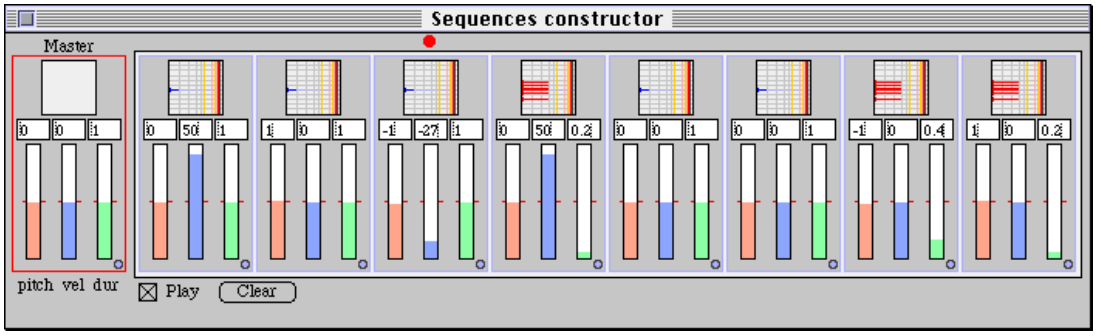


FIG. 7 – Le séquenceur d'expressions

4 Exemples de programmes

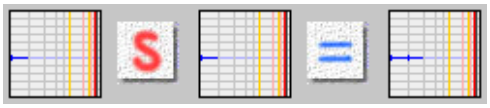
Nous avons rapidement expliqué en introduction les principes du modèle de programmation homogène adopté pour *Elody* afin de rendre la programmation plus accessible à des utilisateurs non-informaticiens. Plutôt que de fournir à l'utilisateur un langage de programmation séparé avec ses concepts, sa syntaxe et ses outils d'édition propres, l'idée est d'étendre de manière consistante le domaine des objets qui intéressent l'utilisateur (ici les objets musicaux) de façon à y inclure les programmes. Ceci est réalisé principalement en introduisant dans le langage des données, les concepts d'abstraction et d'application du λ-calcul, afin de le doter de capacités de programmation. Il n'y a donc plus qu'un seul langage, celui des données, dont la logique est familière à l'utilisateur.

Ceci à plusieurs conséquences intéressantes. D'une part il n'y a aucune difficulté à avoir un langage de programmation visuel, le mode de représentation des programmes est essentiellement le même que celui des données. Les outils d'édition et de manipulation sont également les mêmes. Les programmes définis par l'utilisateur pour manipuler des données s'appliquent tout aussi bien à ses propres programmes. Dans le cas d'*Elody*, un programme de composition défini par l'utilisateur peut très bien

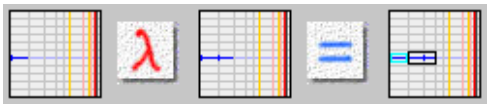
être utilisé pour composer d'autres programmes. On peut ainsi composer des processus de composition. Enfin cette approche conduit naturellement à une forme de programmation par l'exemple.

4.1 La construction d'abstractions

Voyons quelques exemples simples d'abstractions. Prenons une note que nous mettons en séquence avec elle même (figure 8.1). Nous avons construit un exemple particulier de répétition. Nous pouvons généraliser cette répétition particulière, en rendant variable la note employée, et définir ainsi le concept général de répétition. On utilise pour cela le constructeur d'abstraction (λ) en glissant dans la deuxième case l'expression que l'on veut généraliser et dans la première case l'élément que l'on veut rendre variable (figure 8.2).



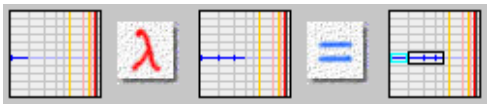
8.1: Construction d'une répétition particulière



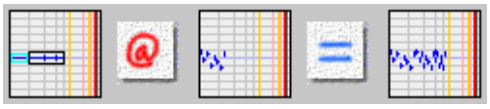
8.2: Fonction *Double* : généralisation de la répétition en rendant *variable* la note employée

FIG. 8 – Création de la fonction *Double*

En procédant de la même manière on peut créer par exemple une fonction *Triple* qui répète trois fois (figure 9.1), et l'appliquer à une séquence (figure 9.2).



9.1: Fonction Triple



9.2: Application de la fonction Triple

FIG. 9 – Création et utilisation de la fonction *Triple*

Il est intéressant de noter ici une double différence par rapport à l'approche classique de la programmation. D'une part, comme nous l'avons dit plus haut, c'est le langage des données musicales qui est utilisé pour écrire les programmes. D'autre part nous n'avons pas besoin d'écrire un programme à priori, nous partons d'un exemple concret, par exemple une séquence jouée au clavier, que nous généralisons après.

4.2 La composition d'abstractions

Comme tout objet musical, une abstraction a une durée, et elle peut être découpée, transposée, mixée et montée. Par exemple nous pouvons prendre les fonctions *Double* et *Triple* précédentes, les compresser pour qu'elles aient une durée identique, en l'occurrence 1 unité de temps, et les monter en séquence en les alternant 8 fois (figure 11).

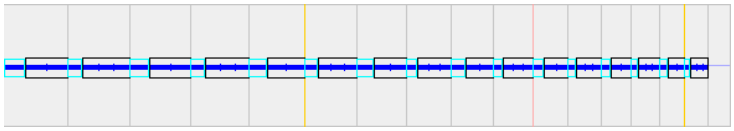


FIG. 10 – Composition d'abstractions

Cette séquence d'abstractions est un nouveau programme et peut être appliqué à un objet pour le transformer. Lorsque l'on applique une séquence, chaque élément de celle-ci s'applique, en fonction de sa durée, sur la partie de l'argument qui lui correspond. Ainsi, si nous redimensionnons cette séquence de fonctions pour qu'elle ait la même durée que l'objet que nous voulons traiter et que nous l'appliquons, nous obtiendrons le premier $1/16$ de notre objet répété deux fois, suivi du deuxième $1/16$ de notre objet répété trois fois, le troisième $1/16$ de notre objet répété deux fois, et ainsi de suite comme le montre la figure 11.

