GRAME
Laboratoire de recherche en informatique musicale

Mémoire de stage en vue de l'obtention du diplome d'ingénieur en
Génie mathématique et modélisation
présenté au
C U S T
Institut des sciences de l'ingénieur de Clermond-Ferrand

1997

Développement d'un opérateur d'abstraction généralisée pour le langage de programmation musicale Élody.

Grégory LEPLATRE

Grame
Centre National de Création Musicale
9, rue du Garet BP 1185
FR - 69202 Lyon Cedex 01

Résumé

Dans le cadre des recherches réalisées au laboratoire d'informatique musicale de Grame autour des langages *homogènes* dérivés du λ-calcul non typé, on se propose d'étudier les possibilités de généralisation de l'*abstraction* héritée du λ-calcul. Dans cette optique, on formalise une notion de *généralité* des expressions du langage. En référence à cette notion, on peut envisager *l'abstraction généralisée* d'une expression e dans une expression f comme le moyen de désigner dans f tous les sous-termes q de f tels que e est plus générale que q. L'opérateur ainsi défini dans un λ-calcul non typé peut être facilement adapté à un langage de programmation musicale dérivé du λ-calcul. Dans *Elody*, un langage de ce type développé à Grame en JAVA, on a remplacé de cette manière, l'opérateur d'abstraction simple implémenté à l'origine, par un opérateur d'abstraction généralisé. Ce qui a pour incidence d'élargir la fonctionnalité du langage.

Abstract

During the course of research at the Grame computer music research laboratory into consistent languages, derived from the non-typed λ-calculous, study of the possibilities of generalizing the abstraction inherited from the λ-calculous has been realized. In this light, one formalises a notion of *generality* for the language terms. Using this idea, one can envisage the *general abstraction* of an expression e in an expression f as the means of showing in f all the sub-terms q of f, such as e being more general than q. The operator thus defined in a non-typed λ-calculous may be easily adapted to a music programming language derived from the λ-calculous theory. In *Elody*, a language of this type developed at Grame in JAVA, the simple abstraction operator has been replaced in this way by the general abstraction operator. This having the consequence of widening the functionality of the language.

Table des matières

Introduction

L'environnement de représentation *Elody* est basé sur un langage de programmation dérivé très directement du λ-calcul. Les concepts d'*abstraction* et d'*application* lui permettent de conférer à un langage purement descriptif les fonctionnalités d'un langage de programmation. L'abstraction permet ainsi de construire des fonctions, en désignant les variables par l'exemple, plutôt que de les manipuler en tant que concepts abstraits. Aussi, dans un système dont la *fonctionnalité* est fondée sur la notion d'abstraction, il est important de disposer d'un solide outil d'abstraction.

Abstraire *simplement* une expression dans une autre, c'est rendre variables les occurrences de la première dans la seconde. C'est donc une opération fondée sur la notion d'égalité syntaxique. Généraliser la notion d'abstraction, c'est se donner la possibilité de rendre variables des expressions en référence à une notion de *généralité* des expressions musicales.

En premier lieu, il s'agira de formaliser cette notion subjective par une relation d'ordre, en cohérence avec les principes du λ-calcul. Alors on pourra substituer l'égalité syntaxique dans le mécanisme de l'abstraction simple, par cette relation d'ordre. On aura ainsi défini le principe de fonctionnement de l'abstraction généralisée. Celui-ci sera formalisé dans le cadre du λ-calcul, avant d'être transposé au cas particulier du langage de programmation dérivé du λ-calcul *Elody*.

L'abstraction généralisée, ainsi définie, permettra de désigner une fonction, un processus compositionnel ou l'application d'un processus dans une expression musicale.

Les techniques auxquelles on fera appel pour mettre en oeuvre un algorithme implémentant l'opérateur d'abstraction généralisée s'apparenteront à des techniques de *filtrage* ou *semi-unification*. L'utilisation du langage de programmation LISP se révélera parfaitement adapté au développement d'algorithmes de ce type, basés sur le parcours récursif d'arborescences.

Il s'agira ensuite d'implémenter ces algorithmes en JAVA, puis de les intégrer à l'interprèteur du langage de programmation *Elody*. L'opérateur finalement intégré à l'environnement de composition, il sera intéressant d'en tester le fonctionnement par une utilisation *courante* de *Elody*, c'est à dire de réaliser quelques petites phrases musicales qui font appel à l'abstraction généralisée.

Présentation de l'association Grame

Grame, centre de musique contemporaine, a été créé en 1982 à l'initiative de James Giroudon et Pierre Alain Jaffrennou, grâce au soutien de la Direction de la Musique et de la Danse, avec pour mission de développer en région Rhône-Alpes un pôle d'activités musicales alliant ensembles instrumentaux, techniques électroacoustiques, techniques informatiques et intégrant la recherche scientifique comme une donnée fondamentale de la création artistique contemporaine. Grame est un centre de ressources artistiques, intellectuelles et techniques au service des professionnels de la musique. C'est aussi une organisation dédiée à la diffusion des musiques d'aujourd'hui en direction d'un large public.

Les activités musicales de Grame couvrent au sein d'une même structure, la création, la recherche scientifique, la diffusion et la formation. Ces activités reposent sur un ensemble de moyens propres se regroupant autour de studios de composition mis à la disposition de compositeurs invités et en résidence, d'un laboratoire de recherche en informatique musicale, du festival «Musiques en Scène», d'une politique de diffusion musicale en France et à l'étranger, d'un ensemble instrumental associé : l'Ensemble Orchestral Contemporain et d'un colloque national organisé annuellement. L'équipe de Grame forme un collège de compositeurs, de chercheurs, d'administratifs et de techniciens représentant une vingtaine de personnes. Par son insertion professionnelle dans la vie artistique et scientifique, Grame constitue aujourd'hui un lieu ouvert et privilégié de rencontre et de communication entre musiciens, compositeurs et chercheurs.

Le laboratoire de recherche en informatique musicale de Grame

L'informatique musicale est une discipline dont les premiers travaux remontent à la fin des années 50. Bien que récente, elle puise ses racines dans une longue tradition d'apports et d'influences réciproques qu'entretiennent Musique et Science depuis des siècles. Elle est aujourd'hui un lieu de rencontre pluridisciplinaire privilégié entre artistes et scientifiques. Par son point de vue spécifique et la diversité des problèmes qu'elle soulève, l'informatique musicale offre un terrain favorable aux recherches scientifiques dans les domaines de l'informatique, de l'acoustique, de la musicologie et des sciences cognitives.

Les travaux de recherche en informatique musicale visent non seulement une meilleure compréhension du monde acoustique et cognitif lié à la pratique musicale, mais ils cherchent aussi à favoriser l'évolution de cette même pratique, tant du point de vue de la composition de l'oeuvre, de son interprétation, de sa réalisation, que de celui de sa diffusion auprès de l'auditeur. Depuis plus de trente ans, ces travaux ont produits des résultats dont l'intérêt dépasse souvent le cadre strictement musical pour rejaillir sur d'autres domaines scientifiques ou technologiques.

Le laboratoire de Grame développe depuis 1982 une activité de recherche fondamentale et appliquée en Informatique Musicale.

Il se consacre à l'étude des systèmes d'écriture, de notation et de représentation formelle comme support des activités cognitive de conception et de création, en particulier dans le domaine artistique et musical, ainsi qu'à l'étude des architectures logicielles aux systèmes musicaux.

Ces recherches recouvrent des champs tels que les langages de programmation, les systèmes d'exploitation temps réel, les systèmes communicants et répartis, l'intelligence artificielle, les sciences

cognitives.

Architecture des Systèmes Musicaux

Objectifs

Cet axe de recherche se propose d'étudier des méthodes et des outils au service des concepteurs d'applications musicales dans le but de faciliter la conception, le développement et la mise au point des applications musicales, de permettre une prise en compte efficace et simplifiée de la dimension temporelle inhérente aux applications musicales et enfin de proposer un cadre de coopération et de collaboration homogène, général et durable aux applications musicales, favorisant une attitude «créative» des utilisateurs. Les questions abordées ici concernent bien entendu les systèmes temps-réel, mais également les modèles de communication (la communication étant entendue ici comme le moyen par lequel les parties interagissent pour former un tout), ainsi que l'échange de données multimédia et la normalisation de ces mêmes données.

Orientations

- les systèmes temps-réels.
- la synchronisation et l'ordonnancement de tâches.
- les processus répartis et communicants.
- la thématique de la communication et les modèles associés.
- l'échange de données multimédia.
- la normalisation des documents.
- le contrôle d'équipements externes.
- les domaines multimédia et industriel.

Applications

Ces travaux ont notamment donné naissance à MidiShare, système d'exploitation musical, lauréat des trophées Apple 89 et du prix Paris-Cité 90, adopté aujourd'hui par plusieurs industriels européens du logiciel musical. Tout en offrant des performances élevées, MidiShare simplifie considérablement la conception d'applications musicales temps-réel complexes et facilite le portage entre plusieurs plates-formes hardware. MidiShare permet en outre à ces applications de fonctionner simultanément sur une même machine et de collaborer entre elles par le biais d'un système sophistiqué de communication inter-applications.

Dans le cadre de ces recherches, le laboratoire de Grame a également développé un système de communication temps réel sur réseau ethernet basé sur l'architecture de MidiShare.

Langages Formels pour l'Ecriture Musicale

Objectifs

Ce deuxième axe de recherche vise à mettre au service des compositeurs les possibilités de formalisation et de manipulation symbolique offertes par l'informatique pour faciliter l'élaboration et la mise en oeuvre de nouvelles techniques d'écriture musicale, favoriser l'articulation entre formalisation et expérimentation dans la démarche de composition et enrichir les modalités de représentation, d'abstraction et de manipulation des matériaux musicaux en continuité avec l'écriture musicale traditionnelle.

Orientations

- Modèles de programmation, en particulier le lambda-calcul, la logique combinatoire et la programmation fonctionnelle.
- Problématique des langages formels comme outils cognitifs pour les activités de conception, d'invention et de création.
- Etude des rapports entre notation, écriture et pensée musicale.
- Prise en compte des problèmes de mémoire et de patrimoine musical induits par l'informatique.

Applications

Ces travaux ont donné lieu à plusieurs générations de langages d'assistance à la composition musicale (MidiLogo, MidiLisp, CLCE). Ces langages prolongent et enrichissent le champ de la pensée musicale en permettant d'explorer des domaines de formalisation jusque là inaccessibles. Ils sont aujourd'hui à la base de nouvelles techniques d'écriture mises en oeuvre notamment dans les musiques interactives.

Première partie

Contexte de l'étude

Chapitre 1

Introduction au λ-calcul

1.1 Présentation

Inventé vers 1930 par Alonzo Church, le λ-calcul propose une formalisation du calcul centrée sur la notion de fonction, par opposition aux machines de Turing, basées sur la notion d'état. Bien que se fondant sur un nombre restreint de principes, le λ-calcul permet, d'après la thèse de Church-Turing, d'exprimer toute fonction effectivement calculable. (cf [Barendregt 1984])

Il constitue l'un des fondements théoriques de l'informatique, en particulier des langages de programmation. Il a eu en particulier une influence considérable sur la conception et l'implémentation des langages de programmation fonctionnels. Le λ-calcul est d'une certaine manière un langage de programmation fonctionnelle réduit à son essence.

En référence à cette essence, il nous sera possible de conférer à un système de notations purement descriptif, les caractéristiques d'un langage de programmation. En l'occurrence, il s'agit du développement d'un langage de programmation musicale.

Le λ-calcul est un calcul formel, c'est à dire un système de notation pouvant être manipulé en se basant uniquement sur la structure des éléments du langage, la syntaxe des expressions, sans qu'il y ait besoin d'en comprendre le sens.

En tant que calcul formel, le λ-calcul se définit par trois éléments :

1. ***des règles de syntaxe***,
2. ***des règles de réécriture***,
3. ***une stratégie d'évaluation***.

Les règles de syntaxe décrivent comment construire les expressions bien formées du langage. On appellera désormais ces expressions λ-termes ou λ-expressions. Les règles de réécriture indiquent les transformations licites que l'on peut opérer sur ces expressions. Enfin, la stratégie d'évaluation précise comment utiliser à bon escient les règles de réécriture afin d'évaluer une expression. L'évaluation d'une expression consiste à en faire des transformations successives conformément à la stratégie d'évaluation choisie, jusqu'à ce qu'il n'y ait plus de transformation possible. On constatera par la suite que l'évaluation d'une expression ne se termine pas toujours.

1.2 La syntaxe du λ-calcul

La formalisation minimale du λ-calcul tient en trois règles définissant la notion de λ-terme :

Définition 1.2.1. *Soit $\mathcal{V}$ un ensemble dénombrable de variables. L'ensemble des λ-termes est défini par :*

- *Toute variable est un λ-terme.*
- *Si T_1 et T_2 sont des λ-termes, $(T_1\ T_2)$ est un λ-terme, application de la fonction T_1 à son argument T_2.*
- *si x est une variable et T un λ-terme, $\lambda x.T$ est un λ-terme, abstraction de x dans T.*

Si la concision de cette définition fait la puissance du λ-calcul, il n'est pas inutile de préciser les notions d'abstraction et d'application.

L'application

L'application d'un λ-terme f à un λ-terme a, notée $(f\ a)$ représente l'opération du calcul usuel notée $f(a)$, par laquelle on applique la fonction f à son argument a. Pour simplifier les notations, on notera par la suite les expressions du type : $((((f\ a)\ b)\ c)\ d)$ par $(f\ a\ b\ c\ d)$.

L'abstraction

L'abstraction est utilisée pour définir une fonction dont le corps est constitué par un λ-terme auquel on associe une variable. Voici quelques exemples de fonctions, représentées par des λ-termes et par leur notation mathématique habituelle :

$$\lambda x.x \iff f(x) = x$$

$$\lambda x.0 \iff f(x) = 0$$

$$\lambda x.(x^2 - 3x) \iff f(x) = x^2 - 3x$$

1.3 Les règles de réécriture du λ-calcul

Les règles de réécriture du λ-calcul définissent les transformations licites que l'on peut opérer à une expression. Elles sont fondées sur la notion de rédex.

Définition 1.3.1. *On appelle rédex tout λ-terme de la forme :*

$$(\lambda x.e\ f)$$

Où x est une variable, e et f des λ-termes.

Le rédex formalise le calcul de e lorsque x vaut f. La réalisation du calcul se fait par substitution de f à x dans e. Cette substitution s'effectue par le biais d'une opération dite de β-conversion qu'on représentera par : $\overset{\beta}{\longleftrightarrow}$. Aussi, on notera une succession de β-conversions par $\overset{\beta*}{\longleftrightarrow}$.

Donnons quelques exemples de β-conversions, après quoi on définira précisément le mécanisme de substitution du λ-calcul.

$$(\lambda x.x\ \ e) \overset{\beta}{\longleftrightarrow} e$$

$$(\lambda x.y\ \ e) \overset{\beta}{\longleftrightarrow} y$$

$$(\lambda z.\lambda w.z\ \ y) \overset{\beta*}{\longleftrightarrow} \lambda w.y$$

En revanche, on ne peut pas écrire :

$$(\lambda x.\lambda y.x\ \ y) \overset{\beta*}{\longleftrightarrow} \lambda y.y$$

Les deux derniers exemples de β-conversion témoignent du fait que la dénomination des variables compte dans le processus de substitution. Intuitivement, les variables sont muettes si elles sont globales ou locales, mais pas les deux. Pour lever l'ambigüité qui pèse sur la dénomination des variables, on distinguera les variables libres des variables liées dans un λ-terme.

1.3.1 Variables libres et variables liées d'une expression

Définition 1.3.2. *Une occurrence d'une variable x dans un terme e est libre si e vérifie l'une des conditions suivantes :*

- $e \equiv x$
- $e \equiv (e_1\ e_2)$ *et x a une occurrence libre dans e_1 ou e_2*
- $e \equiv \lambda y.f$ *avec $x \not\equiv y$ et x a une occurrence libre dans f*

Définition 1.3.3. *Une variable est libre dans un λ-terme e si elle a au moins une occurrence libre dans e*

On notera $FV(e)$ l'ensemble des variables libres d'une λ-expression e.

Définition 1.3.4. *Une variable x est liée dans un λ-terme e si e vérifie l'une des conditions suivantes :*

- $e \equiv \lambda x.f$
- $e \equiv (e_1\ e_2)$ *et x est liée dans e_1 ou e_2*
- $e \equiv \lambda y.f$ *avec $x \not\equiv y$ et x est liée dans f*

On notera $BV(e)$ l'ensemble des variables liées d'une λ-expression e.

