

ANR-12-CORD-0009
INEDIT Project
 Programme ContInt
<http://inedit.ircam.fr/>

Version librairie du compilateur Faust

Yann Orlarey

GRAME, Centre National de Création Musicale

orlarey@grame.fr

Abstract

Faust est un langage de programmation de type fonctionnel synchrone conçu spécifiquement pour la synthèse et le traitement du signal audio-numérique en temps-réel. L'objectif de WP3.4 est de développer une version *embarquable* du compilateur Faust (appelée *libfaust*) pouvant être intégrée aux différentes applications du projet INEDIT. L'intérêt pour ces applications est de disposer ainsi d'une chaîne de compilation complète, permettant de traduire les traitements synchrones, écrites en Faust, en code binaire exécutable fonctionnant donc à vitesse native.

En s'appuyant sur la technologie LLVM cette chaîne de compilation est totalement autonome et ne dépend d'aucun outil de développement extérieur. Ceci facilite grandement le déploiement auprès d'utilisateurs non informaticiens. Elle est en outre très rapide, rendant ainsi dans de nombreux cas la phase de compilation transparente pour l'utilisateur.

Le rapport présente la librairie *libfaust*, l'infrastructure de compilation LLVM, et trois applications de cette technologie : *faustgen* un plugin Max/MSP permettant d'éditer, de compiler et d'exécuter du code Faust depuis Max, *FaustNode*, une extension de la WebAudio API permettant de compiler et d'exécuter du code Faust depuis un navigateur Web et *faustcsound*, un ensemble de quatre opcodes pour intégrer Faust dans le langage Csound.

1 Introduction

Faust (Functional Audio Stream) est un langage de programmation spécifiquement conçu pour les applications musicales. Il permet de développer rapidement des applications efficaces et fiables pour les principales plateformes audios (plugins VST, Max/MSP, Puredata, applications OSX, iPhone, Android, Linux, etc.).

A la différence d'autres environnements audio, Faust est un langage compilé. Un programme Faust décrit un processeur de signaux sous la forme d'une fonction qui prend un ensemble de signaux en entrée et produit un ensemble de signaux en sortie. C'est le calcul des signaux de sortie (en fonction des signaux d'entrée) qui est effectivement compilé, sous la forme d'une fonction *compute*, à appeler à chaque cycle audio avec les buffers audio d'entrée/sortie adaptés.

La version actuelle de Faust traduit le code DSP sous la forme d'une classe C++, à intégrer ensuite dans un fichier d'architecture. Celui-ci connecte le coeur de calcul généré par le compilateur aux entrées/sorties audio et à une interface de contrôle (qui peut être une interface utilisateur classique avec boutons, curseurs, zones de texte ou créée à partir d'une interface de pilotage utilisant le protocole OSC par exemple) [2]. Le code résultant est alors compilé avec un compilateur C/C++ classique, pour obtenir au final une application ou un plug-in opérationnel.

Pour réaliser *libfaust* nous avons procédé en plusieurs étapes :

- le compilateur FAUST a tout d'abord été réorganisé autour d'une nouvelle représentation intermédiaire permettant d'ajouter facilement de nouveaux backends (C, C++, Java, Javascript)
- nous avons ensuite entrepris de développer un backend LLVM IR en vue d'utiliser la librairie de compilation dynamique LLVM qui permet de créer du code exécutable directement en mémoire
- le code du compilateur Faust a été adapté au fonctionnement "multi-shot" qu'implique une version librairie (gestion mémoire, gestion des erreurs, etc.)
- enfin nous avons développé une API simple qui permet, en gros, à partir d'une chaîne de caractères représentant le code Faust, de récupérer un pointeur en mémoire vers le code exécutable.

Tous ces développements ont eu lieu sur la branche *faust2* du repository Faust : <http://sourceforge.net/p/faudiostream/code/ci/faust2/tree/>. Après les avoir détaillés, nous montrerons comment *libfaust* peut être mise en oeuvre à travers trois exemples : *faustgen*, un objet externe pour Max/MSP, *FaustNode*, une intégration de Faust dans la Web Audio API, et *faustcsound* un ensemble d'opcodes pour Csound. Nous expliciterons ensuite l'API définie par *libfaust* et comment l'utiliser pour embarquer le compilateur Faust.

