PROGRAMMER EN MAX AVEC *bell*

Andrea Agostini
Conservatoire de Turin
andrea.agostini@conservatoriotorino.eu

Daniele Ghisi
University of California, Berkeley
CNMAT
danieleghisi@berkeley.edu

Jean-Louis Giavitto
STMS, CNRS,
IRCAM, Sorbonne Université
jean-louis.giavitto@ircam.fr

RÉSUMÉ

Cet article aborde le sujet de la programmation dans l'environnement *Max* : on analysera son modèle de calcul, on présentera quelques stratégies d'implémentation de processus complexes spécifiquement dans le domaine de la composition algorithmique, et on évaluera la pertinence de son paradigme graphique par rapport à ce type de problèmes. Ces considérations ont amené au développement d'un mini-langage de programmation textuelle appelé *bell*, intégré dans la bibliothèque *bach*, une extension de *Max* pour la représentation musicale et l'aide à la composition. On présentera les choix fondamentaux qui en ont guidé la conception et on donnera un aperçu de sa syntaxe et de son modèle de fonctionnement.

1. INTRODUCTION

Le projet *bach* [3], une extension du logiciel *Max* développée par deux des auteurs (Agostini et Ghisi), est dédié à la représentation des données musicales et à la composition assistée et algorithmique. *bach* est une bibliothèque composée de plus de 200 modules dont la plus grande partie a pour but la manipulation d'un type de donnée dédié appelé *llll*, une structure d'arbre inspirée par la liste de Lisp [1] et utilisée pour la représentation de données musicales de tout degré de complexité, allant du simple scalaire, comme par exemple un paramètre de contrôle du traitement sonore, jusqu'à la partition complète.

Cependant, au cours de désormais dix ans de développement et d'utilisation de *bach*, l'implémentation de véritables processus algorithmiques dans *Max*, essentiels par définition à la pratique de la composition algorithmique, s'est souvent révélée difficile. Le développement de *cage* [2] est de ce point de vue éclairant. Ce projet, aux finalités pédagogiques, vise l'implémentation d'un ensemble de processus compositionnels « classiques » sous la forme de patches *Max* reposant sur les fonctionnalités de *bach*. Dans ce contexte, nous avons développé par exemple un module pour le calcul de la modulation en fréquence d'accords, une technique de génération harmonique typique de l'école de composition spectrale, déjà implémentée, entre autres, dans la bibliothèque *Esquisse* pour *PatchWork* et *OpenMusic* de Camilo Rueda, Jacques Duthen et Tristan Murail [12]. La version finale de ce patch, quoique fonctionnelle, représente pour nous un bon exemple de ce qu'il ne faudrait pas faire avec *Max* : il est composé de plusieurs centaines d'objets distribués dans des dizaines de sous-patches (certains sont montrés en figure 1) et son fonctionnement est tellement obscur que, lorsqu'il a été nécessaire d'en corriger les inévitables bogues, il a parfois été plus facile d'en réécrire des parties plutôt que de les isoler et de les amender. Ces problèmes apparaissent alors même que l'expression du processus musical reste relativement simple, la difficulté étant ici de traiter les nombreux cas particuliers induits par l'abstraction et la généralité souhaitées : il faut pouvoir par exemple moduler entre eux non seulement des accords mais aussi des bouts de partition.

Ce type de situation est malheureusement trop fréquent dans notre activité de compositeurs, chercheurs et enseignants, et nous avons commencé à remettre en question la notion même de « programmation en *Max* ». Un des résultats de cette réflexion a été la conception et le développement d'un mini-langage de programmation appelé *bell* [7], intégré dans *bach* à partir de la version 0.8.1, dédié à la manipulation des *llll*s et conçu de manière à s'intégrer parfaitement avec l'environnement *Max*. Ce dernier point a conduit à des choix de conception peut-être insolites d'un point de vue théorique, mais permettant une interaction fluide, concise et expressive avec l'environnement *Max*.

Dans la première partie de cet article nous exposerons

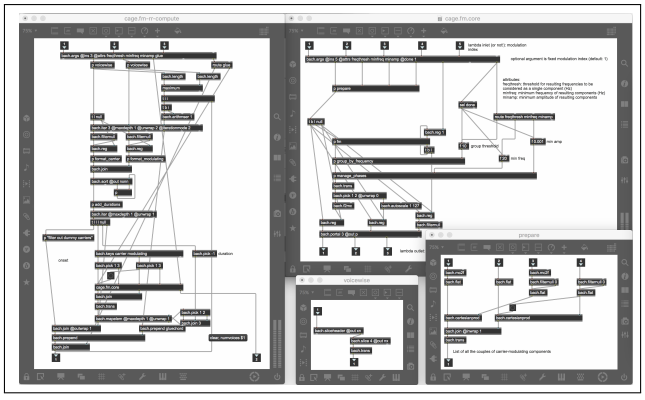

Figure 1. Une vue plongeante de quelques modules de *cage.fm*. Le patch est composé de plus de 500 objets (sans compter ceux compris dans des abstractions) et 30 sous-patches imbriqués sur plusieurs niveaux.

[1] L'acronyme *llll* signifie *Lisp-like linked list*.

les raisons qui nous ont poussés à entreprendre ce travail ; dans la seconde partie, nous présenterons les caractéristiques les plus importantes du langage *bell*.

2. PROGRAMMER EN *MAX* ?

2.1. *Max* est-il un langage de programmation ?

Max est un outil puissant, composé d'un nombre impressionnant de modules dédiés à des opérations de haut et moyen niveau pour manipuler l'audio, les données MIDI, la vidéo, etc. Bien qu'il soit Turing-complet, son paradigme de programmation graphique est beaucoup plus adapté à certaines nécessités – comme la construction de systèmes interactifs en temps réel –, qu'à d'autres – comme l'implémentation de processus complexes du point de vue algorithmique [17]. Il est par exemple facile d'y construire un synthétiseur de qualité remarquable, alors que calculer la série des nombres premiers de 1 à 100 est une tâche d'une difficulté certaine. En comparaison, dans un langage traditionnel tel que C, le premier problème requiert une quantité énorme de travail et de savoir-faire technique, sauf à utiliser des bibliothèques dédiées, alors que le second constitue un exercice presque banal. Les raisons de cette différence d'expressivité sont complexes mais on peut souligner que Max a été conçu comme un *instrument musical* plutôt que comme un *langage de programmation* ; et, de fait, la question de savoir si *Max* est un langage ou pas a été posée de nombreuses fois [10]. Le lecteur intéressé trouvera une description plus détaillée du modèle d'exécution de *Max* dans [6].

La réponse dépend certainement de la définition adoptée, mais le fait même que la question se pose donne une idée de la spécificité de ce système : personne ne se demande si Python est un langage de programmation, ou à l'inverse si *Minecraft*, quoique Turing-complet, en est un.

