Recherche d'invariants : application au langage FAUST

Antoine Vugliano

Janvier-Février 2014

Résumé FAUST est un DSL [1] dédié au traitement du signal, pensé pour la conception d'applications de traitement et de synthèse du son en temps réel. Les applications FAUST peuvent être exécutées sur toutes sortes de matériels, y compris avec des ressources limitées ; c'est pourquoi contrairement aux autres DSL pour le son, FAUST se compile en C++ afin d'être le plus efficace possible. Durant cette étape de compilation, un certain nombre d'analyses peuvent être effectuées pour optimiser le programme. Ce rapport traite de la façon de calculer des invariants lors du calcul d'intervalles dans FAUST et de l'utilisation de ces invariants dans le processus de compilation.

Mot-clefs FAUST, Analyse statique, Calcul d'intervalles, Langages synchrones

Table des matières

1. Domain Specific Language

1 Introduction

Ce TER a été proposé par Laure Gonnord (Compsys, LIP/ENS Lyon, Université Lyon1) et conjointement encadré par Yann Orlarey (Grame), qui est à l'origine du projet FAUST.

1.1 L'équipe Compsys

COMPSYS [2] est une équipe de recherche commune à l'Inria et au Laboratoire de l'Informatique du Parallélisme (LIP). Elle est localisée à l'Ecole Normale Supérieure de Lyon (ENS Lyon).

L'objectif de Compsys est le développement de techniques de compilation, plus précisément d'optimisations de codes, appliquées aux domaine des systèmes embarqués de calcul (embedded computing systems). Compsys s'intéresse à la fois aux optimisations bas niveau (back-end) pour les processeurs spécialisés et aux transformations de haut niveau (principalement source à source) pour la synthèse d'accélérateurs matériels. Ces optimisations s'attachent à favoriser le parallélisme, et la localité des données. Pour réaliser ces optimisations, l'équipe utilise souvent des informations *statiquement* découvertes lors de phases d'analyse dans les phases d'optimisation qui suivent.

1.2 Grame et le language Faust

Grame Créé par Pierre Alain Jaffrennou et James Giroudon en 1982, Grame est aujourd'hui l'un des six centres constitutifs du réseau des centres nationaux de création musicale, labellisation créée par le Ministère de la Culture en 1997.

La mission principale de Grame est de permettre la conception et la réalisation d'oeuvres musicales nouvelles, dans un contexte de transversalité des arts et de synergie arts - sciences.

FAUST FAUST (Functional AUdio STream) est un langage de programmation fonctionnel conçu pour la synthèse de son en temps réel et le traitement du signal. Il vise à simplifier l'écriture de fonctionnalités de DSP (digital signal processing) en fournissant une syntaxe simple mais expressive, qui convienne particulièrement bien aux ingénieurs du son et aux developpeurs audio.

Il combine un style fonctionnel et une syntaxe en *block-diagram* (voir section 2.1). Les signaux sont représentés comme des fonctions du temps, les processeurs de signaux comme des fonctions d'ordre supérieur qui traitent les signaux, et l'on utilise des opérateurs de composition pour combiner les processeurs.
Grâce à cette syntaxe, le compilateur travaille sur une fonction mathématique décrivant le programme, et est capable de le représenter graphiquement sous la forme d'un block-diagram qui illustre son fonctionnement.

Le compilateur permet de générer le code d'un programme FAUST dans divers langages tels que C, C++, Java ou Javascript puisqu'il se base sur le bytecode LLVM. Cela permet la génération d'un code fortement optimisé, car l'efficacité du code résultant est un objectif principal du langage. De fait, FAUST est disponible sur un grand nombre de plateformes, n'a aucune dépendance et est facilement intégrable. Il fournit aussi une abstraction pour la création d'applications standalones avec interface utilisateur.

1.3 Sujet

Les applications écrites en FAUST seront exécutées sur des machines aux ressources limitées, par exemple des systèmes embarqués. Compte tenu de ces contraintes, l'efficacité du code généré est d'autant plus importante.

Il est donc essentiel d'effectuer un maximum d'analyses sur le code source lors de la compilation pour produire le code le plus performant possible. Durant la phase de compilation, le compilateur FAUST a besoin d'informations sur les intervalles de variation des signaux du programme.

