De la webradio lambda à la λ-webradio

D. Baelde[1] & S. Mimram[2]

1: INRIA & LIX, École Polytechnique
david.baelde@ens-lyon.org
2: Équipe PPS, CNRS / Université Paris 7
Case 7014, 75205 Paris cedex 13
samuel.mimram@ens-lyon.org

Résumé

La génération et la manipulation de flux audio – pour une radio *web* par exemple – est une tâche complexe, difficilement réalisable à l'aide des langages de programmation habituels. Nous présentons dans cet article un langage fonctionnel fortement typé appelé Liquidsoap qui offre des abstractions confortables pour décrire la construction de flux élaborés. Il se démarque par sa souplesse d'utilisation et la richesse des possibilités qu'il offre : de l'utilisation de divers types d'entrées (fichiers audio, micro, requêtes d'utilisateurs) que l'on peut sélectionner dynamiquement (selon la disponibilité ou encore l'horaire) à la gestion des transitions entre morceaux et autres traitements audio. La nécessité d'avoir un langage riche et abordable nous a amenés à introduire une variante du λ-calcul typé, avec étiquettes et arguments optionnels, dont la portée va au delà du domaine du traitement audio.

Avec l'avènement des réseaux à haut débit, il est maintenant possible de diffuser rapidement de grandes quantités de données à travers le monde. Ainsi, de nombreuses radios *web* ont pu voir le jour et diffusent en continu divers contenus sonores par le biais d'Internet. Il n'existait cependant pas de langage simple et expressif permettant de construire les flux audio de ces radios.

Au premier abord, la génération d'un flux continu de données audio peut sembler être une tâche simple à réaliser : il suffit de lire bout à bout des fichiers audio. Cependant, sa mise en pratique se heurte rapidement à certaines difficultés.

- Ces difficultés sont d'abord d'ordre purement technique : les fichiers audio sont stockés sous divers formats (il faut les convertir en un format uniforme), sur divers serveurs (il faut des outils gérant les protocoles utilisés), etc.
- Ensuite, la gestion des listes de lecture, ou *playlists*, s'avère complexe : on veut pouvoir choisir un morceau correspondant à certains critères (le titre, l'artiste, le genre, etc.) dans une base de données, ces critères dépendant de l'horaire (on veut jouer de la musique douce le matin et plus dansante le soir), on veut aussi pouvoir avoir des interventions en direct lors de plages horaires réservés aux animateurs, insérer régulièrement des messages rappelant le nom de la radio (*jingles*), etc.
- Enfin, il faut pouvoir traiter le son provenant des fichiers audio afin de le rendre uniforme et agréable à l'écoute : il faut à la fois appliquer des effets audio sur le son (en particulier normaliser et compresser le son afin d'avoir un volume moyen constant), gérer l'enchaînement entre les morceaux (par exemple, appliquer un fondu enchaîné entre les chansons), éviter les blancs, etc.

Il existait déjà plusieurs logiciels qui permettent de gérer des radios. Les solutions les plus professionelles (Master Control, WinRadio, Open Radio ou encore Rivendell) sont des applications graphiques intégrant la gestion de la grille de diffusion, le contrôle des points de transitions entre fichiers, le décrochage vers une émission en direct et enfin la diffusion. D'autres applications plus spécialisées et légères, ont une interface en mode console et sont plus adaptées à un fonctionnement complètement automatisé sur un serveur. Ces derniers générateurs de flux (par exemple Ices ou

EzStream) offrent des possibilités limitées à la diffusion d'une suite de fichiers sans transitions ou d'un flux en provenance de la carte son. Ils sont cependant très utilisés dans la communauté *open-source*, notamment dans le système Mediabox 404 qui offre une interface web conviviale permettant de gérer une grille de diffusion et le décrochage vers les émissions en direct. Dans tous les cas on remarque que la génération du flux se fait selon un schéma rigide : l'ordre dans lequel les opérations sont effectuées est fixe. Ces outils ne sont par conséquent plus utilisables dès que l'on sort du cadre pour lequel ils ont été conçus.

La conception d'un langage applicatif dédié, fournissant des opérations élémentaires sur des valeurs représentant les flux, offre un cadre de développement riche ainsi qu'un moyen simple et expressif pour l'utilisateur de décrire sa configuration. Nous présentons ici le langage Liquidsoap [5], une implémentation de cette idée en OCaml [7]. Cet outil offre de larges possibilités et est d'ores et déjà utilisé avec succès en production par des webradios [1, 2] ou encore pour expérimenter de nouvelles méthodes de diffusion, par exemple fondées sur le recoupement des habitudes musicales de groupes d'auditeurs [4].

L'une des contraintes majeures qu'un tel langage doit respecter, au delà de celles induites par la manipulation de flux audio, est liée à la nature des utilisateurs potentiels : les personnes susceptibles de vouloir créer des radios sur internet sont loin d'être toutes des programmeuses chevronnées. Nous avons donc conçu le langage de sorte qu'il soit abordable et simple d'utilisation. En particulier, le grand nombre de paramètres dont peuvent dépendre les opérations du langage nous a amené à introduire un calcul avec étiquettes (ce qui permet à l'utilisateur de ne pas avoir à se souvenir de l'ordre des arguments) et arguments optionnels (permettant de spécifier des valeurs par défaut pour les arguments) ainsi qu'un système de types pour ce calcul, qui sont deux contributions originales de cet article. Ce calcul n'est pas spécifique au traitement de l'audio et peut être réutilisé dans d'autres domaines où la simplicité du langage passe avant la nécessité d'une compilation efficace, par opposition avec les calculs développés par Aït-Kaci, Garrigue et Furuse [3, 9].

Nous abordons dans cette article deux aspects fondamentaux de la conception de Liquidsoap. Nous commençons par présenter les abstractions fournies dans le langage en illustrant les possibilités qu'elles offrent et nous discutons de l'adéquation entre ces abstractions et l'implémentation. Dans un second temps, nous formalisons le calcul sous-jacent au langage de programmation ainsi qu'un système de types adapté.

1. Un langage de manipulation de flux audio

Notre présentons ici la méthodologie et les concepts généraux importants qui sous-tendent le langage. Nous nous sommes efforcés d'être synthétiques, le lecteur pourra trouver plus de détails pratiques sur le site dédié au langage [5]. Il est cependant intéressant de donner tout d'abord une rapide idée de l'ampleur du développement qui a été nécessaire pour implémenter ce langage.

Le développement de Liquidsoap a été effectué au sein d'un projet appelé Savonet qui, outre le langage de programmation Liquidsoap lui-même, contient des librairies en OCaml interfaçant des libraires C préexistantes qui permettent de décoder et d'encoder des fichiers audio aux formats Ogg/Vorbis, MP3 et AAC, d'appliquer des effets audio (greffons LADSPA pour les effets audio, samplerate pour changer la fréquence d'échantillonage, etc.), de communiquer avec les cartes son (librairies AO, ALSA et portaudio), avec des programmes externes (JACK), ou d'envoyer le flux audio à un serveur d'émission audio utilisant le protocole SHOUTcast (Icecast par exemple). D'autres programmes de Savonet permettent de créer une base de données des fichiers audio disponibles sur un réseau ou de gérer les requêtes d'utilisateurs via un site web ou un bot IRC – on peut par exemple dire « mets de la techno » sur un forum de messagerie instantanée et une chanson de techno, trouvée dans la base de donnée, sera diffusée sur la radio.

