# Fixed-point Format Inference for the FAUST Signal Graph

Beata Burreau
b.burreau@gmail.com

January – July 2021
CITI Lab, INSA Lyon

Supervised by Florent de Dinechin*, Stéphane Letz, Yann Oarley

## 1 Background

All the sounds we make and hear are audio signals, usually a whole bunch of them. An example of such a bunch is the the sound of shouting "Hello World!" out into the void. Let's denote 'the bunch' of signals $(s_1, ..., s_n)$. In reality the signals are continuous, but each audio signal $s$ can be represented as a discrete function of time, $s : \mathbb{Z} \to \mathbb{R}$. At any point in time $t \in \mathbb{Z}$, the audio signal has a value $s(t) \in \mathbb{R}$. The set of all possible signals is then $\mathbb{S} = \mathbb{Z} \to \mathbb{R}$ and our bunch, actually a tuple, a member of the set $\mathbb{S}^n, n \in \mathbb{N}$.

Now if we were to shout "Hello World!" in a ravine we would also hear echoes. The echoes are transformed versions of the original signals and could be denoted $(s'_1, ..., s'_n)$. There is thus some process *echo* producing the echo of a signal such that $echo(s_1) = s'_1$, $echo(s_2) = s'_2$ and so on. $echo : \mathbb{S} \to \mathbb{S}$ is an example of an audio signal processor: something that transforms a set of input audio signals to a set of output signals. More formally, a signal processor $p$ is thus a function $p : \mathbb{S}^n \to \mathbb{S}^m$, $n, m \in \mathbb{N}$ [1].

Faust is a functional programming language for the digital audio processing domain. A Faust program describes an audio signal processor, such as *echo*. From the Faust implementation of some signal processor $p$, it is possible to generate the corresponding implementation of $p$ in a number of programming

*Official internship supervisor

languages. The FAST [2] project aims to extend the Faust compiler with code generation for FPGA-based platforms; an extension that would enable high-level programming – in Faust – of *ultra-low-latency* audio signal processors.

The signals described above, $S$, are discrete-time signals i.e. sequences of real numbers. Some real numbers have an infinite number of decimals, but within the limited memory space of a computer such numbers cannot be stored in their entirety. Instead, a finite approximation of the reals are used. For this task, floating-point number formats are commonly used. They are versatile and provide high accuracy – at a computational cost in time, power and silicone. In some cases, like *ultra-low-latency* audio signal processing, a more resource efficient alternative to floating-point numbers is desirable. This is when the fixed-point numbers enter the stage.

Faust uses floating-point numbers, but in programs for FPGA-based platforms we would like to use fixed-point numbers when appropriate. So, extending the Faust compiler with code generation for FGPA requires a method for determining sensible fixed-point formats for the signal processor described by a Faust program. A fixed-point format is defined by two integers $m, l \in \mathbb{Z}$, the positions of its Most and Least Significant Bits, abbreviated MSB and LSB. A. Dudermel, a previous intern, begun the work on fix-point format determination [3] and his report provides a more in-depth introduction to the floating- and fixed-point formats.

My work has been focused on the LSB, i.e. the number determining the decimal precision of the number format. The LSB and precision are inversely correlated; the smaller the LSB the higher precision. Remembering the domain, audio signal processing, the precision with which the signals are processed may actually affect the sound. When a fixed-point format offers less precision than required for a certain number, one has to apply rounding, which could affect how the signal sounds. On the other hand, a format more precise than necessary is a waste of resources. So, is there a general method for determining what 'just enough precision' is for for Faust signals?

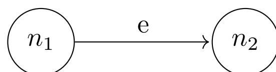
# 2 The Faust Signal Graph

The signal graph is an intermediate representation of an audio processor used in the Faust compiler. It is a directed graph where the nodes represent Faust primitives and the edges represent signals. Formally, any signal $S$ can be represented as a pair of sets, the nodes and edges, as defined below.

$$S = (N, E) \quad E \subseteq \{(n_1, n_2) \mid n_1, n_2 \in N^2,\ n_1 \neq n_2\} \tag{1}$$

The edges are directed, meaning that the edge $(n_1, n_2)$ spans from $n_1$ to $n_2$ as illustrated in figure 1. In a Faust signal graph, the edge represents a signal that is the output of the primitive $n_1$ and the argument of $n_2$.