2. Compilation et systèmes embarqués

Le sujet de ce TER consiste à identifier quelles analyses d'intervalles sont actuellement effectuées par le compilateur FAUST, leurs limites, et comment les améliorer. La singularité de ce projet réside dans le fait que FAUST est un langage synchrone et a vocation à traiter des nombres à virgule flottante, là où les analyses statiques travaillent généralement sur des programmes séquentiels (à la C) et considèrent les variables numériques en précision infinie.

2 Contexte

2.1 Block-diagram

Un *block-diagram* (en français schéma fonctionnel) est une représentation d'un système en diagramme composé de blocs et de lignes qui relient les blocs entre eux. Ce type de diagramme est très utilisé en traitement du signal et permet de décrire des circuits de façon transparente : les blocs sont des processeurs de flux et les lignes relient les entrées et sorties des différents blocs.

Yann Orlarey *et al* [1] proposent une approche algébrique à la construction de block-diagrams qui se base sur des opérateurs de composition : séquentiel, parallèle et récursif. Ces opérateurs permettent de chaîner les blocs avec une gestion fine des entrées et des sorties. Il devient très facile à partir de là de décrire des block-diagrams complexes de façon textuelle.

2.2 Problématique

FAUST est basé sur cette syntaxe de block-diagram, et sur la programmation fonctionnelle. Un programme FAUST définit un processeur de signal (ici sonore), potentiellement composé de plusieurs sous-processeurs assemblés au moyen des opérateurs sus-cités. On peut faire un parallèle entre les processeurs et des fonctions d'ordre supérieur, les signaux correspondant à des fonctions du temps.

En réalité, le programme travaille sur des flots de valeurs (les entrées du programme) et calcule ses valeurs de sortie à l'aide d'opérateurs appliqués sur les valeurs d'entrée et des valeurs précédemment calculées.

Le langage intègre certaines fonctionnalités pratiques pour le programmeur comme le fait de mélanger l'interface utilisateur avec le code de traitement. FAUST permet de créer des widgets pour contrôler différentes valeurs (par exemple le volume) qui seront eux aussi vus comme des processeurs émettant une valeur.

Exemple Prenons un exemple simple de code FAUST (représentation textuelle qui peut être écrite telle quelle, ou produite à partir du block-diagramme) qui illustre l'intérêt de l'analyse d'intervalles, et que l'on gardera comme exemple .

```
smooth(c) = *(1-c) : + ~ *(c);
d = hslider("duree", 0, 0, 2000, 1) : smooth(0.9);
process = _@d;
```

FIGURE 1 – Code de l'exemple en FAUST textuel

Le code de la figure 1 prend le signal d'entrée (symbole _) et lui applique un délai (opérateur @) de d frames. Le mot-clef $process$ définit le processeur principal (la fonction principale) du programme.

La fonction $hslider$ crée un slider (c'est à dire un curseur modifiable par l'utilisateur, qui fournit ici une indication de durée) d'intervalle $[0, 2000]$ dans l'interface graphique et renvoie sa valeur à tout moment. $smooth$ quant à elle permet une variation non linéaire entre plusieurs valeurs successives mais son fonctionnement ne nous intéresse pas pour l'instant. Enfin, l'opérateur : permet de chaîner les processeurs, on peut le voir comme la composition de fonctions.

Le but de ce programme est donc de transmettre le son reçu en entrée avec un délai (retard) de d échantillons. Pour implémenter cela, le compilateur FAUST crée un tableau d'échantillons dans le langage cible (généralement C++) assez grand pour contenir le nombre de frames de délai.

Seulement, ce code ne compile pas en l'état. En effet, le compilateur est capable de savoir que la valeur retournée par le $hslider$ est entre 0 et 2000. Cependant, à cause de la définition "récursive" de la procédure $smooth$, le compilateur est incapable de déterminer une plage de variation pour le signal d et donc de créer un tableau de la bonne taille.

Sur cet exemple, si l'on veut arriver à compiler ce code, il est nécessaire de rajouter séquentiellement les processeurs min et max comme dans la figure 2.

```
smooth(c) = *(1-c) : +~*(c);
d = hslider("duree", 0, 0, 2000, 1) : smooth(0.9) : min(2000) : max(0)
process = _@d;
```

FIGURE 2 – Code de l'exemple en FAUST textuel en ajoutant des informations supplémentaires

Notre programme donne le résultat de la figure 3 une fois compilé. Cette application est configurée pour utiliser *ALSA* [3] : l'entrée du programme correspond au microphone et la sortie aux haut-parleurs. Ce que perçoit le microphone de la machine est donc retardé du nombre d'échantillons choisi dans l'interface.

