Un dialecte MLIR pour les signaux FAUST

Yann Orlarey, Pierre Cochard, Stéphane Letz

19 juin 2025

1 Introduction

L'objectif de ce document est de proposer un dialecte MLIR dédié à la représentation des signaux FAUST.

Conformément à la sémantique de FAUST, un signal y est défini comme une fonction du temps. Par exemple, un signal réel est représenté par le type (i32) -> f32 et un signal entier par (i32) -> i32.

Un point important abordé par ce document est la traduction des expressions récursives dans le cadre SSA de MLIR. Il est traité en introduisant le bloc `faust.recursive_block`. Ce bloc encapsule les dépendances circulaires et utilise une opération interne, `faust.self_projection`, pour permettre à une définition de faire référence à sa propre sortie future, préservant ainsi la validité de la représentation SSA.

A noter que certains aspects du langage ne sont pas encore abordés, comme par exemple les wavetables, les read-write tables, les foreign functions, ou les fichiers audio. Les nouvelles primitives ondemand, upsampling et downsampling ne sont pas non plus abordées.

2 Signaux FAUST en MLIR

Un signal FAUST est une fonction du temps qui associe à chaque instant une valeur. Les instants sont les entiers de $\mathbb{Z}$ et peuvent donc être négatifs. Par convention, les calculs commencent réellement à partir de l'instant 0 et tout signal, avant l'instant 0, vaut 0. Cela correspond, en termes audio, au fait que les lignes à retard sont toujours initialisées avec du silence.

On distingue typiquement deux types de signaux :

- les signaux entiers : $\mathbb{Z} \to \mathbb{Z}$,
- les signaux réels : $\mathbb{Z} \to \mathbb{R}$.

Les entiers de $\mathbb{Z}$ sont implémentés par des int 32 bits et les réels de $\mathbb{R}$ par des float 32 bits ou 64 bits, suivant l'option donnée au compilateur.

Afin de pouvoir exprimer des signaux mutuellement récursifs, on définit des signaux groupés produisant des tuples de valeurs. Par exemple, le signal stéréo `r` produit par une réverbération stéréo aura pour type: $\mathbb{Z} \to (\mathbb{R} \times \mathbb{R})$. Ces signaux groupés ne sont jamais utilisés en tant que tels, mais toujours via des *projections*. Ainsi `r.0` représente le canal gauche et `r.1` le canal droit de ce signal.

2.1 Types de signaux

Ces types s'expriment de manière native en MLIR. Pour les signaux simples, on aura :

- `(i32)-> i32` pour un signal entier ;
- `(i32)-> f32` ou `(i32)-> f64` pour un signal réel.

Pour les signaux groupés, on aura par exemple :

- `(i32)-> tuple<i32>` pour la sortie d'un compteur entier, utilisé avec la projection `.0`,
- `(i32)-> tuple<f32>` pour la sortie d'un simple IIR, utilisé avec la projection `.0`,
- `(i32)-> tuple<f32,f32>` pour la sortie d'une réverbération stéréo, utilisé avec les projections `.0` et `.1`,
- `(i32)-> tuple<t0,t1,...>` en général avec `ti` = `i32`, `f32`/`f64` et les projections `.0`, `.1`,... .

2.2 Signaux primitifs

Les signaux FAUST primitifs sont :

- les signaux constants
  - entiers: par exemple `27`
  - réels: par exemple `0.5`
- les entrées audios de l'application, par exemple : `input(0)`, `input(1)`,...
- les signaux produits par les éléments d'interface utilisateur :

```
- button("play")
- checkbox("mute")
- hslider("pan", 0, -1, 1, 0.01)
- vslider("gain", 0, 0, 1, 0.01)
```

Leur représentation MLIR est la suivante

Signaux Constants

```
// Constantes entières
%c1 = faust.constant 27 : (i32)->i32
%c2 = faust.constant -42 : (i32)->i32

// Constantes réelles
%c3 = faust.constant 0.5 : (i32)->f32
%c4 = faust.constant 3.141592 : (i32)->f32
```

Entrées audio

```
// Canaux d'entrée audio
%in0 = faust.input 0 : (i32)->f32    // Premier canal
%in1 = faust.input 1 : (i32)->f32    // Deuxième canal
%in2 = faust.input 2 : (i32)->f32    // Troisième canal
```

Éléments d'interface utilisateur

Les éléments d'interface utilisateur génèrent des signaux qui changent de valeur selon les actions de l'utilisateur. Ainsi, quand l'utilisateur presse un bouton, le signal produit par ce bouton passe à 1, puis revient à 0 quand l'utilisateur relâche le bouton.