Figure 1: Two nodes $n1$ and $n2$ and a directed edge $e = (n1, n2)$




## 2.1 The Faust Primitives

Primitives are elementary audio processors predefined in Faust. For instance, the numerical operator + is a signal processor of the type $\mathbb{S}^2 \to \mathbb{S}^1$. It takes two argument signals $s_0, s_1 \in \mathbb{S}$ and produces the output signal $s(t) = s_0(t) + s_1(t)$. Even integer numbers are signal processors. Take the integer 2, it is a signal processor with no arguments and one output signal such that $\forall t \in \mathbb{Z}.\ s(t) = 2$, so integers are signal processors of type $\mathbb{S}^0 \to \mathbb{S}^1$ [1]. The Faust primitives are listed in figure 2 below. More thorough descriptions of the primitives is provided in the Faust syntax documentation [4].

Figure 2: The Faust primitives, P

$$P\ ::=\ n \mid c \mid op \mid @ \mid _ \mid\ !\quad [5]$$

| | |
|---|---|
| **n** | numbers, integers or real numbers |
| **c** | UI elements (slider, button etc.) |
| **op** | numerical operations |
| **@** | the delay operation |
| **_** | the identity function |
| **!** | the cut operation |

## 2.2 Argument Order

So, the Faust primitives are all signal processors. We've defined signal processors as functions $p : \mathbb{S}^n \to \mathbb{S}^m$, $n, m \in \mathbb{N}$ [1] and for $n \geq 2$ the function arguments can be ordered in multiple ways. For example, a signal processor of type $\mathbb{S}^2 \to \mathbb{S}^1$ takes two argument signals $s_0, s_1 \in \mathbb{S}$ and these can be ordered in two ways: $(s_0, s_1)$ and $(s_1, s_0)$. For non-commutative operators, like minus and the delay operator, different argument orders may produce different output signals. The argument order is therefore reflected in the signal graph by a natural number $i \in \mathbb{N}$ associated with each edge. Extending definition (1) with this argument enumeration gives definition (2) below. Pay attention to the fact that $N$ is the set of nodes and $\mathbb{N}$ that of natural numbers.

$$S = (N, E) \quad E \subseteq \{(n_1, n_2, i) \mid n_1, n_2 \in N^2,\ i \in \mathbb{N},\ n_1 \neq n_2\} \tag{2}$$

In the following chapters, edges will be denoted $(n_1, n_2, i)$ only when the value of $i$ is relevant. Otherwise the shorthand notation $(n_1, n_2)$ will be used.

In a well-formed signal graph, the incoming edges to a node of indegree $d \in \mathbb{N}$ each have a unique argument number between 0 and $d$ - 1. This is ensured by the logical formulas (3) and (4) below, where $deg^-$ denotes the indegree of a node.

$$\forall\, (n_a, n_b, i), (n_c, n_d, j) \in E^2\ . \quad n_b = n_d\ \wedge\ i = j\ \Longrightarrow\ n_a = n_c \tag{3}$$

$$\forall\, (n_a, n_b, i) \in E\ . \quad i\ \leq\ deg^-(n_b) - 1 \tag{4}$$

## 2.3 Recursion

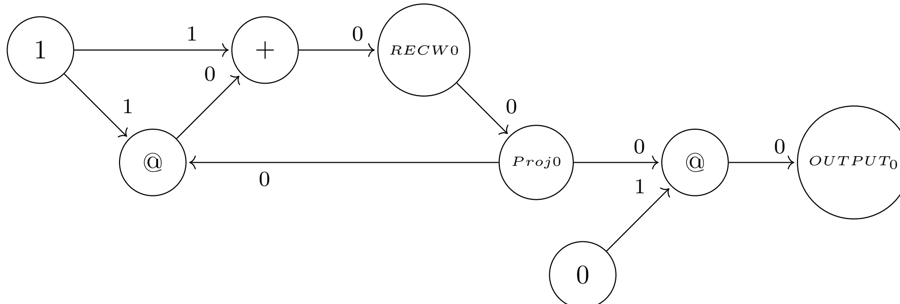
Faust being a functional language, recursion is a common element in the programs and introduced using the operator $\sim$. The recursive composition operation [4] is not a primitive operation like those described in section 2.1. Therefore, it is not represented by a node in the signal graph but instead unfolds into a cycle including primitive operations like the delay operation. Best explained by example, consider the recursive Faust program in figure 3 below. The program describes a signal processor $p : \mathbb{S}^0 \to \mathbb{S}^1$ with output signal $s \in \mathbb{S}$ such that $s(t) = s(t-1) + 1$

Figure 3: A recursive Faust program

```
process = _ ~ (+1);
```

The corresponding signal graph[1] is shown in figure 4 below, where the recursive composition is represented by the cycle $Proj0 \rightarrow @ \rightarrow + \rightarrow RECW0 \rightarrow Proj0$.