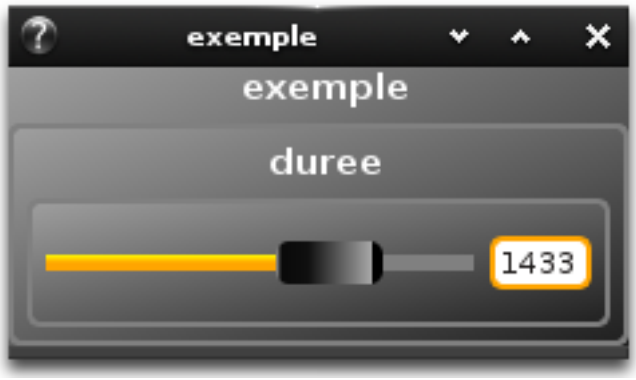


FIGURE 3 – Copie d'écran de l'application finale

Remarque 1. *Cet exemple est inspiré d'applications réelles écrites en FAUST et n'est donc pas un cas d'étude forgé pour illustrer le sujet de recherche. L'analyse d'intervalles a de véritables applications en FAUST et contribuerait à améliorer le processus de compilation pour beaucoup de programmes simples.*

3 État de l'art

3.1 Dans FAUST

Le compilateur FAUST sait calculer les intervalles associés à des expressions de signaux, tant qu'elles n'impliquent pas de récursivité. Cela est suffisant dans la majorité des cas mais lorsqu'un délai récursif est appliqué, comme on l'a vu dans la figure 1, il n'est plus capable de déterminer un intervalle de valeurs. De fait le compilateur ne peut déterminer la taille du tableau à créer pour stocker les frames en mémoire et la compilation échoue. C'est à ce niveau là qu'intervient le besoin de trouver les invariants d'expressions de signaux.

3. Advanced Linux Sound Architecture

4 Pistes

Afin de rechercher une méthode pour calculer les intervalles des signaux dans FAUST, plusieurs façons de faire peuvent être envisagées. Chacune présente une approche différente pour résoudre le problème.

4.1 Dichotomie

Une première proposition de technique de calcul d'invariants par dichotomie a été proposée chez Grame à des fins de test. Le compilateur FAUST est souvent amené à traiter des signaux définis par récurrence, et il a été montré qu'on peut transformer un signal récursif en une équation de point fixe.

Pour notre exemple de la figure 1, il est nécessaire de borner les valeurs renvoyées par $smooth$ et donc de trouver un intervalle qui soit un point fixe de cette fonction.

Le principe est le suivant : en Faust, les équations de recurrence peuvent se ramener à des équations de la forme $X = F(X)$, où X est une variable dénotant l'ensemble des valeurs que peut prendre le signal x au cours du temps, et F une fonction monotone.

On distingue alors trois cas de figure lors de l'application à un intervalle I :

— $I \subseteq F(I)$: I est expansif

— $F(I) \subseteq I$: I est contractant

— $F(I) = I$: I est un point fixe

L'hypothèse est alors qu'un intervalle "légèrement" plus grand que le point fixe va être contractant, et qu'un intervalle plus petit sera expansif.

La dichotomie va consister à diviser l'ensemble des intervalles à chaque itération en rapprochant progressivement les bornes de l'intervalle considéré vers le point fixe. Lorsqu'il est expansif, on va continuer à l'accroître en appliquant F, et de la même façon tant qu'il est contractant, on va le réduire par F. Une fois que l'application de F donne un intervalle assez proche (une différence plus petite qu'un ϵ fixé), on considère avoir trouvé le point fixe.

Initialement cette recherche donnait des résultats assez proches des points fixes réels, mais sous-approximés dans certains cas. Travaillant sur des nombres à virgule flottante, il est inutile de chercher des points fixes exacts, mais des sous-approximations ne conviennent pas non plus, une sur-approximation de l'intervalle est nécessaire ; et j'ai dû chercher un moyen de corriger cette recherche.

Afin de remédier à cela il suffit de toujours appliquer F à l'intervalle tant qu'il est expansif, avant de regarder si l'on est assez proche du point fixe pour arrêter la recherche. De cette manière on est sûr de toujours obtenir un intervalle légèrement plus grand que le point fixe et donc de ne pas avoir de sous-approximation.