2.2. Le paradigme *data-flow* fonctionnel

Cela dit, *Max* est un système programmable. De plus, son paradigme de programmation *data-flow* – y compris au-delà de la couche graphique qui est la plus visible pour l'utilisateur – présente des affinités évidentes avec celui d'autres systèmes qui sont des langages de programmation de plein droit, comme par exemple OpenMusic [1] ou FAUST [16]. D'un autre côté, ces derniers systèmes reposent sur un variante spécifique du paradigme *data-flow*, le *data-flow fonctionnel*, dans lequel le programme est représenté, de manière graphique ou textuelle, comme un graphe équivalent à un système d'équations sur des séquences de valeurs (ou *stream*) associées aux arêtes du graphe [14]. Ce paradigme a des propriétés remarquables du point de vue formel, et on peut dire qu'en général il permet des optimisations importantes et encourage un style de programmation à la fois concis et clair. Au passage, il est nécessaire de préciser que le système audio de *Max* (historiquement appelé *MSP*) est fondé sur ce même paradigme, qui en fait se prête très bien, entre autres, à la représentation des graphes audio dans le domaine du traitement du signal. Tout ce qui suit se réfère donc exclusivement au sous-système de *Max* dédié au contrôle et réalisé par la manipulation et la transmission explicite de messages entre objets.

2.3. Le paradigme *data-flow* dans *Max*

Un patch *Max*, bien que constitué par un graphe, s'éloigne du *data-flow* fonctionnel et n'est pas équivalent à un système d'équations de *streams*, pour plusieurs raisons qui peuvent être résumées en quelques points principaux.

2.3.1. États mutables

Les objets ont des états mutables, *i.e.* qui changent dans le temps, et ils peuvent avoir un comportement *hétérochrone*, *i.e.* l'arrivée d'une donnée en entrée d'un nœud ne génère pas nécessairement une donnée en sortie de ce nœud. Cette particularité est démontrée par exemple par les entrées « froides » des objets arithmétiques, dans lesquelles les données sont enregistrées pour être réutilisées dans le futur ; et, de manière encore plus explicite, par certains objets dédiés spécifiquement à la mémorisation des données, tels que `int`, `zl.reg` et `value`, qui peuvent être vus comme équivalents à des variables dans un langage impératif (les deux premières identifiées par leur position dans le graphe, la troisième, de manière plus traditionnelle, par un nom symbolique).

2.3.2. Temporalité de l'évaluation

Par conséquent, la succession des données à l'entrée de chaque nœud du patch, propriété qui suffit à déterminer (avec les valeurs) le résultat d'un calcul dans le paradigme *data-flow* fonctionnel, ne permet pas de définir le résultat d'un calcul sous *Max* : il faut aussi connaître l'ordre relatif d'arrivée des données entre les différentes entrées d'un nœud (et parfois même la date d'arrivée de chaque valeur) et l'ordre d'activation des nœuds du patch. La sémantique d'un patch *Max* dépend donc de la synchronisation entre les différents *streams* et nécessite des objets, tel que `trigger`, et des règles spécifiques pour contrôler l'ordre de prise en compte des messages, telle la règle de déclenchement de droite à gauche.

2.3.3. Merge *implicite à l'entrée d'un nœud*

Dans *Max*, les *streams* de contrôle sont asynchrones : chaque donnée dans un *stream* correspond à une date de production ou de consommation différente. Cela rend possible et pertinente la connexion de plusieurs arêtes à une même entrée d'un nœud en réalisant une opération implicite de *merge* des streams connectés : les données sont toujours transmises de manière séquentielle, à des dates distinctes (correspondant à la réception de messages successifs) et prises en compte séquentiellement pour déclencher potentiellement le calcul lié à un nœud.

2.3.4. Motivation

Comme évoqué plus haut, ce modèle de calcul, qu'on peut qualifier de *data-flow asynchrone non-fonctionnel*, est motivé par la conception originelle de *Max* comme instrument de musique plutôt que langage de programmation. Le fonctionnement d'un patch peut être souvent rapproché de celui d'un objet mécanique soumis à des actions qui provoquent des réactions immédiates et des changements d'état physique qui auront des conséquences sur son comportement futur, et où toutes les interactions sont asynchrones (il n'y a pas d'événement simultané) : cette approche est souvent très efficace pour la modélisation et l'implémentation de systèmes réactifs [5].

2.4. *Max* vu comme un langage impératif

Si le modèle de *Max* s'écarte du *data-flow* fonctionnel, cela ne l'empêche pas d'être de plus en plus utilisé comme environnement de programmation pour y implémenter des calculs d'une complexité remarquable : le cas de la composition algorithmique, domaine auquel non seulement la bibliothèque *bach* est dédiée, mais aussi de nombreux autres projets, comme par exemple *MaxScore* [11], *AC-Toolkit* et *RTC-lib*, en est une bonne illustration.

Il n'est donc pas inintéressant de caractériser *Max* en tant que langage de programmation : il faudrait alors le considérer comme un langage impératif, du fait de l'importance de la séquentialité et des états [5]. En fait, un usage soigneux des objets-variables et du contrôle séquentiel (à travers les objets mentionnés plus haut et d'autres encore) permet un style de programmation manifestement impératif, qui a l'avantage de produire des patches « étiquetés » et clairement structurés (voir figure 2). Ce style n'est d'ailleurs pas toujours pratique, puisqu'il tend à produire des patches beaucoup plus grands que leurs équivalents qui profitent des nombreux « raccourcis » offerts par *Max*. Ces raccourcis, malheureusement, ont tendance à engendrer des graphes compliqués, dans lesquels les données suivent des parcours très ramifiés, de type « spaghetti » [18], qui distribuent l'état du programme à travers de nombreux objets localisés dans de nombreux patches (voir figure 3).

De plus, *Max* ne possède pas, ou alors il les implémente de manière plutôt singulière, certains concepts qui sont aujourd'hui partagés par presque tous les langages modernes et qui sont essentiels à l'expression efficace et lisible des programmes, comme l'encapsulation des données, les fonctions, la paramétrisation d'un processus par un autre processus, ou encore des structures de données riches et composables arbitrairement.

3. LANGAGES TEXTUELS POUR *MAX*

Pour répondre aux problèmes évoquées dans la section précédente, *Max* permet d'intégrer dans un patch des fragments de code textuel auxquels confier les parties les plus algorithmiques du calcul à réaliser, en réservant le paradigme graphique *data-flow* pour les tâches auxquelles il est le plus adapté, comme la programmation des interactions avec l'environnement (par exemple les interfaces graphiques ou la relation avec le MIDI et le hardware). L'évaluation de ces « capsules » de code textuel est réalisée en général par des objets dédiés qui s'interfacent au patch selon le mécanisme d'évaluation usuel de *Max*, en en recevant des données et en y envoyant des résultats. Le code impliqué peut relever de plusieurs langages : la distribution officielle de *Max* supporte C, C++, Java, JavaScript, Lua et le langage propriétaire GenExpr. En outre, des *bindings* vers plusieurs autres langages (Python, plusieurs dialectes de Lisp, Antescofo [9], SuperCollider, Csound, RTCMix [13], *Odot* [15], etc.) ont été réalisés par des développeurs indépendants.