Le projet dans son ensemble comporte plus de 50 000 lignes de code dont 40 000 sont écrites en

OCaml, et 8 000 en C. Le langage Liquidsoap lui-même comporte 25 000 lignes en OCaml. Malgré le lourd travail d'interfaçage de librairies C, le choix d'OCaml semble avoir été fortement positif : l'expressivité et les capacités d'abstraction offertes par le langage nous ont permis de structurer fortement Liquidsoap, de le maintenir et de l'étendre à moindre coût. En particulier, l'utilisation intensive de la programmation objet nous a permis une grande simplicité et extensibilité de la conception des opérations sur les flux.

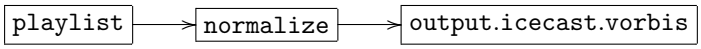
1.1. Génération de flux audio dans Liquidsoap

Un programme Liquidsoap construit des générateurs de flux audio : les *sources*. Une source peut être décrite par un graphe orienté acyclique, dont les sommets sont appelés *opérateurs*. Les sommets initiaux de ce graphe génèrent un flux à partir d'un fichier audio, d'une liste de lecture ou encore d'un microphone. Les sommets internes correspondent à des manipulations opérées sur les flux produits par les sources filles, par exemple la superposition de ces flux, ou le choix entre l'un d'entre eux en fonction de certains paramètres comme l'horaire ou encore l'application d'un effet audio. Les sommets terminaux n'ont typiquement qu'une entrée, et transmettent par exemple leur flux à un serveur de diffusion sur Internet ou à une carte son. Les opérateurs, en plus de dépendre d'autres sources décrites dans le graphe, peuvent dépendre de valeurs données en paramètres (booléens, entiers, flottants, fonctions, etc.).

À partir d'un tel graphe, décrit par un programme Liquidsoap, un flux audio est généré par blocs de données audio. Régulièrement, un bloc est décodé à partir d'une source, des opérateurs procèdent à diverses manipulations sur ce bloc, puis il est encodé dans un format compressé avant d'être transmis à un serveur qui se charge de diffuser le flux aux auditeurs. Par exemple, le programme suivant lit des morceaux dans une liste de lecture appelée `liste` puis applique une normalisation de volume et enfin envoie le flux à un serveur de diffusion :

```
l = playlist("liste")
s = normalize(l)
output.icecast.vorbis(host="www.radio.com", name="ma_radio", s)
```

Le graphe induit par ce script est :

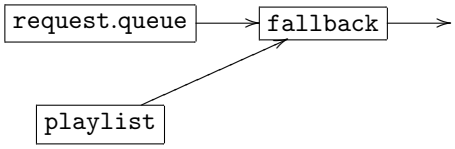


Il nous faut enrichir la notion de flux audio pour pouvoir décrire les configurations usuellement rencontrées. D'une part, certaines sources ne sont pas *disponibles* en permanence, c'est-à-dire qu'elles ne sont pas tout le temps prêtes à générer des données. Par exemple, une source jouant des requêtes d'auditeurs n'est disponible que s'il y a effectivement des requêtes. Lorsque ce n'est pas le cas, il faut pouvoir jouer à la place le flux provenant d'une autre source, typiquement une liste de lecture. Si la source de requêtes devient disponible à nouveau, on ne veut en général pas interrompre le morceau en cours mais attendre qu'il soit terminé avant de jouer la nouvelle requête. Il faut donc d'autre part introduire une notion de *piste* dans le flux qui permette de délimiter les portions faisant partie d'un même morceau.

Liquidsoap permet ainsi de décrire des opérateurs plus complexes comme l'opérateur de choix par défaut `fallback`, qui prend en argument une liste de sources et produit en sortie une piste de la première source disponible, puis à la fin de celle-ci une nouvelle piste de la première source disponible à ce nouvel instant et ainsi de suite. Il permet de réaliser ainsi notre exemple :

```
f = fallback([request.queue(), playlist("liste")])
```

Le graphe sous-jacent à ce programme est :



Le langage permet de construire des sources encore plus élaborées. Nous présentons dans la section suivante l'exemple de l'utilisation des transitions entre pistes, car elle nous semble être une bonne illustration de l'expressivité du langage et motive son caractère fonctionnel.

1.2. Les transitions

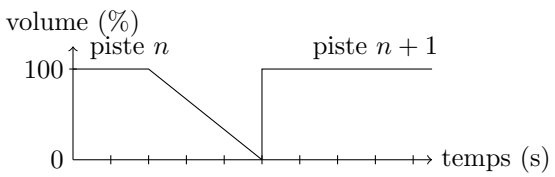
L'opération de fondu enchaîné consiste à faire varier progressivement le volume de 0% à 100% (resp. de 100% à 0%) en début (resp. en fin) de piste. Le fondu enchaîné et croisé consiste de plus à superposer une portion de la fin d'une piste avec le début de la précédente. Cet effet est communément utilisé pour rendre l'écoute plus agréable. De nombreux paramètres sont à prendre en compte : la durée du fondu en début et en fin de piste, la durée de la superposition ou le type de fondu (linéaire ou logarithmique par exemple), etc. De plus, on veut parfois adapter ces paramètres en fonction des volumes sonores, par exemple pour éviter de masquer un début de piste doux en le mixant avec une fin bruyante. Enfin, il est aussi fréquent d'ajouter un *jingle* durant la transition.

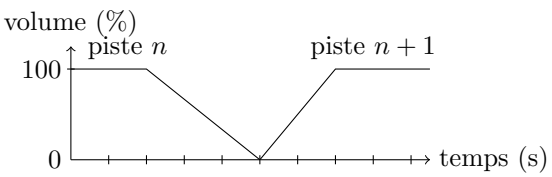
Une solution élégante et générale pour traiter tous ces cas est de décrire une transition par une fonction, qui prend en argument deux sources représentant les pistes à combiner et retourne une source qui est le résultat de la transition. Les opérations usuelles sur les flux sont alors à la disposition de l'utilisateur pour décrire sa transition.

Par exemple, le code suivant permet d'effectuer une transition simple entre deux pistes consécutives d'une source `s` :

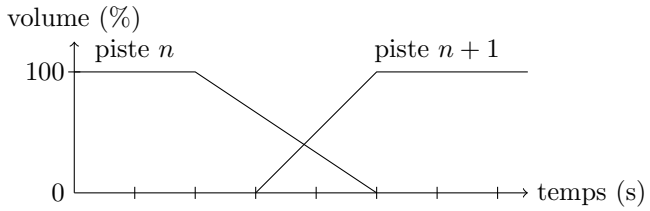
```
s = fade.out(duration=3., s)
s = fade.in(duration=2., s)
fader = fun (a,b) -> add([a,b])
cross(duration=2., fader, s)
```