Il faut noter que si la recherche aboutissait en seulement quelques itérations pour certains cas, elle pouvait avoir beaucoup plus de difficultés sur d'autres cas pourtant simples. Ce comportement rendait son utilisation en pratique peu réaliste. Par exemple, l'invariant de la fonction $smooth$ de la figure 1 est calculé en 17 itérations, tandis qu'en remplaçant simplement l'addition centrale par une soustraction, on monte à pratiquement 900 itérations.

4.2 Utilisation de méthodes de calcul d'invariants

Le problème de calcul de point fixe est un problème classique en informatique, et il peut être résolu en utilisant par exemple l'interprétation abstraite [2]. Cette méthode consiste à interpréter le programme sur les premières itération de ses boucles, puis à extrapoler le comportement des variables par la suite.

Des outils implémentant ces méthodes existent, avec diverses variantes, comme *Pastis*, en cours de développement par Damien Masse. Cet outil prend en entrée la description d'un programme

sous forme d'automate et permet de calculer les intervalles de valeurs des variables de l'automate (et donc du programme). Il accepte les variables de types *int*, *real* et *bool*, ce qui convient parfaitement aux programmes FAUST.

On veut utiliser un outil existant pour profiter de sa méthode qui sera vraisemblablement assez précise, et on choisit donc pour concevoir un algorithme d'exprimer sous la forme d'un langage simple les expressions FAUST que l'on veut traduire. J'ai pour cela écrit une grammaire décrivant un sous-ensemble minimal du langage FAUST afin de pouvoir ensuite travailler sur une transformation vers des automates.

4.3 Implémentation

Enfin, nous avons considéré la possibilité d'utiliser directement la représentation intermédiaire utilisée dans le compilateur FAUST pour implémenter l'analyse sur les structures de données. Cela avait l'avantage d'amener des résultats pratiques et de ne pas ajouter d'étapes supplémentaires de traduction.

Finalement le choix s'est porté vers l'utilisation d'un outil externe (en l'occurence Pastis) pour l'analyse, et la recherche d'une méthode de traduction de FAUST (réduit) vers un format d'automates, en l'occurence NTS.

L'implémentation dans FAUST aurait été la meilleure solution en dehors de toutes contraintes, mais la durée du stage était trop courte pour se lancer dans cette voie. La recherche dichotomique n'a pas été retenue pour les raisons déjà évoquées.

5 Proposition

5.1 Réalisation

Le choix de la méthode de recherche a été validé avec Y. Orlarey, qui m'a montré la forme textuelle de leur représentation interne et comment l'obtenir. Elle est très proche de ma proposition de grammaire, j'ai donc pu l'utiliser facilement de la même façon que ce que j'avais prévu.

J'ai dans le même temps étudié la structure de données de cette représentation dans le code du compilateur afin d'avoir une bonne compréhension de son fonctionnement. Dans l'idéal, il aurait fallu que je puisse implémenter un algorithme pour obtenir un automate depuis cette structure ; ou un parser qui accepterait sa représentation textuelle. Mais à nouveau pour des questions de temps j'ai simplement travaillé sur la méthode de génération d'automate sans l'implémenter.

5.2 Preuve

Pendant la compilation, un programme FAUST va être simplifié plusieurs fois et passer par plusieurs représentations intermédiaires, chacune enlevant certaines fonctionnalités plus haut niveau afin de garder un sous-ensemble du langage. La représentation qui nous intéresse a totalement déroulé l'expression du programme afin d'obtenir la définition d'un seul processeur qui contient toutes les transformations à appliquer aux signaux. Nous l'appellerons *FAUST signal* et voici sa grammaire :

$$\langle \text{signal} \rangle \quad \models \quad \textit{nombre constant} \mid \textit{entrée} \mid \langle \text{signal} \rangle \oplus \langle \text{signal} \rangle \mid \langle \text{signal} \rangle \ @ \ \langle \text{signal} \rangle \mid \text{let } \textit{identifiant} = \langle \text{signal} \rangle$$

où
— *nombre constant* peut être entier ou à virgule flottante
— *entrée* désigne l'une des entrées principales du programme
— $\oplus \in \{+, -, *, /\}$
— @ est l'opérateur *delay*

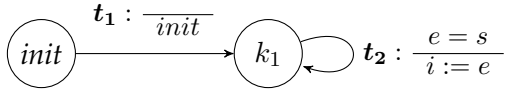
— *let* définit un signal par récursion sur l'*identifiant*. Cela implique que l'expression du signal dépend d'*identifiant*.