4.3 Abstractions généralisées

Tout *ingrédient* utilisé dans une expression peut être rendu variable. Dans l'exemple précédent, nous avons utilisé la fonction Triple. Nous pouvons rendre variable cette fonction et appliquer l'abstraction résultante à une autre fonction, par exemple un

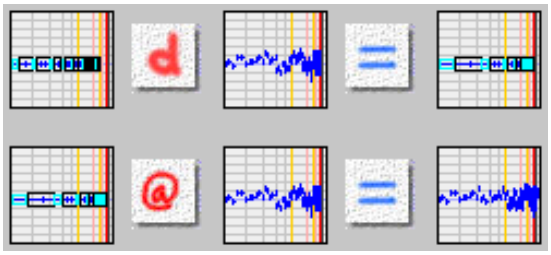
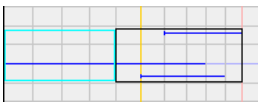
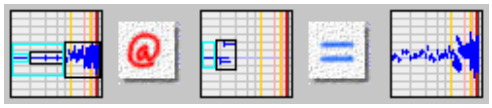


FIG. 11 – Application d'une composition d'abstractions

canon à trois voies (figure 12.2). Le canon lui-même est obtenu en rendant variable une note dans une expression ou cette note apparat à trois hauteurs différentes avec un décalage de temps (figure 12.1).



12.1: Fonction *Canon*



12.2: Abstraction de la fonction *Triple* et application à la fonction *Canon*

FIG. 12 – Exemple d'abstraction de fonctions

5 L'organisation interne d'*Elody*

La figure 13 montre l'organisation des traitements internes d'*Elody* qui, partant d'un objet musical construit par l'utilisateur, conduisent à son écoute et à sa visualisation. Chacune de ces étapes peut-être vue comme la traduction d'une expression d'un langage à un autre.

L'utilisateur ne travaille qu'au niveau des expressions $\mathbb{L}_1$, c'est à dire des *descriptions en intention*. Comme nous l'avons déjà indiqué, les expressions $\mathbb{L}_1$ manipulées par l'utilisateur ne sont jamais modifiées par les calculs qu'elles expriment, elles ne sont jamais remplacées par le résultat de ces calculs. En d'autres termes, l'utilisateur écoute et voit un résultat, mais ce résultat garde trace de son histoire, de tous les processus compositionnels mis en oeuvre pour l'obtenir. Le *sens* et la *dénotation* des objets musicaux, pour reprendre les termes de Frege, restent simultanément accessibles.

Ceci est particulièrement important pour la démarche de programmation proposée par *Elody*. Tous les *ingrédients* constitutifs d'un objet musical restant accessibles, ils peuvent donc être *abstraits*, *rendus variables*, pour permettre à l'utilisateur de dégager la structure générale de l'objet et, par là même, de créer de nouvelles *fonctions*.

Cette approche de la programmation est intéressante car, contrairement à une démarche classique, elle ne force pas à une pensée abstraite (mais ne l'empêche pas non plus, bien au contraire) et n'implique pas d'imaginer *à priori*, dès le départ, toutes les variables souhaitables dans un programme, ce qui est une projection dans le futur toujours difficile à réaliser.

Le calcul du rendu sonore et visuel d'une expression $\mathbb{L}_1$, suppose plusieurs étapes. Une expression $\mathbb{L}_1$ est tout d'abord traduite en une expression $\mathbb{L}_2$ en remplaçant les abstractions généralisées par des λ-abstractions simples faisant appel à des

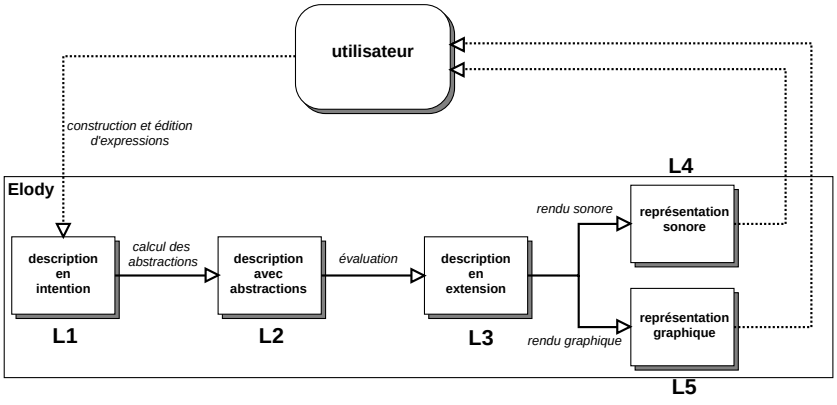


FIG. 13 – Schéma conceptuel d'Elody

variables. L'expression $\mathbb{L}_2$ ainsi obtenue est ensuite évaluée, tous les calculs indiqués dans l'expression sont effectués. Le résultat de l'évaluation est une expression $\mathbb{L}_3$, une expression en *extension*, où tous les processus compositionnels intervenant dans l'expression de départ ont été effectués. Enfin, à partir de l'expression $\mathbb{L}_3$, sont produits le rendu sonore $\mathbb{L}_4$, sous la forme d'une séquence d'événements MIDI, et le rendu graphique $\mathbb{L}_5$ sous la forme d'une suite d'ordres graphiques.

Nous présentons ci-dessous les langages $\mathbb{L}_1$, $\mathbb{L}_2$, $\mathbb{L}_3$ et $\mathbb{L}_4$ ainsi que les opérations de traduction correspondantes. Le langage des représentations graphiques $\mathbb{L}_5$ ne sera pas présenté dans la mesure ou il est relativement analogue à $\mathbb{L}_4$. Précisons que dans l'implémentation d'*Elody*, ces différentes étapes sont imbriquées de façon à ce que le rendu sonore et visuel d'un objet puisse commencer avant même que la totalité de l'objet n'ait été calculé. Ceci améliore le sentiment de réponse de l'utilisateur et permet de traiter le cas des objets infinis.