2 La chaîne de compilation

2.1 FIR : Faust Imperative Representation

Le compilation du code DSP source se passe en plusieurs étapes : le programme est d'abord évalué afin de construire le *bloc-diagramme* de traitement des signaux qu'il représente. Des signaux symboliques sont ensuite propagés dans ce bloc-diagramme pour établir ce que ce bloc-diagramme calcule, c'est à dire comment les signaux de sortie s'expriment en fonction des signaux d'entrée (Figure 1). C'est cette représentation interne des *signaux* qui est compilée directement dans une classe C++ dans la version actuelle de Faust.

Dans *faust2*, un langage intermédiaire nommé FIR (Faust Imperative Representation) a été défini pour décrire les calculs réalisés sur les échantillons de manière générique. Ce langage contient les opérations

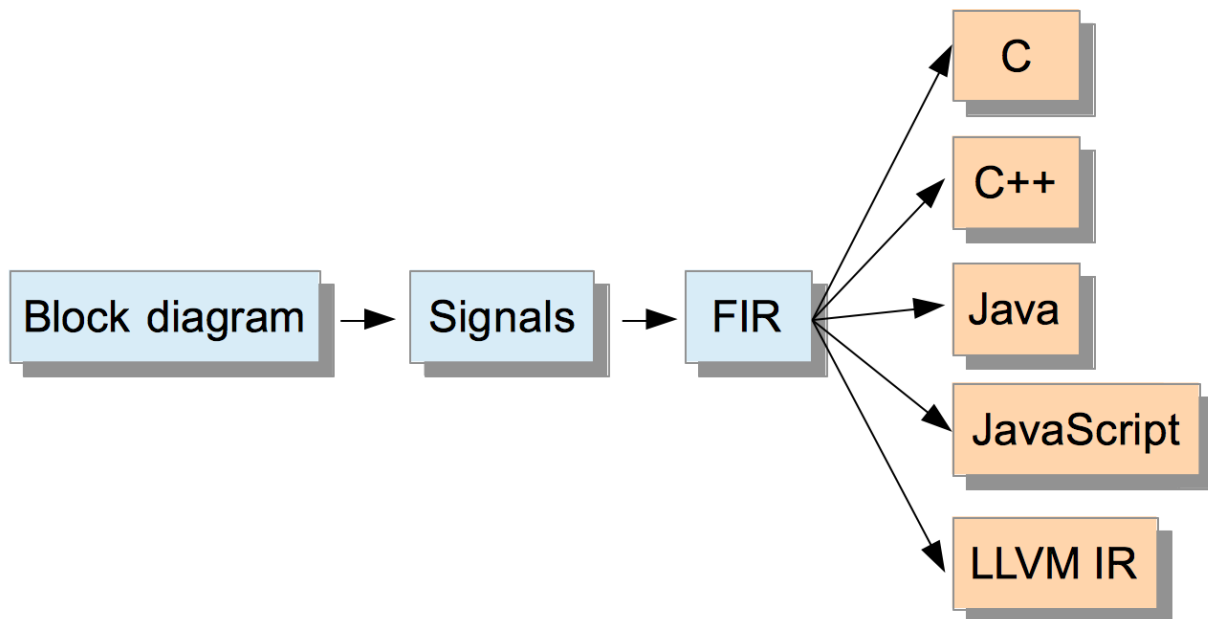


Figure 1: Etapes internes de la compilation dans Faust

pour lire et écrire des variables et des tableaux, effectuer des opérations arithmétiques et définir les structures de contrôle nécessaires (boucle *for*, boucle *while*, structure de test *if*...). Le langage des signaux est maintenant compilé dans ce langage FIR intermédiaire.

Le langage FIR est ensuite traduit en différents langages cible : C, C++, Java, JavaScript ou LLVM. Ce dernier langage cible est particulièrement intéressant dans une optique de compilation dynamique à l'intérieur même des applications ou plug-ins.