Originellement, la grammaire contient aussi la règle suivante :

$$\langle \text{signal} \rangle \quad \models \quad \Pi_k(\langle \text{signal} \rangle)$$

qui permet de sélectionner la k-ème sortie du signal évalué. Cependant, on peut simplifier un processeur à n sorties en n processeurs à 1 sortie. La sélection d'une sortie devient donc inutile.

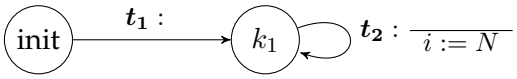
Théorème 1. *Soit s un signal FAUST quelconque, et I_s l'intervalle de variation de ses valeurs. On peut construire un automate à compteur de la forme*



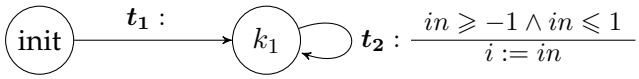
pour lequel le calcul d'invariants détermine I_i, l'intervalle de variation de i, tel que $I_i = I_s$.

Démonstration. On considère un signal s dont la valeur est représentée par la variable i de l'automate. Par induction structurelle sur s on énumère les automates correspondant aux règles de construction de FAUST signal.

$$\langle \text{signal} \rangle \quad \models \quad \textit{nombre constant}$$

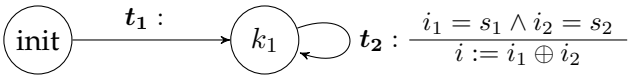


$$\langle \text{signal} \rangle \quad \models \quad \textit{entrée}$$

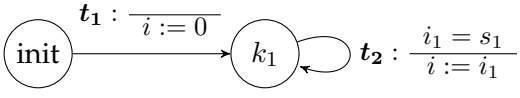


La sémantique du langage FAUST nous garantit que pour toute entrée in_k du programme, $-1 \leq in_k \leq 1$.

$$\langle \text{signal} \rangle \quad \models \quad \langle \text{signal} \rangle \oplus \langle \text{signal} \rangle$$

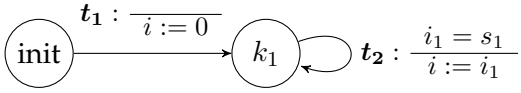


$$\langle \text{signal} \rangle \quad \models \quad \langle \text{signal} \rangle \ @ \ \langle \text{signal} \rangle$$



Lors d'un délai, la valeur précédente à la première itération est 0.

$$\langle \text{signal} \rangle \quad \models \quad \text{let } \textit{identifiant} = \langle \text{signal} \rangle$$



Lors d'une définition récursive, la valeur précédente à la première itération est 0.

Comme on l'a vu précédemment, une définition récursive sur s implique que s soit présent dans son expression. On peut donc écrire que
let $s = s_1$
est équivalent à
let $s = h(f(V_l \setminus s), g(s))$
où :

- $V_l(expr)$ désigne les variables libres de l'expression $expr$
- g est une fonction appliquée au motif récurrent de s (potentiellement l'identité)
- f est une fonction appliquée aux autres variables
- h combine les transformations sur $V_l \setminus s$ et s

Toutes les constructions de base du langage étant définies, il devient ensuite très simple de les combiner à souhait pour obtenir l'automate équivalent à une expression FAUST signal plus complexe. On peut donc obtenir un automate pour toute expression, et ainsi calculer son invariant. De là, tout programme FAUST est analysable par cet algorithme.

□

Conclusion

Nous proposons donc un algorithme qui permet de calculer des invariants pour des programmes FAUST, et en particulier ceux faisant usage de définition récursives, ce qui n'était jusqu'à présent pas possible.

Cet algorithme se base sur une étape intermédiaire de traduction et l'utilisation d'un analyseur externe, ce qui est contraignant pour son implémentation dans le compilateur FAUST. Par ailleurs, l'utilisation d'un outil externe pose quelques limitations, notamment lorsque des propriétés intrinsèques des types de données FAUST (à savoir les *int*, *float* et *double* du C++) sont utilisées, car l'outil travaille avec une précision infinie.

Toutefois il ouvre une voie vers l'intégration d'une technique similaire en prouvant qu'une telle analyse est réalisable en l'état du langage FAUST, et dont l'implémentation est facilement envisageable.

Références

[1] Dominique Fober Yann Orlarey and Stéphane Letz. An algebraic approach to block-diagram construction. In *JIM*, Marseille, France, December 2002.

[2] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th ACM Symposium on Principles of Programming Languages, POPL'78*, Tucson (Arizona), January 1978.