Quoique cette fonctionnalité puisse sembler trahir la nature graphique de *Max*, qui est souvent proposée comme une alternative « facile » à la difficulté de la programmation textuelle, notre avis est que les deux approches – graphique et textuelle – sont complémentaires, et qu'un mélange des deux peut souvent se révéler à la fois plus expressif, concis et lisible que dans une seule des deux approches.

Ces considérations nous ramènent à la motivation évoquée au début de cet article : comment peut-on exprimer de manière claire et efficace des processus compositionnels algorithmiques dans le contexte de la bibliothèque *bach* ? Les arguments précédents pointent les limitations intrinsèques du style de programmation graphique dataflow propre à *Max*, au moins pour les calculs combinatoires et/ou dynamiques. Nous proposons de développer une approche textuelle dédiée à la manière des langages textuels intégrés dans *Max*.

Pour différentes raisons, aucun des langages déjà évoqués n'est vraiment adapté à la tâche. Les APIs de beaucoup d'entre eux demandent un contrôle trop bas niveau de leur relation avec *Max*, et obligent donc l'utilisateur à s'occuper de détails non pertinents vis à vis de l'objectif algorithmique originel : il faut aussi répondre à chaque message que l'objet peut recevoir, gérer l'ordre de sortie des résultats, etc. Ces détails sont bien sûr utiles, par exemple pour contrôler finement la séquence des calculs à travers le patch, mais obscurcissent l'expression du processus computationnel à réaliser. Un autre problème, qu'on retrouve dans la majorité des langages intégrés dans *Max*, est la difficulté d'y implémenter la *llll* en tant que type de donnée, de manière à la fois complète et cohérente avec la syntaxe de *bach*, et de manière suffisamment flexible pour qu'on puisse la manipuler avec la même expressivité, souplesse et concision que les objets dédiés de la bibliothèque. La seule exception pourrait être Lisp, mais les différences entre la *llll* et la liste de Lisp sont plus profondes qu'il apparait à première vue (par exemple, il serait intrinsèquement impossible d'accéder directement aux *cons cells* d'une *llll* comme on le fait en Lisp) et une traduction des traitements entre les deux types aurait été extrêmement lourde et problématique.

Au final, nous avons choisi d'implémenter un nouveau mini-langage de programmation, dédié spécifique-

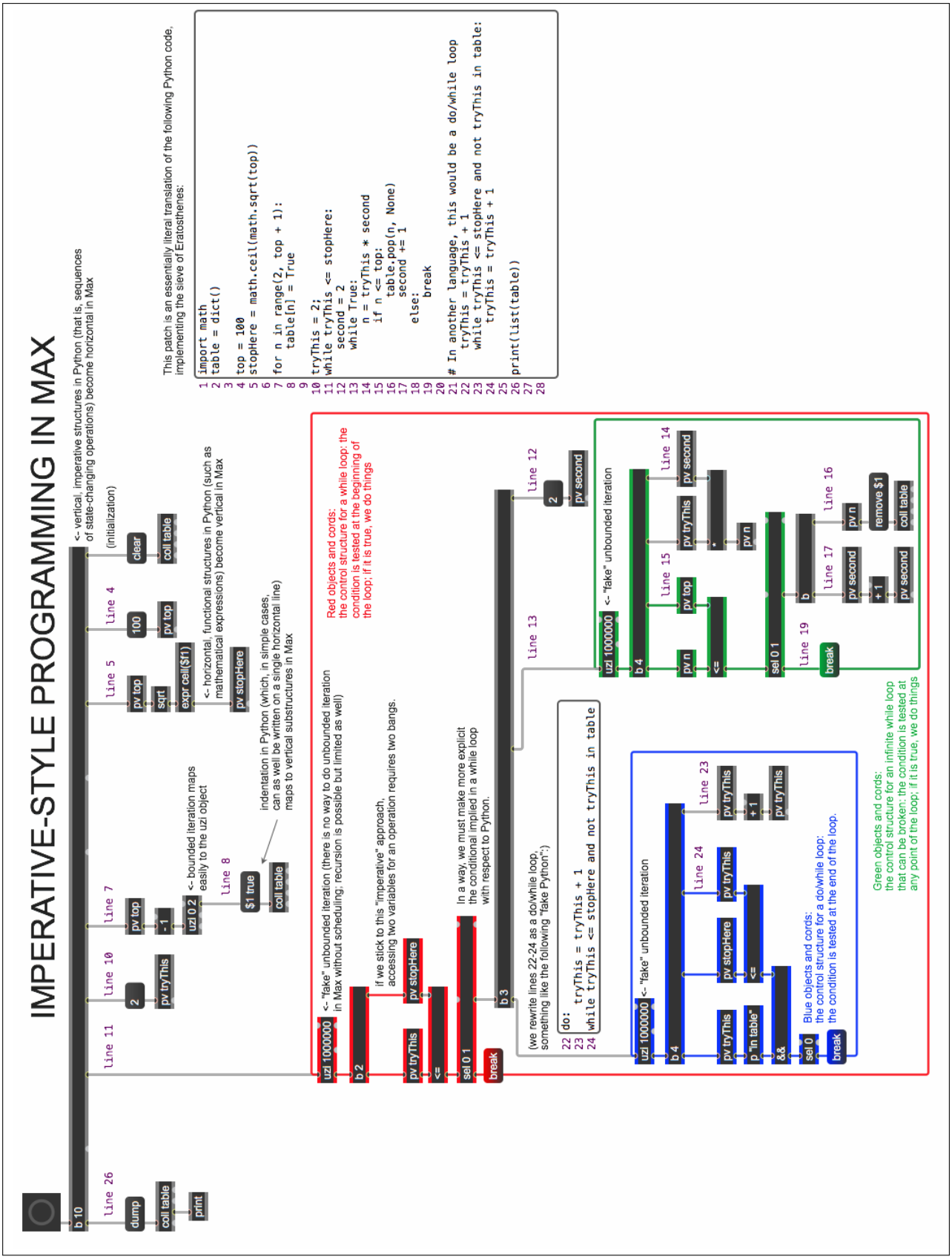

Figure 2. Une implémentation en style impératif « étiqueté » du crible d'Eratosthène, un algorithme pour calculer la suite des nombres premiers. Le patch est une traduction littérale du code Python figuré dans l'encadré. Nous avons choisi de ne pas écrire du code Python très idiomatique et concis pour le rendre plus proche d'un pseudocode facile à lire et ré-implémenter dans d'autres langages. Dans le patch, nous avons évité entre autres tout stockage de données dans des entrées froides des objets : toutes les données qui doivent être réutilisées sont mémorisées dans des objets `pv`, d'ou elles sont récupérées au moment nécéssaire. Le résultat est plutôt redondant, mais la structure des calculs est très claire.