Définition 1.3.5. *Un λ-terme e est clos si toutes ses variables sont des variables liées dans e, ou de façon équivalente, si e ne possède pas de variables libres.*

On peut maintenant proposer une définition générale de la substitution d'un λ-terme f à une variable x dans un λ-terme e.

1.3.2 Les règles de substitution

Définition 1.3.6. *La substitution d'un λ-terme f à une variable x dans un λ-terme e est définie par :*

Si e est une variable
  Si $e \equiv x$,
    alors le résultat de la substitution est f
  Si $e \not\equiv x$,
    alors le résultat de la substitution est e
Si $e \equiv (e_1\ e_2)$
  alors le résultat est l'application de la substitution de f à x dans e_1 à la substitution de f à x dans e_2
Si $e \equiv \lambda y.g$
  Soit y' une variable jamais utilisée, alors le résultat est l'abstraction de y' dans la substitution de f à x dans la substitution de y' à y dans g

Cette définition indique qu'une expression de la forme $(\lambda x.e\ f)$, c'est à dire, l'application d'une abstraction $\lambda x.e$ à un argument f peut être convertie en une nouvelle expression que nous noterons $e[x/f]$ par β-conversion, en substituant dans e toutes les occurrences libres de la variable x par l'argument f. La β-conversion correspond à l'idée intuitive d'un appel de fonction avec passage de paramètre.

En vertu des résultats énoncés précédemment, on formalise les règles de substitution du λ-calcul comme suit :

$(R1)$ $\quad x[x/f] = f$

$(R2)$ $\quad y[x/f] = y$

$(R3)$ $\quad (e_1\ e_2)\,[x/f] = (e_1[x/f]\ e_2[x/f])$

$(R4)$ $\quad (\lambda x.e)[x/f] = \lambda x.e \quad$ si $x \not\equiv y$

$(R5)$ $\quad (\lambda x.e)[y/f] = \lambda x.e[y/f] \quad$ si $x \notin FV(f)$

$(R6)$ $\quad (\lambda x.e)[y/f] = \lambda z.e[x/z]\,[y/f] \quad$ si $x \in FV(f)$
où z est une nouvelle variable telle que :
$z \notin FV(e) \cup FV(f)$

Les deux premières règles correspondent à l'idée intuitive de substitution. La règle $(R3)$ signifie que la substitution dans une application doit s'opérer dans les deux membres de cette application. La règle $(R4)$ indique que les substitutions ne se font que sur les variables libres. Les règles $(R5)$ et $(R6)$ permettent de distinguer les cas où il y a conflits de noms des cas où il n'y a pas conflit de noms.

Si l'on reprend maintenant l'expression $(\lambda x.\lambda y.x\ \ y)$, on a :

$(\lambda y.x)[x/y] = (\lambda z.x[y/z])[x/y]$ $\quad$ par $(R6)$

$(\lambda z.x[y/z])[x/y] = (\lambda z.x)[x/y]$ $\quad$ par $(R2)$

$(\lambda z.x)[x/y] = \lambda z.x[x/y]$ $\quad$ par $(R5)$

$\lambda z.x[x/y] = \lambda z.y$ $\quad$ par $(R1)$

1.3.3 Les règles de conversion

En nous appuyant sur les notions de variables libres et de substitution que nous venons de définir, nous pouvons formuler trois règles de conversion du λ-calcul :

1. *β-conversion (application) :* $(\lambda x.e\ f) \overset{\beta}{\longleftrightarrow} e[x/f]$

2. *α-conversion (renommage) :* $\lambda x.e \overset{\alpha}{\longleftrightarrow} \lambda y.e[x/y]$ si $y \notin FV(e)$

3. *η-conversion :* $\lambda x.(e\ x) \overset{\eta}{\longleftrightarrow} e$ si $x \notin FV(e)$

1.4 La stratégie d'évaluation

Ayant mis en évidence trois règles de conversions, on peut définir une stratégie d'évaluation comme le processus d'applications successives de ces trois règles jusqu'à ce qu'aucune des trois ne puisse plus être appliquée. Aussi, il est évident que l'évaluation d'une expression ne s'effectue qu'en appliquant des règles de conversion dans un sens. On parle dans ce cas de règles de réduction. On appelle β-réduction la règle de β-conversion utilisée dans le sens gauche-droite et qu'on notera $\overset{\beta}{\longrightarrow}$. De même, on parle de α-réduction et de η-réduction. L'évaluation d'une expression n'impose pas d'utiliser les trois règles de réduction. Aussi, dans la pratique on restreint souvent l'évaluation d'une expression aux applications successives de β-réductions.

1.4.1 Forme normale d'une λ-expression

Définition 1.4.1. *La forme normale d'une λ-expression e est une λ-expression sans rédex obtenue par β-réductions successives de e.*

Il se peut que l'évaluation d'un λ-terme ne se termine pas. Considérons par exemple l'abstraction V définie par :

$$V \equiv \lambda x.(x\ x)$$

Si on applique V à lui-même,on obtient par β-réductions successives :

$$\begin{aligned}(V\ V) = (\lambda f.f\ V) &\overset{\beta}{\longrightarrow} (V\ V)\\ &\overset{\beta*}{\longrightarrow} (V\ V)\end{aligned}$$

C'est à dire indéfiniment l'application $(V\ V)$. On dit alors que $(V\ V)$ ne possède pas de forme normale. De façon générale, un λ-terme ne possède pas de forme normale si son évaluation ne se termine pas.

1.4.2 Quelques expressions particulières

L'identité Il s'agit d'une expression de la forme $\lambda x.x$, dans laquelle x est une λ-expression. Sa particularité fondamentale est de permettre de réécrire tout λ-terme comme l'application de l'identité à lui-même.

L'opérateur de point fixe Le λ-calcul offre la possibilité de manipuler des objets infinis par le biais d'une expression appelée opérateur de point fixe, ou combinateur-Y

Considérons l'abstraction :

$$V \equiv \lambda f.(f\ f)$$

Si on applique V à lui-même, on obtient après une seule β-réduction :

$$(V\ V) = (\lambda f.(f\ f)\ V) \xrightarrow{\beta} (V\ V)$$

L'évaluation de cette expression, qui ne possède pas de forme normale, ne se termine donc pas. En utilisant cette propriété, il est possible de construire des objets infinis.

Par exemple et en guise de préliminaire à la présentation du principe de fonctionnement d'*Elody*, supposons que les expressions élémentaires du λ-calcul sont des notes et qu'on dispose de deux opérateurs binaires permettant de construire séquences et accords. On notera $[do; mi]$ la séquence de do et mi et $\left[\frac{do}{mi}\right]$ l'accord de do et mi. On peut alors représenter une séquence infinie de do comme l'application du λ-terme suivant à lui-même :

$$V' \equiv \lambda f.[do; (f\ f)]$$

Si l'on procède à des réductions successives de $(V'\ V')$, on obtient :

$$\begin{aligned}(V'\ V') &= (\lambda f.[do; (f\ f)]\ V') \\ &\xrightarrow{\beta} [do; (V'\ V')] \\ &\xrightarrow{\beta^*} [do; do; do; \ldots]\end{aligned}$$

Expressions *constantes* On appellera fonctions constantes toutes les abstractions de la forme $\lambda\diamond^{e_1}.\lambda\diamond^{e_2}.\lambda\diamond^{e_3}.\ldots.\lambda\diamond^{e_n}.p$ dans lesquelles les variables $\diamond^{e_1}\ldots\diamond^{e_n}$ n'appartiennent pas toutes à p. On adopte le terme *constantes* par analogie avec la notation classique des fonctions à une variable, avec laquelle une fonction constante s'exprime par $f(x) = a$. En λ-calcul, il s'agit d'un terme de la forme suivante : $\lambda x.a$. De façon générale, on a donc :

$$f(x) = a \iff \lambda x.a$$

$$f(x, y) = [a; x] \iff \lambda x.\lambda y.[a; x]$$

$$f(x_1, \ldots, x_n) = \left[\left[\frac{x_1}{x_2}\right]; x_3\right] \iff \lambda x_1.\lambda x_2.\ldots.\lambda x_n.\left[\left[\frac{x_1}{x_2}\right]; x_3\right]$$

Il est important de noter pour la suite de l'étude, que si certaines expressions prennent un sens particulier, elles suivent toujours les règles de syntaxe du λ-calcul.

Chapitre 2

L'environnement de composition musicale *Elody*

2.1 Présentation

Elody est un environnement de composition musicale permettant la description et la manipulation algorithmique de structures musicales et de procédés compositionnels [Orlarey et al. 1997]). Il est fondé sur un langage de programmation *homogène* dérivé du λ-calcul dont les termes sont tous munis d'une dimension temporelle et où les notions usuelles de structuration des objets musicaux (séquences, accords, etc.) s'appliquent également aux programmes de composition que l'utilisateur va pouvoir décrire.

Dans l'environnement de composition, on peut distinguer de façon schématique l'interface utilisateur du langage de programmation. On ne s'attardera pas ici sur l'interface utilisateur (voir [Orlarey et al. 1997] pour une présentation]. En effet, le centre de l'étude se situe au niveau du langage de programmation, c'est à dire de l'interprèteur du langage.

2.2 Un langage de programmation musicale *homogène* dérivé du λ-calcul

Le principe général d'utilisation d'*Elody* consiste à *construire* des objets musicaux que l'on assemble suivant différentes modalités, par exemple en séquence ou en parallèle. Plus précisément, on construit des objets musicaux en appliquant[1] un opérateur binaire à deux arguments. Le résultat obtenu est du même type que les deux arguments. Ainsi, toutes les expressions manipulées par l'utilisateur sont d'un seul et même type. C'est pourquoi on parle de langage *homogène*.

Par ailleurs, l'utilisation des concepts fondamentaux du λ-calcul est le moyen de conférer à un langage de représentation purement descriptif les fonctionnalités d'un véritable langage de programmation [Orlarey et al. 1994].

2.2.1 Le langage descriptif

Dans *Elody*, les objets élémentaires sont des notes ou des silences – on leur associera de façon non exhaustive les paramètres usuels MIDI que sont hauteur, vélocité, durée et canal, ainsi qu'une famille symbolisée par une couleur. Pour que les expressions construites à partir de ces objets de base prennent un sens musical, il faut qu'elles puissent s'organiser conformément à l'idée que des événements musicaux sont ou bien successifs, ou bien simultanés (voir [Chemillier 1994] pour des références algébriques). On dispose donc d'un opérateur de mise en séquence, permettant d'organiser

[1] Il ne s'agit pas ici de l'application au sens du λ-calcul

les événements selon une dimension temporelle et que l'on représentera indifféremment par $\mathsf{seq}[e, f]$ ou par $[e; f]$. On se donne également un opérateur de mixage, permettant d'organiser ces événements suivant une seconde dimension. On le notera de même $\mathsf{mix}[e, f]$ ou $\left[\frac{e}{f}\right]$.

A ces constructeurs canoniques s'ajoutent d'autres constructeurs, qui permettent de modifier les paramètres constitutifs des événements de base (durée, hauteur, vélocité et canal). Ces constructeurs sont décrits plus précisément dans le paragraphe 2.3.1.

2.2.2 Le langage de programmation

On obtient un langage de programmation musicale en munissant le langage de description décrit ci-dessus, des opérateurs d'abstraction et d'application hérités du λ-calcul.

L'abstraction

L'opérateur d'abstraction, qui prend pour arguments deux expressions du langage, permet de créer des fonctions, en rendant variable toutes les occurrences du premier argument dans le second (cf 1.2 pour l'analogie entre fonctions au sens usuel et au sens du λ-calcul).

L'application

L'opérateur d'application, qui prend également pour arguments deux expressions du langage, *applique* le premier argument au second.

Considérons l'exemple suivant :

Soient e et f deux expressions définies par :

$$\begin{cases} e \equiv fa \\ \text{et} \\ f \equiv \left[do; \left[\frac{si}{fa}\right]\right] \end{cases}$$

L'abstraction de e dans f donne l'expression suivante :

$$\lambda fa. \left[do; \left[\frac{si}{fa}\right]\right]$$

On dispose donc d'une fonction que l'on peut appliquer à un argument. Appliquons-la par exemple à la note mi. On obtient l'expression :

$$\left(\lambda fa. \left[do; \left[\frac{si}{fa}\right]\right]\ mi\right)$$

qui donne après réduction :

$$\left[do; \left[\frac{si}{mi}\right]\right]$$

L'opération a donc consisté à substituer à la note fa qui a été abstraite, la note mi, en lui appliquant la fonction résultant de l'abstraction.

Ce moyen simple de réaliser des programmes permet de construire puis d'appliquer des fonctions en désignant les variables correspondantes par l'exemple, plutôt qu'en les manipulant en tant que concepts abstraits.

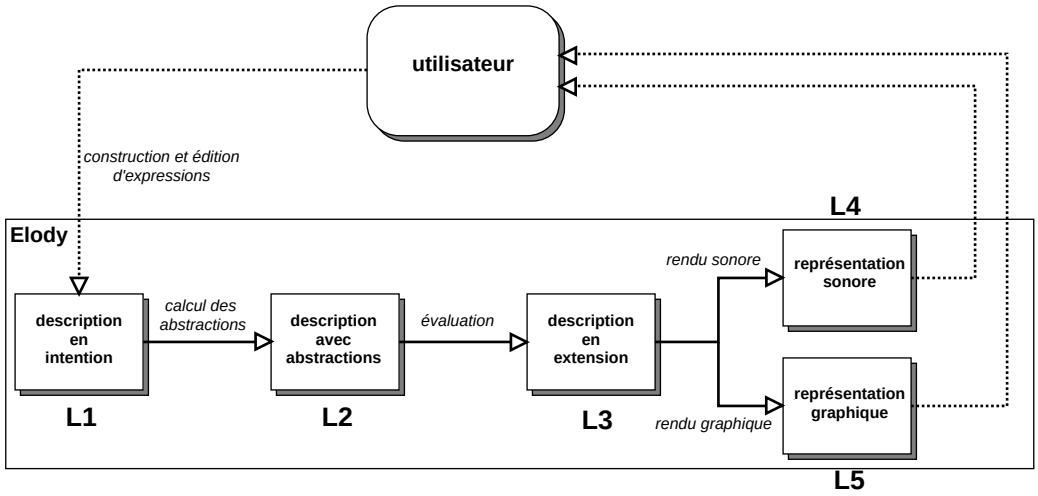


FIG. 2.1 – Schéma conceptuel d'Elody

2.2.3 Arborescence d'une expression

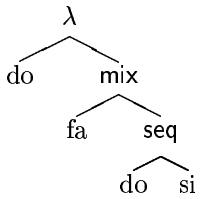
Le fait que les expressions musicales de *Elody* sont construites à partir d'opérateurs binaires signifie que leur structure syntaxique est celle d'un arbre binaire. On peut donc représenter ces expressions par leur arbre syntaxique. Ceci constitue, d'une part une commodité pour visualiser certaines expressions et explique, d'autre part que les techniques utilisées par la suite feront référence au fait qu'il s'agit précisément d'arborescences.

Exemple

On peut représenter l'expression :

$$\lambda do.\left[\frac{fa}{[do;si]}\right]$$

par son arbre syntaxique :



2.3 L'organisation interne d'*Elody*

L'objet de ce paragraphe est de présenter la manière avec laquelle les principes d'un langage de programmation homogène dérivé du λ-calcul sont implémentés dans *Elody*.

La figure 2.1 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 n'a recours qu'au langage $\mathbb{L}_1$, qu'on appellera langage de *description en intention*. A ce niveau toutes les opérations présentées dans le chapitre précédent peuvent être réalisées. L'utilisateur

a accès aux résultats de ces opérations par l'intermédiaire de leurs représentations dans les langages $\mathbb{L}_4$ et $\mathbb{L}_5$.

Dans *Elody* rendre variable un terme dans une expression s'effectue en opérant l'abstraction du terme dans l'expression dans $\mathbb{L}_1$. La notion de variable n'est cependant explicite que dans $\mathbb{L}_2$. Il s'agit comme on l'a vu de désigner les variables par l'exemple, plutôt que de les manipuler en tant qu'objets abstraits.

Une des particularités fondamentales d'*Elody* réside dans le fait que les expressions de $\mathbb{L}_1$ et $\mathbb{L}_2$ ne sont jamais remplacées par les résultats des calculs qu'elles expriment. Tous les *ingrédients* constitutifs d'un objet musical sont donc accessibles à l'utilisateur. Les expressions contiennent donc la trace de leur histoire. D'un point de vue théorique, cela signifie que les expressions de $\mathbb{L}_1$ et $\mathbb{L}_2$ ne sont pas réduites.