5.1 Le langage $\mathbb{L}_1$

Le langage $\mathbb{L}_1$ comporte des expressions élémentaires et un ensemble de constructeurs permettant d'assembler des expressions pour en former de nouvelles. Les expressions élémentaires de $\mathbb{L}_1$ sont les *silences* et les *notes* :

$\texttt{sil}[m, d]$	un *silence* de famille m et une durée d.
$\texttt{note}[m, d, h, v, c]$	une *note* de famille m, de durée d, de hauteur h, de vélocité v et de canal c.

Les durées sont exprimées en secondes. Les hauteurs et les vélocités sont exprimées en hauteurs et vélocités MIDI entre 0 et 127. Les canaux sont exprimés en canaux MIDI, mais entre 0 et 31 afin d'adresser deux ports MIDI. Enfin, la famille d'appartenance, représentée par une couleur : `rouge`, `vert`, `bleu`, ..., est utilisée dans le calcul des abstractions. Elle permet de *lier* les événements de même famille au sein d'une expression par des relations de hauteur, de durée, de vélocité et de canal.

Les constructions proprement musicales de $\mathbb{L}_1$ sont les suivantes (e et f représentent des expressions de $\mathbb{L}_1$, $n \in \mathbb{Z}$ et $r \in \mathbb{R}^+$) :

$\texttt{seq}[e, f]$	mise en séquence de e et f.
$\texttt{mix}[e, f]$	mixage (superposition temporelle) de e et f.
$\texttt{beg}[e, f]$	début de e pris sur une période de la durée de f.
$\texttt{rst}[e, f]$	reste de e privée de son début sur une période de la durée de f.
$\texttt{xpd}[e, f]$	e dilatée ou compressée dans le temps de façon à avoir la durée de f.
$\texttt{spd}^r[e]$	e dilatée ou compressée dans le temps par un coefficient r.
$\texttt{trp}^n[e]$	transposition de e de n unités.
$\texttt{lvl}^n[e]$	augmentation des vélocités de e de n unités.
$\texttt{chn}^n[e]$	augmentation des canaux de e de n unités.

Enfin on trouve également deux constructeurs particulier, venant du λ-calcul, l'*abstraction* et l'*application* qui donnent à $\mathbb{L}_1$ des capacités de programmation :

$\lambda^{\text{G}}[e, f]$	abstraction généralisée, généralisation de f obtenue en *rendant variables* les occurrences de e dans f.
$\texttt{@}[e, f]$	application de la *fonction* e à l'*argument* f.

5.2 Le langage $\mathbb{L}_2$

Pour que l'évaluation des expressions puisse avoir lieu, il est nécessaire de traduire les abstractions généralisée $\lambda^{\text{G}}[e, f]$ utilisées dans $\mathbb{L}_1$ en λ-abstractions simples faisant appel à des variables. C'est l'objet du langage $\mathbb{L}_2$ et de la fonction de traduction $\mathcal{L}() : \mathbb{L}_1 \rightarrow \mathbb{L}_2$.

$\mathbb{L}_2$ se distingue essentiellement de $\mathbb{L}_1$ par l'apparition de variables $\diamond^e$ formée par le symbole $\diamond$ étiqueté par une expression e de $\mathbb{L}_2$ qui lui sert de nom et qui correspond à l'expression *rendue variable*, ainsi que par l'apparition de λ-abstractions simples de la forme $\lambda[\diamond^e, f]$ en remplacement des abstractions généralisées.

Comme pour $\mathbb{L}_1$, les expressions élémentaires de $\mathbb{L}_2$ sont les *silences* et les *notes* :

$\texttt{sil}[m, d]$	un *silence* de famille m et une durée d.
$\texttt{note}[m, d, h, v, c]$	une *note* de famille m, de durée d, de hauteur h, de vélocité v et de canal c.

Les constructeurs musicaux sont les même (e et f représentent des expressions de $\mathbb{L}_2$, $n \in \mathbb{Z}$ et $r \in \mathbb{R}^+$) :

$\texttt{seq}[e, f]$	mise en séquence de e et f.
$\texttt{mix}[e, f]$	mixage (superposition temporelle) de e et f.
$\texttt{beg}[e, f]$	début de e pris sur une période de la durée de f.
$\texttt{rst}[e, f]$	reste de e privée de son début sur une période de la durée de f.
$\texttt{xpd}[e, f]$	e dilatée ou compressée dans le temps de façon à avoir la durée de f.
$\texttt{spd}^r[e]$	e dilatée ou compressée dans le temps par un coefficient r.
$\texttt{trp}^n[e]$	transposition de e de n unités.
$\texttt{lvl}^n[e]$	augmentation des vélocités de e de n unités.
$\texttt{chn}^n[e]$	augmentation des canaux de e de n unités.

Enfin, on trouve des variables, des λ-abstractions simples et des applications :

$\diamond^e$	variable d'étiquette e.
$\lambda[\diamond^e, f]$	λ-abstraction simple.
$\texttt{@}[e, f]$	application e à f.

5.3 La traduction de $\mathbb{L}_1$ vers $\mathbb{L}_2$

La traduction d'une expression de $\mathbb{L}_1$ vers $\mathbb{L}_2$ est réalisée par la fonction $\mathcal{L}() : \mathbb{L}_1 \rightarrow \mathbb{L}_2$ qui parcours récursivement l'expression de départ et remplace les $\lambda^G[]$ par des $\lambda[]$ avec l'aide de la fonction $\mathcal{V}() : \mathbb{L}_2 \times \mathbb{L}_2 \rightarrow \mathbb{L}_2$:

$$\mathcal{L}((\ldots \lambda^G[e, f] \ldots)) \rightarrow (\ldots \lambda\left[\diamond^{e'}, \mathcal{V}(e', f')\right] \ldots) \quad \text{avec : } e' = \mathcal{L}(e) \text{et } f' = \mathcal{L}(f) \tag{5.1}$$

La fonction $\mathcal{V}(e, f)$ à pour tche de calculer le corps de l'abstraction, c'est à dire de reconnatre les occurrences de e dans f et de les remplacer par la variable $\diamond^e$. Comme nous allons le voir, la notion d'occurrence ne se limite pas à l'identité syntaxique, mais exploite notamment le fait qu'une λ-abstraction peut être vue comme un pattern qui reconnat une classe d'expressions. La fonction $\mathcal{V}()$ distingue trois cas :