Figure 4: The signal graph for the Faust program $process = _ \sim (+1)$
The node labels are primitive Faust operations and the edge labels the signals' argument numbers for the operations




## 3 Finding the LSB

Given the precision of the argument signals to an elementary processor (if it has any) and the kind of processor, it is for most processors possible to compute the precision required to keep its output signal exact. Let's look at a few examples, using $lsb_i$ to denote the LSB of an argument signal $s_i$, $i$ its argument number, and $lsb_{out}$ for the output signal's LSB. Boolean operations will always output integer values (1 or 0) requiring $lsb_{out} = 0$. Addition of two signals $s_0, s_1 \in \mathbb{S}$ in fixed-point formats with some LSBs $lsb_0$ and $lsb_1$ requires $\min(lsb_0, lsb_1)$ for the output signal since there is an inverse correlation between LSB and precision; the smallest LSB gives the highest precision. For other operations, like division, it is not as clear-cut. Take the division of two argument signals $s_0, s_1 \in \mathbb{S}$ with $lsb_0 = lsb_1 = 0$. If $s_0/s_1$ comes out even $lsb_{out} = 0$ is sufficient, but if their quotient is e.g. $0.1_{10}$ for which the binary representation has an infinite number of decimals, no LSB is small enough to hold the exact value of the signal. Thus for an operation like division, one has to make a qualified guess for an $lsb_{out}$ value that will keep the output signal precise enough, knowing that it may entail some rounding.

[1] A signal graph description in DOT language can be generated for a Faust program with the command `faust -sg filename.dsp`

The Faust primitives can be divided into two types of processors:

1. $p : \mathbb{S}^0 \to \mathbb{S}^m, m \in \mathbb{N}$
   These are numbers, UI elements and input signals and are in the signal graph represented by nodes of indegree 0, i.e. sources.

2. $p : \mathbb{S}^n \to \mathbb{S}^m, n, m \in \mathbb{N}, n \geq 1$
   These are numerical operations, the delay operation, identity function and cut operation and are represented by nodes of indegree 1 or greater.

The output − represented by $OUTPUT_0$ in the signal graph (4) − is not exactly a primitive, but nonetheless a processor of type:

3. $p : \mathbb{S}^1 \to \mathbb{S}^0$
   This is the node of outdegree 0 in the signal graph, i.e. the sink.

For the first group of primitives, the precisions of their output signals are known and for the second they can be computed given the precisions of the argument signals. Thus it is possible to infer the LSB for all signals in the graph for which it is not already known, starting in the sources of the graph and working towards the sink.

From the inference we obtain a precision for the output signal of the graph, i.e. the argument signal to the output processor of type 3. above, but the actual precision of the output signal is also known. If $lsb_{actual} > lsb_{inferred}$ for the output signal, the internal precision of the signal graph is unnecessarily high; in the end, the value of the output signal will be rounded to fit the actual fixed-point format. There is thus a possibility to lower the internal precision and save computational resources by propagating the actual, bigger LSB backwards in the graph. This process is here called *trimming* and a kind of inverse LSB inference.

We arrive at the following process.

1. **Infer LSBs**
   Given the LSB for the output signals from processors of type 1 above, propagate the precision forward in the signal graph to the output signal

   If the inferred LSB for the output signal is greater than the actual, then

2. **Trim LSBs**
   Trim the LSB for the output signal by setting it to the actual output LSB. Propagate the decreased precision backwards towards the input signals

## 3.1 Signal Graph Traversal

An intuitive way to traverse a signal graph for LSB inference would be to, for an arbitrary $node \in N$:

If all argument signals to the node have LSBs, compute the LSB for its output signal, then infer the LSB for the node with that output signal as argument. Otherwise infer LSBs for the argument signals that lack it.