Pour en tirer des représentations visuelles et sonores, les expressions de $\mathbb{L}_2$ doivent être évaluées. L'évaluation est le processus par lequel on passe de $\mathbb{L}_2$ à $\mathbb{L}_3$.

On donne ici une présentation des langages $\mathbb{L}_1$ et $\mathbb{L}_2$, qui comme on le verra se trouvent au centre de la mise en oeuvre de l'opérateur d'abstraction généralisée.

2.3.1 Le langage $\mathbb{L}_1$

Une expression est implémentée dans $\mathbb{L}_1$ avec des événements élémentaires[2] :

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

et avec les opérateurs répertoriés ci-dessous :
(e et f représentent des expressions de $\mathbb{L}_1$, $n \in \mathbb{Z}$ et $r \in \mathbb{R}^+$

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

Notations

Soient ev un événement élémentaire.

- Alors on note $\mathcal{T}^{vec}(\mathsf{ev})$ avec $vec = (r, n_1, n_2, n_3)$, la transformation :

$$\mathsf{spd}^r[\mathsf{trp}^{n_1}[\mathsf{lvl}^{n_2}[\mathsf{chn}^{n_3}[\mathsf{ev}]]]]$$

[2] On emploiera par la suite l'expression ev pour désigner une expression élémentaire, qu'il s'agisse d'un silence ou d'une note.

• On note aussi $\mathcal{T}(\mathsf{ev}, e)$ l'opération qui transforme tous les événements de même couleur que ev dans e par $\mathcal{T}^{vec}(\mathsf{ev})$, où vec est le vecteur qui convient.

On a par exemple :

$$\mathcal{T}(si, [do; mi]) \quad = \quad [\mathsf{trp}^1[si]; \mathsf{trp}^5[si]] \quad rcl$$

si do et mi sont de la même couleur que si, et n'en diffère que par la hauteur.

On utilisera ces notations en 2.3.3 pour signifier qu'abstraire un événement dans une expression, c'est mettre en évidence la différence entre les paramètres de l'événement et ceux de même couleur dans l'expression.

• On utilise la notation $\mathsf{op}[e, f]$ pour désigner un quelconque opérateur binaire exceptée l'abstraction.

• De même, on désigne par $\mathsf{op}^r[e]$ les opérateurs unaires qui s'appliquent à des notes.

2.3.2 Le langage $\mathbb{L}_2$

Pour que l'évaluation des expressions puisse avoir lieu, il est nécessaire de traduire les abstractions $\mathsf{abstr}[e, f]$ créés dans $\mathbb{L}_1$ lors de l'application de l'opérateur d'abstraction à deux expressions e et f, en des abstractions faisant appel à des variables. C'est l'objet du langage $\mathbb{L}_2$ et de la fonction $\mathcal{L}$ de traduction de $\mathbb{L}_1$ vers $\mathbb{L}_2$.

On représente par $\diamond^e$ la variable e, c'est à dire par le symbole $\diamond$ déterminant un *type* variable, étiqueté par l'expression e qu'il rend variable, et qui lui sert de nom. On note $\mathbb{V}$ l'ensemble des variables créées de cette manière.

On a donc toujours les mêmes expressions élémentaires :

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

et un nouvel opérateur d'abstraction, avec $E, F \in \mathbb{L}_2$, $\diamond^E \in \mathbb{V}$, $n \in \mathbb{Z}$ et $r, d \in \mathbb{R}^+$:

$\mathsf{abstr}[\diamond^E, F]$	abstraction de $\diamond^e$ dans F.
$\mathsf{appl}[E, F]$	application de la *fonction* E à l'*argument* F.
$\mathsf{seq}[E, F]$	mise en séquence de E et F.
$\mathsf{mix}[E, F]$	mixage (superposition temporelle) de E et F.
$\mathsf{beg}[E, F]$	début de E pris sur une période de la durée de F.
$\mathsf{rst}[E, F]$	reste de E privée de son début sur une période de la durée de F.
$\mathsf{xpd}[E, F]$	E dilatée ou compressée dans le temps de façon à avoir la durée de F.
$\mathsf{spd}^r[E]$	E dilatée ou compressée dans le temps par un coefficient r.
$\mathsf{trp}^n[E]$	transposition de E de n unités.
$\mathsf{lvl}^n[E]$	augmentation des vélocités de E de n unités.
$\mathsf{chn}^n[E]$	augmentation des canaux de E de n unités.

Exemple

Reprenons l'exemple proposé en 2.2.2 pour mettre en évidence la différence entre $\mathbb{L}_1$ et $\mathbb{L}_2$:

On a :

$$\begin{cases} e \equiv fa \\ \text{et} \\ f \equiv \left[do; \left[\frac{si}{fa}\right]\right] \end{cases}$$

L'abstraction de e dans f s'exprime dans $\mathbb{L}_1$, c'est à dire au niveau de l'utilisateur, sans faire appel à la notion de variable par :

$$\lambda fa. \left[do; \left[\frac{si}{fa}\right]\right]$$

Dans $\mathbb{L}_2$ s'exprime le fait que la note e a été rendue variable dans f. L'expression correspondante dans $\mathbb{L}_2$ est la suivante :

$$\lambda \diamond^{fa} . \left[do; \left[\frac{si}{\diamond^{fa}}\right]\right]$$

Si on applique à cette fonction la note mi, on crée dans $\mathbb{L}_1$ l'expression :

$$\left(\lambda fa. \left[do; \left[\frac{si}{fa}\right]\right]\ mi\right)$$

et dans $\mathbb{L}_2$:

$$\left(\lambda \diamond^{fa} . \left[do; \left[\frac{si}{\diamond fa}\right]\right]\ mi\right)$$

Remarquons que l'expression $\left[do; \left[\frac{si}{mi}\right]\right]$, obtenue par réduction de l'application précédente, n'est utilisée qu'à partir de $\mathbb{L}_3$.

Notation

On notera par la suite $\breve{\mathbb{L}}_2$ l'ensemble des expressions dont l'arbre syntaxique est un sous-arbre d'une expression de $\mathbb{L}_2$

Remarque

$\mathbb{L}_2$ se distingue de $\breve{\mathbb{L}}_2$ par le fait que toute expression de $\mathbb{L}_2$ est nécessairement close par construction, ce qui n'est pas le cas pour les expressions de $\breve{\mathbb{L}}_2$.

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

La traduction d'une expression de $\mathbb{L}_1$ en une expression de $\mathbb{L}_2$, représentée par une fonction $\mathcal{L}$, occupe une place centrale dans la présente étude. Elle traduit le fonctionnement de l'opérateur d'abstraction. Il s'agira donc de la redéfinir lorsqu'on remplacera l'abstraction simple par l'abstraction généralisée. $\mathcal{L}$ est ici définie pour l'abstraction simple.

$$\mathcal{L} : \begin{cases} \mathbb{L}_1 \longrightarrow \mathbb{L}_2 \\ E \longmapsto \mathcal{L}(E) \end{cases}$$

Avec :

$$\left\{\begin{array}{rcl} \mathcal{L}(\lambda e.f) & \longrightarrow & \lambda \diamond^{\mathcal{L}(e)}.\mathcal{V}(\mathcal{L}(e),\mathcal{L}(f)) \\ \mathcal{L}(\mathsf{op}[e,f]) & \longrightarrow & \mathsf{op}[\mathcal{L}(e),\mathcal{L}(f)] \\ \mathcal{L}(\mathsf{op}^r[e]) & \longrightarrow & \mathsf{op}^r[\mathcal{L}(e)] \\ \mathcal{L}(\mathsf{ev}) & \longrightarrow & \mathsf{ev} \end{array}\right.$$

Où la fonction $\mathcal{V}$ appliquée à e et f a pour tâche de calculer le corps de l'abstraction, c'est à dire de reconnaître les occurrences de e dans f. $\mathcal{V}$ est définie comme suit :

$$\mathcal{V} : \left\{\begin{array}{l} \mathbb{L}_2 \times \mathbb{L}_2 \longrightarrow \mathbb{L}_2 \\ E, F \longmapsto \mathcal{V}(E,F) \end{array}\right.$$

Avec :

$$\left\{\begin{array}{rcl} \text{si } F \equiv E & & \\ \mathcal{V}(E,F) & \longrightarrow & \diamond^E \\ \text{si } E \equiv \mathsf{ev} & & \\ \mathcal{V}(E,F) & \longrightarrow & \mathcal{T}(E,F) \end{array}\right.$$

$$\left\{\begin{array}{rcl} \text{si } F \equiv \mathsf{op}[G,H] & & \\ \mathcal{V}(E,F) & \longrightarrow & \mathsf{op}[\mathcal{V}(E,G),\mathcal{V}(E,H)] \\ \text{si } F \equiv \mathsf{op}^r[G] & & \\ \mathcal{V}(E,F) & \longrightarrow & \mathsf{op}^r[\mathcal{V}(E,G)] \\ \text{si } F \equiv \lambda \diamond^G.H & & \\ \mathcal{V}(E,F) & \longrightarrow & \lambda \diamond^G.\mathcal{V}(E,H)] \end{array}\right.$$

$$\left\{\begin{array}{rcl} \text{si } F \equiv \mathsf{ev} & & \\ \mathcal{V}(E,F) & \longrightarrow & F \\ \text{si } \mathcal{F} \equiv \diamond^G & & \\ \mathcal{V}(E,F) & \longrightarrow & F \end{array}\right.$$

c

Deuxième partie

L'abstraction généralisée

Chapitre 3

Position du problème

La fonctionnalité du langage de programmation *Elody* dépend de l'opérateur d'abstraction. L'abstraction *simple* permet de créer des fonctions à partir de deux expressions e et f, en rendant variable dans f, les occurrences de e. La démarche consiste donc à désigner, pour les abstraire, des termes syntaxiquement égaux à e dans f.

L'abstraction généralisée se propose de substituer à la notion d'égalité syntaxique, une notion plus générale dans le mécanisme d'abstraction. Il s'agit ainsi de se donner la possibilité de désigner dans une expression f non plus des occurrences d'un terme e, mais tous les sous-termes q de f tels que e est plus général que q en un certain sens.

En terme fonctionnel, on souhaite qu'une expression e soit plus générale qu'une expression f, si on peut interpréter f comme l'application de e à quelque chose.

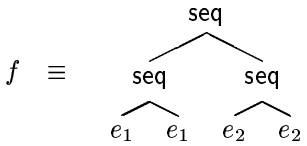
Il s'agira pour nous de formaliser cette idée de *généralité* pour y associer un mécanisme d'abstraction qu'on qualifiera pour cette raison, de *généralisée*

Grâce au mécanisme qu'on vient d'évoquer, on souhaite obtenir des résultats précis pour un ensemble de cas de référence. Il s'agit donc de s'assurer que l'extension du mécanisme à un ensemble exhaustif de cas d'utilisation de l'opérateur est cohérente avec les concepts du λ-calcul, sur lequel est fondé le langage. Si ce n'est pas le cas, il est nécessaire de recadrer la définition de l'opérateur.

3.1 Exemple

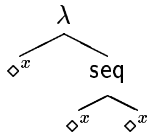
Voici une opération qu'on souhaite pouvoir réaliser avec l'abstraction généralisée, et qui n'est pas possible avec l'abstraction simple :

Soit e une expression définie par son arbre syntaxique :



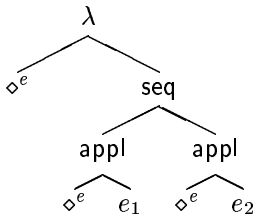
Si on voulait rendre variable toutes les répétitions dans cette expression on serait contraint, avec l'abstraction simple, d'abstraire successivement :

Or, une répétition peut être représentée par le pattern e :



L'abstraction généralisée de e dans f nous permet donc de désigner dans f toutes les *instances* du concept e. En ce sens, e est plus *général* que ses instances.

Alors, l'abstraction généralisée s'exprime comme :



où $\diamond^e$ est une nouvelle variable.

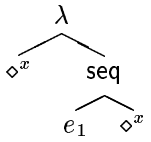
3.2 Résolution d'un problème de *semi-unification*

La semi-unification (ou filtrage) est un cas particulier de l'unification : les variables sont concentrées dans un seul terme [Gengler 1987]. Pour davantage de détails concernant semi-unification et unification, voir [Knight 1989], [Ruf et Weise 1989] et plus spécifiquement [Wertz 1989] et [Queinnec 1990] pour des applications en LISP.

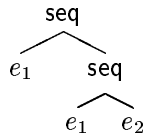
Voyons sur un exemple l'analogie entre le développement de l'opérateur d'abstraction généralisée et la résolution d'un problème de semi-unification.

Supposons qu'on veuille abstraire un pattern e dans une expression f, où e et f sont définies par :

$e \equiv$



$f \equiv$



Schématiquement, le mécanisme d'abstraction s'apparente à la donnée d'une règle de réécriture (ou filtre) $\mathcal{R}$ qui s'exprime comme suit, pour toute variable $\diamond^x$ et tous termes a et b :

$$[a;b] \quad \xrightarrow{\mathcal{R}} \quad (\lambda \diamond^x .[a;\diamond^x]\ b)$$

L'application de cette règle à f fournit un ensemble d'expressions :

$$[a;[a;b]] \quad \xrightarrow{\mathcal{R}} \quad \begin{cases} [a;(\lambda \diamond^x . [a;\diamond^x]\ b)] \\ [(\lambda \diamond^x . [a;\diamond^x]\ b);a] \\ (\lambda \diamond^x . [a;\diamond^x]\ (\lambda \diamond^x . [a;\diamond^x]\ b)) \end{cases}$$

parmi lesquelles se trouve le résultat souhaité de l'abstraction généralisée de e dans f :

$$(\lambda \diamond^x . [a;\diamond^x]\ (\lambda \diamond^x . [a;\diamond^x]\ b))$$

Cela signifie que pour que l'abstraction généralisée soit une opération déterministe, Il faut appliquer la règle de réécriture autant de fois que possible.

Le fait qu'il y ait une parenté entre les mécanisme d'abstraction généralisée et de filtrage nous permettra par la suite de se référer à des algorithmes connus. Cependant, l'opérateur ne sera pas formalisé en termes de semi-unification. Il le sera en référence au λ-calcul, de façon à vérifier qu'il est cohérent avec celui-ci.

Chapitre 4

Formulation d'un opérateur d'abstraction généralisée du λ-calcul

L'objet de ce paragraphe est de formaliser la définition d'un opérateur d'abstraction généralisée dans un λ-calcul non typé comme celui qui a été présenté dans le chapitre 1.

4.1 Définitions préliminaires

On définit deux relations qui, pour l'opération d'abstraction généralisée, vont se substituer à la relation d'égalité syntaxique, sur laquelle repose l'abstraction simple.

4.1.1 égalité syntaxique au renommage des variables liées près

Définissons tout d'abord une relation d'*égalité syntaxique au nom des variables liées des expressions près*. On note cette relation $\overset{\alpha}{\equiv}$, en référence à la notion d'α-conversion à laquelle elle se rapporte (cf 1.3.3)

Définition 4.1.1. *Pour tout λ-termes e et f, si l'on pose :*

$$\begin{cases} \{x_1, \ldots, x_n\} = BV(e) \\ et \\ \{y_1, \ldots, y_n\} = BV(f) \end{cases}$$

on définit la relation $e \overset{\alpha}{\equiv} f$ par :

$e \overset{\alpha}{\equiv} f$ si et seulement si :

$$\begin{cases} n = m \\ et \\ e[x_1/y_1, \ldots, x_n/y_n] \equiv f \end{cases}$$

4.1.2 Formalisation de la notion de *généralité* par une relation d'ordre

On se donne maintenant une relation notée $>$, fondamentale dans le cadre de la définition de l'opérateur d'abstraction généralisée, qui formalise la notion de *généralité* d'une expression. $e > f$ **signifiera en effet que** e **est plus** ***générale*** **que** f.