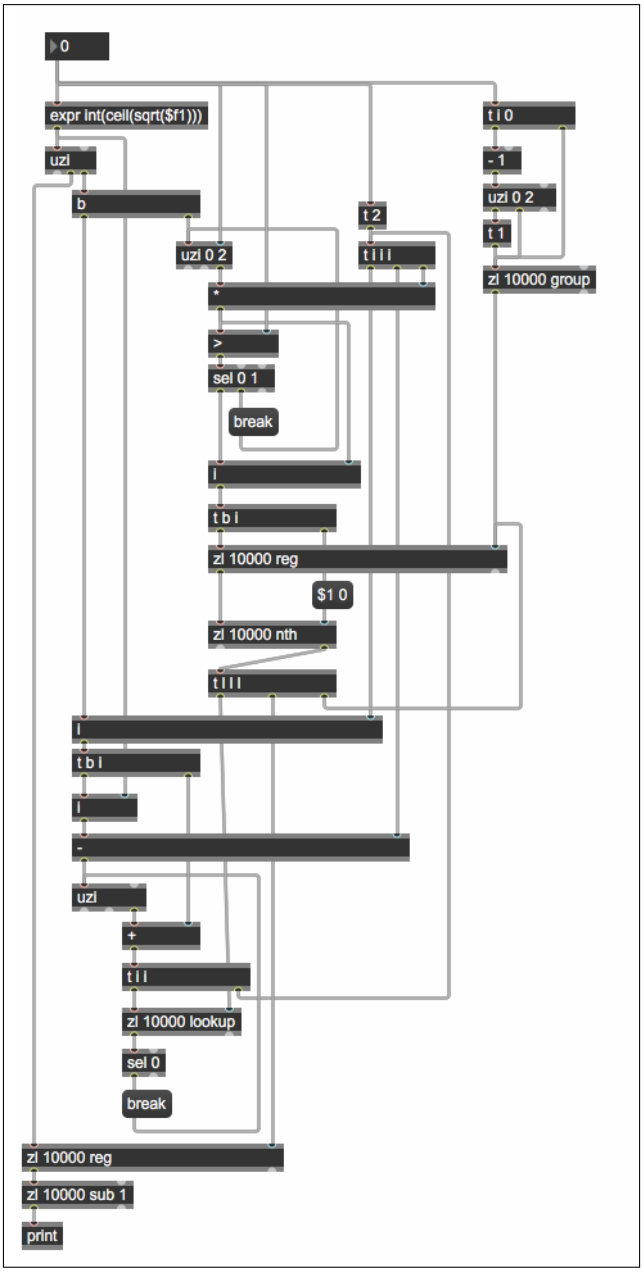

Figure 3. Une implémentation plus synthétique du crible d'Eratosthène. Par rapport à l'exemple de la figure 2, le patch est plus compact mais aussi beaucoup moins lisible.

ment aux *llll* et à leurs manipulations dans les applications musicales, et cohérent autant que possible avec la syntaxe, la nomenclature et les conventions des objets de *bach* et de *Max*.

4. LA FAMILLE EXPR

La première considération qui a guidé la définition des caractéristiques fondamentales du nouveau langage est l'existence dans *Max* d'une petite famille d'objets qui, sans implémenter un véritable langage de programmation, permet pourtant la définition d'expressions mathématiques et de conditionnelles avec un certain dégré de souplesse. Notamment, `expr` implémente des expressions mathématiques sur des scalaires ; `vexpr` en fait de même pour les listes ; et `if` implémente des conditionnelles avec une syntaxe très proche des précédentes. La bibliothèque *bach* implémente un objet nommé `bach.expr`, qui est un équivalent de `vexpr` capable d'opérer sur des *llll*s plutôt que sur des listes *Max*.

Tous ces objets, qui dans de nombreuses occasions permettent de simplifier considérablement l'expression de petits processus de calcul, partagent une syntaxe simple fondée sur la notation infixe et intègrent les opérateurs et les fonctions arithmétiques habituels. Par exemple, une expression évaluable par l'objet *expr* est

```
pow(2, ($f1 - 69) / 12) * 440.
```

où `$f1` indique une valeur flottante reçue de la première entrée de l'objet.

Dans la réalisation des tâches simples pour lesquels ils ont été conçus, ces objets présentent certains avantages importants par rapport aux « vrais » langages évoqués plus haut :

— Le code peut être spécifié directement comme paramètre de l'objet : cela montre de manière très claire le rôle de l'objet dans le patch.
— Les entrées et les sorties sont gérées automatiquement par l'objet lui-même, sans besoin d'en définir le comportement de manière détaillée.
— Il n'y a pas besoin de définir explicitement les méthodes correspondant aux différents messages que l'objet peut recevoir.
— Au final, le code exprime seulement le noyau du calcul attendu de l'objet, sans besoin de gérer les interactions avec *Max* et le reste du patch.

Cette approche permet donc une intégration immédiate d'un langage textuel dans l'environnement hôte : dans leur cas d'usage typique, plusieurs objets de la famille de `expr` sont présents dans le patch et sont en charge de petites tâches computationnelles en interaction avec d'autres objets qui gèrent l'interface, la communication MIDI, le lancement d'événements dans le temps musical, le traitement du signal audio, etc.

5. PRINCIPES DE CONCEPTION

L'approche esquissée ci-dessus a été choisie afin d'intégrer un nouveau langage, appelé *bell* pour « bach evaluation language for *llll*s », mais aussi comme hommage aux Bell Labs, où une partie importante de ce qu'aujourd'hui nous appelons l'informatique musicale a été conçue et développée. *bell* répond à quatre principes de conception posés *a priori* et décrits ci-dessous.

5.1. Intégration à *Max*

bell est implémenté à travers un nouvel objet appelé `bach.eval`. On peut spécifier du code :

— directement dans l'instance d'un objet de `bach.eval`, comme pour `expr` et sa famille ;
— via un éditeur externe ;
— en chargeant un fichier de texte contenant le code ;

— ou encore à travers un message dédié.

De plus, on doit pouvoir utiliser *bell* pour paramétrer le comportement d'autres objets *bach*, notamment certains objets qui implémentent un patron de conception appelé *lambda-loop*. Dans ce patron de conception, certaines des valeurs produites par le fonctionnement de l'objet sont transformées par une sous-partie du patch et réinjectées en entrée de l'objet. Un exemple est l'intersection de deux listes de notes MIDI paramétrée par une fonction de comparaison (par exemple la comparaison des classes d'hauteurs). Dans un langage fonctionnel tel que Lisp, on pourrait paramétrer la fonction réalisant l'intersection par une fonction anonyme (ou lambda) ; dans *Max* on peut le faire avec une *lambda-loop* de l'objet `bach.intersection`. Avec *bell*, on peut paramétrer l'objet `bach.intersection` par une expression *bell* spécifiée à travers l'attribut *lambda* (voir figure 4).