Dans ce programme, l'opérateur `fade.out` applique une diminution de volume linéaire durant les trois dernières secondes d'une piste, pour arriver au volume nul ; l'opérateur `fade.in` agit similairement mais en début de piste. La seconde définition de `s` masque la première, le `=` de la syntaxe concrète étant à lire comme un `let` dont le `in` est implicite. La fonction `fader` est ensuite définie pour décrire comment va se faire la transition entre les pistes. Son premier paramètre sera un tronçon de flux à croiser en fin de piste, le second paramètre correspondra au flux commençant à la piste suivante. Ici, on se contente de superposer les deux pistes grâce à l'opérateur `add` mais on aurait aussi pu ajouter un *jingle* durant la transition en utilisant la fonction `fader = fun (a,b) -> add([a,once(jingles),b])`. Enfin, l'opérateur `cross` va jouer le flux en appliquant la transition à chaque changement de piste : à deux secondes de la fin d'une piste, la fonction de transition est utilisée pour calculer l'enchaînement des pistes. Graphiquement, après la première et la deuxième ligne du script, le volume de deux pistes consécutives sera respectivement :





Enfin, à la fin du script, après l'utilisation de l'opérateur `cross`, l'enchaînement entre pistes sera schématiquement :



L'opérateur `cross` doit fournir en même temps à sa fonction `fader` des données provenant de la même source `s` mais correspondant à des instants différents dans le flux. Il doit donc récupérer en avance et stocker les données à superposer. Si un autre opérateur `t` accédait à la source `s`, il faudrait stocker ses anciennes données pour pouvoir fournir des réponses cohérentes à `t`. Sans hypothèse simplificatrice, ceci est irréalisable : la quantité de données à stocker croît sans cesse. Dans Liquidsoap, on choisit d'exclure complètement cette possibilité : aucune des sources en amont d'un opérateur `cross` ne doit être accédée par un autre opérateur. Malheureusement, c'est encore à l'utilisateur de vérifier cette condition, en attendant une extension adéquate du système de types (cf. Section 3.2).

Pour finir, notons que cette représentation des transitions motive pleinement la conception de Liquidsoap comme un langage fonctionnel : contrairement à de nombreux langages spécialisés (*Domain Specific Languages*), la notion de fonction est essentielle dans Liquidsoap. En effet, on ne peut pas éliminer les fonctions par une première passe d'évaluation partielle, car leur exécution est étroitement liée au processus infini de génération de flux.

1.3. Granularité, efficacité et partage

Nous avons décrit un système expressif pour construire un flux, en composant des opérations élémentaires prédéfinies. Nous discutons ici de l'écart entre la notion abstraite de flux que l'utilisateur manipule, et son implémentation.

Précisons le cadre dans lequel se situe Liquidsoap, ses objectifs, sa *granularité*, par comparaison avec les *langages synchrones* [6] dans lesquels l'évaluation se fait régulièrement, et de façon théoriquement instantanée, à chaque tic d'une horloge globale. Il existe de tels langages dédiés à la synthèse audio, ChucK [10] ou StreamIt [11] par exemple. Ces langages offrent une granularité très fine : ils permettent à l'utilisateur un contrôle précis du flux, au niveau de l'échantillon. La synthèse sonore s'effectue échantillon par échantillon, en exécutant périodiquement le programme saisi par l'utilisateur. Une telle richesse n'est ni nécessaire ni souhaitable dans notre cas. Le créateur de radio *web* n'a pas besoin d'implémenter un nouvel algorithme de synthèse sonore ou un filtre innovant ; il n'en a en général d'ailleurs pas les capacités techniques et préfère en tout cas s'abstraire de ces détails de bas niveau. Liquidsoap offre donc une granularité plus grossière : le langage ne donne pas accès aux flux mais traite ceux-ci comme des objets complètement abstraits. L'exécution du programme crée essentiellement une source en assemblant des boîtes noires dont le comportement est programmé en OCaml et non pas dans le langage Liquidsoap. La synthèse du flux audio n'a lieu que dans un second temps, une fois la source créée, et n'implique (presque) plus l'évaluation du programme Liquidsoap saisi par l'utilisateur.

La simplicité d'utilisation n'est cependant pas l'unique motivation de la conception de Liquidsoap qui est aussi guidée par une recherche d'efficacité. En effet, la majeure partie des calculs est ainsi effectuée en OCaml qui est un langage efficacement compilé, mais surtout le niveau d'abstraction permet un traitement du flux optimisé en interne. Au lieu de calculer le flux échantillon après échantillon, les opérations définies en OCaml vont pouvoir travailler directement avec des blocs d'échantillons, c'est-à-dire un tableau d'échantillons consécutifs de taille fixe. Ceci évite de nombreuses copies de données et est crucial pour les performances : par exemple, en passant d'une taille de bloc

de 1 à 10 échantillons on constate une accélération d'un facteur 5, puis encore de 2 en passant de 10 à 100.

Cependant, cette optimisation des copies a une conséquence sur le flux calculé. Il se peut qu'à un instant donné un opérateur demande à une de ses sources filles de remplir un bloc d'échantillons à partir d'une position quelconque qui n'est pas nécessairement au début du bloc. Par exemple, un opérateur de choix par défaut `fallback` va, si l'une de ses sources se tarit, finir de remplir un bloc avec une autre source. On rappelle par ailleurs que le graphe décrivant un flux n'étant pas nécessairement un arbre, une même source S peut être *partagée* par deux opérateurs A et B. En général, il est impossible pour S de savoir au début d'un instant lequel de ces opérateurs va effectivement lui réclamer des données, car cela dépend notamment des données générées par les autres sources dans le même instant. La source S doit donc se préparer à tous les scénarios. À un instant donné, A peut lui demander de remplir un bloc, à partir du milieu. Puis, dans le même instant, B peut demander à S des données, cette fois à partir du début d'un bloc. Or, S doit produire des résultats cohérents : même un décalage d'un demi-bloc entre les flux envoyés à A et B est intolérable, par exemple dans le cas où ces flux sont ensuite combinés. La source S se voit donc contrainte de générer un bloc complet, et d'en copier la seconde moitié pour A. Si B réclame effectivement des données, on pourra les copier à partir de notre bloc complet. Sinon, on aura généré un début de bloc inutile, ce qui crée une légère différence entre le flux qui aurait été produit par un traitement échantillon par échantillon.

Afin de minimiser cet écart entre l'implémentation et l'abstraction que l'utilisateur manipule, on doit chercher à limiter les effets du partage, c'est-à-dire détecter le plus finement possible les points de partage potentiel. Par ailleurs, cela entraine une limitation des copies de blocs, donc encore une optimisation – cette fois sans effet sur le résultat. La détection du partage devient plus délicate en présence de transitions, mais nous ne la détaillerons pas ici.

2. Un langage de programmation étiqueté

L'utilisation d'un langage de programmation *fonctionnel* pour décrire les graphes de configuration apparait naturellement, les flux étant construits à l'aide d'opérateurs ayant plusieurs entrées et au plus une sortie. De plus, cela permet de décrire de façon élégante des constructions complexes comme les transitions (voir Section 1.2). Enfin, l'*application partielle* s'est aussi révélée être pratique pour factoriser le code, y compris au sein de scripts simples.

