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. A termes, il doit également servir de spécification à l'implémentation d'un backend MLIR dans le compilateur FAUST.

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 elle même, tout en 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 Types de base et signaux FAUST

Un signal FAUST dénote une fonction du temps. FAUST considère deux types de signaux suivant que la grandeur qui varie dans le temps soit entière ou réelle :

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

Le temps est représenté par $\mathbb{Z}$, car il est discret et potentiellement négatif.

2.1 Types de base

Pour rester proche de la spécification formelle, le dialecte `faust` introduit deux types de base :

- `!faust.integer` qui représente l'ensemble des entiers $\mathbb{Z}$;
- `!faust.real` qui représente l'ensemble des réels $\mathbb{R}$.

Lors des phases de *lowering*, ces types pourront être traduits vers des types machine concrets, par exemple :

- `!faust.integer` vers `i32`;
- `!faust.real` vers `f32` ou `f64`.

2.2 Types signaux

Les types signaux s'expriment de la manière suivante :

- `(!faust.integer)-> !faust.integer` pour les signaux entiers ;
- `(!faust.integer)-> !faust.real` pour les signaux réels.

Pour modéliser des groupes de signaux qui peuvent être mutuellement récursifs, on introduit le type signal multicanal :

- `(!faust.integer)-> tuple<t0,t1,...>`

avec `ti` = `!faust.integer` ou `!faust.real`

Ainsi, le signal produit par une réverbération stéréophonique pourra être modélisé par :

- `(!faust.integer)-> tuple<!faust.real, !faust.real>`

  et le signal d'un simple IIR, par :

- `(!faust.integer)-> tuple<!faust.real>`

Un système de projection permet d'accéder aux canaux individuels d'un signal multicanal.

2.3 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 : (!faust.integer)->!faust.integer
%c2 = faust.constant -42 : (!faust.integer)->!faust.integer

// Constantes réelles
%c3 = faust.constant 0.5 : (!faust.integer)->!faust.real
%c4 = faust.constant 3.141592 : (!faust.integer)->!faust.real
```

Entrées audio

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

```
// Bouton (génère 0.0 ou 1.0)
%play = faust.button "play" : (!faust.integer)->!faust.real

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

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

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

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

Éléments clés :

- **`faust.recursive_block "label"`** : Définit un bloc de signaux récursifs avec un label symbolique permettant les auto-références internes ;
- **`faust.self_projection "label", chan`** : Accède au canal `chan` du bloc en cours de définition ayant le label `"label"` ;
- **`faust.yield`** : Retourne un signal multicanal à partir des canaux qui le constituent ;
- **`faust.projection sig, chan`** : Accède au canal `chan` d'un signal multicanal `sig` déjà définit.

Exemple : Compteur récursif

Voici comment le signal $y(t) = 1 + y(t-1)$ peut être exprimée en MLIR grâce à un bloc récursif spécialisé :

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

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

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

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

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

// Projection externe pour accéder au canal 0 du compteur
%y = faust.projection %counter, 0 : (!faust.integer) -> !faust.integer
```

Blocs récursifs imbriqués

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

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

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

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

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 : (!faust.integer) -> !faust.
    integer
```

Paramètres :

- `%input_signal` : Signal d'entrée de type `(!faust.integer)-> !faust.real`
- Résultat : Signal entier de type `(!faust.integer)-> !faust.integer`

Cast vers réel : `faust.realcast`

Convertit un signal en représentation à virgule flottante.

```
%float_signal = faust.realcast %input_signal : (!faust.integer) -> !faust.
    real
```

Paramètres :

- `%input_signal` : Signal d'entrée de type `(!faust.integer)-> !faust.integer`
- Résultat : Signal flottant de type `(!faust.integer)-> !faust.real`

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 : (!faust.
    integer) -> T
```

Paramètres :

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

Soustraction : `faust.sub`

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

Multiplication : `faust.mul`

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

Division : `faust.div`

```
%result = faust.div %signal1, %signal2 : (!faust.integer) -> !faust.real
```

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

```
// Division entière : résultat automatiquement en float
%int1 = faust.constant 7 : (!faust.integer) -> !faust.integer
%int2 = faust.constant 3 : (!faust.integer) -> !faust.integer
%result = faust.div %int1, %int2 : (!faust.integer) -> !faust.real  // Ré
    sultat = 2.333...
```

Modulo : `faust.mod`

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

3.3 Opérations unaires

Valeur absolue : `faust.abs`

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

Négation : `faust.neg`

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

Inverse : `faust.inv`