2.2 LLVM

LLVM (Low Level Virtual Machine) est une infrastructure de compilation composée de différents modules pour analyser des langages sources, générer du code intermédiaire, faire l'édition des liens avec du code défini de manière externe ou dans des bibliothèques, et finalement générer du code exécutable (soit de manière statique en vue de produire un fichier exécutable, soit de manière dynamique avec un compilateur à la volée) (Figure 2).

Le code de cette nouvelle architecture de compilation écrite en C++ est plus moderne et plus flexible que celle construite autour de GCC. Il est donc plus facile à utiliser dans des projets variés. LLVM est ainsi devenue la chaîne de compilation par défaut disponible sur OSX (sous la forme du compilateur *clang*, du débogueur *lldb* etc.).

Le code intermédiaire LLVM (LLVM IR [4]) est un langage bas niveau typé, qui représente les instructions sous la forme SSA¹. Pour générer ce code, un backend FIR vers LLVM IR a été ajouté dans *faust2*. Il produit au final un module LLVM, qui regroupe les déclarations de structures de

1. Static Single Assignment : où chaque variable est assignée une seule fois

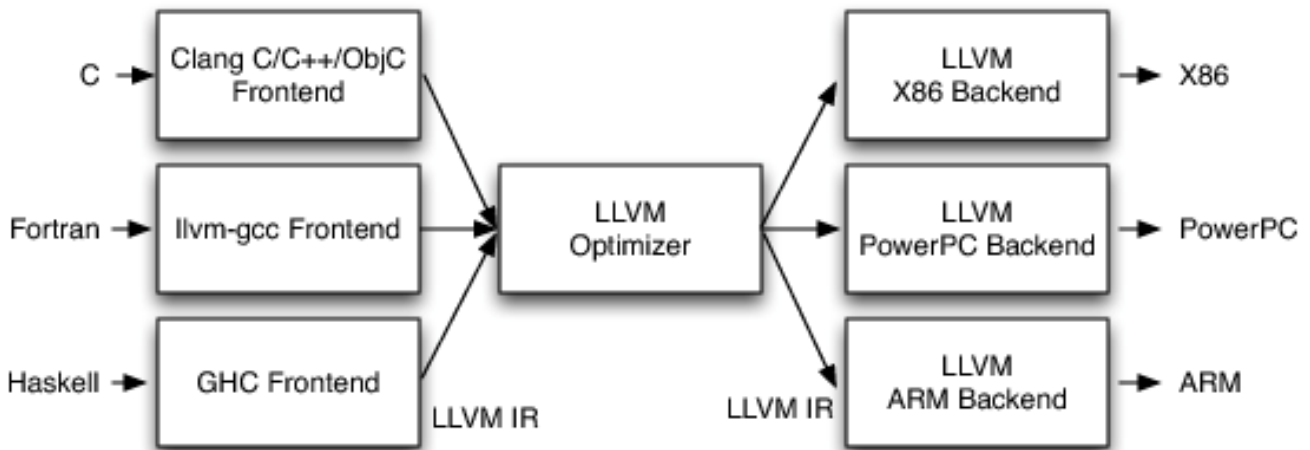


Figure 2: Schéma des composants LLVM

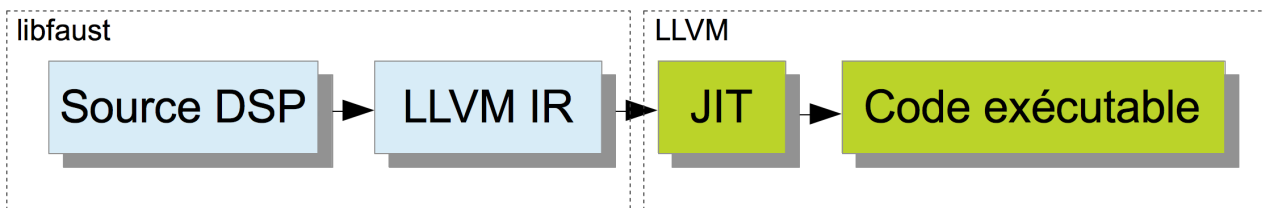


Figure 3: Chaîne de compilation complète

données (objet DSP et interface utilisateur...) ainsi que l'ensemble des fonctions pour le manipuler (*init*, *buildUserInterface*, *compute*...).