Nous avons voulu un langage *fortement* et *statiquement typé*. La garantie d'absence d'erreur à l'exécution que cela apporte nous semble d'autant plus importante ici que les scripts sont amenés à fonctionner pendant de très longues durées sans maintenance : il serait malheureux qu'une coquille dans le code d'une transition spécifique au troisième dimanche du mois provoque l'arrêt de toute la diffusion. D'autre part, le typage fournit un support utile à la documentation, comme on peut le voir pour l'opérateur `cross`, déjà rencontré à la Section 1.2 :

```
$ liquidsoap -h cross
Generic cross operator, allowing the composition of the N last seconds of a track
with the beginning of the next track.
Type: (?duration:float, ((source, source)->source), source)->source
Parameters:
* duration    :: float (default 5.)                Duration of the composition (s).
* (unlabeled) :: (source, source)->source                        Transition function.
* (unlabeled) :: source
```

Les opérateurs prédéfinis de Liquidsoap dépendent souvent de nombreux paramètres. Par exemple, la sortie la plus couramment utilisée, qui envoie le flux encodé au format Ogg/Vorbis à un serveur de diffusion, dépend de vingt paramètres. Pour que le langage reste attractif et utilisable simplement, nous

avons donc été conduits à utiliser des *arguments étiquetés*, ce qui évite d'avoir à se souvenir de l'ordre des arguments. De plus, de nombreux paramètres ayant des valeurs par défaut raisonnables, nous avons ajouté la possibilité d'avoir des *arguments optionnels*, qui permettent d'utiliser les valeurs par défaut quand aucune valeur n'a été spécifiée. Ainsi, dans l'exemple précédent, le paramètre spécifiant la durée de la superposition est étiqueté `duration` et est optionnel, avec comme valeur par défaut 5 secondes.

2.1. Un langage orienté vers la simplicité

Plusieurs articles établissent les fondations d'un λ-calcul avec étiquettes [3] et arguments implicites [9], et ont mené à l'implémentation de ces traits dans le langage OCaml. Avant d'introduire le calcul avec étiquettes et arguments implicites utilisé dans Liquidsoap, mentionnons brièvement pourquoi nous avons trouvé utile de nous démarquer de [9], en soulignant les difficultés que posent ce genre de calculs. Les auteurs de cet article s'attachent à la compilation efficace du langage, ce qui entraîne certaines restrictions peu naturelles pour l'utilisateur non averti. Par exemple en OCaml, les arguments étiquetés ne commutent pas toujours :

```
# let app f = f ~a:1 ~b:2 ;;
val app : (a:int -> b:int -> 'a) -> 'a = <fun>
# app (fun ~b ~a -> a+b) ;;
This function should have type a:int -> b:int -> 'a
but its first argument is labeled ~b
```

Dans la même optique, les auteurs souhaitent implémenter les arguments optionnels comme un « sucre syntaxique typé » : après une première phase de typage, le code appliquant les valeurs par défaut est ajouté implicitement lorsque tous les paramètres obligatoires sont déja appliqués. Ceci limite les abstractions sur une fonction prenant un argument implicite. Considérons par exemple la fonction `fun f x -> f x`. Avec le type `('a -> 'b) -> 'a -> 'b`, inféré par OCaml, cette fonction sera compilée sans prendre en compte la possibilité d'arguments optionnels à substituer implicitement dans `f`, et le système de types interdira de passer une fonction avec un argument optionnel. En revanche, si on force le type de `f` à être par exemple `?a:int -> int -> int`, la fonction sera compilée pour implicitement substituer l'argument étiqueté `a`, mais n'acceptera plus de fonction sans argument implicite. Dans [9], les auteurs obtiennent un système de type et un algorithme qui infère des types principaux (cf. Section 2.4), mais ils doivent pour cela interdire toute abstraction sur une fonction avec argument implicite.

En présence d'arguments optionnels, on notera le besoin de distinguer entre les applications successives d'une fonction à des arguments et la *multi-application* d'une fonction à des arguments c'est-à-dire l'application de la fonction à plusieurs arguments *simultanément*. Ceci est nécessaire pour contrôler le moment où les arguments optionnels sont implicitement appliqués. L'exemple suivant le montre avec OCaml :

```
# let f ?(a=false) () = a ;;
val f : ?a:bool -> unit -> bool = <fun>
# f () ~a:true ;;
- : bool = true
# (f ()) ~a:true ;;
This expression is not a function, it cannot be applied
```

Cependant, la multi-application d'OCaml ne peut être vide (0-aire) : on ne peut pas appliquer une fonction de type `?l:t -> t'` à « rien » pour obtenir un terme de type `t'`, calculé en substituant l'argument optionnel par sa valeur par défaut dans le corps de la fonction. Dans ce cas l'argument optionnel est *ineffaçable*, c'est-à-dire que la valeur par défaut ne peut jamais être utilisée.

Dans notre approche, l'efficacité passe après la simplicité d'utilisation, car les réductions dans le langage prennent généralement un temps négligeable devant les traitements audio. Cela nous a amenés à introduire un calcul légèrement différent de [9]. Nous nous proposons de plus de considérer une notion de multi-abstraction, une approche qu'il paraît naturel d'explorer, par symétrie avec la multi-application. Le système obtenu n'est peut-être pas compilable efficacement, mais sa sémantique est intuitive, et son interprétation simple.

Nous allons donc formaliser un calcul et un système de types originaux. On notera que le cœur du langage n'est pas spécifique au traitement audio et pourrait être réutilisé dans d'autres domaines.

2.2. Termes du langage

On se donne un ensemble L d'*étiquettes* et un ensemble V de *valeurs de base*. En pratique dans Liquidsoap, l'unité, les booléens, les entiers, les flottants et les chaînes de caractères font partie des valeurs de base, mais aussi les sources et les requêtes. On note X l'ensemble des *variables*. Les termes sont définis inductivement par la grammaire suivante :

$$\begin{array}{rcll} M & ::= & v & (1)\\ & | & x & (2)\\ & | & \texttt{let } x = M \texttt{ in } M & (3)\\ & | & \lambda\{\ldots, l_i : x_i, \ldots, l_j : x_j = M, \ldots\}.M & (4)\\ & | & M\{l_1 = M, \ldots, l_n = M\} & (5) \end{array}$$

La multi-application (5) est l'application simultanée d'une fonction à plusieurs arguments étiquetés par des $l_i \in L$. La multi-abstraction (4) abstrait simultanément sur plusieurs arguments en les étiquetant[1] par $l_i \in L$ et en précisant pour certains arguments une *valeur par défaut*. La notation compacte utilisée dans (4) dénote une liste d'arguments quelconque, sans contrainte sur la position relative des arguments avec et sans valeur par défaut. Toujours dans la multi-abstraction, on supposera les x_i deux-à-deux distincts[2]. Les arguments pour lesquels une valeur par défaut est donnée sont appelés *optionnels*, dans le cas contraire ils sont *obligatoires*. On dira enfin qu'un groupe de *liaisons* $\Gamma = \{\ldots, l_i : x_i, \ldots, l_j : x_j = M_j, \ldots\}$ est *effaçable* si toutes les liaisons qu'il contient sont optionnelles (en particulier, l'ensemble vide est effaçable). Dans la multi-abstraction $\lambda\{\Gamma\}.M$, les variables x_i sont liées dans M, mais pas dans les valeurs par défaut M_j. Ceci permet de définir l'ensemble $\mathcal{FV}(M)$ des variables libres d'un terme M ; lorsque cet ensemble est vide, le terme M est dit *clos*.