1. Le cas d'occurrences syntaxiquement identiques, ainsi :

   $$\mathcal{V}(e, (\ldots e \ldots e \ldots)) \rightarrow (\ldots \diamond^e \ldots \diamond^e \ldots) \tag{5.2a}$$

2. Le cas d'événements, notes ou silences qui n'apparaissent pas tel que, mais transposés, dilatés, etc. C'est alors le paramètre de famille qui est utilisé pour reconnatre les occurrences transformées d'un événement :

   $$\begin{aligned} &\mathcal{V}(\texttt{note}[\texttt{bleue}, 1, 60, 80, 1], (\ldots \texttt{note}[\texttt{rouge}, 1, 60, 80, 1] \ldots \texttt{note}[\texttt{bleue}, 2, 64, 80, 1] \ldots)) \\ &\rightarrow (\ldots \texttt{note}[\texttt{rouge}, 1, 60, 80, 1] \ldots \texttt{trp}^4\left[\texttt{spd}^2\left[\diamond^{\texttt{note}[\texttt{bleue},1,60,80,1]}\right]\right] \ldots) \end{aligned} \tag{5.2b}$$

3. Enfin il y a le cas des abstractions. Prenons par exemple l'abstraction $p = \lambda[x, \texttt{seq}[DO, x]]$: on peut voir p comme la fonction qui met en séquence la note DO avec son argument, ou comme un pattern représentant toutes les séquences qui commencent par DO. Rendre variable p dans une expression c'est rendre variable toutes les séquences qui commencent par DO. En d'autre termes, c'est considérer toute séquence de la forme $\texttt{seq}[DO, f]$ comme l'application de p à f et la remplacer par $\texttt{@}[\diamond^p, f]$. Voici quelques exemples :

   $$\mathcal{V}(p, \texttt{seq}[DO, \texttt{seq}[RE, MI]]) \rightarrow \texttt{@}[\diamond^p, \texttt{seq}[RE, MI]]\,. \tag{5.2c}$$

   $$\mathcal{V}(p, \texttt{seq}[DO, \texttt{seq}[DO, MI]]) \rightarrow \texttt{@}[\diamond^p, \texttt{@}[\diamond^p, MI]]\,. \tag{5.2d}$$

   D'une manière générale pour p de la forme $\lambda[x_0, \ldots, \lambda[x_n, q]]$ on a :

   $$\mathcal{V}(p, (\ldots e \ldots)) \rightarrow (\ldots \texttt{@}[\texttt{@}[\diamond^p, a_1], \ldots, a_n] \ldots) \tag{5.2e}$$

   Si en substituant dans q les variables libres x_i par a_i on obtient une expression syntaxiquement identique à e.

5.4 Le langage $\mathbb{L}_3$

$\mathbb{L}_3$ est le langage des *valeurs*, c'est à dire des expressions en *extension* résultantes de l'*évaluation* des expressions en *intention*. Le principe de l'évaluation est d'effectuer tous les calculs présents dans l'expression de départ afin d'obtenir une expression plus simple, mais généralement beaucoup plus longue, décrite essentiellement en terme d'événements, de séquences et de mixages, et qui puisse être directement jouée ou affichée.

Les expressions élémentaires de $\mathbb{L}_3$ sont, comme pour les deux précédents langages, les *silences* et les *notes* :

$\texttt{sil}[m, d]$	un *silence* de famille m et une durée d.
$\texttt{note}[m, d, h, v, c]$	une *note* de famille m, de durée d, de hauteur h, de vélocité v et de canal c.

Les constructeurs musicaux ne sont plus que la séquence et le mixage. Les *calculs* représentés par les autres constructeurs ont été effectués :

$\texttt{seq}[u, v]$	mise en séquence de u et v.
$\texttt{mix}[u, v]$	mixage (superposition temporelle) de u et v.

Enfin apparaissent deux nouvelles formes, les fermetures $\Lambda[]$ et les applications non-réductibles $@'[]$:

$\Lambda[d, \lambda[\diamond^e, f], \rho]$	une *fermeture* associant une durée, une λ-abstraction et un environnement ρ.
$@'[u, v]$	application *non réductible* de u à v.

Comme on va le voir au paragraphe suivant, l'application d'une abstraction à un argument consiste à remplacer, dans le corps de l'abstraction, les occurrences de la variable par l'argument. Plutôt que d'effectuer réellement ce remplacement, on le mémorise dans un *environnement*, une sorte de liste associant des variables et des valeurs. Lorsque l'on évalue une abstraction, le résultat doit tenir compte de cette liste de substitutions d'où les fermetures $\Lambda[d, \lambda[\diamond^e, f], \rho]$ qui associent l'abstraction, son environnement et sa durée (une abstraction étant une généralisation d'une expression, sa durée est, au départ, celle de l'expression qu'elle généralise). Les applications non-réductibles résultent de l'application d'une note ou d'un silence à une autre expression ce qui ne donne lieu à aucun calcul spécifique.

5.5 Evaluation des expressions

L'évaluation consiste à effectuer tous les *calculs* contenus dans une expression $\mathbb{L}_2$, en particulier les applications d'abstractions mais également les différentes opérations musicales primitives comme la transposition, le découpage temporel, etc. L'implémentation de l'évaluation est classique. Elle est basée sur l'utilisation d'environnements, et de deux fonctions mutuellement récursives. $\mathcal{E}(e, \rho) : \mathbb{L}_2 \times \mathbb{ENV} \rightarrow \mathbb{L}_3$ évalue une expression e de $\mathbb{L}_2$dans un environnement ρ. $\mathcal{A}(u, v) : \mathbb{L}_3 \times \mathbb{L}_3 \rightarrow \mathbb{L}_3$ réalise l'application de la fonction u à l'argument v.