Le code LLVM IR peut ensuite être optimisé à l'aide de différentes passes d'optimisation, et enfin compilé à la volée en code natif exécutable dépendant de l'architecture sous-jacente (i386, x86_64, PPC...). Les fonctions en code natif vont être générées et pourront être appelées avec les paramètres adaptés.

Par ailleurs du code externe défini par exemple dans un fichier source C ou C++, peut être compilé en code LLVM IR en utilisant le compilateur *clang*, et assemblé avec le module LLVM généré pour un source Faust donné. Il est ainsi possible d'ajouter le code du Work Stealing Scheduler [3] de manière très efficace, grâce en particulier à la phase de "Link Time Optimisation" qui est disponible dans l'infrastructure LLVM.

2.3 La chaîne de compilation complète

La chaîne de compilation complète part du code DSP source, compilé en code intermédiaire LLVM grâce au backend LLVM, et produit au final du code exécutable à l'aide d'un *compilateur à la volée* (JIT) (Figure 3).

3 Interface de programmation

La librairie *libfaust* est distribuée avec deux fichiers d'entêtes : `llvm-dsp.h` pour les développements C++, `llvm-c-dsp.h` pour les développements en C, ainsi qu'une documentation spécifique de l'API (voir le document *libfaust 0.90.pdf*). Pour être intégrée à une application, elle doit être ajoutée à l'édition des liens ainsi que toutes les librairies LLVM.

L'interface de programmation est la suivante : une *fabrique* d'objets DSP peut être construite à partir d'un programme Faust source (lu à partir d'un fichier ou d'une chaîne de caractères) à l'aide de la fonction `createDSPFactoryFromFile()` ou de la fonction `createDSPFactoryFromString()` (par analogie, cette fabrique correspond à la définition de la classe C++ obtenue avec le mode de compilation traditionnel de Faust).

Une fois la fabrique produite, des instances de l'objet DSP vont être construites avec la fonction `createDSPInstance()` (par analogie, cette opération correspond à la création d'objets C++ avec l'opérateur `new` appelé sur la classe `mydsp` obtenue avec le mode de compilation traditionnel de Faust). Une fabrique peut également être construite en rechargeant une description du module LLVM (au format bitcode ou textuel) sauvegardée au préalable, ce qui en accélère la re-génération.

L'objet créé possède l'interface classique d'un processeur de signaux Faust:

```
class llvm_dsp : public dsp {
public:
    int metadata(Meta* m);

    int getNumInputs();
    int getNumOutputs();

    void init(int rate);

    void buildUserInterface(UI* interface);

    void compute(int count,
                 float** input,
                 float** output);
}
```

Il peut donc être utilisé à l'intérieur d'un des nombreux fichiers d'architecture de manière similaire à un objet dont le code C++ serait produit avec la méthode classique. Enfin les objets et la fabrique sont détruits avec les fonctions `deleteDSPInstance()` et `deleteDSPFactory()`.

L'utilisation interne de la technologie LLVM reste donc cachée dans l'interface de programmation décrite, ceci pour faciliter la tâche des développeurs. Une interface plus bas niveau permettant d'accéder au module LLVM est également disponible.

4 L'objet *faustgen~* pour Max/MSP

Jusqu'à présent, l'utilisation d'objets externes Faust dans Max/MSP nécessitait de produire un objet compilé en statique avec le fichier d'architecture adapté et compilé avec un compilateur C. L'exécutable

résultant devait ensuite être chargé dans l'environnement Max/MSP. Cette technique rendait difficile la modification dynamique du code source du programme Faust, puisque celui devait être re-compilé et le programme Max/MSP relancé pour pouvoir utiliser la nouvelle version.