Nos termes sont considérés modulo la relation d'α-équivalence habituelle. On considérera aussi que deux arguments d'étiquettes distinctes peuvent permuter au sein d'une même multi-abstraction[3]. Formellement, cela revient à considérer les termes modulo la relation $\equiv$ définie comme la plus petite congruence satisfaisant :

$$\left.\begin{array}{rcl} \lambda\{\Gamma, l : x, l' : x', \Delta\}.M & \equiv & \lambda\{\Gamma, l' : x', l : x, \Delta\}.M\\ \lambda\{\Gamma, l : x = N, l' : x', \Delta\}.M & \equiv & \lambda\{\Gamma, l' : x', l : x = N, \Delta\}.M\\ \lambda\{\Gamma, l : x = N, l' : x' = N', \Delta\}.M & \equiv & \lambda\{\Gamma, l' : x' = N', l : x = N, \Delta\}.M \end{array}\right\} \text{ si } l \neq l' \qquad (6)$$

où Γ et Δ sont des listes d'arguments étiquetés, optionnels ou pas. On pourra vérifier par ailleurs que l'ajout de la relation similaire de permutation dans les multi-applications est compatible avec la réduction du calcul.

[1] On représente facilement les arguments non étiquetés, possiblement optionnels, en les étiquetant par une étiquette particulière.

[2] Mais deux étiquettes peuvent être égales, comme on le voit dans l'Exemple 2.

[3] L'implémentation dans Liquidsoap est légèrement différente, car elle permet de définir une valeur par défaut en fonction des valeurs des arguments précédents. Les notions de liaison et de substitution tiennent alors compte de l'ordre des arguments dans une multi-abstraction, mais la réduction se fait toujours modulo permutation. Cela complique l'écriture du système mais ne change essentiellement rien.

L'application d'une fonction $\lambda\{\ldots, l : x, \ldots\}.M$ à un terme N sur l'étiquette l va provoquer une substitution de la variable x par N dans le corps M, et la suppression de la liaison $l : x$ dans la multi-abstraction. Si à l'issue d'une multi-application aucun argument obligatoire ne subsiste, les valeurs par défaut seront automatiquement substituées aux argument optionnels et la multi-abstraction disparaîtra. Afin de formaliser ceci, on introduit la relation de substitution suivante :

$$\begin{aligned}
x[M/x] &= M \\
y[M/x] &= y, \text{ si } x \neq y \\
(\texttt{let } y = N \texttt{ in } P)[M/x] &= \texttt{let } y = N[M/x] \texttt{ in } P[M/x] \\
(\lambda\{\ldots, l_i : y_i, \ldots, l_j : y_j = N_j, \ldots\}.P)[M/x] &= \lambda\{\ldots, l_i : y_i, \ldots, l_j : y_j = N_j[M/x], \ldots\}.(P[M/x]) \\
(N\{\ldots, l_i = P_i, \ldots\})[M/x] &= N[M/x]\{\ldots, l_i = P_i[M/x], \ldots\}
\end{aligned}$$

Pour le cas du `let` on suppose y distinct de x et non libre dans M. De même pour la multi-abstraction, on suppose que les variables y_i sont distinctes de x et non libres dans M. La sémantique opérationnelle de notre calcul est alors définie par les règles de réduction suivantes :

$$\texttt{let } x = M \texttt{ in } N \rightsquigarrow N[M/x] \tag{7}$$

$$(\lambda\{\overrightarrow{l_i : x_i, l_j : x_j = M_j}, \Gamma\}.M)\{\overrightarrow{l_i = N_i}\} \rightsquigarrow \lambda\{\Gamma\}.(M[\overrightarrow{N_i/x_i}]), \text{ si } \Gamma \text{ est ineffaçable} \tag{8}$$

$$(\lambda\{\overrightarrow{l_i : x_i, l_j : x_j = P_j}, \overrightarrow{l'_k : y_k = M_k}\}.M)\{\overrightarrow{l_i = N_i}\} \rightsquigarrow M[\overrightarrow{N_i/x_i}, \overrightarrow{M_k/y_k}] \tag{9}$$

La notation vecteur ci-dessus désigne une liste de liaisons ou de substitutions. En particulier, la notation $\overrightarrow{l_i : x_i, l_j : x_j = M_j}$ représente une liste de liaisons dont certaines ont une valeur par défaut, sans contrainte particulière sur la position de celles-ci dans la liste. De plus, $M[\overrightarrow{N/x}]$ désigne une séquence de substitutions. Dans la règle (8), on suppose que les variables x_i, ainsi que les variables liées par Γ, ne sont pas libres dans les N_i ; de même dans la règle (9), on supposera que les variables x_i et y_k ne sont pas libres dans les N_i et M_k. Les règles (8) et (9) permettent de réduire l'application d'une abstraction à un ensemble de paramètres étiquetés $\{\overrightarrow{l_i = N_i}\}$, ces arguments correspondant à des paramètres optionnels ou obligatoires de l'abstraction. Dans les deux cas, les règles de réduction provoquent la substitution, dans le corps de l'abstraction, des variables x_i associées aux arguments par la valeur du paramètre N_i, ignorant les valeurs par défaut des x_i optionnels – M_i dans (8) et P_i dans (9). Si des valeurs n'ont pas été assignées à tous les arguments obligatoires, c'est la règle (8) qui s'applique, laissant l'abstraction en attente d'une autre application. Sinon, la règle (9) supprime l'abstraction et substitue les variables associées aux arguments optionnels restants $l'_k : y_k = M_k$ par leur valeur par défaut.

Exemple 1. Le terme $\lambda\{l_1 : x, l_2 : y = 12, l_3 : z = 13\}.M$, appliqué à $\{l_3 = 3\}$, se réduit en $\lambda\{l_1 : x, l_2 : y = 12\}.M[3/z]$. On notera que lors de cette réduction la valeur par défaut de z est ignorée. Si on applique le résultat à $\{l_1 = 1\}$, on obtient alors $M[1/x, 12/y, 3/z]$: la variable y a été implicitement substituée par sa valeur par défaut.

Exemple 2. On retrouve dans ce calcul un phénomène similaire aux arguments ineffaçables d'OCaml. Dans le terme $\lambda\{l : x = 3, l : y\}.M$, il est impossible de permuter les deux arguments. Ainsi, la première application sur l'étiquette l désignera l'argument optionnel x : on est contraint de donner une valeur à x avant de pouvoir en donner une à y et permettre la réduction (9). On préférera donc utiliser deux étiquettes distinctes ou mettre l'argument optionnel en seconde position : $\lambda\{l : y, l : x = 3\}.M$. Dans ce dernier cas, l'application d'un unique paramètre sur l désignera l'argument y et permettra la réduction (9), la multi-application de deux paramètres sur l étant toujours possible si l'on veut aussi spécifier une valeur pour x.

Proposition 1. *Le calcul est confluent.*

Dans la pratique, l'implémentation de Liquidsoap utilise une stratégie de réduction en appel par valeur, ce qui a son importance, car certaines fonctions prédéfinies ont des effets de bord.

2.3. Types

Nous décrivons maintenant un système de types pour nos termes. On se donne un ensemble A de *variables de type*, un ensemble B de *types de base* (dans Liquidsoap, il y a par exemple un type de base pour les sources), et une fonction $\mathcal{T} : V \to B$ qui donne le type des valeurs élémentaires. Le type flèche du λ-calcul devient ici une multi-flèche étiquetée :

$$\begin{array}{rcl} t & ::= & \iota \\ & | & \alpha \\ & | & \{\ldots, l_i : t_i, \ldots, ?l_j : t_j, \ldots\} \to t \end{array}$$

Ci-dessus, on suppose $\iota \in B$ et $\alpha \in A$. Les étiquettes annotées $?l$ dans la multi-flèche dénotent les arguments optionnels. De même qu'avec la multi-abstraction, on considérera les types modulo une permutation dans la multi-flèche définie de façon similaire à (6). La substitution $t[t'/\alpha]$ d'une variable de type $\alpha \in A$ par un type t' dans un type t est définie de façon habituelle.