Les paramètres des sliders sont décrits par des attributs de type `f32` ou `f64` suivant les options données au compilateur.

```
// Bouton (génère 0.0 ou 1.0)
%play = faust.button "play" : (i32)->f32

// Case à cocher (génère 0.0 ou 1.0, état persistant)
%mute = faust.checkbox "mute" : (i32)->f32

// Slider horizontal: label, init, min, max, step
%pan = faust.hslider "pan" {init = 0.0 : f32, min = -1.0 : f32, max = 1.0
    : f32, step = 0.01 : f32} : (i32)->f32

// Slider vertical
%gain = faust.vslider "gain" {init = 0.0 : f32, min = 0.0 : f32, max = 1.0
     : f32, step = 0.01 : f32} : (i32)->f32
```

2.3 Description des signaux récursifs

Les définitions récursives sont essentielles pour exprimer filtres IIR, échos, réverbérations et d'une manière générale tout système de rétroaction.

La traduction MLIR pose des problèmes particuliers pour éviter les dépendances circulaires entre signaux, car MLIR suit une approche SSA (Static Single Assignment) où chaque valeur doit être définie avant d'être utilisée. Ce problème peut être résolu en introduisant des blocs récursifs spécialisés qui encapsulent les références circulaires tout en préservant la forme SSA.

Syntaxe générale

```
%block_name = faust.recursive_block "label" : (i32) -> tuple<T1, T2, ...>
    {
  // Corps du bloc avec références internes
  %self0 = faust.self_projection "label", 0 : (i32) -> T1
  // ...
  faust.yield (%sig1, %sig2, ...) : (i32) -> tuple<T1, T2, ...>
}
```

Éléments clés :

- **`faust.recursive_block`** : Définit un bloc de signaux récursifs avec un nom symbolique
- **`"label"`** : Nom symbolique permettant les auto-références internes
- **`faust.self_projection`** : Accède à l'une des projections d'un bloc en cours de définition
- **`faust.projection`** : Accède à l'une des projections d'un bloc déjà défini
- **`faust.yield`** : Retourne le tuple de signaux produit par le bloc

Exemple : Compteur récursif

Le défi principal est de traduire des expressions récursives du type `y(t)= 1 + y(t-1)` où `y` apparaît des deux côtés de l'équation. En effet, en MLIR-SSA, une variable ne peut pas être utilisée avant d'être définie.

Voici comment cette définition peut être exprimée en MLIR grâce à un bloc récursif spécialisé :

```
// Définition de la constante 1
%one = faust.constant 1 : (i32) -> i32

// Bloc récursif minimal
%counter = faust.recursive_block "counter" : (i32) -> tuple<i32> {
  // Projection interne (référence au bloc en cours de définition)
  %y_current = faust.self_projection "counter", 0 : (i32) -> i32

  // Application du délai pour obtenir y(t-1)
  %y_prev = faust.delay %y_current, %one : (i32) -> i32

  // Addition : y(t) = y(t-1) + 1
  %y_next = faust.add %y_prev, %one : (i32) -> i32

  faust.yield (%y_next) : (i32) -> tuple<i32>
}

// Projection externe pour utiliser le compteur
%y = faust.projection %counter, 0 : (i32) -> i32
```

Blocs récursifs imbriqués

Pour des systèmes plus complexes, on peut imbriquer plusieurs blocs récursifs :

```
%outer = faust.recursive_block "outer" : (i32) -> tuple<i32> {
  %outer_ref = faust.self_projection "outer", 0 : (i32) -> i32

  %inner = faust.recursive_block "inner" : (i32) -> tuple<i32> {
    %inner_ref = faust.self_projection "inner", 0 : (i32) -> i32
    %delayed = faust.delay %inner_ref, %delay_amt : (i32) -> i32
    %with_outer = faust.add %delayed, %outer_ref : (i32) -> i32
    faust.yield (%with_outer) : (i32) -> tuple<i32>
  }

  %inner_proj = faust.projection %inner, 0 : (i32) -> i32
  %result = faust.mul %outer_ref, %inner_proj : (i32) -> i32
  faust.yield (%result) : (i32) -> tuple<i32>
}
```

3 Primitives du langage

3.1 Opérations de conversion de type

Les opérations de cast permettent de convertir entre les types numériques de base.