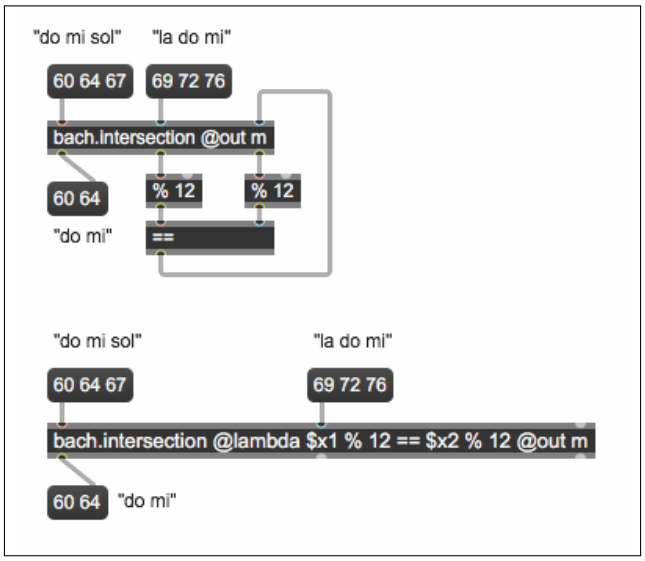

Figure 4. L'intersection de deux listes par classes d'hauteur, implementée avec une lambda-loop et du code *bell*. L'attribut *out m* indique que la *llll* calculée doit être sortie dans un format lisible pour les objets *Max*, au lieu du format de communication interne des objets *bach* (pour plus de détails, voir [3]).

5.2. Puissance expressive et rétro-compatibilité

bell est un langage simple mais Turing-complet, avec des mécanismes de variables impératives, et des structures de contrôle comme les itérations, les conditionnelles et des fonctions d'ordre supérieur définies par l'utilisateur. Dans le même temps, il doit rester compatible avec la syntaxe de `bach.expr`, `vexpr` et `expr`, dans le sens où toute expression acceptée par ces objets doit être aussi légale dans *bell* et donner le même résultat, de manière à ce que son apprentissage et l'implémentation de processus simples soient aisés pour un utilisateur déjà habitué à *Max* et *bach*.

5.3. Styles de programmation

Puisque, comme dans la famille `expr`, les programmes *bell* sont des expressions, le langage doit être applicatif (*i.e.*, permettre un style de programmation fonctionnel) : toutes les constructions du langage correspondent à une expression dont l'évaluation (exécution) retourne une valeur. Par souci de commodité et d'efficacité, un style impératif est possible, d'où la nécessité d'avoir, par exemple, des variables mutables. L'exécution du programme est déclenchée par l'arrivée de données en entrée et produit un résultat qui est restitué par la sortie principale de `bach.eval` (même s'il est aussi possible produire des données sur d'autres sorties de l'objet).

5.4. Algèbre des *llll*s

L'objectif de *bell* est la manipulation des *llll*s, qui constituent son seul type de donnée, au moins dans les premières versions. Puisque *bach* contient déjà beaucoup de primitives pour le traitement des *llll*s, qui sont exposées comme objets utilisables dans le patch, *bell* expose ces même primitives comme fonctions et opérateurs prédéfinis dans le langage, en suivant autant que possible les conventions déjà établies dans les objets correspondants.

6. LE LANGAGE *BELL*

L'implémentation du langage *bell* est fondée sur les principes définis dans la section précédente. Une description relativement détaillée et formelle de la syntaxe du langage a déjà été présentée dans [7], et une spécification complète sera prochainement intégrée dans la documentation. En tout cas, la syntaxe et le modèle sémantique du langage sont plutôt traditionnels, et beaucoup de ses concepts et structures sont tirés de manière directe d'autres langages établis comme C, Common Lisp ou Python. Comme on le verra, la caractéristique plus spécifique de *bell*, qui l'éloigne un peu de ses modèles, est la manière très directe d'exprimer la construction des *llll*s et la concaténation d'éléments. On se limitera donc ici à un aperçu des caractéristiques principales avec quelques exemples illustratifs.

6.1. Expressions élémentaires et opérateurs

De prime abord, *bell* permet d'évaluer simplement des expressions mathématiques. L'expression donnée dans la section 4 comme exemple pour `expr` est tout à fait valide en *bell*. De plus, *bell* peut manipuler deux types introduits en Max par le package *bach*, les rationnels et les *pitches* dédiés à la représentation des hauteurs musicales avec préservation de l'enharmonicité [4] et à leurs opérations. Ainsi, une autre expression valide en *bell* est par exemple

```
$p1 + D0
```

où `$p1` signifie un pitch reçu via la première entrée de l'objet, et `D0` est un littéral de type pitch qui représente à la fois

un ré très grave et un intervalle de seconde majeure. Dans ce contexte, l'opérateur `+` produit une transposition.

Comme évoqué plus haut, *bell* contient un certain nombre de fonctions de manipulation des *llls* qui s'ajoutent aux fonctions mathématiques acceptées par `expr` et `bach.expr`. Une autre exemple d'expression valide est donc

```
length($x1) * 2
```

où `$x1` indique une donnée quelconque (et donc typiquement une *llll*) reçue par la première entrée et `length` une opération prédéfinie qui correspond à l'objet `bach.length` et calcule le cardinal d'une *llll*.

Il y a aussi des opérateurs spécialisés pour le traitement des listes, comme l'opérateur infixe « `:` » qui récupère un élément d'une *llll* selon sa position et « `:*` » qui produit une *llll* constituée de la répétition de son premier argument :

```
$x1 : 2
```

produira `b` en sortie si la *llll* `a b c` est la valeur sur la première entrée de l'objet ; et l'expression

```
$x1 :* 2
```

produira la *llll* `a b c a b c`.

6.2. Littéraux et concaténation implicite

Un aspect très important de *bell* est le fait que la concaténation des éléments d'une *llll* se fait syntaxiquement de manière implicite, en juxtaposant simplement ses éléments. Il est donc possible d'écrire directement des littéraux de type *llll* dans des expressions arbitraires, sans besoin d'aucun formalisme supplémentaire :

```
1 2 3 4
```

est juste une expression dont la valeur résultante est la *llll* `1 2 3 4`.