On introduit du polymorphisme prénexe à la Damas-Milner [8], en se donnant la possibilité de quantifier universellement sur une variable de type dans les *schémas de type* :

$$\sigma \quad ::= \quad t \quad | \quad \forall \alpha.\, \sigma$$

Les règles de réduction introduisent la possibilité pour une fonction $\lambda\{\Gamma, \Delta\}.M$, dont les arguments sont Γ, Δ, de se comporter localement comme une fonction dont les arguments sont Γ et qui renvoie une fonction dont les arguments sont Δ. Ceci est exprimé naturellement ici par la relation de sous-typage suivante :

$$\frac{\Delta \text{ ineffaçable}}{\{\Gamma, \Delta\} \to t \leq \{\Gamma\} \to \{\Delta\} \to t} \qquad \frac{\Delta \text{ effaçable}}{\{\Gamma, \Delta\} \to t \leq \{\Gamma\} \to t}$$

$$\frac{}{t \leq t} \qquad \frac{t \leq t' \qquad t' \leq t''}{t \leq t''}$$

$$\frac{t \leq t' \qquad t'_1 \leq t_1 \qquad \cdots \qquad t'_n \leq t_n}{\{\ldots, l_i : t_i, \ldots, ?l_j : t_j, \ldots\} \to t \leq \{\ldots, l_i : t'_i, \ldots, ?l_j : t'_j, \ldots\} \to t'}$$

Remarque 1. Il peut sembler naturel d'ajouter l'inéquation

$$\{\Gamma, ?l : t, \Delta\} \to t' \quad \leq \quad \{\Gamma, l : t, \Delta\} \to t' \tag{10}$$

à la définition de la relation de sous-typage, mais celle-ci est incompatible avec le calcul. En effet, la fonction $f := \lambda\{l : x = 42\}.x$, qui a le type $\{?l : int\} \to int$, ne se comporte pas comme un terme de type $\{l : int\} \to int$. Par exemple, elle ne peut être appliquée successivement à $\{\}$ puis à $\{l = 12\}$: dès la première application, elle se réduit en 42. En revanche, la fonction $g := \lambda\{l : x\}.x$, qui a le type $\{l : int\} \to int$, se réduit en elle-même quand on l'applique à $\{\}$, puis se réduit en 12 quand on l'applique à $\{l = 12\}$.

Les règles de typage de notre calcul sont :

$$\frac{}{\Gamma \vdash v : \mathcal{T}(v)}(\text{Val}) \qquad \frac{}{\Gamma, x : \forall\alpha_1.\ \ldots \forall\alpha_n.\ t \vdash x : t[t_1/\alpha_1, \ldots, t_n/\alpha_n]}(\text{Ax})$$

$$\frac{\Gamma \vdash M : t' \qquad \Gamma, x : \forall\alpha_1.\ \ldots \forall\alpha_n.\ t' \vdash N : t \qquad \{\alpha_1, \ldots, \alpha_n\} = \mathcal{FV}(t') \setminus \mathcal{FV}(\Gamma)}{\Gamma \vdash \texttt{let}\ x = M\ \texttt{in}\ N : t}(\text{Let})$$

$$\frac{\Gamma \vdash M : \{\ldots, l_i : t_i \ldots, ?l_j : t_j, \ldots\} \to t \qquad \Gamma \vdash N_1 : t_1 \qquad \cdots \qquad \Gamma \vdash N_n : t_n}{\Gamma \vdash M\{\ldots, l_i = N_i, \ldots, l_j = N_j, \ldots\} : t}(\text{App})$$

$$\frac{\Gamma, \ldots, x_i : t_i, \ldots, x_j : t_j, \ldots \vdash M : t \qquad \cdots \qquad \Gamma \vdash M_j : t_j \qquad \cdots}{\Gamma \vdash \lambda\{\ldots, l_i : x_i, \ldots, l_j : x_j = M_j, \ldots\}.M : \{\ldots, l_i : t_i, \ldots, ?l_j : t_j, \ldots\} \to t}(\text{Abs})$$

$$\frac{\Gamma \vdash M : t' \qquad t' \leq t}{\Gamma \vdash M : t}(\text{Sub})$$

De même que dans [8], on limite l'introduction de schémas de type à la règle (Let) ; les schémas sont éliminés par instantiation complète dans la règle (Ax). La règle (App) type la multi-application d'une fonction à des arguments qui peuvent aussi bien se substituer à des paramètres optionnels qu'obligatoires. Cette règle sera le plus souvent utilisée en conjonction avec (Sub) pour autoriser les applications partielles et l'effacement des arguments optionnels. La règle (Abs) vérifie d'une part que le corps de la fonction a le type promis si les arguments ont bien le type demandé, d'autre part que les valeurs par défaut des arguments optionnels ont le bon type.

Exemple 3. La fonction $f := \lambda\{a : x, b : y, c : z = 42\}.x$ admet le type $\{a : int, b : int, ?c : int\} \to int$, qui est un sous-type de $\{a : int\} \to \{b : int, ?c : int\} \to int$. On en déduit que la fonction $f' := f\{a = 16\}$ admet le type $\{b : int, ?c : int\} \to int$. Ce dernier est un sous-type de $\{b : int\} \to int$, et par suite $f'\{b = 69\}$ est de type int. De fait, ce terme se réduit en 16.

Proposition 2 (Réduction du sujet)**.** *Si* $\Gamma \vdash M : t$ *et* $M \rightsquigarrow M'$*, alors* $\Gamma \vdash M' : t$*.*

Lemme 1. *Les dérivations de typage vérifient plusieurs propriétés usuelles, exprimées par l'admissibilité des règles suivantes :*

$$\frac{\Gamma \vdash M : t}{\Gamma, x : \sigma \vdash M : t}\textit{(Weak)} \qquad \frac{\Gamma, x : t' \vdash M : t \qquad \Gamma \vdash N : t'}{\Gamma \vdash M[N/x] : t}\textit{(Subst)}$$

$$\frac{\Gamma, x_1 : t_1, \ldots, x_n : t_n \vdash M : t \qquad t \leq t' \qquad t'_1 \leq t_1 \qquad \cdots \qquad t'_n \leq t_n}{\Gamma; x_1 : t'_1, \ldots, x_n : t'_n \vdash M : t'}\textit{(Subst-Sub)}$$

Proposition 3 (Terminaison)**.** *Si* M *est un terme tel que* $\vdash M : t$ *est dérivable alors toute suite de réductions*

$$M \rightsquigarrow M_1 \rightsquigarrow M_2 \rightsquigarrow \ldots$$

est nécessairement finie.

En utilisant la Proposition 1, on en déduit que tout terme typé admet une *forme normale*.

2.4. Inférence de type

L'algorithme d'inférence utilisé dans Liquidsoap est l'algorithme W de Damas et Milner [8] naïvement étendu à notre calcul pour prendre en compte les multi-abstractions et les multi-applications, ainsi que les arguments étiquetés et optionnels.