4.1 L'objet *faustgen*

C'est un objet externe pour Max/MSP qui embarque le compilateur Faust. S'appuyant sur *libfaust* et LLVM, *faustgen* permet de compiler et d'exécuter dynamiquement du code Faust. Celui-ci est édité dans un éditeur intégré, et l'exécutable natif est compilé lorsque l'éditeur est fermé. Il est alors inséré dans le graphe de calcul audio.

Les options de compilation classiques de Faust² peuvent être passées en paramètre. Les noms et les caractéristiques des paramètres de contrôle sont imprimés dans la console de Max/MSP, à charge ensuite à l'utilisateur de créer et connecter les éléments graphiques (curseurs, boutons, zones numériques...) nécessaires pour contrôler le processus audio.

Chaque objet *faustgen* contient potentiellement un algorithme DSP différent. Pour partager le même algorithme, l'objet *faustgen* doit être nommé et il pourra ensuite être dupliqué pour créer autant d'instances différentes. Si le programme est modifié, toutes les instances de celui-ci seront mises à jour (Figure 4).

Le programme source au format texte ainsi que les options de compilation utilisées sont sauvegardés dans le patch. Pour optimiser les temps de re-chargement en économisant une partie des étapes de la compilation, une version binaire (au format LLVM bytecode) du code LLVM intermédiaire est également sauvegardée.

5 FaustNode

La programmation audio dans les navigateurs était jusqu'à récemment assez primitive, et essentiellement possible grâce à l'utilisation de technologies externes comme QuickTime ou Flash. L'introduction de l'élément audio dans HTML 5 a été un progrès important, en permettant la gestion de flux en "streaming". Néanmoins, pour pouvoir décrire des applications audio plus complexes, l'introduction de fonctionnalités nouvelles était nécessaire.

Web Audio est une API [5] JavaScript de haut niveau pour le traitement et la génération de son dans les applications web en HTML5. Le processus de calcul est décrit sous la forme d'un graphe de noeuds audio "natifs" (typiquement implémentés en C/C++ pour des raisons d'efficacité) et dont le rendu est ensuite effectué en temps-réel par l'implémentation sous-jacente. Des noeuds de calculs audio écrits en JavaScript (JavaScriptNode) peuvent également être insérés dans le graphe.

5.1 Génération directe de code JavaScript

Nous avons expérimenté la génération directe de code JavaScript en implémentant un backend JavaScript dans *faust2*. Le code est ensuite enrobé dans un fichier d'architecture adapté qui permet de l'utiliser comme un JavaScriptNode, en l'insérant ensuite dans le graphe de noeuds audio de Web Audio.

2. par exemple `-vec, -lv 1, -vs 32`, ou `-sch` pour le mode Work Stealing Scheduler