Comme nous l'avons dit plus haut, un environnement permet de mémoriser les valeurs à substituer aux variables. On peut voir un environnement ρ comme une fonction faisant correspondre à une variable la valeur à laquelle elle est liée. On notera $\rho(x)$ la valeur associée à x dans l'environnement ρ. On représentera par $\rho[x := v]$ l'environnement ρ étendu de façon à lier la variable x à la valeur v de sorte que : $\rho[x := v](x) = v$ et $\rho[x := v](y) = \rho(y)$. On notera ε l'environnement vide.

Les différentes fonctions intervenant dans le processus d'évaluation sont les suivantes :

$\mathcal{E}(e, \rho)$	évalue e dans l'environnement ρ.
$\mathcal{A}(u, v)$	applique u à v
$\mathcal{D}(u)$	calcule la durée de u
$\mathcal{B}(u, d)$	prend le début de u sur une durée d
$\mathcal{R}(u, d)$	supprime le début de u sur une durée d
$\mathcal{T}(u, n)$	transpose u de n unités
$\mathcal{G}(u, n)$	augmente les vélocités de u de n unités
$\mathcal{C}(u, n)$	augmente les canaux de u de n unités
$\mathcal{X}(u, r)$	dilate u par un coefficient r

Nous allons détailler toutes ces fonctions en commençant par $\mathcal{E}()$.

5.5.1 La fonction d'évaluation $\mathcal{E}()$

La fonction d'évaluation $\mathcal{E}(e,\rho) : \mathbb{L}_2 \times \mathbb{ENV} \to \mathbb{L}_3$ se définit comme suit :

$$
\begin{aligned}
\mathcal{E}(\texttt{sil}[m,d]\,,\rho) &\to \texttt{sil}[m,d] \\
\mathcal{E}(\texttt{note}[m,d,h,v,c]\,,\rho) &\to \texttt{note}[m,d,h,v,c] \\
\mathcal{E}(\texttt{seq}[e,f]\,,\rho) &\to \texttt{seq}[\mathcal{E}(e,\rho),\mathcal{E}(f,\rho)] \\
\mathcal{E}(\texttt{mix}[e,f]\,,\rho) &\to \texttt{mix}[\mathcal{E}(e,\rho),\mathcal{E}(f,\rho)] \\
\mathcal{E}(\diamond^e,\rho) &\to \rho(\diamond^e) \\
\mathcal{E}(\lambda[\diamond^e,f]\,,\rho) &\to \Lambda[d,\lambda[\diamond^e,f]\,,\rho] \text{ avec } d = \mathcal{D}(\mathcal{E}(f,\rho[\diamond^e = \mathcal{E}(e,\varepsilon)])) \\
\mathcal{E}(\texttt{@}[e,f]\,,\rho) &\to \mathcal{A}(\mathcal{E}(e,\rho),\mathcal{E}(f,\rho)) \\
\mathcal{E}(\texttt{beg}[e,f]\,,\rho) &\to \mathcal{B}(\mathcal{E}(e,\rho),d) \text{ avec } d = \mathcal{D}(\mathcal{E}(f,\rho)) \\
\mathcal{E}(\texttt{rst}[e,f]\,,\rho) &\to \mathcal{R}(\mathcal{E}(e,\rho),d) \text{ avec } d = \mathcal{D}(\mathcal{E}(f,\rho)) \\
\mathcal{E}(\texttt{xpd}[e,f]\,,\rho) &\to \mathcal{X}(u,r) \text{ avec } u = \mathcal{E}(e,\rho), r = \mathcal{D}(\mathcal{E}(f,\rho))/\mathcal{D}(u) \\
\mathcal{E}(\texttt{spd}^r[e]\,,\rho) &\to \mathcal{X}(\mathcal{E}(e,\rho),r) \\
\mathcal{E}(\texttt{trp}^n[e]\,,\rho) &\to \mathcal{T}(\mathcal{E}(e,\rho),n) \\
\mathcal{E}(\texttt{lvl}^n[e]\,,\rho) &\to \mathcal{G}(\mathcal{E}(e,\rho),n) \\
\mathcal{E}(\texttt{chn}^n[e]\,,\rho) &\to \mathcal{C}(\mathcal{E}(e,\rho),n)
\end{aligned}
\tag{5.3}
$$

5.5.2 Le calcul des applications

La fonction $\mathcal{A}(u,v) : \mathbb{L}_3 \times \mathbb{L}_3 \to \mathbb{L}_3$ applique u à v :

$$
\begin{aligned}
\mathcal{A}(\texttt{sil}[m,d]\,,w) &\to \texttt{@}'[\texttt{sil}[m,d]\,,w] \\
\mathcal{A}(\texttt{note}[m,d,h,v,c]\,,w) &\to \texttt{@}'[\texttt{note}[m,d,h+n,v,c]\,,w] \\
\mathcal{A}(\texttt{seq}[u,v]\,,w) &\to \texttt{seq}[\mathcal{A}(u,\mathcal{B}(w,\mathcal{D}(u))),\mathcal{A}(v,\mathcal{R}(w,\mathcal{D}(u)))] \\
\mathcal{A}(\texttt{mix}[u,v]\,,w) &\to \texttt{mix}[\mathcal{A}(u,w),\mathcal{A}(v,w)] \\
\mathcal{A}(\texttt{@}'[u,v]\,,w) &\to \texttt{@}'\left[\texttt{@}'[u,v]\,,w\right] \\
\mathcal{A}(\Lambda[d,\lambda[\diamond^e,f]\,,\rho]\,,w) &\to \mathcal{E}(f,\rho[\diamond^e := w])
\end{aligned}
\tag{5.4}
$$

5.5.3 Le calcul de la durée $\mathcal{D}()$

La fonction $\mathcal{D}(u) : \mathbb{L}_3 \to \mathbb{R}$ calcule la durée de u :

$$
\begin{aligned}
\mathcal{D}(\texttt{sil}[m,d]) &\to d \\
\mathcal{D}(\texttt{note}[m,d,h,v,c]) &\to d \\
\mathcal{D}(\texttt{seq}[u,v]) &\to \mathcal{D}(u) + \mathcal{D}(v) \\
\mathcal{D}(\texttt{mix}[u,v]) &\to max(\mathcal{D}(u),\mathcal{D}(v)) \\
\mathcal{D}(\texttt{@}'[u,v]) &\to \mathcal{D}(u)) \\
\mathcal{D}(\Lambda[d,\lambda[\diamond^e,f]\,,\rho]) &\to d
\end{aligned}
\tag{5.5}
$$