Notre algorithme étend strictement celui de Damas et Milner dans le sens suivant. Étant donnée une étiquette $l \in L$, tout terme M de ML peut être vu comme un terme $[\![M]\!]$ dans notre langage par l'interprétation définie par

$$[\![v]\!] = v, \qquad [\![x]\!] = x, \qquad [\![\lambda x.M]\!] = \lambda\{l : x\}.[\![M]\!], \qquad [\![MN]\!] = [\![M]\!]\{l = [\![N]\!]\}$$

et comme le morphisme attendu sur les constructions `let`. De même, on peut voir tout type t de ML comme un type $[\![t]\!]$ de notre langage et ces deux interprétations sont injectives et compatibles avec la réduction. On peut alors montrer que si t est le type d'un terme M de ML, inféré par l'algorithme W, alors $[\![t]\!]$ est le type inféré par notre algorithme pour $[\![M]\!]$.

Une propriété majeure de l'algorithme W est qu'il infère un (schéma de) type canonique, appelé *type principal*, pour un terme. Dans notre système, un terme M admet un type principal t si tout autre type t' de M peut se déduire de t par sur-typage ou par une sorte particulière de substitution appelée *instantiation générique*. Malheureusement, un terme n'admet pas nécessairement de type principal. Par exemple, la fonction $f := \lambda\{l : g\}.(g\{l' = 42\})$ admet à la fois

$$t := \forall\alpha.\ l : \{\{l' : int\} \to \alpha\} \to \alpha \qquad \text{et} \qquad t' := \forall\alpha.\ l : \{\{?l' : int\} \to \alpha\} \to \alpha \tag{11}$$

comme types, et aucun de ces deux types n'est sous-type ou instance de l'autre. Ce défaut est essentiellement dû au fait que la règle de sous-typage évoquée dans la Remarque 1 est invalide :

$$\{\Gamma, ?l : t, \Delta\} \to t' \quad \leq \quad \{\Gamma, l : t, \Delta\} \to t' \tag{10}$$

On peut cependant *plonger* les termes de type $\{\Gamma, ?l : t, \Delta\} \to t'$ dans les termes de type $\{\Gamma, l : t, \Delta\} \to t'$. Par exemple, à partir d'un terme f de type $\{?l : int\} \to int$, on peut construire un terme de type $\{l : int\} \to int$ en utilisant une sorte d'η-expansion : $\lambda\{l : x\}.f\{l = x\}$. Dans ce sens, le type t de f donné en (11) est donc plus général que le type t' donné en (11), car on peut transformer les arguments pour le second en arguments pour le premier. Le type t est précisément celui qui sera inféré pour la fonction f par notre algorithme. Ainsi, les types inférés sont canoniques mais dans un sens plus faible que la principalité.

2.5. Une variante du calcul

Nous évoquons ici une variante du calcul qui permet de s'accommoder de la règle de sous-typage (10), rendant ainsi possible l'existence d'un type principal pour tous les termes. Nous ne l'avons pas choisie lors de l'implémentation de Liquidsoap, car si elle pourrait mieux se prêter à une étude théorique, elle est moins naturelle à utiliser dans un langage de programmation et le typage qu'elle nécessite semble plus complexe à définir et à mettre en œuvre.

Au lieu d'avoir une application et deux règles de réduction, l'une pour l'application partielle et l'autre pour l'application totale, décomposons l'application en deux constructions[4] : l'*application partielle* $M\{l_1 = M_1, \ldots, l_n = M_n\}$ et la *destruction des multi-abstractions* dont les liaisons sont optionnelles $M\bullet$, avec les règles de réduction suivantes.

$$\begin{aligned} (\lambda\{\overrightarrow{l_i : x_i}, l_j : x_j = M_j, \Gamma\}.M)\{\overrightarrow{l_i = N_i}\} &\rightsquigarrow \lambda\{\Gamma\}.(M[\overrightarrow{N_i/x_i}]) \\ (\lambda\{\overrightarrow{l_i : x_i = M_i}\}.M)\bullet &\rightsquigarrow M[\overrightarrow{M_i/x_i}] \end{aligned}$$

Dans la première règle, on n'impose plus que le groupe de liaison Γ ne soit pas effaçable : même si le contexte restant est vide, la multi-abstraction subsiste et seul un destructeur $\bullet$ peut la supprimer.

[4]Cette distinction évoque certains langages, comme Python, où l'application usuelle est totale mais où une construction spécifique a été ajoutée pour gagner le confort de l'application partielle. La question importante ici est le typage statique d'un tel calcul, ce qui n'entre pas du tout en ligne de compte en Python.

La règle de sous-typage $\{\Gamma, \Delta\} \to t \leq \{\Gamma\} \to \{\Delta\} \to t$ (où Δ n'est pas effaçable) n'est plus pertinente, mais la nouvelle règle (10) qui permet de promouvoir un argument optionnel en argument obligatoire semble désormais valide : une application partielle est toujours partielle, que la fonction utilisée ait des arguments obligatoires ou optionnels.

Par exemple, la fonction $f := \lambda\{l : x\}.(x\{l' = 1\}\bullet)$ accepte toutes les fonctions attendant un entier sur l'étiquette l', que cet argument soit obligatoire ou non. L'application partielle vide, qui aurait exclu les arguments x de type $\{?l' : int\} \to int$ dans le système précédent, n'a désormais plus jamais d'effet. En donnant à notre fonction le type $\forall\alpha.\ \{l : \{l' : int\} \to \alpha\} \to \alpha$, on peut l'appliquer à $g_1 := \lambda\{l' : x\}.x$ aussi bien qu'à $g_2 := \lambda\{l' : x = 3\}.x$, car le type canonique de ce dernier terme est désormais sous-type de $\{l' : int\} \to int$. Mais on ne peut l'appliquer à $g_3 := \lambda\{l' : x, l'' : y\}.y$, à juste titre, car on ne pourrait alors pas réduire le destructeur $\bullet$.

Considérons maintenant la fonction $f' := \lambda\{l : x\}.x\{l' = 1\}$, qui peut cette fois être appliquée à g_1, à g_2 ou à g_3. Le type de f' doit indiquer que cette fonction prend en paramètre sur l n'importe quelle fonction acceptant un paramètre entier sur l', et éventuellement d'autres paramètres, et renvoie une fonction qui n'attend plus que ces autres paramètres. Pour rendre compte de ceci, il faudrait enrichir notre système de types en lui ajoutant des variables de rangée Γ représentant des ensembles de liaisons et donner à f' un type de la forme $\forall\Gamma.\forall\alpha.\ \{l : \{l' : int, \Gamma\} \to \alpha\} \to \{\Gamma\} \to \alpha$. La définition complète et l'étude d'un tel système seraient intéressantes et sont laissées pour de futurs travaux.