Une *llll* peut mêler des éléments de tous les types élémentaires (entiers, rationnels, flottants, pitches et symboles), ainsi que, de manière récursive, des sous-*llls* délimitées par des crochets :

```
1 2/3 [ 4.5 A7 'huit' ] 9
```

La définition d'une *llll* est une expression qui peut impliquer des sous-expressions pour définir ses éléments, par exemple

```
1 2 $x1 * 3 4
```

s'évalue en `1 2 30 60 90 4` si `$x1` dénote la *llll* `10 20 30` en entrée. On peut au passage remarquer que les opérations arithmétiques sont *listables*, c'est-à-dire qu'elles sont automatiquement étendues aux *llll* en s'appliquant élément par élément, et que leur priorité est supérieure à celle de la concaténation implicite.

Il est possible de forcer la priorité de l'évaluation en utilisant les parenthèses :

```
1 2 $x1 * (3 4)
```

s'évalue en `1 2 30 80` pour la *llll* `10 20` en entrée : la sous-expression `(3 4)` s'évalue en une *llll* de longueur deux et la multiplication de deux *llll* de même longueur se fait élément par élément. Notez que l'expression

```
sqrt(4)
```

est interprétée comme une application de fonction plutôt que comme la concaténation de la fonction `sqrt` à l'entier parenthésé `4` parce qu'il n'y a pas d'espace entre les deux sous-expressions. L'expression

```
sqrt (4)
```

serait en revanche interprété comme une concaténation conduisant à une *llll* avec deux éléments.

6.3. Variables, séquences et assignation

bell implémente plusieurs types de variables : *locales* (dont la visibilité est limitée à la fonction à l'intérieur de laquelle elles sont définies), *globales* (partagées parmi toutes les fonctions de tous les objets à l'intérieur de tous les patches ouverts) et *relatives* au patch (similaires aux globales, mais visibles seulement des objets contenus dans la même hiérarchie de patches).

Les variables sont généralement utiles dans un contexte où des opérations peuvent être mises en séquence, ce qui est possible à travers l'opérateur « `;` » qui évalue séquentiellement deux expressions et retourne la valeur de la seconde. Chaque variable réfère à une *llll* : il n'y a pas, en *bell*, le concept de données scalaires. Les valeurs scalaires sont gérées dans *bell* comme des *llll* singletons. Il faut noter que les *llll* sont toujours passées par valeur, et donc il n'y a jamais besoin d'une opération explicite de copie.

Le type plus simple d'affectation est obtenue par l'opérateur `=` qui s'évalue en la valeur assignée. Il est par ailleurs possible d'utiliser des opérateurs d'assignation tels que `+=`, `-=`, etc. L'expression :

```
$a = 1 2 3 ; $a += 4 ; $a * 5
```

assigne la *llll* `1 2 3` à la variable `$a` (une variable locale, marquée par le préfixe `$`) ; puis on y additionne `4`, et donc la valeur de la variable devient `5 6 7` ; enfin, la valeur de la variable est multipliée par `5`, et le programme produit `25 30 35`.

On peut utiliser l'opérateur « `:` » en partie gauche d'une affectation pour remplacer un élément d'une liste. Après l'exécution de

```
$a = 10 20 30 ; $a:2 = 200
```

la valeur de `$a` est `10 200 30`.

Les variables globales et relatives au patch sont utilisées de la même manière que les variables locales, mais elles sont visibles en dehors de la fonction (et donc de l'objet) dans laquelle elles sont définies. Ces différents types de variable se distinguent syntaxiquement : les globales n'ont pas de préfixe (ex. `MyValue`), les relatives au patch sont marquées par un dièse (ex. `#myPatcherValue`) et les locales débutent par un dollar (ex. `$x`). Il est recommandé de faire débuter les variables globales introduites par l'utilisateur par une première lettre majuscule, afin de ne pas risquer de collision avec les futures fonctions intégrées qui seront rajoutées dans les versions ultérieures du langage.

Il est possible de déclencher automatiquement l'évaluation du programme lié à un objet `bach.eval` sur le

changement d'une ou plusieurs variables globales ou relatives, ce qui permet de gérer simplement, et de manière réactive, des paramètres globaux ou liés à un patch.

6.4. Conditionnels et itérations

La principale structure conditionnelle est la construction `if ... else ...` qui retourne la valeur de la branche effectivement évaluée suivant une condition logique :

```
if $x1 < 10 then -$x1 else 2*$x1
```

a pour valeur `-5` si la valeur en entrée est `5`, et `30` si la valeur en entrée est `15`.

Il y a deux types d'itérations dans *bell*. La construction `for` exécute une itération sur une ou plusieurs *llll*s. Parmi ses nombreuses variantes, on peut citer la version `for ... do`, qui retourne la valeur de la dernière itération de la clause `do`, et la version `for ... collect` qui construit une *llll* avec la valeur après chaque itération :

```
$sum = 0 ; for $i in 1 2 3 do $sum += $i
```

s'évalue en `6`, alors que

```
$sum = 0 ; for $i in 1 2 3
           collect $sum += $i
```

retourne `1 3 6`.

La construction `while` correspond à une répétition conditionné par un prédicat arbitraire. Elle est donc potentiellement non bornée. Comme pour `for`, deux versions existent, `while ... do` et `while ... collect`. Par exemple :

```
$n = 1 ; while $n < 100 collect $n *= 2
```

calcule `2 4 8 16 32 64 128`.

6.5. Fonctions

bell prédéfinit un certain nombre de fonctions dédiées au calcul mathématique et à la manipulation des listes. Lors de l'appel à ces fonctions, les arguments peuvent être définis par position ou par nom :

```
$a = pow(2, 3)
```

est équivalent à

```
$a = pow(@exponent 3, @base 2)
```

les noms des deux paramètres de la fonction `pow` étant `@base` et `@exponent`.

On peut mixer les arguments passés par position et ceux passés par nom, à condition que ceux passés par position précèdent ceux passés par nom. De plus, tous les arguments ont une valeur par défaut, et donc en principe tous sont optionnel. Ce mécanisme est très utile : de nombreuses fonctions prédéfinies impliquent un grand nombre d'arguments et il est ainsi possible de n'en spécifier que quelques uns, les autres prenant leurs valeurs par défaut. Le choix de l'arobase `@` comme marqueur des arguments passés par nom est inspiré de la syntaxe des attributs des objets *Max*.

L'utilisateur peut définir des fonctions additionnelles, qui peuvent être appelées avec la même syntaxe que pour les fonctions prédéfinies. Voici un exemple de définition de fonction avec deux paramètres :

```
$note, $middleA = 440 ->
     pow(2, ($note - 69) / 12) * $middleA
```

Les arguments sont traités comme des variables locales dans le code de la fonction. On notera que l'argument `$middleA` est spécifié avec une valeur par défaut de `440`. Lorsqu'il n'y a pas de défaut spécifiée, comme c'est le cas pour `$note`, la convention est d'utiliser la *llll* vide. Les noms des deux arguments constituent aussi les noms de ses paramètres (en remplaçant `$` par `@`).