```
%result = faust.inv %signal : (!faust.integer) -> !faust.real  // 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 : (!faust.integer) -> !faust.integer
```

Inégalité : `faust.ne`

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

Inférieur : `faust.lt`

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

Inférieur ou égal : `faust.le`

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

Supérieur : `faust.gt`

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

Supérieur ou égal : `faust.ge`

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

3.5 Opérations logiques et binaires

ET logique : `faust.and`

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

OU logique : `faust.or`

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

OU exclusif : `faust.xor`

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

NON logique : `faust.not`

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

Décalage à gauche : `faust.lsh`

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

Décalage à droite : `faust.rsh`

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

3.6 Fonctions trigonométriques

Sinus : `faust.sin`

```
%result = faust.sin %signal : (!faust.integer) -> !faust.real
```

Cosinus : `faust.cos`

```
%result = faust.cos %signal : (!faust.integer) -> !faust.real
```

Tangente : `faust.tan`

```
%result = faust.tan %signal : (!faust.integer) -> !faust.real
```

3.7 Fonctions trigonométriques inverses

Arc sinus : `faust.asin`

```
%result = faust.asin %signal : (!faust.integer) -> !faust.real
```

Arc cosinus : `faust.acos`

```
%result = faust.acos %signal : (!faust.integer) -> !faust.real
```

Arc tangente : `faust.atan`

```
%result = faust.atan %signal : (!faust.integer) -> !faust.real
```

Arc tangente à deux arguments : `faust.atan2`

```
%result = faust.atan2 %y, %x : (!faust.integer) -> !faust.real
```

3.8 Fonctions hyperboliques

Sinus hyperbolique : `faust.sinh`

```
%result = faust.sinh %signal : (!faust.integer) -> !faust.real
```

Cosinus hyperbolique : `faust.cosh`

```
%result = faust.cosh %signal : (!faust.integer) -> !faust.real
```

Tangente hyperbolique : `faust.tanh`

```
%result = faust.tanh %signal : (!faust.integer) -> !faust.real
```

3.9 Fonctions hyperboliques inverses

Arc sinus hyperbolique : `faust.asinh`

```
%result = faust.asinh %signal : (!faust.integer) -> !faust.real
```

Arc cosinus hyperbolique : `faust.acosh`

```
%result = faust.acosh %signal : (!faust.integer) -> !faust.real
```

Arc tangente hyperbolique : `faust.atanh`

```
%result = faust.atanh %signal : (!faust.integer) -> !faust.real
```

3.10 Fonctions exponentielles et logarithmiques

Exponentielle : `faust.exp`

```
%result = faust.exp %signal : (!faust.integer) -> !faust.real
```

Logarithme naturel : `faust.log`

```
%result = faust.log %signal : (!faust.integer) -> !faust.real
```

Logarithme base 10 : `faust.log10`

```
%result = faust.log10 %signal : (!faust.integer) -> !faust.real
```

Puissance : `faust.pow`

```
%result = faust.pow %base, %exponent : (!faust.integer) -> !faust.real
```

Racine carrée : `faust.sqrt`

```
%result = faust.sqrt %signal : (!faust.integer) -> !faust.real
```

3.11 Fonctions d'arrondi et de troncature

Plafond : `faust.ceil`

```
%result = faust.ceil %signal : (!faust.integer) -> !faust.real
```

Plancher : `faust.floor`

```
%result = faust.floor %signal : (!faust.integer) -> !faust.real
```

Arrondi : `faust.round`

```
%result = faust.round %signal : (!faust.integer) -> !faust.real
```

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

```
%result = faust.rint %signal : (!faust.integer) -> !faust.real
```

3.12 Fonctions de sélection

Minimum : `faust.min`

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

Maximum : `faust.max`

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

Sélection par multiplexage : `faust.select2`

L'opération `faust.select2` agit comme un multiplexeur. Elle produit un signal de sortie en sélectionnant, à chaque instant, en fonction de la valeur 0 ou 1 d'un signal de contrôle (c), une valeur parmi deux signaux d'entrée ($s0$, $s1$).

Sémantique :

Soient $c(t)$ le signal de contrôle, et $s0(t)$, $s1(t)$ les deux signaux d'entrée. Le signal de sortie $r(t)$ est défini par :

- $r(t) = s0(t)$ si $c(t) = 0$
- $r(t) = s1(t)$ si $c(t) = 1$

```
// %select_sig doit produire 0 ou 1
%result = faust.select2 %select_sig, %input_for_0, %input_for_1 : (!faust.
    integer) -> T
```