Définition 4.1.2. *Pour tous λ-termes e et f, exprimés par :*

$$\begin{cases} e \equiv \lambda x_1 \ldots . \lambda x_n . e' \\ et \\ f \equiv \lambda y_1 \ldots . \lambda y_m . f' \end{cases}$$

où e' et f' ne sont pas des abstractions, et où n ou m peuvent être nuls, on définit la relation $e > f$ par :

$e > f$ si et seulement si :

$\exists\, !(A_1, \ldots, A_k)$ avec $1 \leq k \leq n$, tel que :

$$\begin{cases} e \equiv \lambda x_1 \ldots . \lambda x_k . e'' \\ et \\ e''[x_1/A_1, \ldots, x_k/A_k] \overset{\alpha}{\equiv} f \end{cases}$$

Remarques

(i) Soient deux λ-termes :

$$\begin{cases} e \equiv \lambda x_1 \ldots . \lambda x_n . e' \\ \text{et} \\ f \equiv \lambda y_1 \ldots . \lambda y_m . f' \end{cases}$$

Par définition de $>$, on a :

$$e > f \Rightarrow (n > m \text{ et } k = n - m).$$

(ii) Si e un λ-terme qui n'est pas une abstraction. Alors, pour tout λ-terme f, on a :

$$e \not> f.$$

(iii) Pour tous λ-termes e et f, on a :

$$e \not> f \not\Leftarrow (f > e \text{ ou } f \overset{\alpha}{\equiv} e).$$

Aux deux relations définies précédemment, on associe la relation $\geq$ définie ci-dessous, qui formalise une notion de ***généralité* au sens large.**

Définition 4.1.3. *Pour tout λ-termes e et f, on définit $e \geq f$ par :*

$e \geq f$ si et seulement si $e > f$ ou $e \overset{\alpha}{\equiv} f$.

Par définition de $\overset{\alpha}{\equiv}$ et $>$, il est clair que la définition précédente peut se formuler comme suit :

Définition 4.1.4. *Pour tous λ-termes e et f, exprimés par :*

$$\begin{cases} e \equiv \lambda x_1 \ldots . \lambda x_n . e' \\ et \\ f \equiv \lambda y_1 \ldots . \lambda y_m . f' \end{cases}$$

où e' et f' ne sont pas des abstractions, et où n ou m peuvent être nuls, on définit la relation $e \geq f$ par :

$e \geq f$ si et seulement si :

$\exists\, !(A_1, \ldots, A_k)$ avec $0 \leq k \leq n$, tel que[1] *:*

$$\begin{cases} e \equiv \lambda x_1 \ldots . \lambda x_k . e'' \\ et \\ e''[x_1/A_1, \ldots, x_k/A_k] \overset{\alpha}{\equiv} f \end{cases}$$

[1] $k = 0$ signifie que $e \overset{\alpha}{\equiv} f$

Vérifions que **la relation $\geq$ est une relation d'ordre sur l'ensemble des λ-termes.**

Proposition 4.1.1. *$\geq$ est une relation d'ordre sur l'ensemble des λ-termes.*

Preuve

(i) *Réflexivité*

Pour tout λ-terme, $e \equiv e$, d'où $e \overset{\alpha}{\equiv} e$ et donc $e \geq e$. Donc $\geq$ est réflexive.

(ii) *Antisymétrie*

Par définition de $\geq$, montrer que $\geq$ est antisymétrique revient à montrer que pour tous λ-termes e et f tels que $e \geq f$, $g \not\geq e$. Ce qui est trivial, d'après la conséquence (i) de la définition 4.1.2. $\geq$ est donc antisymétrique.

(iii) *Transitivité*

Soient e, f et g des λ-termes exprimés comme suit :

$$\begin{cases} e \equiv \lambda x_1 . \ldots . \lambda x_n . e' \\ f \equiv \lambda y_1 . \ldots . \lambda y_m . f' \\ g \equiv \lambda z_1 . \ldots . \lambda z_p . g' \end{cases}$$

Supposons que $e \geq f$ et $f \geq g$

Alors

$$\exists\, !(A_1, \ldots, A_k) \text{ avec } k = n - m, \text{ tel que :}$$
$$\begin{cases} e \equiv \lambda x_1 . \lambda x_2 . \ldots . \lambda x_k . e'' \\ \text{et} \\ e''[x_1/A_1, \ldots, x_k/A_k] \overset{\alpha}{\equiv} f \end{cases}$$

De même,

$$\exists\, !(B_1, \ldots, B_l) \text{ avec } l = m - p, \text{ tel que :}$$
$$\begin{cases} f \equiv \lambda y_1 . \lambda y_2 . \ldots . \lambda y_l . f'' \\ \text{et} \\ f''[y_1/B_1, \ldots, y_l/B_l] \overset{\alpha}{\equiv} g \end{cases}$$

D'où :

$$e''[x_1/A_1, \ldots, x_k/A_k] \overset{\alpha}{\equiv} \lambda y_1 . \lambda y_2 . \ldots . \lambda y_l . f''$$

Par suite

$$\lambda x_{n-m+1} . \ldots . \lambda x_{n-m+l} . e'''[x_1/A_1, \ldots, x_k/A_k] \overset{\alpha}{\equiv} \lambda y_1 . \lambda y_2 . \ldots . \lambda y_l . f''$$

Or

$$f''[y_1/B_1, \ldots, y_l/B_l] \overset{\alpha}{\equiv} g$$

Donc

$$(e'''[x_1/A_1,\ldots,x_k/A_k])[x_{n-m+1}/B_1,\ldots,x_{n-m+l}/B_l] \overset{\alpha}{\equiv} g$$

Soit de façon équivalente

$$e'''[x_1/A_1,\ldots,x_{k+l}/A_{k+l}] \overset{\alpha}{\equiv} g, \text{ (avec } k+l=n-p)$$

Donc $\geq$ est transitive.

On a montré qu'il s'agit d'une relation d'ordre sur les λ-termes.

Remarque

En vertu de la conséquence (iii) de la définition 4.1.2, $\geq$ n'est pas un ordre total sur l'ensemble des λ-termes.

4.1.3 Une substitution généralisée

On se donne enfin une opération de substitution plus générale que la substitution usuelle présentée en 1.3.2. Ainsi, on notera $f[e/x]^G$ la ***substitution généralisée*** de e par x dans f :

Définition 4.1.5. *Pour tous λ-termes e et f et toute variable x, exprimés par :*

$$\begin{cases} e \equiv \lambda x_1 \ldots . \lambda x_n . e' \\ et \\ f \equiv \lambda y_1 \ldots . \lambda y_m . f' \end{cases}$$

où e' et f' ne sont pas des abstractions, et où n ou m peuvent être nuls, on définit $f[e/x]^G$ par :

$$f[e/x]^G = (x\ A_1[e/x] \ldots\ A_k[e/x]) \quad si\ e > f$$

$$f[e/x]^G = x \quad si\ e \overset{\alpha}{\equiv} f$$

sinon ($e \not\geq f$)

$$\begin{cases} f[e/x]^G = (g[e/x]^G h[e/x]^G) & si\ f \equiv (g\ h) \\ f[e/x]^G = \lambda g.(h[e/x]^G) & si\ f \equiv \lambda g.h \\ f[e/x]^G = y & sinon\ (\ y\ est\ nécessairement\ un\ atome\ du\ langage) \end{cases}$$

avec :

$$\begin{cases} e \equiv \lambda x_1 \ldots . \lambda x_k . e'' \\ et \\ e''[x_1/A_1,\ldots,x_k/A_k] \overset{\alpha}{\equiv} f \end{cases}$$

Remarquons qu'en adoptant pour convention que pour $k=0$,

$$(x\ A_1[e/x] \ldots\ A_k[e/x]) = x)$$

la définition précédente peut se formuler de la manière suivante :

Définition 4.1.6. *Pour tous λ-termes e et f et toute variable x, exprimés par :*

$$\begin{cases} e \equiv \lambda x_1 \ldots . \lambda x_n . e' \\ et \\ f \equiv \lambda y_1 \ldots . \lambda y_m . f' \end{cases}$$

où e' et f' ne sont pas des abstractions, et où n ou m peuvent être nuls, on définit $f[e/x]^G$ par :

$f[e/x]^G = (x\ A_1[e/x] \ldots\ A_k[e/x]) \quad si\ e \geq f$

sinon $(e \not\geq f)$

$$\begin{cases} f[e/x]^G = (g[e/x]^G h[e/x]^G) & si\ f \equiv (gh) \\ f[e/x]^G = \lambda g.(h[e/x]^G) & si\ f \equiv \lambda g.h \\ f[e/x]^G = y & sinon\ (\ y\ est\ nécessairement\ un\ atome\ du\ langage) \end{cases}$$

avec :

$$\begin{cases} e \equiv \lambda x_1 \ldots . \lambda x_k . e'' \\ et \\ e''[x_1/A_1, \ldots, x_k/A_k] \overset{\alpha}{\equiv} f \end{cases}$$

4.2 L'abstraction généralisée du λ-calcul

Grâce à la relation $\geq$, qui traduit la notion de *généralité* et à l'opération de substitution *généralisée*, on peut aisément définir un opérateur d'**abstraction *généralisée*** comme suit :

Définition 4.2.1. *Pour tous λ-termes e et f, on définit l'abstraction généralisée notée λ^G de e dans f par :*

$$\lambda^G e.f = \lambda x.f[p/x]^G$$

Où x est une nouvelle variable.

4.3 Résultats

L'objet de ce paragraphe est de présenter deux résultats relatifs aux notions qui ont été définies dans ce chapitre et qui se révéleront fort utiles lors de la mise en oeuvre de l'algorithme implémentant l'opérateur.

4.3.1 Proposition

Proposition 4.3.1. *Soient e et f deux expressions définies par :*

$$\begin{cases} e \equiv \lambda x_1 \ldots . \lambda x_n . e' \\ f \equiv \lambda y_1 \ldots . \lambda y_m . f' \end{cases}$$

Avec $n>m$, et où e' et f' ne sont pas des abstractions.

Notons $e \equiv \lambda x_1 \ldots . \lambda x_{n-m} . e''$

Alors les deux propositions suivantes sont équivalentes :

(i) $e''[x_1/A_1, \ldots, x_{n-m}/A_{n-m}] \stackrel{\alpha}{\equiv} f$

(ii) $\begin{cases} e'[x_1/A_1, \ldots, x_{n-m}/A_{n-m}] \stackrel{\alpha}{\equiv} f' \\ et \\ \quad \forall i \in 1, \ldots, m \ , \forall j \in 1, \ldots, n-m \\ \qquad\qquad y_i \notin FV(A_j) \end{cases}$

Preuve

- *Sens direct* $(i) \Rightarrow (ii)$

  Supposons que

  $$e''[x_1/A_1, \ldots, x_{n-m}/A_{n-m}] \stackrel{\alpha}{\equiv} f$$

  Supposons en outre que

  $$e'[x_1/A_1, \ldots, x_{n-m}/A_{n-m}] \stackrel{\alpha}{\equiv} f'$$

  et qu'il existe $i_0 \in \{1, \ldots, m\}$ et $j_0 \in \{1, \ldots, n-m\}$ tels que :

  $$y_{i_0} \in FV(A_{j_0}).$$

  Notons par ailleurs :

$$\begin{cases} \{\hat{x_1}, \ldots, \hat{x_r}\} = BV(e'') \\ \text{et} \\ \{\hat{y_1}, \ldots, \hat{y_s}\} = BV(f) \end{cases}$$

D'après l'hypothèse de départ, on a :

$$\begin{cases} r = s \\ \text{et} \\ (e''[x_1/A_1, \ldots, x_{n-m}/A_{n-m}])[\hat{x_1}/\hat{y_1}, \ldots, \hat{x_r}/\hat{y_r}] \equiv f \end{cases}$$

De plus,

$$\begin{cases} y_{i_0} \in \hat{x_1}, \ldots, \hat{x_r} \\ \text{et} \\ y_{i_0} \in \hat{y_1}, \ldots, \hat{y_s} \end{cases}$$

Donc

$$y_{i_0} \in FV((e''[x_1/A_1, \ldots, x_{n-m}/A_{n-m}])[\hat{x_1}/\hat{y_1}, \ldots, \hat{x_r}/\hat{y_r}])$$

et

$$y_{i_0} \in BV(f)$$

Ce qui est absurde.

Par conséquent

$$\begin{array}{c} e''[x_1/A_1, \ldots, x_{n-m}/A_{n-m}] \overset{\alpha}{\equiv} f \\ \Downarrow \\ \begin{cases} e'[x_1/A_1, \ldots, x_{n-m}/A_{n-m}] \overset{\alpha}{\equiv} f' \\ \text{et} \\ \forall i \in 1, \ldots, m \ , \forall j \in 1, \ldots, n-m \\ y_i \notin FV(A_j) \end{cases} \end{array}$$

- *Réciproque* $(i) \Leftarrow (ii)$

  Supposons qu'on ait :

$$\begin{cases} e'[x_1/A_1, \ldots, x_{n-m}/A_{n-m}] \overset{\alpha}{\equiv} f' \\ \text{et} \\ \forall i \in 1, \ldots, m \ , \forall j \in 1, \ldots, n-m \\ y_i \notin FV(A_j) \end{cases}$$

  Alors, on peut affirmer que :

$$(\lambda x_{n-m+1} \ldots . \lambda x_n . e'[x_1/A_1, \ldots, x_{n-m}/A_{n-m}])[x_{n-m+1}/y_1, \ldots, x_n/y_m \overset{\alpha}{\equiv} f$$

  D'où

$$e''[x_1/A_1, \ldots, x_{n-m}/A_{n-m}] \overset{\alpha}{\equiv} f$$

On a donc bien :

$$\left\{\begin{array}{l} e'[x_1/A_1,\ldots,x_{n-m}/A_{n-m}] \stackrel{\alpha}{\equiv} f' \\ \text{et} \\ \forall i \in 1,\ldots,m\ ,\forall j \in 1,\ldots,n-m \\ y_i \notin FV(A_j) \end{array}\right.$$
$$\Downarrow$$
$$e''[x_1/A_1,\ldots,x_{n-m}/A_{n-m}] \stackrel{\alpha}{\equiv} f$$

4.3.2 Conséquence

En vertu de la proposition énoncée dans le paragraphe précédent, on peut formuler un résultat qui servira de point de départ à la construction de l'algorithme d'abstraction généralisée

Soient e et f deux λ-termes définis par :

$$\left\{\begin{array}{l} e \equiv \lambda x_1.\ldots.\lambda x_n.e' \\ f \equiv \lambda y_1.\ldots.\lambda y_m.f' \\ \text{et} \\ e \equiv \lambda x_1.\ldots.\lambda x_{n-m}.e'' \end{array}\right.$$

où e' et f' ne sont pas des abstractions, et où m et $n-m$ peuvent être nuls.

Notons également

$\{\hat{x_1},\ldots,\hat{x_q}\} = BV(e')$

et

$\{\hat{y_1},\ldots,\hat{x_r}\} = BV(f')$

Alors, on a l'équivalence suivante :

$$e \geq f$$
$$\Updownarrow$$
$$\left(\begin{array}{c} \exists\ !(A_1,\ldots,A_{n-m}) \text{ tel que} \\ e'[x_1/A_1,\ldots,x_{n-m}/A_{n-m},x_{n-m+1}/y_1,\ldots,x_n/y_m,\hat{x_1}/\hat{y_1},\ldots,\hat{x_q}/\hat{y_q}] \equiv f \\ \text{et} \\ \forall i \in \{1,\ldots,m\}\ ,\forall j \in \{1,\ldots,n-m\} \\ y_i \notin FV(A_j) \end{array}\right)$$

Chapitre 5

Formalisation de l'opérateur d'abstraction généralisée dans *Elody*

On se propose dans ce paragraphe de formaliser le fonctionnement de l'opérateur d'abstraction généralisée dans *Elody*. Celui-ci sera naturellement calqué sur l'opérateur du λ-calcul défini dans le chapitre précédent. Comme on l'a souligné en 2.3.3, c'est par le biais de la traduction de $\mathbb{L}_1$ dans $\mathbb{L}_2$ que le mécanisme de l'abstraction utilisée dans *Elody* est défini.

Redéfinissons tout d'abord dans $\mathbb{L}_2$, c'est à dire pour des expression du langage de programmation musicale et non plus pour des λ-termes, les relations qui ont été définies dans le chapitre précédent.