3. Extensions

3.1. Les contraintes de type

En pratique, Liquidsoap implémente un langage déja étendu par rapport au système décrit formellement en Section 2. On y trouve en effet une notion de *contrainte* permettant du polymorphisme *ad-hoc*. Par exemple, malgré la distinction entre entiers (de type *int*) et flottants (de type $float$), la même fonction d'addition permet d'additionner des valeurs de l'un ou l'autre des types. Celle-ci a en effet le type $\forall\alpha \in Num.\ \{\alpha, \alpha\} \to \alpha$, où la quantification universelle sur α est restreinte par une contrainte qui impose à la variable de type α d'être instanciée par *int* ou par $float$. Concrètement, le type de l'addition dans Liquidsoap est :

```
(+) :: ('a,'a)->'a where 'a is a number type
```

Outre la restriction à un type numérique, d'autres contraintes sont possibles. Par exemple, la contrainte Ord représente une classe de types naturellement ordonnables auxquels seront restreintes les opérations de comparaison : la clôture de $\{int, float, string, bool, unit\}$ par $- \times -$ (constructeur des types des paires) et $[-]$ (constructeur des types des listes). Mais Liquidsoap dispose aussi de contraintes plus exotiques, spécifiques à son domaine d'application.

Ce système très utile est néanmoins très faible. Les contraintes sont des atomes prédéfinis dans le langage et non définissables par l'utilisateur, sans autre sémantique qu'un test de satisfaction. Le système de types n'est par exemple pas capable de simplifier un ensemble de contraintes ou de détecter une incompatibilité entre contraintes.

Lors de l'inférence de type usuelle, des méta-variables sont utilisées pour représenter les types encore inconnus. Pour l'étendre aux contraintes, on attache aux méta-variables des ensembles de contraintes à respecter. Quand on instancie une méta-variable par un type, on vérifie que ses contraintes sont respectées par le type, en propageant si besoin les contraintes aux nouvelles méta-variables. Par exemple, si l'on procède à une comparaison (`x==x`) sur la variable `x`, la méta-variable de type associée (`?Tx`) est contrainte à être dans Ord ; si ensuite on utilise `x` comme une liste (`list.tl(x)==x`), la méta-variable `?Tx` est instanciée en `[?T]` (c'est-à-dire une liste dont le type est la méta-variable `?T`) et la contrainte Ord est propagée à la méta-variable `?T`.

3.2. Raffinements des types spécifiques à la génération de flux

Dans le contexte de la génération de flux audio pour des radios, il est intéressant de garantir certaines propriétés. La *vivacité* est par exemple importante : une radio doit diffuser en continu. Liquidsoap procède actuellement à la vérification qu'une source émise vers un serveur de diffusion a toujours des données à émettre. Ceci permet au concepteur de la source de s'assurer qu'il a bien prévu des dispositifs de secours dans tous les cas. Par exemple, une liste de lecture de fichiers distants n'est pas infaillible, car une panne ou une latence du réseau pourrait empêcher d'obtenir les fichiers à jouer ou encore ces fichiers pourraient être corrompus. Un opérateur de choix est vivace si l'une de ses sources filles l'est ; la vivacité sera ainsi typiquement assurée grâce à un choix par défaut qui en cas de problème va jouer un fichier ou une liste de lecture locale dont on aura préalablement vérifié les fichiers.

Cette vérification est pour le moment faite de façon dynamique : ce sont les opérateurs de sortie créés par les programmes Liquidsoap qui se chargent de demander à leurs sources filles de calculer leur vivacité. Cette approche est limitée, et ne prend notamment pas en compte les transitions. La vivacité d'un opérateur peut en effet être compromise par l'utilisation d'une fonction de transition, si celle-ci transforme une source représentant la nouvelle piste en une source qui ne produit rien[5]. Pour rejeter les programmes qui peuvent avoir ce comportement, il faudrait développer une véritable technique d'analyse statique, impliquant une annotation des types des sources.

Un autre problème est lié à la notion de temps. Au premier abord, il semble que toutes les sources évoluent dans la même échelle de temps, produisant toutes leur flux avec le même débit. Cela n'est plus vrai avec des opérateurs comme `cross`, qui superposent deux portions consécutives du même flux, accélérant localement son débit en amont. Or, on risque d'obtenir des résultats incohérents si deux opérateurs accèdent à une même source dans des échelles de temps différentes. Là encore, une solution serait d'adapter le système de types pour assurer qu'une source potentiellement utilisée (directement ou pas) par un opérateur comme `cross` n'est utilisée par aucun autre opérateur.

Enfin, Liquidsoap est actuellement restreint à l'émission de données audio. De plus, toutes les sources au sein d'une même instance partagent le même format audio (nombre de canaux et fréquence d'échantillonnage). Cependant, les abstractions fournies par notre langage ne sont pas spécifiques à la manipulation de données audio, et on pourrait par exemple espérer traiter un flux vidéo – le traitement vidéo à la volée devenant accessible aux processeurs modernes. Les formats des flux ne seraient alors pas uniformes, certains flux contenant des données audio, d'autres de la vidéo. Plus modestement, il serait utile qu'une radio puisse émettre plusieurs flux audio dont certains seraient en stéréo et d'autres en 5 canaux (*surround 5.1*). Il faudrait donc étendre le système de types pour pouvoir spécifier dans le type des sources le format du flux qu'elles émettent, ce qui permettrait de vérifier statiquement la compatibilité des formats des sources en interaction.

Conclusion

Nous avons présenté un outil puissant dédié à la génération de flux audio. Liquidsoap permet de décrire de façon simple les configurations les plus complexes, grâce à la manipulation des flux de façon abstraite.

Ce langage illustre la possibilité d'utiliser avec succès, pour des utilisateurs néophytes de la programmation, des technologies réputées compliquées comme le typage statique et inféré. Cette expérience nous a conduits à revisiter la conception des arguments étiquetés développée pour le langage OCaml et à en développer une variante plus souple. Nous espérons voir l'idée et l'implémentation réutilisées dans d'autres projets similaires.

[5]Au delà de la vivacité, c'est même un arrêt total du système qui survient dans ces cas, qui ne se produisent heureusement que très rarement et avec des programmes complexes.

Remerciements. Un grand merci à Clément Renard pour ses contributions à Liquidsoap et ses remarques sur cet article. Merci aussi à Romain Beauxis, Stéphane Gimenez et Vincent Tabard pour leurs contributions à Savonet/Liquidsoap.

Références

[1] Dolebraï. `http://www.dolebrai.net/`.

[2] RadioPi. `http://www.radiopi.org/`.

[3] H. Aït-Kaci and J. Garrigue. Label-selective λ-calculus : syntax and confluence. *Theoretical Computer Science*, 151 :353–383, 1995.

[4] C. Baccigalupo and E. Plaza. A Case-Based Song Scheduler for Group Customised Radio. *Lecture Notes in Computer Science*, 4626 :433, 2007.

[5] D. Baelde, R. Beauxis, S. Gimenez, S. Mimram, V. Tabard, and al. Savonet. `http://savonet.sf.net/`.

[6] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9) :1270–1282, 1991.

[7] E. Chailloux, P. Manoury, and B. Pagano. *Développement d'applications avec Objective Caml*. O'Reilly, 2000.

[8] L. Damas and R. Milner. Principal type schemes for functional programs. In *POPL*, 1982.

[9] J. P. Furuse and J. Garrigue. A label-selective lambda-calculus with optional arguments and its compilation method. RIMS Preprint 1041, Kyoto University, October 1995.

[10] G. Wang and P. R. Cook. ChucK : A Concurrent, On-the-fly Audio Programming Language. *Proceedings of International Computer Music Conference*, pages 219–226, 2003.

[11] William Thies, Michal Karczmarek and Saman Amarasinghe. StreamIt : A Language for Streaming Applications. *Proceedings of International Conference on Compiler Construction*, 2002.