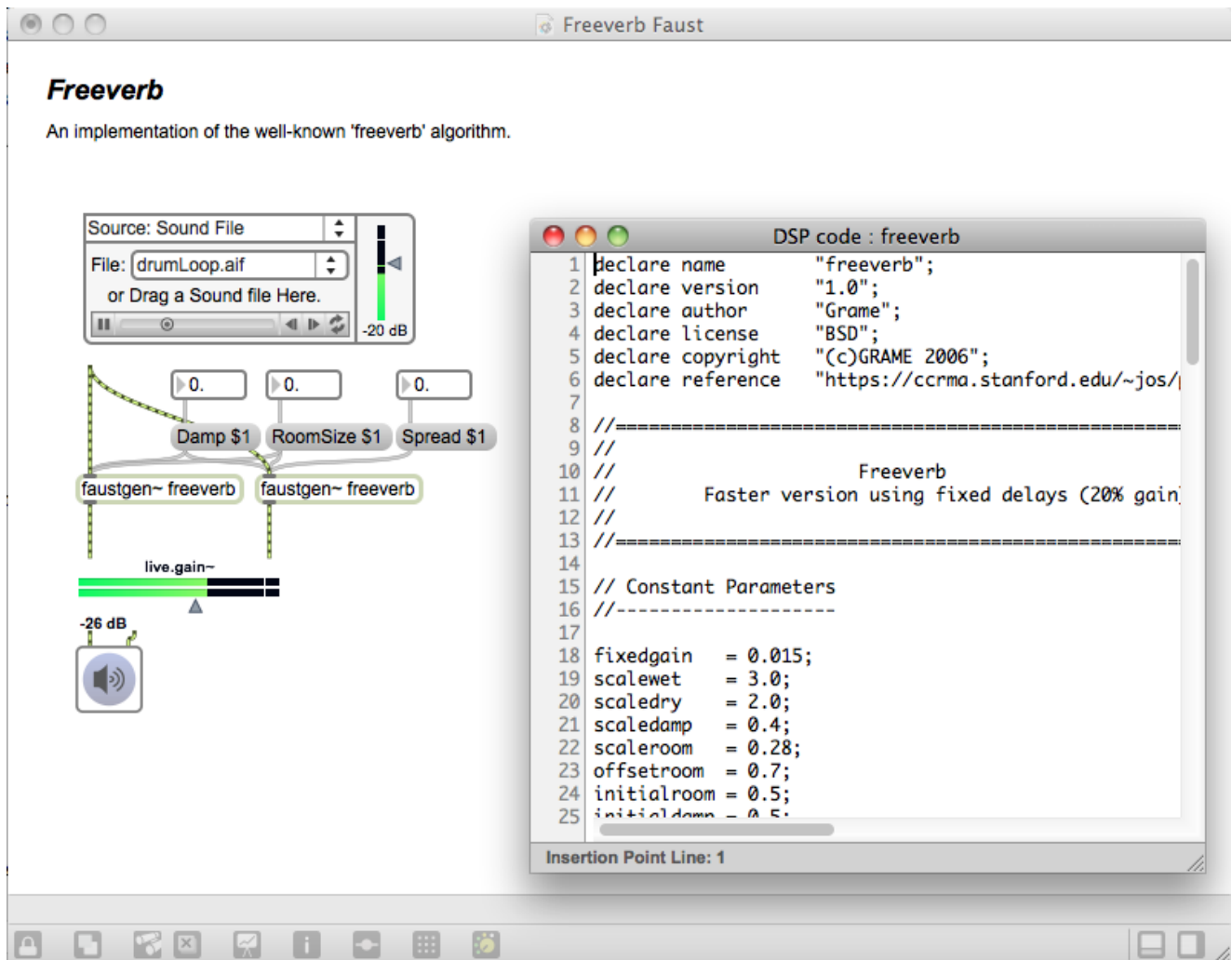


Figure 4: Une utilisation de *faustgen~* pour programmer *freeverb*

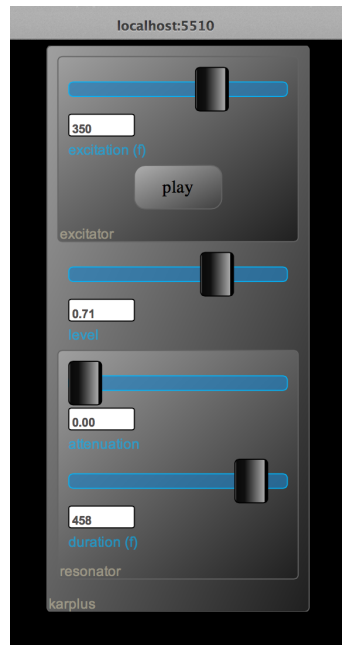


Figure 5: Un exemple de *karplus* fonctionnant dans un navigateur avec une interface graphique décrite en SVG à partir de la description JSON

Cette méthode produit des noeuds de calculs audio opérationnels, mais dont le code est trop lent pour en envisager une réelle utilisation dans un contexte temps-réel (avec par ailleurs les problèmes de performances liés au côté "dynamique" de JavaScript, et en particulier sa gestion mémoire avec "ramasse-miettes" qui potentiellement peut interrompre le thread audio).