5.1 Définitions préliminaires

Définition 5.1.1. *Pour toutes expressions $E, F \in \mathbb{L}_2$, si l'on pose :*

$$\begin{cases} \diamond^{x_1}, \ldots, \diamond^{x_n} = BV(E) \\ et \\ \diamond^{y_1}, \ldots, \diamond^{y_m} = BV(F) \end{cases}$$

on définit la relation $E \overset{\alpha}{\equiv} F$ par :

$E \overset{\alpha}{\equiv} F$ si et seulement si :

$$\begin{cases} n = m \\ et \\ E[\diamond^{x_1}/\diamond^{y_1}, \ldots, \diamond^{x_n}/\diamond^{y_n}] \equiv F \end{cases}$$

La relation qui indique si une expression E est plus générale, au sens strict, qu'une expression F est redéfinie comme suit

Définition 5.1.2. *Pour toutes expressions $E, F \in \mathbb{L}_2$, exprimés par :*

$$\begin{cases} E \equiv \lambda \diamond^{x_1} \ldots . \lambda \diamond^{x_n} . E' \\ et \\ F \equiv \lambda \diamond^{y_1} \ldots . \lambda \diamond^{y_m} . F' \end{cases}$$

où E' et F' ne sont pas des abstractions, et où n ou m peuvent être nuls, on définit la relation $E > F$ par :

$E > F$ si et seulement si :

$\exists\, !(A_1, \ldots, A_k) \in \breve{\mathbb{L}}_2^k$ avec $1 \leq k \leq n$, tel que :

$$\begin{cases} E \equiv \lambda \diamond^{x_1} \ldots . \lambda \diamond^{x_k} . E'' \\ et \\ E''[\diamond^{x_1}/A_1, \ldots, \diamond^{x_k}/A_k] \overset{\alpha}{\equiv} f \end{cases}$$

Des deux relations définies précédemment, on déduit la relation $\geq$ de généralisation au sens large, définie ci-dessous.

Définition 5.1.3. *Pour toutes expressions $E, F \in \mathbb{L}_2$, on définit $E \geq F$ par :*

$E \geq F$ si et seulement si $E > F$ ou $E \overset{\alpha}{\equiv} F$.

Comme en 4.1, on peut formuler une définition équivalente :

Définition 5.1.4. *Pour toutes expressions E et F, exprimées par :*

$$\begin{cases} E \equiv \lambda \diamond^{x_1} \ldots . \lambda \diamond^{x_n} . E' \\ et \\ F \equiv \lambda \diamond^{y_1} \ldots . \lambda \diamond^{y_m} . F' \end{cases}$$

où E' et F' ne sont pas des abstractions, et où n ou m peuvent être nuls, on définit la relation $E \geq F$ par :

$E \geq F$ si et seulement si :

$\exists\, !(A_1, \ldots, A_k) \in \overset{\vee}{\mathbb{L}}_2^k$ avec $0 \leq k \leq n$, tel que[1] *:*

$$\begin{cases} E \equiv \lambda \diamond^{x_1} \ldots . \lambda \diamond^{x_k} . E'' \\ et \\ E''[\diamond^{x_1}/A_1, \ldots, \diamond^{x_k}/A_k] \overset{\alpha}{\equiv} F \end{cases}$$

La relation $\geq$ ainsi définies pour des expressions de $\mathbb{L}_2$ va nous permettre de formuler l'abstraction généralisée dans *Elody* dont le principe est défini par l'intermédiaire de la fonction de traduction de $\mathbb{L}_1$ vers $\mathbb{L}_2$. Lorsque l'utilisateur procède à une abstraction généralisée dans $\mathbb{L}_1$, l'expression correspondante dans $\mathbb{L}_2$ est calculée selon une procédure qui traduit les mécanismes de l'opérateur du λ-calcul. Cette procédure est décrite par la fonction de traduction de $\mathbb{L}_1$ vers $\mathbb{L}_2$.

5.2 réorganisation interne d'*Elody*

5.2.1 Redéfinition des opérateurs de $\mathbb{L}_1$

Commençons par mettre à jour $\mathbb{L}_1$ en substituant à l'abstraction simple du paragraphe 2.3.1, l'opérateur d'abstraction généralisée.

[1] $k = 0$ signifie que $E \overset{\alpha}{\equiv} F$

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

Où e et f représentent des expressions de $\mathbb{L}_1$, $n \in \mathbb{Z}$ et $r \in \mathbb{R}^+$

5.2.2 Rappel de la structure de $\mathbb{L}_2$

$\mathbb{L}_2$ diffère de $\mathbb{L}_1$ par le fait que les abstraction généralisée sont remplacées par des abstractions simples dans lesquelles la notion de variable apparaît et dont le corps est déterminé par la fonction de traduction de $\mathbb{L}_1$ dans $\mathbb{L}_2$ définie dans le paragraphe suivant.

$\mathsf{abstr}[\diamond^E, F]$	abstraction de $\diamond^e$ dans F.
$\mathsf{appl}[E, F]$	application de la *fonction* E à l'*argument* F.
$\mathsf{seq}[E, F]$	mise en séquence de E et F.
$\mathsf{mix}[E, F]$	mixage (superposition temporelle) de E et F.
$\mathsf{beg}[E, F]$	début de E pris sur une période de la durée de F.
$\mathsf{rst}[E, F]$	reste de E privée de son début sur une période de la durée de F.
$\mathsf{xpd}[E, F]$	E dilatée ou compressée dans le temps de façon à avoir la durée de F.
$\mathsf{spd}^r[E]$	E dilatée ou compressée dans le temps par un coefficient r.
$\mathsf{trp}^n[E]$	transposition de E de n unités.
$\mathsf{lvl}^n[E]$	augmentation des vélocités de E de n unités.
$\mathsf{chn}^n[E]$	augmentation des canaux de E de n unités.

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

Les fonctions $\mathcal{L}$ et $\mathcal{V}$, décrites dans ce qui suit, rendent compte du fonctionnement de l'abstraction généralisée dans *Elody*, de la même façon qu'elle rendaient compte du fonctionnement de l'abstraction simple en 2.3.3.

$$\mathcal{L} : \begin{cases} \mathbb{L}_1 \longrightarrow \mathbb{L}_2 \\ E \longmapsto \mathcal{L}(E) \end{cases}$$

Avec :

$$\begin{cases} \mathcal{L}(\lambda^G e.f) & \longrightarrow & \lambda \diamond^{\mathcal{L}(e)} .\mathcal{V}(\mathcal{L}(e), \mathcal{L}(f)) \\ \mathcal{L}(\mathsf{op}[e,f]) & \longrightarrow & \mathsf{op}[\mathcal{L}(e),(f)] \\ \mathcal{L}(\mathsf{op}^r[e]) & \longrightarrow & \mathsf{op}^r[\mathcal{L}(e)] \\ \mathcal{L}(\mathsf{ev}) & \longrightarrow & \mathsf{ev} \end{cases}$$

Où la fonction $\mathcal{V}$ appliquée à E et F a pour tâche de calculer le corps de l'abstraction, c'est à dire, non plus de reconnaître les occurrences de E dans F, mais de reconnaître les sous-termes Q de F tels que E est plus *général* que Q. La notion de généralité étant liée à la relation $\geq$. $\mathcal{V}$ est donc définie comme suit :

$$\mathcal{V} : \begin{cases} \mathbb{L}_2 \times \mathbb{L}_2 \longrightarrow \mathbb{L}_2 \\ E, F \longmapsto \mathcal{V}(E,F) \end{cases}$$

Avec :

$$\begin{cases} E \equiv \lambda \diamond^{x_1} \ldots . \lambda \diamond^{x_n} .E' \\ F \equiv \lambda \diamond^{y_1} \ldots . \lambda \diamond^{y_m} .F' \\ E \equiv \lambda \diamond^{x_1} \ldots . \lambda \diamond^{x_{n-m}} .E'' \\ \text{et} \\ E''[\diamond^{x_1}/A_1, \ldots, \diamond^{x_{n-m}}/A_{n-m}] \stackrel{\alpha}{\equiv} F \\ \text{(si E>F)} \end{cases}$$

ou E' et F' ne sont pas des abstractions.

$$\begin{cases} \text{si } E > F & \\ \quad \mathcal{V}(E,F) & \longrightarrow \quad (\diamond^E \ \mathcal{V}(E,A_1) \ \ldots \ \mathcal{V}(E,A_k)) \\ \text{si } F \stackrel{\alpha}{\equiv} E & \\ \quad \mathcal{V}(E,F) & \longrightarrow \quad \diamond^E \\ \text{si } E \equiv \mathsf{ev} & \\ \quad \mathcal{V}(E,F) & \longrightarrow \quad \mathcal{T}(E,F) \end{cases}$$

$$\begin{cases} \text{si } F \equiv \mathsf{op}[G,H] & \\ \quad \mathcal{V}(E,F) & \longrightarrow \quad \mathsf{op}[\mathcal{V}(E,G), \mathcal{V}(E,H)] \\ \text{si } F \equiv \mathsf{op}^r[G] & \\ \quad \mathcal{V}(E,F) & \longrightarrow \quad \mathsf{op}^r[\mathcal{V}(E,G)] \\ \text{si } F \equiv \lambda \diamond^G .H & \\ \quad \mathcal{V}(E,F) & \longrightarrow \quad \lambda \diamond^G .\mathcal{V}(E,H)] \end{cases} \qquad c$$

$$\begin{cases} \text{si } F \equiv \mathsf{ev} & \\ \quad \mathcal{V}(E,F) & \longrightarrow \quad F \\ \text{si } \mathcal{F} \equiv \diamond^G & \\ \quad \mathcal{V}(E,F) & \longrightarrow \quad F \end{cases}$$

Remarque

Les deux cas suivants :

$$\begin{cases} \text{si } E > F & \\ \quad \mathcal{V}(E,F) & \longrightarrow \quad (\diamond^E\ \mathcal{V}(E,A_1)\ \ldots\ \mathcal{V}(E,A_k)) \\ \text{si } F \overset{\alpha}{\equiv} E & \\ \quad \mathcal{V}(E,F) & \longrightarrow \quad \diamond^E \end{cases}$$

sont équivalents au seul cas :

$$\begin{cases} \text{si } E \geq F & \\ \quad \mathcal{V}(E,F) & \longrightarrow \quad (\diamond^E\ \mathcal{V}(E,A_1)\ \ldots\ \mathcal{V}(E,A_k)) \end{cases}$$

En convenant, comme en 4.1.3 que dans le cas trivial où $k = 0$, on a :

$$(\diamond^E\ \mathcal{V}(E,A_1)\ \ldots\ \mathcal{V}(E,A_k)) = \diamond^E$$

5.3 Deux cas particuliers

5.3.1 Les fonctions constantes

A la question : $\lambda \diamond^x .a$ est-elle plus *générale* que a, on peut apporter deux réponses.

– Oui, en considérant qu'un terme de $\breve{\mathbb{L}}_2$ quelconque A permet de vérifier la relation : $a[\diamond^x/A] \overset{\alpha}{\equiv} a$.

– Non, car la définition de $\geq$ précise que le A en question doit être unique.

Les fonctions constantes sont donc un cas particulier d'abstractions qui ne sont plus générales (au sens strict) qu'aucune expression.

5.3.2 L'identité

S'il n'existe pas d'expression $E \in \breve{\mathbb{L}}_2$ telle que pour une expression constante quelconque C on ait :

$$C > E$$

il n'existe pas non plus d'expression $E \in \breve{\mathbb{L}}_2$ telle que l'identité I vérifie :

$$I \not> E$$

L'identité est donc un cas particulier d'abstractions plus générales (au sens stricte et donc au sens large) que toute autre expression.

Par conséquent, si on abstrait par exemple I dans un événement ev, on va créer dans $\mathbb{L}_2$ une abstraction dont le corps est constitué d'une infinité d'applications :

$$\lambda I.(I\ I\ I\ \ldots\ I\ ev)$$

Aussi, il semble raisonnable de convenir que l'infinité d'applications soit remplacée par une seule application. Abstraire l'identité dans une expression E revient donc à substituer à chaque sous-arbre Q de E, l'application de l'identité à Q.

De plus, par souci d'homogénéité du langage, on convient que pour toute variable $\diamond^x$, $I \not\geq \diamond^x$.

5.4 Transition vers l'algorithme

En guise de transition vers un algorithme implémentant l'opérateur d'abstraction généralisée, on formule ici pour $\mathbb{L}_2$, le résultat énoncé en 4.3.2, dans le cadre du λ-calcul. Ce résultat servira de point de départ au développement de l'algorithme.

Soient $E, F \in \mathbb{L}_2$ définies par :

$$\begin{cases} E \equiv \lambda \diamond^{x_1} \ldots . \lambda \diamond^{x_n} . E' \\ F \equiv \lambda \diamond^{y_1} \ldots . \lambda \diamond^{y_m} . F' \\ \text{et} \\ E \equiv \lambda \diamond^{x_1} \ldots . \lambda \diamond^{x_{n-m}} . E'' \end{cases}$$

où E' et F' ne sont pas des abstractions, et où m peut être nul.

Notons également

$\{\diamond^{\hat{x}_1}, \ldots, \diamond^{\hat{x}_q}\} = BV(E')$

et

$\{\diamond^{\hat{y}_1}, \ldots, \diamond^{\hat{x}_r}\} = BV(F')$

Alors, on a l'équivalence :

$$\begin{array}{c} E \geq F \\ \Updownarrow \\ \left(\begin{array}{c} \exists\, !(A_1, \ldots, A_{n-m}) \in \breve{\mathbb{L}}_2^{n-m} \text{ tel que} \\ E'[\diamond^{x_1}/A_1, \ldots, \diamond^{x_{n-m}}/A_{n-m}, \diamond^{x_{n-m+1}}/\diamond^{y_1}, \ldots, \diamond^{x_n}/\diamond^{y_m}, \diamond^{\hat{x}_1}/\diamond^{\hat{y}_1}, \ldots, \diamond^{\hat{x}_q}/\diamond^{\hat{y}_q}] \equiv F \\ \text{et} \\ \forall i \in \{1, \ldots, m\}\ , \forall j \in \{1, \ldots, n-m\} \\ \diamond^{y_i} \notin FV(A_j) \end{array} \right) \end{array}$$

Chapitre 6

L'algorithme d'abstraction généralisée

Le mécanisme de l'opérateur est décrit par des fonctions possédant des structures récursives (cf 5.2.3). Les algorithmes qui implémentent ces fonctions seront donc également récursifs.

Rappel

Soient $e, f \in \mathbb{L}_1$ et $E, F \in \mathbb{L}_2$ tels que

$E = \mathcal{L}(e)$

et

$F = \mathcal{L}(f)$

Procéder à l'abstraction généralisée de e dans f, c'est former l'expression $\lambda^G e.f$ dans $\mathbb{L}_1$.

Dans $\mathbb{L}_2$, c'est y est associer l'abstraction suivante :

$\lambda \diamond^E .\mathcal{V}(E, F)$

6.1 Pattern matching de deux expressions

On a vu que le calcul du corps de l'abstraction de $\mathbb{L}_2$ créée lorsque s'opère une abstraction généralisée dans $\mathbb{L}_1$ est réalisé en référence à la relation d'ordre $\geq$.

On se donne ici deux fonctions , qui reprennent d'un point de vue algorithmique, le fonctionnement algébrique de $\geq$.

En effet, on sait que $E \geq F$ si deux conditions sont respectées :

$$(i) \begin{cases} \exists\ !(A_1, \ldots, A_{n-m}) \in \breve{\mathbb{L}}_2^{n-m} \text{ tel que} \\ E''[\diamond^{x_1}/A_1, \ldots, \diamond^{x_{n-m}}/A_{n-m}, \diamond^{x_{n-m+1}}/\diamond^{y_1}, \ldots, \diamond^{x_n}/\diamond^{y_m}, \diamond^{\hat{x}_1}/\diamond^{\hat{y}_1}, \ldots, \diamond^{\hat{x}_q}/\diamond^{\hat{y}_q}] \equiv F \end{cases}$$

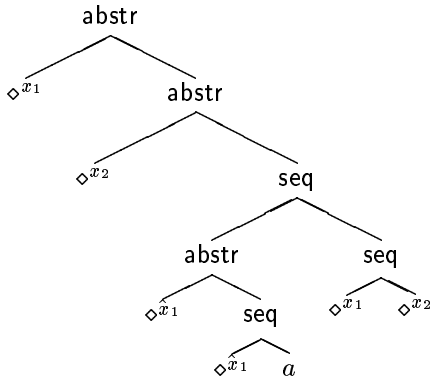
et

$$(ii) \begin{cases} \forall i \in \{1, \ldots, m\}\ , \forall j \in \{1, \ldots, n-m\} \\ \diamond^{y_i} \notin FV(A_j) \end{cases}$$

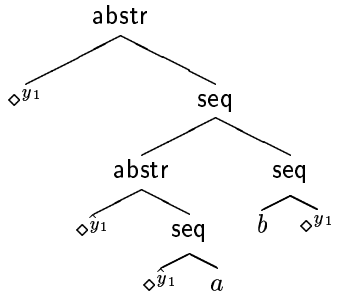
On définit donc $\mathcal{M}_1$ et $\mathcal{M}_2$ de façon à traduire algorithmiquement les conditions (i) et (ii).