However, the signal graph often contains cycles which the above method cannot handle. Once within a cycle, one will never reach a node with LSBs for all its arguments and thus loop forever. The signal graph needs to be traversed in such a manner that covers all edges - to infer the LSBs for the whole graph - and avoids getting stuck in a loop. This requires a more complex method than the one proposed above; one that detects when looping and then infers the LSB of the current node's output signal despite not knowing the LSBs of all argument signals. This means that the precision of signals within a loop needs to be fixed and predefined, since it cannot be inferred.

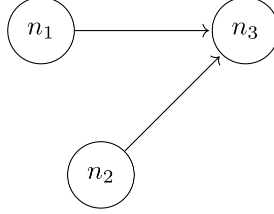
The method enforced in the LSB inference and trimming algorithms is to traverse the signal graph backwards, i.e. starting in the sink node and going in the opposite direction to the edges, while keeping track of the path to detect loops. Operating on a node level, a loop is detected if visiting the current node from a node that it's been visited from before. The algorithm has then followed a cyclic path in the graph and should not continue further backwards. The node is treated as a sink, for which the precision of the input signals is known, and the algorithm infers the LSB of the output signal and then visits the node that it came from.

## 3.2 LSB Inference and Trimming

Before presenting the LSB algorithms, let's introduce some terminology and complementary set definitions. The heads of the incoming edges to a node are its predecessors, denoted $P_{node}$. A predecessor node is considered unvisited if it has not been visited *from the current node.* A node's unvisited predecessors are denoted $P'_{node}$. The path is kept on a node level as a list of successors, denoted $S_{node}$, i.e. the tails of the outgoing edges from a node that it node has been visited from. The list is initially empty, and when visiting a node the successor is added at the head of the list. The set of predecessors is static for a particular graph, but the successors and with it the unvisited predecessors change as the path progresses. To exemplify, the predecessors of $n_3$ in figure 5 are $n_1$ and $n_2$, written $P_{n_3} = \{n_1, n_2\}$ and in the case that we have a path $n_3 \rightarrow n_1,\ n_3$

is a successor of $n_1$ but not $n_2$ and so $S_{n_1} = [n_3]$, $S_{n_2} = [\,]$ and the unvisited predeccessors of $n_3$ would be $n_2$, i.e. $P'_{n_3} = \{n_2\}$.

Figure 5: Three nodes $n_1, n_2$ and $n_3$ such that $n_1$ and $n_2$ are the predecessors of $n_3$




Complementing definition (2) of a signal graph, we formalize the terms *predecessors* ($P$), *successors* ($S$) and *unvisited predecessors* ($P'$) of a node:

$$P_{node} = \{p \mid p, node \in N,\ (p, node) \in E\} \tag{5}$$

$$S_{node} \subseteq \{s \mid s, node \in N,\ (node, s) \in E\}^2 \tag{6}$$

$$P'_{node} = \{p' \mid p' \in P_{node},\ node \notin S_{p'}\} \tag{7}$$

The LSBs of a signal graph can be represented as a set of tuples of an edge, a natural number LSB and a boolean value trimmed. The set is initially empty and members are added with the progression of the LSB inference. As the trimming algorithm progresses, the size of the set is constant but the values of *lsb* and *trimmed* may change for some members.

$$LSBs \subseteq \{(e, lsb, trimmed) \mid e \in E,\ lsb \in \mathbb{Z},\ trimmed \in \{True, False\}\} \tag{8}$$

With inspiration from the Haskell data type Maybe [6], we define the collection of sets $M_n$:

$$M_n = \{Nothing\} \cup \mathbb{Z}^n,\ n \in \mathbb{Z}^+ \tag{9}$$

[2] A node's successors must be an ordered collection to represent a path, but $S$ here is the corresponding set

#### 3.2.1 LSB Inference Algorithm – from the inputs to the output

Preconditions for the inference algorithm are:

$\forall\, node \in N\ .\quad S_{node} = [\,]$

$LSBs = \emptyset$

**Algorithm 1** LSB inference

```
1:  function inferLSBs(node)
2:      if P'_node ≠ ∅ then
3:          predecessor ∈ P'_node
4:          S_predecessor ← node ++ S_predecessor
5:          inferLSBs(predecessor)
6:      else
7:          S_node ← [s | s ∈ S_node, ((node, s), _, _) ∉ LSBs]
8:          if S_node ≠ [ ] then
9:              successor ← S_node[0]
10:             lsb ← computeLSB(node)
11:             if lsb ∈ ℤ then
12:                 LSBs ← LSBs ∪ {((node, successor), lsb, False)}
13:                 inferLSBs(successor)
14:             else
15:                 return
16:         else
17:             return
```

*computeLSB* is a function that computes the precision of the given node's output signal based on its processor type and the LSBs of its argument signals. The return value is a member of $M_1$. For primitives of type $\mathbb{S}^n \to \mathbb{S}^0$, *Nothing* is returned and for primitives of type $\mathbb{S}^n \to \mathbb{S}^m, m \geq 1$ an integer value.