5.2 Le noeud natif FaustNode

L'autre possibilité que nous avons exploré est la création d'un noeud C++ natif *FaustNode*. Ce noeud permet à la chaîne de compilation décrite précédemment d'être intégrée dans les navigateurs, et à des programmes Faust arbitraires d'être compilés dynamiquement et exécutés à vitesse native.

Cette intégration dans la Web Audio API a été testée en modifiant le moteur open-source WebKit³. Un nouveau noeud natif (qui s'insère dans une hiérarchie de classes C++) nommé *FaustNode* a été développé. Il reçoit en paramètre le code source Faust sous la forme d'une chaîne de caractères. Le source DSP est compilé et le code exécutable natif est produit. Il est ensuite inséré dans le graphe de calcul audio. Un système de cache est utilisé pour éviter la recompilation inutile d'un même source DSP.

La description de l'interface utilisateur peut être produite sous un format JSON (JavaScript Object Notation), un format de données textuel générique qui permet de représenter de l'information structurée, et sera ensuite analysée pour créer une interface utilisateur fonctionnelle (Figure 5).

3. WebKit : <http://www.webkit.org>

6 Faustcsound : quatre opcodes pour utiliser Faust dans Csound

Victor Lazarrini du département de musique de l'université NUI Maynooth Ireland a développé un ensemble de quatre opcodes pour utiliser Faust de manière dynamique dans Csound⁴ :

- `faustcompile` : compile un programme Faust
- `faustaudio` : crée une instance DSP à partir d'un programme compilé Faust. Un nombre arbitraire d'instances peuvent être créées à partir d'un même programme
- `faustctl` : permet de changer la valeur d'un paramètre d'une instance d'un programme Faust donné
- `faustgen` : compile et génère une instance unique d'un programme Faust.

Ces quatre opcodes sont alors compilés sous la forme d'une librairie dynamique nommé `libfaustcsound` et ajoutés avec `libfaust` et les libraires statiques LLVM dans les dépendances de l'application Csound.

Dans une partition Csound, voici un exemple d'utilisation des opcodes `faustgen` et `faustctl`, avec un programme simple qui ajuste le gain d'une entrée :

```
idsp, a1 faustgen {{
    gain = hslider("vol", 1, 0, 1, 0.01);
    process = (_ * gain);
}}, ain1
faustctl idsp, "vol", 0.5
```

7 Conclusion

Nous avons présenté `libfaust`, une version embarquable du compilateur FAUST sous forme de librairie. Cette librairie dispose d'une API simple et peut être facilement intégrée aux différentes applications du projet INEDIT comme le montre les trois exemples d'intégration présentés. La librairie `libfaust` offre une chaîne de compilation FAUST complète, sans faire appel à aucun outil extérieur. Elle permet à une application de fournir à l'utilisateur des capacités de traitement du signal programmable de haut niveau, tout en assurant la plus grande efficacité de l'exécution. De plus la compilation étant très rapide, elle est dans bien des cas transparente pour l'utilisateur pouvant donner l'illusion d'un langage interprété.

Remerciements

Cette recherche est menée dans le cadre du projet INEDIT soutenu par l'Agence Nationale pour la Recherche [ANR-12-CORD-0009] que nous tenons à remercier ici.

References

- [1] Y. Orlarey and D. Fober and S. Letz "FAUST : an Efficient Functional Approach to DSP Programming", *New Computational Paradigms for Computer Music*, Editions DELATOUR FRANCE, 2009.
- [2] D. Fober and Y. Orlarey and S. Letz "FAUST Architectures Design and OSC Support", *IRCAM, (Ed.): Proc. of the 14th Int. Conference on Digital Audio Effects (DAFx-11), Page(s): 231–216*, 2011

4. Csound: un langage de programmation audio : <http://www.csounds.com>

- [3] S. Letz and D. Fober and Y. Orlarey "Work Stealing Scheduler for Automatic Parallelization in Faust", *Linux Audio Conference*, 2010
- [4] Low Level Virtual Machine "Language Reference Manual", llvm.org/docs/LangRef.html
- [5] Web Audio Application Programming Interface "W3C Editor's Draft", dvcs.w3.org/hg/audio/raw-file/tip/webaudio/specification.html