6.1.1 Exemple

Soit $E \in \mathbb{L}_2$ définie par l'arbre syntaxique :



Et soit $F \in \mathbb{L}_2$ définie de même par :



On peut vérifier que $E \geq F$

•(i) est vérifiée :

$$E''[\diamond^{x_1}/b, \diamond^{x_2}/\diamond^{y_1}, \diamond^{\hat{x}_1}/\diamond^{\hat{y}_1}] \stackrel{\alpha}{\equiv} F$$

•(ii) est vérifiée :

$$b \neq \diamond^{y_1}$$

$\mathcal{M}_1$ a pour tâche d'associer aux variables $\diamond^{x_1}, \diamond^{x_2}, \diamond^{\hat{x}_1}$ de E, les sous-expressions de F correspondantes, au cours d'un parcours récursif simultané des deux expressions. C'est à dire ici, respectivement $b, \diamond^{y_1}, \diamond^{\hat{y}_1}$. Si créer une telle liste s'avère impossible, $\mathcal{M}_1$ retourne un message d'erreur.

Le rôle de $\mathcal{M}_2$ consiste à vérifier que $\diamond^{y_1}$ n'appartient pas au terme qui est associé à $\diamond^{x_1}$, en l'occurrence b.

6.1.2 Formulation de la fonction $\mathcal{M}_1$

Comme on l'a vu dans l'exemple précédent, au cours d'un parcours récursif simultané de E'' et de F, on associe aux variables de E'' les termes correspondants dans F dans une liste qu'on appellera *liste d'associations* et qui sera notée ℓ_{ass}. On vérifie ainsi que (i) est respectée ou non.

Notations

- On note une variable $\diamond^x$ qui n'est pas associée dans ℓ_{ass} par $\diamond^{x_1}/nil$.
- La liste d'associations ℓ_{ass} dans laquelle on associe un terme A à une variable $\diamond^x$ est notée : $\ell_{ass}/[\diamond^x/A]$.

– La concaténations de deux listes d'associations ℓ^1_{ass} et ℓ^2_{ass} est notée $\ell^1_{ass} \oplus \ell^2_{ass}$.

– La concaténation d'une liste d'association ℓ_{ass} et d'une association $[\diamond^x/A]$ est notée de même $\ell_{ass} \oplus [\diamond^x/A]$.

– La suppression d'une association $[\diamond^x/A]$ dans une liste ℓ_{ass} est notée $\ell_{ass} \ominus [\diamond^x/A]$.

Ces notations seront réutilisées par la suite pour des listes quelconques.

L'algorithme

• **En entrée de la fonction $\mathcal{M}_1$**

– $E \in \breve{\mathbb{L}}_2$

– $F \in \breve{\mathbb{L}}_2$

– ℓ_{ass}, une liste d'associations.

(Initialement, $\mathcal{M}_1$, qui est une fonction récursive, prend pour argument deux expressions $E, F \in \mathbb{L}_2$, et une liste d'associations dans laquelle se trouvent les variables $\diamond^{x_1}, \ldots, \diamond^{x_{n-m}}$[1] non encore associées.

• **En sortie de la fonction $\mathcal{M}_1$**

– ℓ_{ass}, une liste d'associations. Si une liste d'association licite ne peut être construite, la fonction retourne un message d'erreur symbolisé par ℓ^-_{ass} et qui met fin au parcours récursif.

$\mathcal{M}_1$ s'exprime comme suit :

$$\mathcal{M}_1(E \equiv \mathsf{ev}, F, \ell_{ass})$$
$$\rightarrow \begin{cases} \ell_{ass} & \text{si } \mathsf{ev} \equiv F \\ \ell^-_{ass} & \text{sinon} \end{cases}$$

$$\mathcal{M}_1(E \equiv \diamond^e, F, \ell_{ass})$$
$$\rightarrow \begin{cases} \text{si } E \notin \ell_{ass} & \\ \quad \ell_{ass} \oplus [E/F] & \\ \quad\quad \ell_{ass} & \text{si } [E/F] \in \ell_{ass} \\ \text{sinon } \ell_{ass}/[E/F] & \text{si } [E/nil] \in \ell_{ass} \\ \quad\quad \ell^-_{ass} & \text{sinon} \end{cases}$$

$$\mathcal{M}_1(E \equiv \lambda \diamond^{G_1} .H_1, F, \ell_{ass})$$
$$\rightarrow \begin{cases} \mathcal{M}_1(H_1, H_2, \mathcal{M}_1(G_1, G_2, \ell_{ass})) & \text{si } F \equiv \lambda \diamond^{G_2} .H_2 \\ \ell^-_{ass} & \text{sinon} \end{cases}$$

$$\mathcal{M}_1(E \equiv \mathsf{op}[G_1, H_1], q, \ell_{ass})$$
$$\rightarrow \begin{cases} \mathcal{M}_1(G_1, G_2, \mathcal{M}_1(H_1, H_2, \ell_{ass})) & \text{si } F \equiv \mathsf{op}[G_2, H_2] \\ \ell^-_{ass} & \text{sinon} \end{cases}$$

$$\mathcal{M}_1(E \equiv \mathsf{op}^r[G_1], q, \ell_{ass})$$
$$\rightarrow \begin{cases} \mathcal{M}_1(G_1, G_2, \ell_{ass}) & \text{si } F \equiv \mathsf{op}^r[G_2] \\ \ell^-_{ass} & \text{sinon} \end{cases}$$

6.1.3 Formulation de la fonction $\mathcal{M}_2$

A partir de la liste d'associations retournée par $\mathcal{M}_1$, on construit la liste ℓ_{var} constituée par les variables suivantes[2] :

[1] conformément aux notations utilisées jusqu'ici

[2] toujours en référence aux notations en vigueur jusqu'ici

$\diamond^{y_1}, \ldots, \diamond^{y_m}, \diamond^{\hat{y}_1}, \ldots, \diamond^{\hat{y}_q}$

Il est clair que la liste constituée des m premières variables suffirait, mais dans la pratique on ne peut pas les distinguer des autres dans ℓ_{ass}.

$\mathcal{M}_2$ doit alors s'assurer que toutes les variables de ℓ_{var} qui sont présentes dans les termes associés à $\diamond^{x_1}, \ldots, \diamond^{x_{n-m}}$ sont liées dans ces termes. Soit de façon équivalente que $\diamond^{y_1}, \ldots, \diamond^{y_m}$ n'appartiennent pas à ces termes. En somme, que (ii) est vérifié.

L'algorithme

- **En entrée de la fonction $\mathcal{M}_2$**

– ℓ_{ass}, une liste d'associations.

– ℓ^0_{ass}, la liste d'associations $[\diamond^{x_1}/nil, \ldots, \diamond^{x_{n-m}}/nil]$

- **En sortie de la fonction $\mathcal{M}_2$**

– un booléen

$\mathcal{M}_2$ s'exprime comme suit :

$$\mathcal{M}_2(\ell_{ass}, \ell^0_{ass}, \ell_{var})$$
$$\rightarrow \left\{ \begin{array}{ll} \text{si } \ell_{ass} = [\diamond^x/A] \oplus \ell^1_{ass} & \\ \quad \mathcal{M}_2(\ell_{ass}, \ell^0_{ass}, \ell_{var}) & \text{si} \mathcal{M}'_2(A, \ell_{var}) \\ \quad false & sinon \\ \text{si } \ell_{ass} = [\diamond^x/A] & \\ \quad \mathcal{M}'_2(A, \ell_{var}) & \end{array} \right.$$

Où $\mathcal{M}'_2$ est la fonction récursive exprimée ci-dessous :

- **En entrée de la fonction $\mathcal{M}'_2$**

– $E \in \breve{\mathbb{L}}_2$

– ℓ_{var}, une liste de variables.

- **En sortie de la fonction $\mathcal{M}'_2$**

– un booléen

$$\mathcal{M}'_2(E \equiv \mathsf{ev}, \ell_{var})$$
$$\rightarrow \{ \ true$$

$$\mathcal{M}'_2(E \equiv \diamond^e, \ell_{var})$$
$$\rightarrow \left\{ \begin{array}{ll} true & \text{si } E \notin \ell_{var} \\ false & \text{sinon} \end{array} \right.$$

$$\mathcal{M}'_2(E \equiv \lambda \diamond^F .G, \ell_{var})$$
$$\rightarrow \{ \ \mathcal{M}'_2(G, \ell_{var} \ominus \diamond^F)$$

$$\mathcal{M}'_2(E \equiv \mathsf{op}[F, G], \ell_{var})$$
$$\rightarrow \{ \ (\mathcal{M}'_2(F, \ell_{var}) and \mathcal{M}'_2(G, \ell_{var}))$$

$$\mathcal{M}'_2(E \equiv \mathsf{op}^r[F], \ell_{var})$$
$$\rightarrow \{ \ \mathcal{M}'_2(F, \ell_{var})$$

6.2 Calcul du corps de l'abstraction

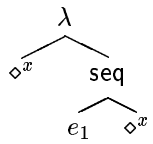
En s'appuyant sur la définition de $\mathcal{V}$, on va ici tacher de formuler un algorithme qui réalise cette opération. La définition de $\mathcal{V}$ étant récursive (cf 5.2.3), on va naturellement construire un algorithme fondé sur un parcours récursif des arguments de la fonction. Ce parcours, toujours d'après la définition de $\mathcal{V}$, devrait être effectué *en largeur*. Cependant, compte tenu de la remarque relative à l'identité en 5.3.2, il semble intéressant d'étudier la possibilité d'effectuer la même opération par un parcours *en profondeur d'abord*.

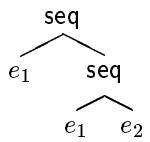
6.2.1 Choix du parcours récursif : *en largeur* ou *en profondeur d'abord*

Voyons sur deux exemples les mécanismes de ces deux modes de parcours.

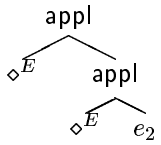
Exemple 1

Soient $E, F \in \mathbb{L}_2$ définies par :

$E \equiv$ 

$F \equiv$ 

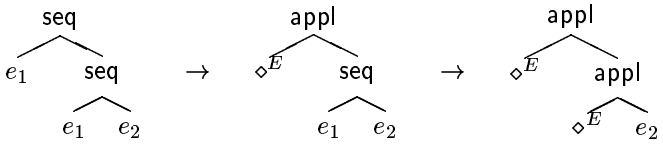
On sait par définition que l'application de $\mathcal{V}$ à ces deux arguments donne :

$\mathcal{V}(E,F) \equiv$ 

Si on s'en tenait simplement à la définition de $\mathcal{V}$, on adopterait un parcours *en largeur*. Cependant, il existe une bonne raison d'envisager un parcours *en profondeur d'abord*. Il faut en effet veiller que l'abstraction de l'identité ne retourne pas une infinité d'applications. C'est à dire, pratiquement, que l'algorithme ne boucle pas.

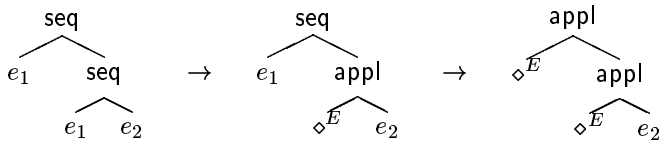
Parcours *en largeur*

Voici les deux étapes qui conduisent au résultat :



Parcours *en profondeur d'abord*

Voici les deux étapes qui conduisent au résultat :

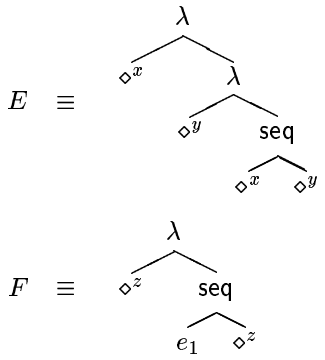


Excepté le fait qu'il permet de mettre en évidences les mécanismes des deux modes de parcours, cet exemple ne fournit pas d'argument favorable à l'une ou l'autre des méthodes.

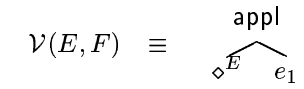
Pour l'instant, seul le cas particulier pourrait nous faire préférer le parcours *en profondeur d'abord*. Considérons donc un autre exemple :

Exemple 2

Soient $E, F \in \mathbb{L}_2$ définies par :

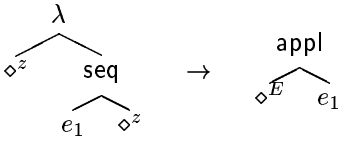


Par définition de $\mathcal{V}$, on sait que la fonction $\mathcal{V}$ appliquée à ces deux arguments retourne :



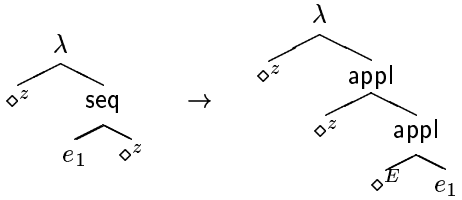
Parcours *en largeur*

On a le résultat en une seule étape :



Parcours *en profondeur d'abord*

On obtient un autre (!) résultat en une étape :



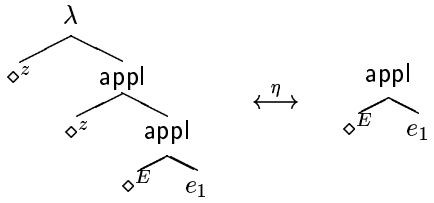
On pourrait alors en conclure hâtivement que si le parcours *en profondeur d'abord* fournit dans certains cas des résultats erronés, c'est qu'il n'est pas adapté au problème. Cependant on a vu qu'il permettait de régler le cas de l'identité. On aurait donc tort de s'en débarrasser aussi vite.

D'autant qu'en accordant un peu d'attention au dernier résultat, il apparaît qu'il n'est pas réellement erroné.

Rappelons en effet qu'en 1.3.3, ont été définies trois règles de conversions parmi lesquelles l'η-conversion. Dans le cadre du λ-calcul, on l'a exprimée ainsi :

$$\lambda x.e \overset{\eta}{\longleftrightarrow} e \text{ si } x \notin FV(e)$$

Dans le présent exemple, il se trouve que :



Bien entendu, ce n'est pas un hasard. L'exemple 2 est représentatif d'un ensemble de cas pour lequel le parcours *en profondeur d'abord* fournit un résultat correct à un certain nombre d'η-conversions près.

Soient en effet deux abstractions E et F telles que $E \geq F$.

Notons :

$$E \equiv \lambda \diamond^{x_1} \ldots . \lambda \diamond^{x_n} . E'$$

$$F \equiv \lambda \diamond^{y_1} \ldots . \lambda \diamond^{y_m} . F'$$

où E' et F' ne sont pas des abstractions, et

$$E \equiv \lambda \diamond^{x_1} \ldots . \lambda \diamond^{x_{n-m}} . E''$$

On sait alors par hypothèses qu'il existe un unique $(A_1, \ldots, A_{n-m}) \in \breve{\mathbb{L}}_2$ tel que :

$$E''[\diamond^{x_1}/A_1, \ldots, \diamond^{x_k}/A_k] \overset{\alpha}{\equiv} F$$

Par ailleurs il est clair que $E \geq F \Longrightarrow E \geq F'$

Aussi, dans le parcours *en profondeur d'abord*, la relation $E \geq F'$ est vérifiée avant que ne le soit $E \geq F$.

C'est pourquoi, on a :

$$\mathcal{V}(E, F) \equiv \lambda \diamond^{y_1} \ldots . \lambda \diamond^{y_m} . (\diamond^E \ A_1 \ \ldots \ A_{n-m} \ y_1 \ \ldots \ y_m)$$

On obtient le résultat escompté à partir de cette dernière expression en procédant à m η-réductions successives.

Choix du mode de parcours

Pour mettre en oeuvre la fonction $\mathcal{V}$, compte tenu des atouts respectifs des deux modes de parcours d'arborescences, il semble raisonnable de ne choisir ni l'une ni l'autre des méthodes précédentes mais un parcours hybride, dont les avantages sont les suivants :

– L'identité n'est l'objet d'aucun traitement spécifique.

– Il n'est pas nécessaire de recourir à un processus d'η-réduction.