### 3.2.2 LSB Trimming Algorithm – from the output to the inputs

Preconditions for the trimming algorithm are:

$\forall\, node \in N \;.\quad S_{node} = [\,]$

$\forall\, edge \in E \quad \exists\,(e, lsb, trimmed) \in LSBs \;.\quad edge = e,\ lsb \in \mathbb{Z}$

---

**Algorithm 2** LSB trimming

---

1: **function** TRIMLSBS($node$)
2:     **if** $P'_{node} \neq \emptyset$ **then**
3:         $predecessor \in P'_{node}$
4:         **if** $\{(_, node) \mid ((_, node), _, False) \in LSBs\} \neq \emptyset$ **then**
5:             $lsbs \leftarrow \text{refineLSB}(node)$
6:             **if** $lsbs \subset \mathbb{N}^2$ **then**
7:                 $LSBs \leftarrow LSBs$
8:                         $\setminus \{((p, node), _, _) \mid ((p, node), _, _) \in LSBs\}$
9:                         $\cup \{((p, node, i), lsb_i, True) \mid p \in P_{node},\ lsb_i \in lsbs\}$
10:                 $S_{predecessor} \leftarrow node +\!+\, S_{predecessor}$
11:                 trimLSBs($predecessor$)
12:             **else**
13:                 **if** $S_{node} \neq [\,]$ **then**
14:                     $successor \leftarrow S_{node}[0]$
15:                     trimLSBs($successor$)
16:                 **else**
17:                     return
18:         **else**
19:             $S_{predecessor} \leftarrow node +\!+\, S_{predecessor}$
20:             trimLSBs($predecessor$)
21:     **else**
22:         **if** $S_{node} \neq [\,]$ **then**
23:             $successor \leftarrow S_{node}[0]$
24:             trimLSBs($successor$)
25:         **else**
26:             return         $\triangleright$ $lsb_i$ is shorthand notation for $(i, lsb) \in \mathbb{N}^2$

---

*refineLSB* is a function that computes the precision of the given node's argument signals based on its processor type and the current LSBs of its argument signals and output signal. If the argument signals' current LSBs produce an output LSB smaller than the actual, the argument signals' LSBs should be refined i.e. increased. The difference in output LSBs, *actual LSB − computed LSB from argument signals' LSBs*, is then distributed over the argument signals' LSBs. The return value is then a collection members of $M_2$; pairs of argument numbers and the corresponding new LSB for the argument. For primitives of

type $\mathbb{S}^0 \rightarrow \mathbb{S}^m$ or primitives for which the argument LSBs can remain the same, *Nothing* is returned.

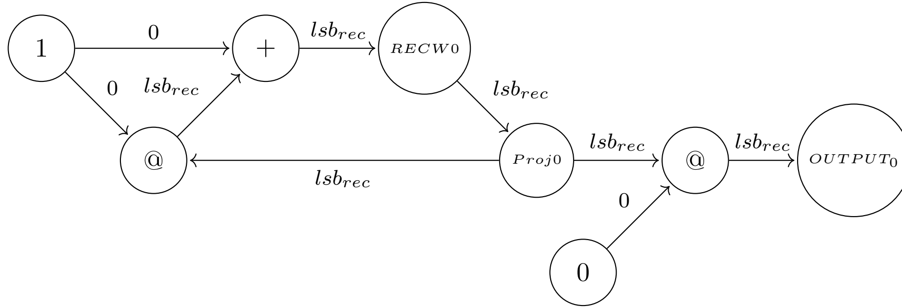
#### 3.2.3 Algorithm Demonstration: A Small Recursive Example

To demonstrate the algorithms described in section 3.2.1 and 3.2.2, they can be run on the recursive signal graph in figure 4 starting in the output node. The actual precision of the output signal is known and here denoted $lsb_{out}$. The precision for signals with loops in recursive graphs is predefined and here denoted $lsb_{rec}$. Both LSBs are integers smaller than or equal to zero.