5.5.4 La fonction de transposition $\mathcal{T}()$

La fonction $\mathcal{T}(u,n) : \mathbb{L}_3 \times \mathbb{Z} \to \mathbb{L}_3$ transpose u de n unités.

$$
\begin{aligned}
\mathcal{T}(\texttt{sil}[m,d]\,,n) &\to \texttt{sil}[m,d] \\
\mathcal{T}(\texttt{note}[m,d,h,v,c]\,,n) &\to \texttt{note}[m,d,h+n,v,c] \\
\mathcal{T}(\texttt{seq}[u,v]\,,n) &\to \texttt{seq}[\mathcal{T}(u,n),\mathcal{T}(v,n)] \\
\mathcal{T}(\texttt{mix}[u,v]\,,n) &\to \texttt{mix}[\mathcal{T}(u,n),\mathcal{T}(v,n)] \\
\mathcal{T}(\texttt{@}'[u,v]\,,n) &\to \texttt{@}'[\mathcal{T}(u,n),v] \\
\mathcal{T}(\Lambda[d,\lambda[\diamond^e,f]\,,\rho]\,,n) &\to \Lambda[d,\lambda[\diamond^e,\texttt{trp}^n[f]]\,,\rho]
\end{aligned}
\tag{5.6}
$$

5.5.5 La fonction de modification des vélocités $\mathcal{G}()$

La fonction $\mathcal{G}(u, n) : \mathbb{L}_3 \times \mathbb{Z} \to \mathbb{L}_3$ ajoute n aux vélocités de u.

$$
\begin{aligned}
\mathcal{G}(\texttt{sil}[m, d]\,, n) &\to \texttt{sil}[m, d] \\
\mathcal{G}(\texttt{note}[m, d, h, v, c]\,, n) &\to \texttt{note}[m, d, h, v + n, c] \\
\mathcal{G}(\texttt{seq}[u, v]\,, n) &\to \texttt{seq}[\mathcal{G}(u, n), \mathcal{G}(v, n)] \\
\mathcal{G}(\texttt{mix}[u, v]\,, n) &\to \texttt{mix}[\mathcal{G}(u, n), \mathcal{G}(v, n)] \\
\mathcal{G}(@'[u, v]\,, n) &\to @'[\mathcal{G}(u, n), v] \\
\mathcal{G}(\Lambda[d, \lambda[\diamond^e, f]\,, \rho]\,, n) &\to \Lambda[d, \lambda[\diamond^e, \texttt{lvl}^n[f]]\,, \rho]
\end{aligned}
\tag{5.7}
$$

5.5.6 La fonction de modification des canaux $\mathcal{C}()$

La fonction $\mathcal{C}(u, n) : \mathbb{L}_3 \times \mathbb{Z} \to \mathbb{L}_3$ ajoute n aux canaux de u.

$$
\begin{aligned}
\mathcal{C}(\texttt{sil}[m, d]\,, n) &\to \texttt{sil}[m, d] \\
\mathcal{C}(\texttt{note}[m, d, h, v, c]\,, n) &\to \texttt{note}[m, d, h, v, c + n] \\
\mathcal{C}(\texttt{seq}[u, v]\,, n) &\to \texttt{seq}[\mathcal{C}(u, n), \mathcal{C}(v, n)] \\
\mathcal{C}(\texttt{mix}[u, v]\,, n) &\to \texttt{mix}[\mathcal{C}(u, n), \mathcal{C}(v, n)] \\
\mathcal{C}(@'[u, v]\,, n) &\to @'[\mathcal{C}(u, n), v] \\
\mathcal{C}(\Lambda[d, \lambda[\diamond^e, f]\,, \rho]\,, n) &\to \Lambda[d, \lambda[\diamond^e, \texttt{chn}^n[f]]\,, \rho]
\end{aligned}
\tag{5.8}
$$

5.5.7 La fonction de dilatation compression du temps $\mathcal{X}()$

La fonction $\mathcal{X}(u, r) : \mathbb{L}_3 \times \mathbb{R} \to \mathbb{L}_3$ dilate d'un coefficient r le temps de u.

$$
\begin{aligned}
\mathcal{X}(\texttt{sil}[m, d]\,, r) &\to \texttt{sil}[m, d * r] \\
\mathcal{X}(\texttt{note}[m, d, h, v, c]\,, r) &\to \texttt{note}[m, d * r, h, v, c] \\
\mathcal{X}(\texttt{seq}[u, v]\,, r) &\to \texttt{seq}[\mathcal{X}(u, r), \mathcal{X}(v, r)] \\
\mathcal{X}(\texttt{mix}[u, v]\,, r) &\to \texttt{mix}[\mathcal{X}(u, r), \mathcal{X}(v, r)] \\
\mathcal{X}(@'[u, v]\,, r) &\to @'[\mathcal{X}(u, r), v] \\
\mathcal{X}(\Lambda[d, \lambda[\diamond^e, f]\,, \rho]\,, r) &\to \Lambda[d * r, \lambda[\diamond^e, f]\,, \rho]
\end{aligned}
\tag{5.9}
$$

5.5.8 La fonction de découpage du début $\mathcal{B}()$

La fonction $\mathcal{B}(u, d) : \mathbb{L}_3 \times \mathbb{R} \to \mathbb{L}_3$ calcule le début de u sur la durée d.