Les fonctions définies par l'utilisateur en *bell* sont anonymes (elles ne sont pas nommées) et ce sont des valeurs *banalisées* (*first class citizens*). Pour référer à une fonction après sa définition, il faut l'assigner à une variable, qui donc contiendra une *llll* dont le seul élément sera de type fonction – quoique rien n'empêche à une fonction d'être un élément d'une *llll* avec d'autres éléments. Ainsi l'expression

```
$cbrt = $x -> pow($x, 1/3) ; $cbrt(27)
```

s'évalue en `3`.

Il est souvent utile d'assigner une fonction à une variable globale de manière à la rendre disponible pour tous les programmes *bell* dans le système. Ces assignations sont effectuées quand le code qui les contient est exécuté. Afin de simplifier la gestion des dépendances, et de s'assurer que les fonctions globales sont accessibles dès leur chargement, `bach.eval` offre un attribut *auto* qui permet une évaluation du code *bell* associé dès qu'il est disponible.

6.6. Gestion des entrées et des sorties de l'objet

Un programme en *bell* peut très simplement référer aux valeurs de plusieurs entrées à travers les pseudo-variables d'entrée `$x`<*n*> (où <*n*> est un entier positif qui représente la n^{e} entrée) et leurs versions typées : `$i`<*n*> pour les entiers, `$f`<*n*> pour les flottants, `$r`<*n*> pour les rationnels et `$p`<*n*> pour les pitches. Ces versions typées ne sont en général pas très utiles, mais elles sont présentes pour la compatibilité avec `expr` et elles participent à la lisibilité du programme.

La gestion de sorties multiples est plus complexe. Chaque programme *bell* calcule une valeur disponible sur l'*outlet* principal de l'objet. Il est possible de déclarer d'autres sorties de l'objet et d'y envoyer des données à travers des pseudo-variables `$o`<*n*>. Cela est utile, par exemple, en conjonction avec des conditionnelles :

```
if length($x1) < 10 then $o1 = $x1
                    else $o2 = rev($x1)
```

Si ce code paramètre un objet `bach.eval`, le système génère automatiquement deux sorties supplémentaires à la gauche de la sortie principale ; si le code est spécifié autrement, il faut déclarer explicitement le nombre souhaité de sorties à travers un attribut de l'objet. Dans les deux cas, l'évaluation produit une valeur envoyée sur la sortie principale de l'objet (celle la plus à droite). Les valeurs des pseudo-variables `$o`<*n*> sont affectées aux sorties auxiliaires, de droite à gauche dans l'ordre des in-

dices n décroissants, selon une convention qui peut sembler contraire à l'intuition, mais qui est cohérent avec le comportement de plusieurs autres objets Max.

Il est aussi possible de sortir des données pendant l'évaluation, à travers des pseudo-variables, les *direct outlets*, dénotées par `$do<n>`. Lorsqu'on assigne une valeur à une de ces pseudo-variables, cette valeur est envoyée directement à la sortie correspondante. Les sorties directes sont distinctes des sorties auxiliaires décrites plus haut, et se trouvent à la droite de la sortie principale. Comme pour les sorties auxiliaires, elles sont générées automatiquement si le code est spécifié comme paramètre d'instantiation de l'objet, et doivent être déclarées explicitement dans les autres cas.

7. DÉTAILS SUR L'IMPLÉMENTATION

7.1. Mécanisme d'interprétation

Quand l'objet `bach.eval` reçoit un programme en *bell*, il le traduit dans un *arbre de syntaxe abstrait* grâce à un analyseur syntaxique écrit en C++ à l'aide des outils *Flex* (analyse lexicographique) et *Bison* (analyse grammaticale). Il y a quelques catégories d'événements qui peuvent déclencher l'évaluation de ce code :

— l'arrivée d'un message dans une des entrées « chaudes » de `bach.eval` (normalement celle la plus à gauche, mais on peut le spécifier à travers l'attribut *triggers*) ;

— la mise à jour d'une variable globale spécifiée à traver l'attribut *triggers* de l'objet `bach.eval` ;

— immédiatement après le chargement du code dans `bach.eval`, si l'attribut *auto* est à 1 (ce qui est particulièrement utile pour le nommage des fonctions globales et le traitement des paramètres de configuration) ;

— l'évaluation de l'expression *lambda* dans les objets autres que `bach.eval` (une activation de l'objet peut entraîner plusieurs évaluation de l'expression *lambda* si cette expression est utilisée plusieurs fois en interne).

7.2. Efficacité

L'interprète de *bell* n'est actuellement pas particulièrement optimisé. Par exemple, il exécute beaucoup plus de traversées de l'arbre syntaxique que nécessaire, et il utilise la pile d'exécution du système pour gérer les appels de ses propres fonctions et opérateurs. Cela pose une limitation sur la profondeur de la récursion, et il est facile d'atteindre une profondeur importante avec des opérations apparemment innocentes : par exemple chaque concaténation de deux éléments d'une liste est gérée comme une opération indépendante enchaînée aux autres, et donc le littéral

```
1 2 3 4 5
```

causera déjà une pile de quatre appels, un pour chaque concaténation entre éléments consécutifs.

De plus, on utilise très souvent des *llll*s d'un seul élément, pour représenter les scalaires, ce qui est une représentation inutilement lourde qui implique des opérations d'encodage et de décodage à chaque opération.

Cependant, en général, un programme *bell* sera remarquablement plus efficace que le patch *Max* correspondant s'il utilise les *llll*s de manière étendue. Ce point doit être tempéré : un patch *Max* « pur » (c'est-à-dire, sans objets et types de données de *bach*) sera normalement beaucoup plus efficace que l'implémentation du même processus en *bell*, à cause du coût inhérent à la manipulation des *llll*s. D'ailleurs, *Max* n'est pas vraiment adapté pour les types d'opérations pour lesquelles *bach* et *bell* ont été conçus, et donc la comparaison n'est pas toujours pertinente.

Pour référence, nous avons effectué un petit *benchmark* en comparant la performance du même algorithme de calcul de fondamentales virtuelles, implémenté de manière raisonnablement idiomatique dans les langages suivants :

— en *Max*, à l'aide de la bibliothèque *bach*, mais sans utiliser *bell* ;

— en *bell*, à l'intérieur de l'environnement *Max* ;

— en OpenMusic, sans écrire du code textuel Lisp ;

— en Python, sans utiliser aucune bibliothèque optionnelle telle que *Numpy*.

Les temps d'exécution obtenus sur la même machine à l'issue de 1000 répétitions du programme, sont rapportés dans la table 1.

Max et *bach* (sans *bell*)	3006 ms
bell	1614 ms
OpenMusic	10243 ms
Python	529 ms

Table 1. Comparatif de temps d'exécution selon 4 modes d'implémentation (pour 1000 exécutions).