Running the LSB inference algorithm, the graph is traversed with the following path.
$OUTPUT_0 \rightarrow @ \rightarrow Proj0 \rightarrow RECW0 \rightarrow + \rightarrow @ \rightarrow Proj0 \rightarrow @ \rightarrow 1 \rightarrow @ \rightarrow + \rightarrow 1 \rightarrow + \rightarrow RECW0 \rightarrow Proj0 \rightarrow @ \rightarrow 0 \rightarrow @ \rightarrow OUTPUT_0$

When the algorithm halts, LSBs have been inferred for all signals in the graph and the result is shown as edge labels in figure 6 below.

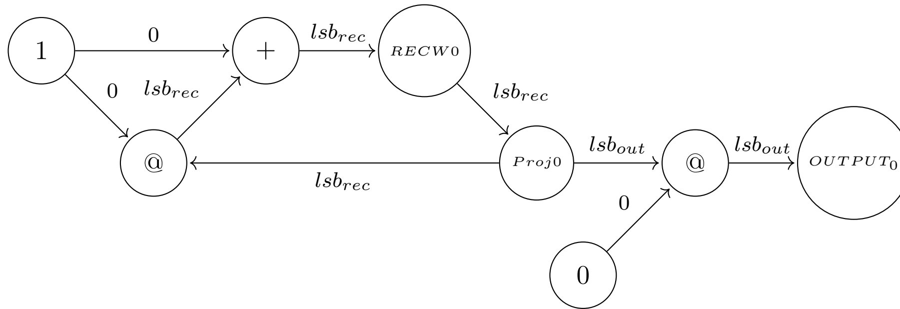
Figure 6: The signal graph for the Faust program *process* = _ ∼ (+1)
The edge labels show the signals' inferred LSBs




With LSBs inferred for the signal graph, the LSB trimming algorithm can be run on it. If $lsb_{out} \geq lsb_{rec}$, the graph is traversed with the following path.
$OUTPUT_0 \rightarrow @ \rightarrow Proj0 \rightarrow @ \rightarrow 0 \rightarrow @ \rightarrow OUTPUT_0$

When the algorithm halts, LSBs have been trimmed for the signals in the graph and the result is shown as edge labels in figure 7 below. Note the trimmed LSBs for the signals represented by edges ($Proj0$, @) and (@, $lsb_{out}$).

Figure 7: The signal graph for the Faust program $process = _ \sim (+1)$
The edge labels show the signals' trimmed LSBs




### 3.3 Implementation

A toy Haskell implementation of the LSB inference and trimming algorithms presented is available at GitHub.

## 4 Conclusion

It is possible to systematically infer LSBs for the Faust signal graphs using the LSB inference and trimming algorithms presented here. With most 'interesting' graphs being cyclic due to recursion, it requires a predefined, fixed LSB for the signals within the cycles. The choice of a sufficiently small LSB is lies on the user, and what constitutes 'sufficiently large' differs between signal processors. This means that with fixed-point formats for the signals within cycles, there is a risk of rounding errors accumulating within the cycle and affecting the sound of the processor's output signal. To avoid this, one could instead use fixed-point formats for all signals outside of the cycles and floating-point within the cycles. However, this approach was not investigated within the scope of this project but would entail LSB inference from the sources up until the cycle(s) and a backwards inference, similar to the LSB trimming, from the sinks to the cycle(s).

## References


[1] GRAME CNCM, *Signal processor semantic.* [Online]. Available: `https://faustdoc.grame.fr/manual/introduction/#signal-processor-semantic` (visited on 06/27/2022).

[2] GRAME CNCM, CITI Lab, and LMFA, *Fast project.* [Online]. Available: `https://fast.grame.fr` (visited on 06/27/2022).

[3] A. Dudermel, "Signal annotation for fixed-point faust targets," 2021.

[4] GRAME CNCM, *Faust syntax, Recursive composition.* [Online]. Available: `https://faustdoc.grame.fr/manual/syntax/#primitives` (visited on 07/12/2022).

[5] Equipe Emeraude, "La compilation de programmes faust avec ondemand," 2022.

[6] HaskellWiki, *Maybe.* [Online]. Available: `https://wiki.haskell.org/Maybe` (visited on 07/20/2022).