$$
\begin{aligned}
\mathcal{B}(\texttt{sil}[m, d]\,, d') &\to \begin{cases} \texttt{sil}[m, d'] & \text{si } d' \leq d \\ \texttt{seq}[\texttt{sil}[m, d]\,, \texttt{sil}[m, d' - d]] & \text{si } d' > d \end{cases} \\
\mathcal{B}(\texttt{note}[m, d, h, v, c]\,, d') &\to \begin{cases} \texttt{note}[m, d', h, v, c] & \text{si } d' \leq d \\ \texttt{seq}[\texttt{note}[m, d, h, v, c]\,, \texttt{sil}[m, d' - d]] & \text{si } d' > d \end{cases} \\
\mathcal{B}(\texttt{seq}[u, v]\,, d') &\to \begin{cases} \mathcal{B}(u, d') & \text{si } d' \leq \mathcal{D}(u) \\ \texttt{seq}[u, \mathcal{B}(v, d' - \mathcal{D}(u))] & \text{si } d' > \mathcal{D}(u) \end{cases} \\
\mathcal{B}(\texttt{mix}[u, v]\,, d') &\to \texttt{mix}\big[\mathcal{B}(u, d'), \mathcal{B}(v, d')\big] \\
\mathcal{B}(@'[u, v]\,, d') &\to @'\big[\mathcal{B}(u, d'), v\big] \\
\mathcal{B}(\Lambda[d, \lambda[\diamond^e, f]\,, \rho]\,, d') &\to \begin{cases} \Lambda[d', \lambda[\diamond^e, f]\,, \rho] & \text{si } d' \leq d \\ \texttt{seq}[\Lambda[d, \lambda[\diamond^e, f]\,, \rho]\,, \texttt{sil}[m, d' - d]] & \text{si } d' > d \end{cases}
\end{aligned}
\tag{5.10}
$$

5.5.9 La fonction de découpage du reste $\mathcal{R}()$

La fonction $\mathcal{R}(u,d) : \mathbb{L}_3 \times \mathbb{R} \rightarrow \mathbb{L}_3$ calcule le reste de u privé de son début sur la durée d.

$$
\begin{aligned}
\mathcal{R}(\texttt{sil}[m,d]\,,d') &\rightarrow \begin{cases} \texttt{sil}[m,d-d'] & \text{si } d' < d \\ \texttt{sil}[m,0] & \text{si } d' \geq d \end{cases} \\
\mathcal{R}(\texttt{note}[m,d,h,v,c]\,,d') &\rightarrow \begin{cases} \texttt{note}[m,d-d',h,v,c] & \text{si } d' < d \\ \texttt{sil}[m,0] & \text{si } d' \geq d \end{cases} \\
\mathcal{R}(\texttt{seq}[u,v]\,,d') &\rightarrow \begin{cases} \mathcal{R}(v,d'-\mathcal{D}(u)) & \text{si } d' \geq \mathcal{D}(u) \\ \texttt{seq}[\mathcal{R}(u,d'),v] & \text{si } d' < \mathcal{D}(u) \end{cases} \\
\mathcal{R}(\texttt{mix}[u,v]\,,d') &\rightarrow \texttt{mix}\big[\mathcal{R}(u,d'),\mathcal{R}(v,d')\big] \\
\mathcal{R}(\texttt{@}'[u,v]\,,d') &\rightarrow \texttt{@}'\big[\mathcal{R}(u,d'),v\big] \\
\mathcal{R}(\Lambda[d,\lambda[\diamond^e,f]\,,\rho]\,,d') &\rightarrow \begin{cases} \Lambda[d-d',\lambda[\diamond^e,f]\,,\rho] & \text{si } d' < d \\ \texttt{sil}[\texttt{noir},0] & \text{si } d' \geq d \end{cases}
\end{aligned}
\tag{5.11}
$$

5.6 Le langage $\mathbb{L}_4$

$\mathbb{L}_4$ est le langage des séquences d'événements MIDI correspondant au contenu musical des expressions de $\mathbb{L}_3$, calculé par la fonction de traduction $\mathcal{J}() : \mathbb{L}_3 \rightarrow \mathbb{L}_4$.

Une séquence d'événements MIDI S est définie par une durée totale et une liste, éventuellement vide, d'événements MIDI. On notera une séquence d'événements MIDI S de durée D de la manière suivante :

$$
\begin{aligned}
S &= \{\ldots,\langle t_i,d_i,h_i,v_i,c_i\rangle,\ldots\}^D \\
&\text{avec} \quad t_i + d_i \leq D
\end{aligned}
$$

ou $\langle t_i,d_i,h_i,v_i,c_i\rangle$ représente un événement MIDI de date $t \geq 0$, de durée $d > 0$, de hauteur h, de vélocité v et de canal c.

Afin de simplifier la description de la fonction de traduction musicale, définissons également trois opérations sur les séquences MIDI : la concaténation, le mixage et l'étirement.

5.6.1 Concaténation de séquences

Soit deux séquences S et S' :

$$
\begin{aligned}
S &= \{\ldots,\langle t_i,d_i,h_i,v_i,c_i\rangle,\ldots\}^D \\
S' &= \{\ldots,\langle t'_j,d'_j,h'_j,v'_j,c'_j\rangle,\ldots\}^{D'}
\end{aligned}
$$

la concaténation de S et S', notée $(S \bullet S')$, se définie comme suit :

$$
(S \bullet S') = \{\ldots,\langle t_i,d_i,h_i,v_i,c_i\rangle,\ldots,\langle D+t'_j,d'_j,h'_j,v'_j,c'_j\rangle,\ldots\}^{D+D'}
$$

Cela correspond à l'idée de jouer d'abord S puis S'.

5.6.2 Mixage de séquences

Soit deux séquences S et S' :

$$
\begin{aligned}
S &= \{\ldots,\langle t_i,d_i,h_i,v_i,c_i\rangle,\ldots\}^D \\
S' &= \{\ldots,\langle t'_j,d'_j,h'_j,v'_j,c'_j\rangle,\ldots\}^{D'}
\end{aligned}
$$

le mixage de S et S', noté $(S \parallel S')$, se définit comme suit :

$$
(S \parallel S') = \{\ldots,\langle t_i,d_i,h_i,v_i,c_i\rangle,\ldots,\langle t'_j,d'_j,h'_j,v'_j,c'_j\rangle,\ldots\}^{max(D,D')}
$$

Cela correspond à l'idée de jouer simultanément S et S'.