Cast vers entier : `faust.intcast`

Convertit un signal en représentation entière par troncature.

```
%int_signal = faust.intcast %input_signal : (i32) -> i32
```

Paramètres :

- `%input_signal` : Signal d'entrée de type `(i32)-> f32`
- Résultat : Signal entier de type `(i32)-> i32`

Cast vers réel : `faust.floatcast`

Convertit un signal en représentation à virgule flottante.

```
%float_signal = faust.floatcast %input_signal : (i32) -> f32
```

Paramètres :

- `%input_signal` : Signal d'entrée de type `(i32)-> i32`
- Résultat : Signal flottant de type `(i32)-> f32`

L'opération de délai : `faust.delay`

L'opération `faust.delay` est fondamentale en FAUST pour introduire des retards dans les signaux. Le premier argument est le signal à retarder et le deuxième argument est le retard exprimé en nombre entier d'échantillons. Si x est le signal que l'on veut retarder et y le retard, alors le signal résultant z est tel que $z(t) = x(t - y(t))$.

Pour que le retard soit valide, il faut qu'il soit entier $y(t) \in \mathbb{N}$, positif et borné : $\exists m \in \mathbb{N}$ tel que $0 \leq y(t) \leq m$

```
%delayed_signal = faust.delay %input_signal, %delay_amount : (i32) -> T
```

Paramètres :

- `%input_signal` : Le signal à retarder de type `(i32)-> T`
- `%delay_amount` : La quantité de délai, également un signal de type `(i32)-> i32`
- Résultat : Signal retardé de type `(i32)-> T`

3.2 Opérations arithmétiques

Les opérations arithmétiques de base permettent de combiner et transformer les signaux numériquement. Elles requièrent que les deux opérandes soient du même type. Il n'y a pas de promotion automatique - les conversions de type doivent être explicites.

Addition : `faust.add`

```
%result = faust.add %signal1, %signal2 : (i32) -> T
```

Soustraction : `faust.sub`

```
%result = faust.sub %signal1, %signal2 : (i32) -> T
```

Multiplication : `faust.mul`

```
%result = faust.mul %signal1, %signal2 : (i32) -> T
```

Division : `faust.div`

```
%result = faust.div %signal1, %signal2 : (i32) -> f32
```

Spécificité : La division produit toujours un résultat de type `f32`, même si les deux opérandes sont entiers.

```
// Division entière : résultat automatiquement en float
%int1 = faust.constant 7 : (i32) -> i32
%int2 = faust.constant 3 : (i32) -> i32
%result = faust.div %int1, %int2 : (i32) -> f32  // Résultat = 2.333...
```

Modulo : `faust.mod`

```
%result = faust.mod %signal1, %signal2 : (i32) -> T
```

3.3 Opérations unaires

Valeur absolue : `faust.abs`

```
%result = faust.abs %signal : (i32) -> T
```

Négation : `faust.neg`

```
%result = faust.neg %signal : (i32) -> T
```

Inverse : `faust.inv`

```
%result = faust.inv %signal : (i32) -> f32  // 1/x
```

3.4 Opérations de comparaison

Toutes les opérations de comparaison produisent un signal entier (0 pour faux, 1 pour vrai).