Cette méthode consiste simplement à parcourir les abstractions *en largeur* et les sous-termes qui ne sont pas des abstractions, *en profondeur d'abord*

Si l'on reprend les termes de 5.2.3, cela implique que dans le cas du parcours *en largeur* on dispose d'une fonction $\mathcal{A}_{rec}$ qui retourne les applications :

$$(\diamond^E \ \mathcal{V}(E, A_1) \ \ldots \ \mathcal{V}(E, A_{n-m}))$$

et dans le cas du parcours *en profondeur d'abord*, une fonction $\mathcal{A}$ qui retourne les applications :

$$(\diamond^E \ A_1 \ \ldots \ A_{n-m})$$

Le parcours est réalisé par la fonction $\mathcal{P}$, exprimée dans le paragraphe suivant.

6.2.2 Formulation de la fonction $\mathcal{P}$

• **En entrée de la fonction $\mathcal{P}$**

– $E \in \breve{\mathbb{L}}_2$

– $E' \in \mathbb{L}_2$

– $varE \in \mathbb{V}$

– $F \in \breve{\mathbb{L}}_2$

– ℓ^0_{ass}, la liste d'associations $[\diamond^{x_1}/nil, \ldots, \diamond^{x_{n-m}}/nil]$.

• **En sortie de la fonction $\mathcal{P}$**

– une expression de $\breve{\mathbb{L}}_2$

$$\mathcal{P}(E, E', \diamond^E, F \equiv \mathsf{ev}, \ell^0_{ass})$$
$$\rightarrow \begin{cases} \text{Soit } \ell_{ass} = \mathcal{M}_1(E, \mathsf{ev}, \ell^0_{ass}) & \\ \quad \mathsf{appl}[\diamond^E, \mathsf{ev}] & \text{si } \mathcal{M}_2(\ell_{ass}, \mathcal{B}(\ell_{ass}, \ell^0_{ass})) = true \\ \quad \mathsf{ev} & \text{sinon} \end{cases}$$

$$\mathcal{P}(E, E', \diamond^E, F \equiv \diamond^G, \ell^0_{ass})$$
$$\rightarrow \{ \ F$$

$\mathcal{P}(E, E', \diamond^E, F \equiv \lambda\, \diamond^G .H, \ell^0_{ass})$

$$\rightarrow \left\{ \begin{array}{l} \text{Soient } \ell^1_{ass} = nil \\ \text{et } E'' = E \\ \text{tant que } E'' \equiv \lambda\, \diamond^I .J \\ \quad \left| \begin{array}{l} \ell^2_{ass} = \mathcal{M}_1(E'', F, \ell^1_{ass}) \\ \text{si } \mathcal{M}_2(\ell^1_{ass}, \mathcal{B}(\ell^1_{ass}, \ell^0_{ass})) = true \\ \quad \mathcal{A}_{rec}(E, E', \diamond^E, \ell^1_{ass}, \ell^0_{ass}) \\ \text{sinon} \\ \quad E' = J \\ \quad \ell^1_{ass} = \ell^1_{ass} \oplus \diamond^I \end{array} \right. \\ \text{fin tant que} \\ \ell_{ass} = \mathcal{M}_1(E', F, \ell^0_{ass}) \\ \text{si } \mathcal{M}_2(\ell_{ass}, \mathcal{B}(\ell^1_{ass}, \ell^0_{ass})) = true \\ \quad \mathcal{A}(\diamond^E, \ell_{ass}, \ell^0_{ass}) \\ \text{sinon} \\ \quad \mathcal{P}(E, E', \diamond^E, H, \ell^0_{ass}) \end{array} \right.$$

$\mathcal{P}(E, E', \diamond^E, F \equiv \mathsf{op}[G, H], \ell^0_{ass})$

$$\rightarrow \left\{ \begin{array}{ll} \text{Soit } \ell_{ass} = \mathcal{M}_1(E, \mathsf{op}[\mathcal{P}(E, E', \diamond^E, G), \mathcal{P}(E, E', \diamond^E, H)], \ell^0_{ass}) & \\ \quad \mathcal{A}(\diamond^E, \ell_{ass}, \ell^0_{ass}) & \text{si } \mathcal{M}_2(\ell_{ass}, \mathcal{B}(\ell_{ass}, \ell^0_{ass})) = true \\ \quad \mathsf{op}[\mathcal{P}(E, E', \diamond^E, G), \mathcal{P}(E, E', \diamond^E, H)] & \text{sinon} \end{array} \right.$$

$\mathcal{P}(E, E', \diamond^E, F \equiv \mathsf{op}^r[G], \ell^0_{ass})$

$$\rightarrow \left\{ \begin{array}{ll} \text{Soit } \ell_{ass} = \mathcal{M}_1(E, \mathsf{op}^r[\mathcal{P}(E, E', \diamond^E, G)], \ell^0_{ass}) & \\ \quad \mathcal{A}(\diamond^E, \ell_{ass}, \ell^0_{ass}) & \text{si } \mathcal{M}_2(\ell_{ass}, \mathcal{B}(\ell_{ass}, \ell^0_{ass})) = true \\ \quad \mathsf{op}^r[\mathcal{P}(E, E', \diamond^E, G)] & \text{sinon} \end{array} \right.$$

Où $\mathcal{B}$ est la fonction qui retourne la liste ℓ_{var} utilisée par $\mathcal{M}_2$. On rappelle que, conformément aux notations habituelles, cette liste est constituée des variables :

$$\diamond^{y_1}, \ldots, \diamond^{y_m}, \diamond^{\hat{y}_1}, \ldots, \diamond^{\hat{y}_q}$$

6.2.3 Formulation de $\mathcal{B}$

- **En entrée de la fonction $\mathcal{B}$**

– ℓ_{ass} une liste d'associations

– ℓ^0_{ass}, la liste d'associations $[\diamond^{x_1}/nil, \ldots, \diamond^{x_{n-m}}/nil]$.

- **En sortie de la fonction $\mathcal{B}$**

– la liste des variables $\diamond^{y_1}, \ldots, \diamond^{y_m}, \diamond^{\hat{y}_1}, \ldots, \diamond^{\hat{y}_q}$: ℓ_{var}

$\mathcal{B}(\ell_{ass}, \ell^0_{ass})$

$$\rightarrow \left\{ \begin{array}{ll} \diamond^y & \text{si } \ell_{ass} = [\diamond^x/\diamond^y] \text{ et } \diamond^x \notin \ell_{ass} \\ \diamond^y \oplus \mathcal{B}(\ell^1_{ass}, \ell^0_{ass}) & \text{si } \ell_{ass} = [\diamond^x/\diamond^y] \oplus \ell^1_{ass} \text{ et } \diamond^x \notin \ell_{ass} \end{array} \right.$$

6.2.4 Formulation des fonctions $\mathcal{A}$ et $\mathcal{A}_{rec}$

La fonction $\mathcal{A}$

- **En entrée de la fonction $\mathcal{A}$**

– $varE \in \mathbb{V}$

– ℓ_{ass} une liste d'associations

– ℓ^0_{ass}, la liste d'associations $[\diamond^{x_1}/nil, \ldots, \diamond^{x_{n-m}}/nil]$.

- **En sortie de la fonction $\mathcal{A}$**

– une expression de $\breve{\mathbb{L}}_2$

$$\mathcal{A}(E, E', \diamond^E, \ell_{ass}, \ell^0_{ass})$$
$$\rightarrow \begin{cases} \mathsf{appl}[\diamond^E, A] & \text{si } \ell_{ass} = [\diamond^x/A] \text{ et } \diamond^x \in \ell_{ass} \\ \mathsf{appl}[\mathcal{A}(\diamond^E, \ell^1_{ass}, \ell^0_{ass}), A] & \text{si } \ell_{ass} = [\diamond^x/A] \oplus \ell^1_{ass} \text{ et } \diamond^x \in \ell_{ass} \end{cases}$$

La fonction $\mathcal{A}_{rec}$

$\mathcal{A}_{rec}$ possède une structure récursive croisée avec $\mathcal{P}$, comme en témoigne la formulation suivante :

- **En entrée de la fonction $\mathcal{A}_{rec}$**

– $E \in \breve{\mathbb{L}}_2$

– $E' \in \mathbb{L}_2$

– $varE \in \mathbb{V}$

– ℓ_{ass} une liste d'associations

– ℓ^0_{ass}, la liste d'associations $[\diamond^{x_1}/nil, \ldots, \diamond^{x_{n-m}}/nil]$.

- **En sortie de la fonction $\mathcal{A}_{rec}$**

– une expression de $\breve{\mathbb{L}}_2$

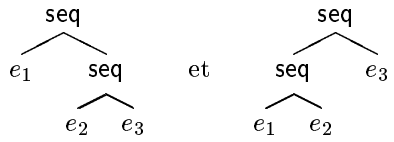
$$\mathcal{A}_{rec}(E, E', \diamond^E, \ell_{ass}, \ell^0_{ass})$$
$$\rightarrow \begin{cases} \mathsf{appl}[\diamond^E, A] & \text{si } \ell_{ass} = [\diamond^x/A] \text{ et } \diamond^x \in \ell_{ass} \\ \mathsf{appl}[\mathcal{A}_{rec}(\diamond^E, \ell^1_{ass}, \ell^0_{ass}), \mathcal{P}(E, E', \diamond^E, A, \ell^0_{ass}] & \text{si } \ell_{ass} = [\diamond^x/A] \oplus \ell^1_{ass} \text{ et } \diamond^x \in \ell_{ass} \end{cases}$$

Chapitre 7

Un autre opérateur d'abstraction généralisée

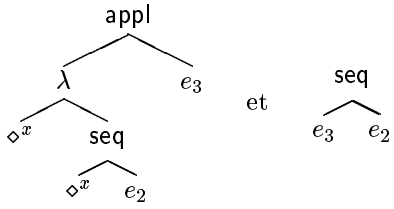
7.1 Présentation

L'abstraction généralisée est fondée sur une relation qui formalise la notion de *généralité* d'une expression. Dans le chapitre 4, on a proposé une relation formalisant une certaine notion de *généralité*. Cette idée est elle-même liée au fait que les étapes de la construction d'une expression font partie intégrante de l'expression. Or, lorsque l'utilisateur accède à une expression par le biais de sa représentation sonore, il perçoit une expression dont les *ingrédients* ont été *consommés*. Rien, par exemple, ne permet de distinguer les représentations sonores des deux expressions suivantes, alors qu'elles ne sont pas syntaxiquement égales :



De même en ce qui concerne :

Ou encore :



Ainsi, si une expression fait sens dans *Elody* en référence à sa valeur dans $\mathbb{L}_2$, il semble intéressant qu'elle puisse aussi faire sens en référence à sa représentation sonore, c'est à dire à sa valeur dans $\mathbb{L}_4$. Ces deux possibilités étant exclusives, on pourrait imaginer que l'utilisateur choisisse l'une ou l'autre selon les opérations qu'il souhaite réaliser.

Choisir la deuxième possibilité signifie décider d'identifier une classe d'expressions, qui sont syntaxiquement égales, moyennant deux types de transformations, comme en témoignent les exemples précédents :

1. *Une réduction des expressions*
2. *Une mise sous forme canonique*

On pourrait donc imaginer un opérateur d'abstraction généralisée qui tienne compte de ces deux notions. Une expression E serait de cette manière plus générale qu'une expression F, si E, réduite et mise sous forme canonique est plus générale que F, réduite et mise sous forme canonique.

Le processus de réduction étant connu et décrit dans [Orlarey et al. 1997], on va uniquement proposer une forme canonique envisageable pour les expressions de $\mathbb{L}_2$. Celle-ci sera définie en référence aux propriétés des constructeurs mix et seq qu'on a pu entrevoir dans les deux premiers exemples

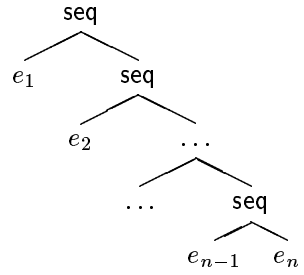
7.2 Une organisation standard des constructeurs mix et seq

Musicalement, on appréhende généralement un accord ou une séquence de n événements comme des vecteurs de dimension n, et non une structure engendrée par un opérateur binaire. Ainsi, $[mi; fa; sol]$ représenterait à la fois $[mi; [fa; sol]]$ et $[[mi; fa]; sol]$. Il apparaît que la définition d'une forme standard est liée aux propriétés d'associativité de mix et seq et de commutativité de mix, que l'on peut traduire par des règles de réécriture simples :

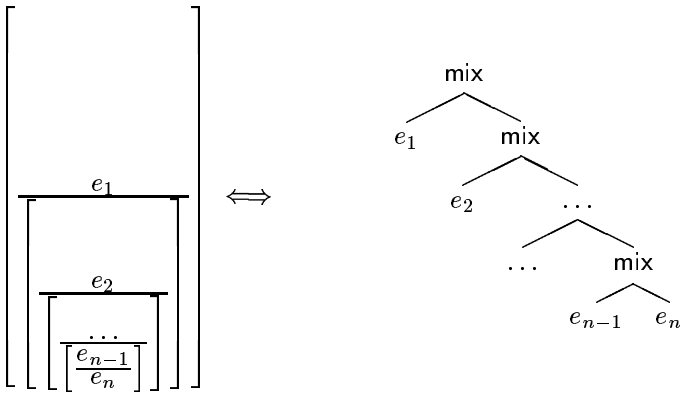
- associativité de droite à gauche : $[e_1; [e_2; e_3]] \rightarrow [[e_1; e_2]; e_3]$
- associativité de gauche à droite : $[[e_1; e_2]; e_3] \rightarrow [e_1; [e_2; e_3]]$
- associativité de bas en haut : $\left[\frac{e_1}{\left[\frac{e_2}{e_3}\right]}\right] \rightarrow \left[\frac{\left[\frac{e_1}{e_2}\right]}{e_3}\right]$
- associativité de haut en bas : $\left[\frac{\left[\frac{e_1}{e_2}\right]}{e_3}\right] \rightarrow \left[\frac{e_1}{\left[\frac{e_2}{e_3}\right]}\right]$
- commutativité : $\left[\frac{e_1}{e_2}\right] \rightarrow \left[\frac{e_2}{e_1}\right]$

• On convient alors que la forme canonique d'une expression possédant n feuilles et dont les opérateurs ne sont que des séquences est :

$[e_1; [e_2; [\ldots; [e_{n-1}; e_n]]]] \iff$



• En ce qui concerne mix, la commutativité contraint à définir une relation d'ordre sur les expressions qu'on notera $\lessdot$. Alors, la forme canonique d'une expression possédant n feuilles et dont les opérateurs sont tous des opérateurs de mixage est :



avec $e_1 \lessdot e_2 \lessdot \ldots \lessdot e_{n-1} \lessdot e_n$

On en déduit un algorithme simple permettant de réarranger toute expression de $\mathbb{L}_2$ en une forme standard unique.

7.3 Algorithme

Notons $\mathcal{C}$ la fonction qui met une expression sous forme canonique. $\mathcal{C}$ se décompose comme suit :

$\mathcal{C}(\mathsf{ev}) \rightarrow \mathsf{ev}$

$\mathcal{C}(\mathsf{var}) \rightarrow \mathsf{var}$

$\mathcal{C}(\mathsf{seq}[e, f]) \rightarrow AssocSeq(\mathsf{seq}[e, f])$

$\mathcal{C}(\mathsf{mix}[e, f]) \rightarrow Construit(TriMix(ListeMix(\mathsf{mix}[e, f])))$

$\mathcal{C}(\mathsf{abstr}[d, \diamond^e, f] \rightarrow \mathsf{abstr}[d, \diamond^e, \mathcal{C}(f)])$

$\mathcal{C}(\mathsf{appl}[e, f]]) \rightarrow \mathsf{appl}[\mathcal{C}(e), \mathcal{C}(f)]$

$\mathcal{C}(\mathsf{trp}[e, n]]) \rightarrow \mathsf{trp}[\mathcal{C}(e), n]$

Dans le cas de l'opérateur mix, $ListeMix$ crée la liste des n expressions qui doivent être réordonnées, $TriMix$ ordonne ces expressions et $Construit$ retourne l'expression standard à partir de la liste ordonnée.

$$AssocSeq(\mathsf{seq}[e, f]) \rightarrow \begin{cases} AssocSeq(\mathsf{seq}[a, \mathsf{seq}[b, f]]) & \text{si } e \equiv \mathsf{seq}[a, b] \\ \mathsf{seq}(\mathcal{C}(e), AssocSeq(f)] & \text{sinon} \end{cases}$$

$AssocSeq(e \not\equiv \mathsf{seq}[f, g]) \rightarrow \mathcal{C}(e)$

$ListeMix(\mathsf{mix}[e, f]) \rightarrow [ListeMix(e)] \oplus [ListeMix(e)]$

$ListeMix(e \not\equiv \mathsf{mix}[f, g]) \rightarrow [\mathcal{C}(e)]$

7.4 Bilan

La forme canonique définie précédemment fait uniquement référence aux propriétés des constructeurs fondamentaux mix et seq. Cette démarche mériterait donc d'être étendue aux autres constructeurs du langage.