Ces résultats montrent que les performances de *bell*, dans cette première implémentation, sont évidemment moindres que dans un langage interprété mais très optimisé comme Python, mais remarquablement supérieures à celles qui peuvent être obtenues dans des environnements graphiques comme OpenMusic et *Max* (en utilisant la bibliothèque *bach*, sans laquelle d'ailleurs l'implémentation d'un processus de manipulation de listes tel que celui proposé aurait été très compliquée).

8. CONCLUSIONS ET PERSPECTIVES

Nous avons présenté *bell*, un petit langage textuel conçu pour être intégré dans l'environnement *Max* et la bibliothèque *bach*, en détaillant les motivations et le contexte au fondement de ce travail. Du point de vue des mécanismes d'évaluation, *bell* se présente comme un langage fonctionnel avec des traits impératifs, avec un modèle d'exécution classique. Le caractère spécifique de *bell*, outre son intégration harmonieuse dans *Max*, est la manipulation des *llll*s. Une étude formelle de cette structure de données est encore à effectuer, dans la lignée des algèbres de liste développées par Bird et Merteeens [8].

La première version de *bell* est intégrée dans la distribution courante de *bach* (version 0.8.1), qui peut être obtenue en version Windows ou Mac via le *Package Manager* de *Max*, sous licence GPL v3. La réception par la communauté d'utilisateurs a été très positive et encourageante, quoiqu'il soit difficile d'évaluer le nombre de personnes qui ont effectivement adopté le langage dans leur flux de travail.

Au delà de quelques rapports de bogues, qui en majorité ont déjà été résolus, la remarque la plus importante qui a été faite par les utilisateurs concerne la documentation : à l'heure actuelle, toute la syntaxe de *bell* est décrite dans les fichiers d'aide des objets qui l'accompagnent, mais il s'agit d'une documentation très concise et peu pédagogique, surtout en considérant que l'utilisateur typique de *bell* n'est pas vraiment un programmeur dans le sens traditionnel. L'écriture d'un manuel de programmation visant à introduire les concepts et la syntaxe du langage de manière plus claire et approfondie est envisagée en parallèle avec un répertoire d'exemples pratiques.

Il faudrait aussi augmenter le nombre des fonctions prédéfinies, de manière à rendre plus immédiates beaucoup d'opérations typiques qui disposent déjà de modules dédiés dans la bibliothèque *bach*.

Il apparaît nécessaire d'adresser de manière systématique les points évoqués plus haut : rendre plus efficace l'implémentation des successions de concaténation, éliminer la dépendance à la pile du système et implémenter un type de donnée interne pour représenter plus efficacement les *singletons*, tout en cachant cette distinction à l'utilisateur.

Pour finir, nous envisageons de paramétrer beaucoup d'autres objets *bach* par des expressions en *bell*, et notamment les deux éditeurs de partition `bach.roll` et `bach.score`. Une meilleure prise en compte de ces objets entraînera une évolution importante du langage, dont les lignes générales sont déjà tracées, en introduisant par exemple une extension de la syntaxe vers une notation orientée objet.

REMERCIEMENTS

Le langage *bell* a été développé par Andrea Agostini dans le cadre d'une résidence *IRC* à l'IRCAM, avec l'aide de Jean-Louis Giavitto pour certains aspects théoriques mais aussi syntaxiques. L'intégration dans *bach* a été entreprise dans le cadre de la bibliothèque développée par Andrea Agostini et Daniele Ghisi.

Nous tenons à remercier Emmanuel Jourdan, Paola Palumbo, Arshia Cont et Greg Beller, qui ont soutenu le projet dès le début et sans l'aide desquels nous ne n'aurions pu le réaliser.

9. RÉFÉRENCES

[1] Agon, C. « OpenMusic : Un langage visuel pour la composition musicale assistée par ordinateur », thèse de doctorat, sous la dir. de J.-F. Perrot, Université Paris 6, 1998.

[2] Agostini, A., Daubresse, E., Ghisi, D. « *cage*: a High-Level Library for Real-Time Computer-Aided Composition ». Proceedings of the International Computer Music Conference, Athens, Greece, 2014.

[3] Agostini, A., Ghisi, D. « A Max Library for Musical Notation and Computer-Aided Composition ». *Computer Music Journal* 39/2 (2015), p. 11-27.

[4] Agostini, A., Ghisi, D. « Pitches in bach », Proceedings of the International Conference on Technologies for Music Notation and Representation, Montréal, Canada, 2018.

[5] Agostini, A., Ghisi, D., Giavitto, J.-L. « Programming in style with *bach* », Proceedings of the International Symposium on Computer Music Multidisciplinary Research, Marseille, France, 2019.

[6] Agostini, A., Ghisi, D., Giavitto, J.-L. Programming in style with bach, *Perception, Representations, Image, Sound, Music*. Springer, en cours d'édition.

[7] Agostini, A., Giavitto, J.-L. « *bell*, a textual language for the bach library » Proceedings of the International Computer Music Conference, New York, États-Unis, 2019.

[8] Backhouse, R. C. *An exploration of the Bird-Meertens formalism*. University of Groningen, Department of Mathematics and Computing Science, 1988.

[9] Cont, A.. « ANTESCOFO: Anticipatory Synchronization and Control of Interactive Parameters in Computer Music. » Proceedings of the International Computer Music ConferenceBelfast, Ireland, Aug. 2008.

[10] Desain, P., Honing, H. Putting Max in Perspective. *Computer Music Journal* 17/2 (1992), p. 3-11.

[11] Didkovsky, N., Hajdu, G. « Maxscore: Music Notation in Max/MSP ». Proceedings of the International Computer Music ConferenceBelfast, Ireland, Aug. 2008.

[12] Fineberg, J. « Esquisse - library-reference manual (code de Tristan Murail, J. Duthen and C. Rueda) », 1993.

[13] Garton, B. , Topper, D. « Rtcmix-using cmix in real time », Proceedings of the International Computer Music Conference, Tessalonique, Grèce, 1997.

[14] Kahn, G. « The semantics of a simple language for parallel programming », Proceedings of the International Federation for Information Processing Congress, Stockholm, Suède, 1974.

[15] MacCallum, J., Gottfried, R., Rostovtsev, I., Bresson, J., Freed, A. « Dynamic message-oriented middleware with open sound control and odot », Proceedings of the International Computer Music Conference, Denton, États-Unis, 2015.

[16] Orlarey, Y., Fober, D., Letz, S. « Faust: an efficient functional approach to dsp programming. » *New Computational Paradigms for Computer Music*, Assayag, G., Gerzso, A. (dir.), Delatour, Paris, 2009, p. 65-96.

[17] Puckette, M. « Max at seventeen », *Computer Music Journal* 26/4 (2002), p. 31-43.

[18] Steele, G. L. Jr. « Macaroni is better than spaghetti » *ACM SIGPLAN Notices* 12/8 (1977), p. 60-66.

Texte édité par Corentin Guichaoua.