Égalité : `faust.eq`

```
%result = faust.eq %signal1, %signal2 : (i32) -> i32
```

Inégalité : `faust.ne`

```
%result = faust.ne %signal1, %signal2 : (i32) -> i32
```

Inférieur : `faust.lt`

```
%result = faust.lt %signal1, %signal2 : (i32) -> i32
```

Inférieur ou égal : `faust.le`

```
%result = faust.le %signal1, %signal2 : (i32) -> i32
```

Supérieur : `faust.gt`

```
%result = faust.gt %signal1, %signal2 : (i32) -> i32
```

Supérieur ou égal : `faust.ge`

```
%result = faust.ge %signal1, %signal2 : (i32) -> i32
```

3.5 Opérations logiques et binaires

ET logique : `faust.and`

```
%result = faust.and %signal1, %signal2 : (i32) -> i32
```

OU logique : `faust.or`

```
%result = faust.or %signal1, %signal2 : (i32) -> i32
```

OU exclusif : `faust.xor`

```
%result = faust.xor %signal1, %signal2 : (i32) -> i32
```

NON logique : `faust.not`

```
%result = faust.not %signal : (i32) -> i32
```

Décalage à gauche : `faust.lsh`

```
%result = faust.lsh %signal, %shift : (i32) -> i32
```

Décalage à droite : `faust.rsh`

```
%result = faust.rsh %signal, %shift : (i32) -> i32
```

3.6 Fonctions trigonométriques

Sinus : `faust.sin`

```
%result = faust.sin %signal : (i32) -> f32
```

Cosinus : `faust.cos`

```
%result = faust.cos %signal : (i32) -> f32
```

Tangente : `faust.tan`

```
%result = faust.tan %signal : (i32) -> f32
```

3.7 Fonctions trigonométriques inverses

Arc sinus : `faust.asin`

```
%result = faust.asin %signal : (i32) -> f32
```

Arc cosinus : `faust.acos`

```
%result = faust.acos %signal : (i32) -> f32
```

Arc tangente : `faust.atan`

```
%result = faust.atan %signal : (i32) -> f32
```

Arc tangente à deux arguments : `faust.atan2`

```
%result = faust.atan2 %y, %x : (i32) -> f32
```

3.8 Fonctions hyperboliques

Sinus hyperbolique : `faust.sinh`

```
%result = faust.sinh %signal : (i32) -> f32
```

Cosinus hyperbolique : `faust.cosh`

```
%result = faust.cosh %signal : (i32) -> f32
```

Tangente hyperbolique : `faust.tanh`

```
%result = faust.tanh %signal : (i32) -> f32
```

3.9 Fonctions hyperboliques inverses

Arc sinus hyperbolique : `faust.asinh`

```
%result = faust.asinh %signal : (i32) -> f32
```

Arc cosinus hyperbolique : `faust.acosh`

```
%result = faust.acosh %signal : (i32) -> f32
```

Arc tangente hyperbolique : `faust.atanh`

```
%result = faust.atanh %signal : (i32) -> f32
```

3.10 Fonctions exponentielles et logarithmiques

Exponentielle : `faust.exp`

```
%result = faust.exp %signal : (i32) -> f32
```

Logarithme naturel : `faust.log`

```
%result = faust.log %signal : (i32) -> f32
```

Logarithme base 10 : `faust.log10`

```
%result = faust.log10 %signal : (i32) -> f32
```

Puissance : `faust.pow`

```
%result = faust.pow %base, %exponent : (i32) -> f32
```

Racine carrée : `faust.sqrt`

```
%result = faust.sqrt %signal : (i32) -> f32
```

3.11 Fonctions d'arrondi et de troncature

Plafond : `faust.ceil`

```
%result = faust.ceil %signal : (i32) -> f32
```

Plancher : `faust.floor`

```
%result = faust.floor %signal : (i32) -> f32
```

Arrondi : `faust.round`

```
%result = faust.round %signal : (i32) -> f32
```

Arrondi vers l'entier le plus proche : `faust.rint`

```
%result = faust.rint %signal : (i32) -> f32
```

3.12 Fonctions de sélection

Minimum : `faust.min`

```
%result = faust.min %signal1, %signal2 : (i32) -> T
```

Maximum : `faust.max`

```
%result = faust.max %signal1, %signal2 : (i32) -> T
```

Sélection conditionnelle : `faust.select2`

```
%result = faust.select2 %condition, %else_value, %then_value : (i32) -> T
```

Sémantique : Si `%condition != 0`, retourne `%then_value`, sinon `%else_value`.