5.6.3 Etirement temporel d'une séquence

Soit une séquences S :

$$
S = \{\ldots,\langle t_i,d_i,h_i,v_i,c_i\rangle,\ldots\}^D
$$

L'étirement (ou la compression) temporel de la séquence S à la durée D', noté $(S \star D')$, se définit comme suit :

$$
\begin{aligned}
(S \star 0) &= \{\}^0 \\
(S \star D') &= \{\ldots,\langle r*t_i,r*d_i,h_i,v_i,c_i\rangle,\ldots\}^{D'} \\
&\text{avec } r = D'/D
\end{aligned}
$$

Les dates et les durées des notes sont ajustées en proportion de façon à ce que la séquence dure exactement D'.

5.7 La fonction de traduction musicale $\mathcal{J}()$

La fonction $\mathcal{J}(v) : \mathbb{L}_3 \rightarrow \mathbb{L}_4$ calcule le contenu musical de l'expressions évaluée v sous la forme d'une séquence d'événements MIDI.

$$
\begin{aligned}
\mathcal{J}(\texttt{sil}[m,d]) &\rightarrow \{\}^d \\
\mathcal{J}(\texttt{note}[m,d,h,v,c]) &\rightarrow \{\langle 0,d,h,v,c\rangle\}^d \\
\mathcal{J}(\texttt{seq}[u,v]) &\rightarrow \mathcal{J}(u) \bullet \mathcal{J}(v) \\
\mathcal{J}(\texttt{mix}[u,v]) &\rightarrow \mathcal{J}(u) \parallel \mathcal{J}(v) \\
\mathcal{J}(\texttt{@}'[u,v]) &\rightarrow \mathcal{J}(u) \\
\mathcal{J}(\Lambda[d,\lambda[\diamond^e,f]\,,\rho]\,,r) &\rightarrow \mathcal{J}(\mathcal{E}(f,\rho[\diamond^e := \mathcal{E}(e,\varepsilon)])) \star d
\end{aligned}
\tag{5.12}
$$

6 Conclusion

Il est courant en mathématique de définir un ensemble en extension (énumérant nommément tous les éléments qui composent l'ensemble) ou en compréhension (indiquant les propriétés des éléments qui composent l'ensemble sans avoir à les nommer explicitement). L'un des rôles essentiels d'un langage de composition musicale ou de synthèse sonore est de permettre à l'utilisateur de décrire des objets musicaux en intention, ce qui est en quelque sorte l'analogue d'une définition en compréhension, et d'assurer la relation avec les descriptions en extension correspondantes.

Ces deux types de description sont nécessaires à l'utilisateur. La description en extension est celle dont on aura besoin *in fine*, celle qui nous permet de percevoir l'objet et d'en travailler les détails. La description en intention, quant à elle, présente de nombreux intérêts :

1. **concision**. Une description en intention peut être extrêmement concise par rapport à une description en extension. Dans ce cas la description en intention est un moyen efficace et rapide pour décrire un objet.
2. **structuration**. Une description en intention permet de rendre explicite *l'idée* sous-jacente à un objet, sa structure générale, ses principes de construction. Elle se prête à l'analogie, à la généralisation, à la systématisation et à la réutilisabilité dans d'autres contextes.
3. **expérimentation**. Les modifications d'une description en intention ont généralement une grande portée. Une *petite* modification de la description en intention peut produire une *grande* modification de la description en extension. Lorsque l'ordinateur prend en charge le calcul de la description en extension, il est alors très facile d'avoir une démarche expérimentale du type *qu'est-ce qui se passe si* je remplace ceci par cela dans ma description en intention.

Pouvoir faire des descriptions en intention avec toute la généralité voulue suppose une forme de langage de programmation. L'approche fonctionnelle, par sa puissance expressive (voir [Hughes 89] à ce sujet), nous parait d'un grand intérêt dans le domaine musical. Plusieurs environnements musicaux ont été réalisés sur ce modèle, citons en particulier les travaux de R. Dannenberg [1984, 1989, 1991] et de P. Desain et H. Honing [1991]. Plus récemment P. Hudak [1996] a proposé, avec Haskore, un élégant formalisme et un ensemble de modules musicaux pour le langage fonctionnel Haskell.

La première originalité de notre approche se situe dans le traitement du langage de programmation qui n'est pas extérieur au langage musical, mais qui en dérive très directement par l'adjonction des concepts d'abstraction et d'application du λ-calcul. Ce modèle de *programmation homogène*, présenté dans [Orlarey et al. 94], s'applique aisément à d'autres domaines que la musique. La deuxième originalité est l'introduction de la notion d'abstraction généralisée qui permet une forme de programmation par l'exemple, laissant à la machine le soin de construire les abstractions d'après les indications de l'utilisateur, et qui exploite le fait qu'une abstraction peut être vue comme un pattern dans la généralisation d'une autre expression. Ces éléments, associés à l'interface utilisateur d'*Elody*, favorisent, nous semble-t-il, une attitude expérimentale et ludique de la part de l'utilisateur y compris dans son activité de programmation.

7 Bibliographie

[Barendregt 1984] H. P. Barendregt, The Lambda Calculus, Its Syntax and Semantics. North-Holland, Amsterdam, 1984.

[0.1cm][Dannenberg 1989], R. B. Dannenberg, The Canon Score Language, Computer Music Journal, 13 (1), pp. 47-56, 1989.

[Dannenberg et al. 1991], R. B. Dannenberg, C. L. Fraley and P. Velikonja, Fugue : A Functional Language for Sound Synthesis, Computer, 24 (7), pp. 36-42, 1991.

[Desain and Honing 1991], P. Desain and H. Honing, Time Functions Function Best as Functions of Multiple Times, Computer Music Journal, 16 (2), pp. 17-34, 1991.

[Hudak 1996], P. Hudak, T. Makucevich, S. Gadde, B. Whong, Haskore music notation - an algebra of music, Journal of Functional Programming, 6(3), Juin 1996.

[Hughes 1989] J. Hughes, Why Functional Programming Matters, The computer Journal 32 (2), pp. 98-107, 1989.

[Orlarey 1994], Y. Orlarey, D. Fober, S. Letz, M. Bilton, Lambda Calculus and Music Calculi, Proceedings of the ICMC 1994.