Elle fournit en outre l'occasion de souligner que l'opérateur d'abstraction généralisée qui a été présenté dans le cadre de l'étude n'est pas immuable. Il ne tient qu'à *une* formalisation possible de la

notion subjective de *généralité* d'une expression musicale, cohérente avec les principes du λ-calcul qui sont à la base du langage.

Chapitre 8

Méthodologie et développement

Le développement d'un opérateur d'abstraction généralisée a été initialement motivée par la perspective de son utilisation dans le langage de programmation musicale *Elody*. Son implémentation dans l'interprèteur a donc constitué l'aboutissement du travail.

8.1 Méthodologie

Il est important de préciser la méthodologie à laquelle l'étude s'est conformée, de la spécification du comportement de l'opérateur dans certains cas, à son intégration finale dans le langage de programmation *Elody*. La figure 8.1 montre que la démarche a consisté à formuler un algorithme qui vérifie deux spécifications indépendantes :

1. l'opérateur qu'il implémente doit être cohérent avec les principes du λ-calcul.
2. Il doit fournir des résultats précis pour un ensemble de cas déterminés.

En élargissant progressivement la seconde spécification, on arrive à cerner le comportement général de l'opérateur. Il est alors possible de le formuler mathématiquement et d'en déduire un algorithme définitif. Il s'ensuit une première implémentation en JAVA, préalable à l'implémentation finale dans l'interprèteur de *Elody*.

8.2 Développement

8.2.1 Programmation fonctionnelle en LISP

LISP a servi a implémenter les algorithmes. Il s'est révélé être un langage tout à fait adapté à cet usage pour de multiples raisons.

- En LISP, les structures de données de base sont des listes. Or les listes permettent de construire des arborescences de façon simple. LISP est donc particulièrement adapté à la manipulation d'arbres ([Wertz 1989], [Winston et al. 1981], [Gunter 1992])
- Les techniques de filtrage sont l'objet d'une documentation assez vaste en ce qui concerne leur implémentation en LISP (voir par exemple [Queinnec 1990]).
- LISP étant un langage de programmation fonctionnelle, il permet un développement aisé d'un interprèteur de λ-calcul.
- LISP est un langage interprété. On n'effectue donc pas la procédure de compilation, édition des liens et exécution qui est coûteuse en temps. Cela permet donc d'effectuer des modifications dans le code puis de tester le programme rapidement.

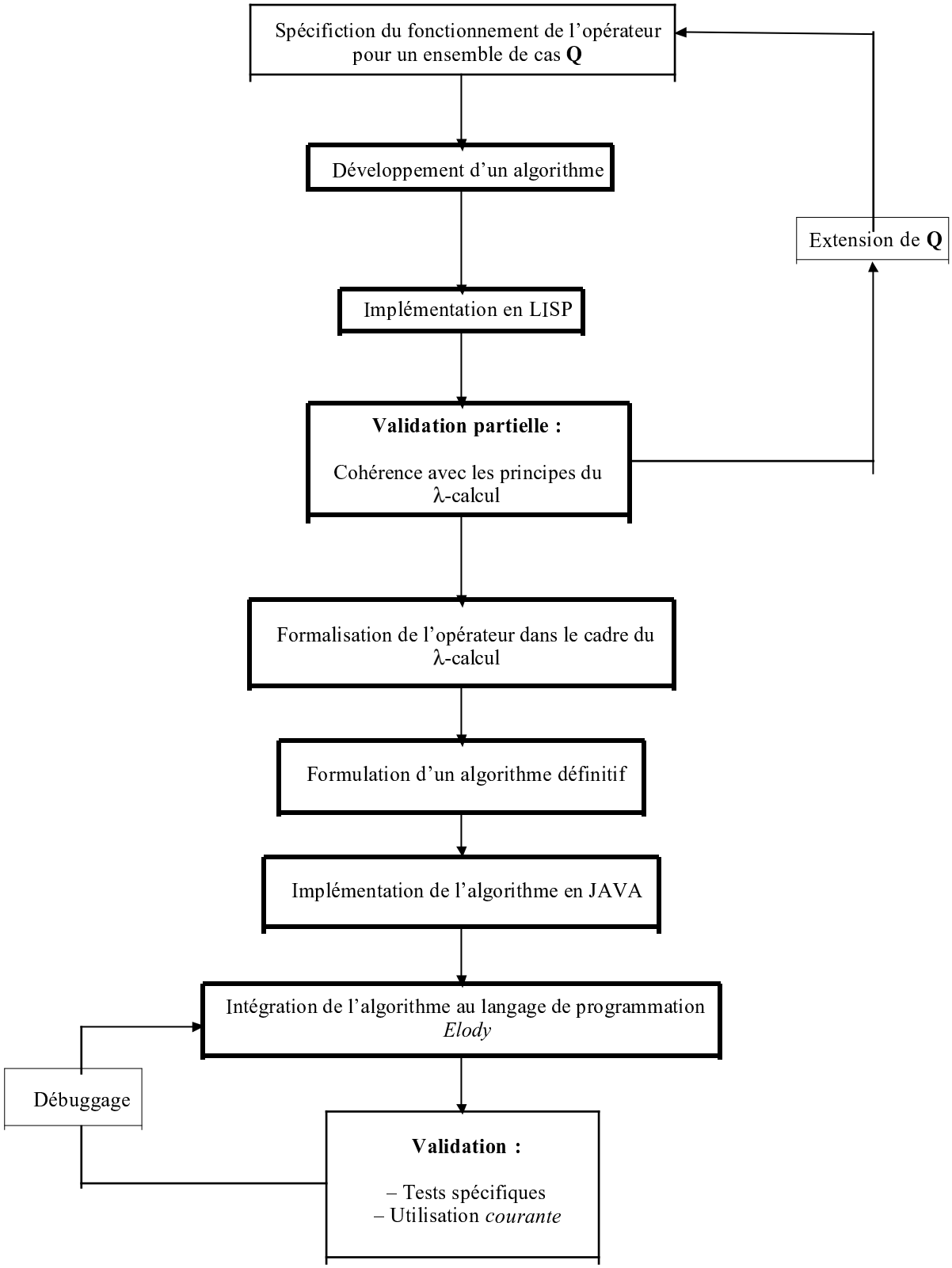


FIG. 8.1 – organigramme méthodologique

De façon générale, l'utilisation de LISP s'est révélée très enrichissante, car incite à formuler des algorithmes de façon précise et concise.

Pour preuve, un extrait d'une fonction LISP qui réalise le test de pattern matching de deux expressions :

```
(defun match (exp1 exp2 liste-associations)
       (tuplecase exp1
                  ((EV) (if (equal exp1 exp2)
                        liste-associations
                        (matche-pas liste-associations)))

                  ((VAR) (if (present variables-locales exp1)
                        (test-variable exp1 exp2 liste-associations)))

                  ((ABSTR v1 b1) (tuplecase exp2)
                        ((ABSTR v2 b2) (match b1 b2 (match(v1 v2))))
                        (otherwise (matche-pas liste-associations))))

                  ((APPL e1 f1) (tuplecase exp2)
                        ((APPL e2 f2) (match e1 e2 (match f1 f2)))
                        (otherwise (matche-pas liste-associations))))

                  ((SEQ e1 f1) (tuplecase exp2)
                        ((SEQ e2 f2) (match e1 e2 (match f1 f2)))
                        (otherwise (matche-pas liste-associations))))

       (...)))
```

8.2.2 Programmation objets en JAVA

JAVA est un langage à objets, multiplateformes, conçu pour fonctionner avec la norme HTTP. Il est doté d'une bibliothèque de classes permettant la programmation d'interfaces graphiques.

La base de la programmation objets consiste à se donner une hiérarchie de classes auxquelles sont associées des méthodes.

L'interprèteur utilisé par *Elody* est constitué de classes dont la hiérarchie n'a pas été affectée par la substitution de l'abstraction généralisée à l'abstraction simple. Cette hiérarchie permet de décrire la structure arborescente des expressions de *Elody*, et d'associer des méthodes spécifiques à chaque noeud d'une telle arborescence conformément aux principes de programmation objets.

Ainsi, toutes les expressions du langage sont des objets de classes représentant tous les opérateurs du langage. Toutes ces classes dérivent d'une classe même classe abstraite qui représente la notion *abstraite* d'expression. Cela permet de définir dans la classe abstraite, des méthodes générales, qui sont redéfinies dans les classes relatives aux expressions qui ont un comportement spécifique, comme les abstractions, les variables ou les événements élémentaires. Ce mécanisme est schématisé dans l'arbre d'héritage simplifié de la figure 8.2.

Pour plus de détail sur l'implémentation de l'opérateur en JAVA dans sa version définitive, se reporter au code de l'interprèteur de *Elody* qui figure en annexe.

8.3 Utilisation de l'opérateur en situation courante

Ce paragraphe n'est en réalité qu'un prétexte pour faire part au lecteur de l'enthousiasme que j'ai éprouvé à utiliser *Elody* et l'opérateur d'abstraction généralisée. Aprés que l'opérateur a été intégré

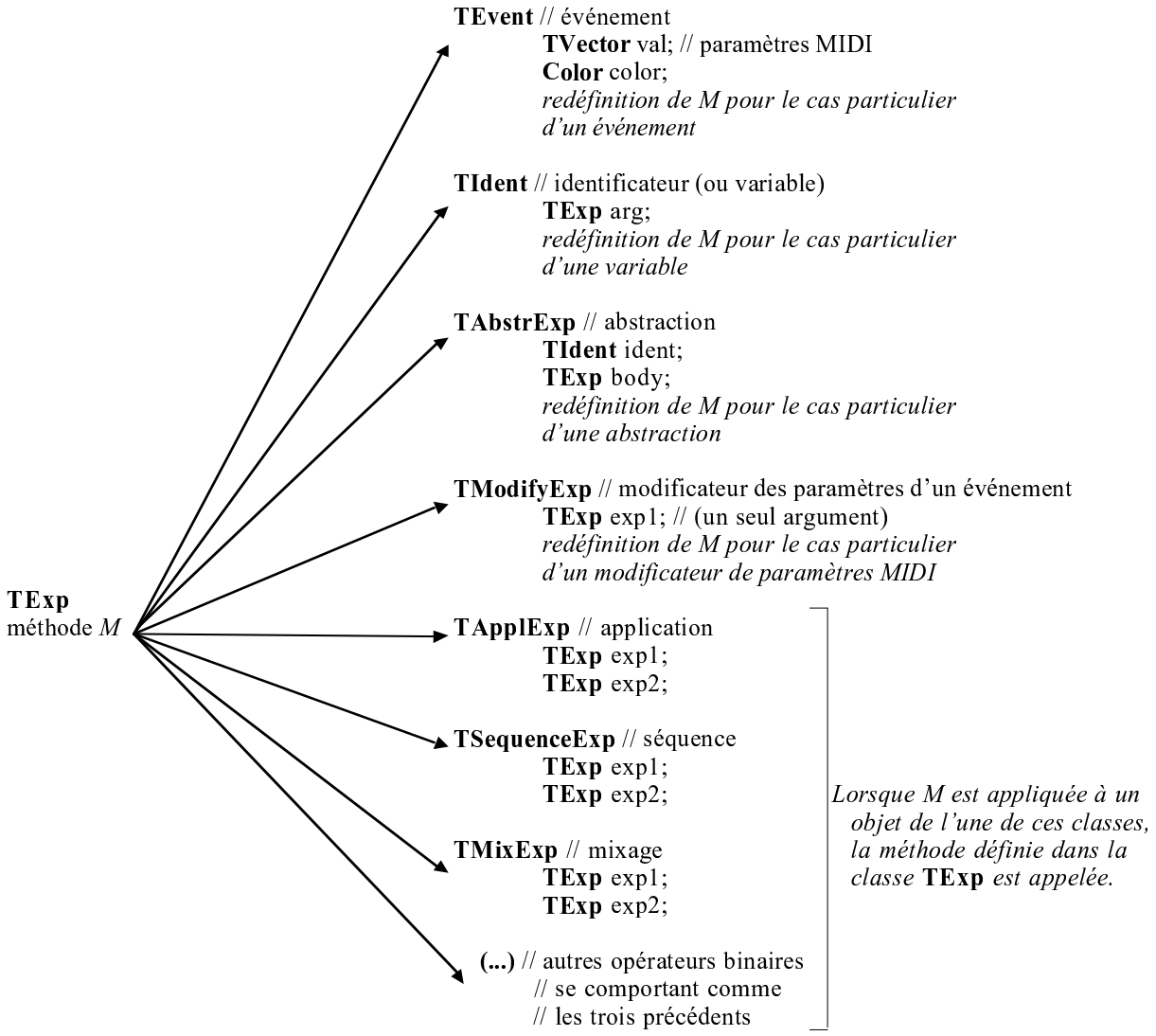


FIG. 8.2 – arbre d'héritage simplifié implémentant la structure arborescente des expressions de $\mathbb{L}_2$, redéfinition d'une méthode

au langage de programmation , je me suis en effet amusé à construire quelques expressions musicales à l'aide de l'abstraction généralisée. J'ai pu à cette occasion constater qu'il était possible de réaliser des choses très intéressantes à l'aide de ce langage.

Conclusion

La généralisation de la notion d'abstraction a donné lieu à la formulation d'un opérateur d'abstraction généralisée qu'on a formalisé et finalement intégré au langage de composition *Elody*. Ainsi, la fonctionnalité du langage de programmation musicale dérivé du λ-calcul, qui tient au mécanisme de l'abstraction, a pu être élargie.

La notion subjective de *généralité* des expressions musicales, sur lequel l'opérateur est fondé, a révélé qu'on pourrait envisager d'autres abstractions généralisées, associées à d'autres notions de *généralité*.

Quelque soit la notion de *généralité* considérée, il est essentiel de modéliser un opérateur qui soit cohérent avec les principes du λ-calcul. Vérifier cette condition tout en faisant en sorte d'obtenir un comportement spécifique de l'opérateur dans un ensemble de cas précis a constitué la principale difficulté de l'étude.

Sur un plan strictement personnel, l'implémentation des algorithmes associés à l'opérateur m'a permis d'adopter deux points de vue très différents concernant la programmation. Un point de vue *fonctionnel* avec LISP et un point de vue *objets* avec JAVA. En outre, j'ai pris conscience de la place fondamentale qu'occupe la notion d'abstraction dans la théorie des langages.

D'autre part, au-delà de la généralisation du mécanisme d'abstraction, on peut songer à la généralisation de l'utilisation des principes qui sont à la base d'*Elody*. On pourrait par exemple appliquer les principes du λ-calcul à un espace non plus constitué de notes et de silences, mais à des signaux. Peut-être peut-on même envisager un langage totalement générique, qui répondrait à une volonté exprimée, de modeler le vocabulaire et la grammaire du langage musical à l'aide d'un outil unique.

Bibliographie

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

[Chemillier 1994] M. Chemillier,*Analysis and Computer Reconstruction of a Musical Fragment of Gyögy Ligeti's Melodien*, Proceedings of the Symposium Music and Mathematics, Bucarest, 1994

[Gengler 1987] M. Gengler, *Un algorithme optimisé de semi-unification du second-ordre sur arborescences*, Bigre+Globule (59), 1987

[Gunter 1992] C. A. Gunter, *Semantics of Programming Languages*, MIT Press, Cambridge, 1992.

[Knight 1989] K. Knight, *Unification, A Multidisciplinary Survey*, ACM Computing Surveys, 21 (1), pp. 94-119, 1989.

[Ruf et Weise 1989] E. Ruf, D. Weise, *Nondeterministic and Unification in LogScheme : Integrating Logic and Functional Programming*, ACM, 1989.

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

[Orlarey et al. 1997] Y. Orlarey, D. Fober, S. Letz, *L'environnement de composition musicale* Elody, Actes des Jourées d'Informatique Musicale 1997.

[Queinnec 1990] C. Queinnec, *Le Filtrage, une application de (et pour) LISP*, Collection Science informatique, InterEditions, Paris, 1990.

[Wertz 1989] H. Wertz, *(Common) LISP une introduction à la programmation*, Masson, Paris, 1989.

[Winston et al. 1981] P. H. Winston, B. Klaus, P. Horn, *LISP*, Addisson-Wesley, 1981.