On-demand Computations in Faust

[YO, Preliminary draft v5]

1. Introduction

There are requests for using Faust in a more *composition-oriented* way. For that purpose, it has been suggested to introduce *on-demand* computations. In this model, computations are not performed on a sample-by-sample basis, as is normally the case with Faust, but only on request. Conceptually, these requests are propagated backwards, starting from the output of the expression and going back to the inputs.

The challenge is to introduce *on-demand* computations while keeping the simple and well-defined *signal processor* semantics of Faust. In this note we propose a new `ondemand`$(P) \rightarrow P'$ primitive that transforms a signal processor P into an *on-demand* signal processor P', and we define its semantics. As we will see, this semantics can be expressed using the regular Faust semantics, but applied to *downsampled* signals.

2. Semantics of Faust Expressions

Before describing the particular semantics of `ondemand`(), let's take a look at the semantics of Faust in general. The main notions are *signals* and *signal processors*. A *signal* is a function of time and a *signal processor* is a function of signals. A Faust program describes a *signal processor*. Programming in Faust is essentially *combining signal processors* together, for example using composition operators like : or $\sim$.

2.1. Time

Time in Faust is discrete and it is represented by $\mathbb{Z}$. All computations start at time 0, but negative $times are possible in order to take delay operations into account .

2.2. Signals

A *signal* s is a function of time $s : \mathbb{Z} \rightarrow \mathbb{R}$. Actually Faust considers two types of signals: *integer signals* $(s : \mathbb{Z} \rightarrow \mathbb{Z})$ and *floating point signals* $(s : \mathbb{Z} \rightarrow \mathbb{Q})$ but this distinction doesn't matter here. The value of a signal s at time t is written $s(t)$.

The set of all possible signals in Faust is $\mathbb{S} \subset \mathbb{Z} \rightarrow \mathbb{R}$. The set $\mathbb{S}$ is a subset of $\mathbb{Z} \rightarrow \mathbb{R}$ because the value of any Faust signal s at a negative time is always 0:
$\forall s \in \mathbb{S}, s(t < 0) = 0$. In operational terms this corresponds to initialising all delay lines with 0s.

2.3. Tuples

A group of n signals (a n-tuple of signals) is written $(s_1, \ldots, s_n) \in \mathbb{S}^n$. The *empty tuple*, single element of $\mathbb{S}^0$ is notated $()$.

2.4. Signal Processors

A *signal processors* $P : \mathbb{S}^n \to \mathbb{S}^m$, is a function that maps a n-tuple of signals to a m-tuple of signals . The set $\mathbb{P} = \bigcup_{n,m} \mathbb{S}^n \to \mathbb{S}^m$ is the set of all possible signal processors.

2.5. Composition operators

The five composition operators of Faust ($<:$ $:$ $:>$, $\sim$) are binary operations on signal processors: $\mathbb{P} \times \mathbb{P} \to \mathbb{P}$.

2.6. Semantic Brackets

In order to distinguish a Faust expression from its *meaning* as a signal processor, we use the semantic brackets notation $[\![\ ]\!]$. For example $[\![+]\!]$ represents the *meaning* of Faust expression $+$, a signal processor that takes two input signals x and y and produces an output signal by adding together the samples of the input signals:

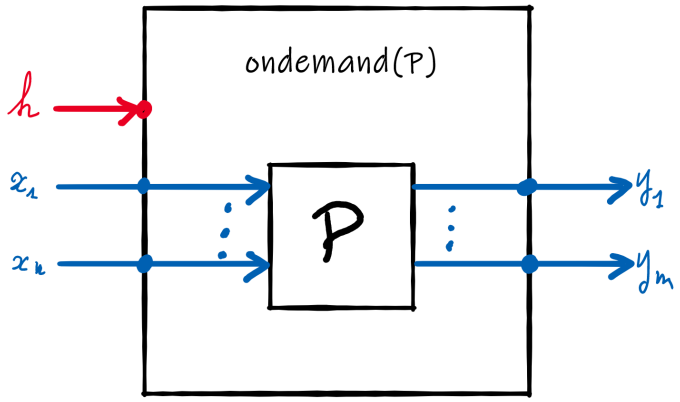
$$[\![+]\!] : \mathbb{S}^2 \to \mathbb{S}^1$$
$$[\![+]\!](x, y) = \lambda t.\, (x(t) + y(t))$$

Numbers are also signal processors. The *meaning* of the Faust expression 1 is the following:

$$[\![1]\!] : \mathbb{S}^0 \to \mathbb{S}^1$$
$$[\![1]\!]() = \lambda t. \begin{cases} 0 & (t < 0) \\ 1 & (t \geq 0) \end{cases}$$

3. The `ondemand`(P) primitive

The vast majority of Faust primitives, like $+$ or `enable`, are operations on *signals*. The `ondemand` primitive is very different. It is an operation on *signal processors* of type $\mathbb{P} \to \mathbb{P}$. It transforms a signal processor P into an on-demand version, as illustrated in the figure below:



If P has n inputs and m outputs, then $\texttt{ondemand}(P)$ has $n+1$ inputs and m outputs. The additional input of $\texttt{ondemand}(P)$ is a clock signal h that indicates by a 1 when there is a computation demand, and by 0 otherwise. In other words, $h(t) = 1$ means that there is a computation demand at time t.

$$\frac{P : n \to m}{\texttt{ondemand}(P) : 1 + n \to m}$$

3.1. The clock signal h

From a clock signal h we can derive a signal h^* that indicates the time of each demand. For example if $h = 1, 0, 0, 1, 0, 0, 0, 1, 0 \ldots$ then $h^* = 0, 3, 7, \ldots$ indicating that the first demand is at time 0, the second one at time 3, the third one at time 7, etc. In other words, h^* translates internal time (time inside $\texttt{ondemand}(P)$) into external time (time outside $\texttt{ondemand}(P)$). We have

$$\begin{aligned} h^*(0) &= \min\{t' | (h(t') = 1)\} \\ h^*(t) &= \min\{t' | (h(t') = 1) \wedge (t' > h^*(t-1))\} \end{aligned}$$

We also derive another signal h^+ that *counts* the number of demands:

$$h^+(t) = \left(\sum_{i=0}^{t} h(i)\right) - 1$$

For the same $h = 1, 0, 0, 1, 0, 0, 0, 1, 0 \ldots$ we have $h^+ = 0, 0, 0, 1, 1, 1, 1, 2, 2, \ldots$. Here h^+ translates external time (time outside $\texttt{ondemand}(P)$) into internal time (time inside $\texttt{ondemand}(P)$).

Now that we have defined the clocks signals h , h^+ and h^*, we can introduce the *downsampling* and *upsampling* operations needed to express the *on-demand* semantics.

3.2. Downsampling

The downsamplig operation is notated $\downarrow h$. For a signal x, the downsampled signal $x \downarrow h$ representes the external signal x seen as seen from inside $\texttt{ondemand}(P)$. It is defined as:

$$x \downarrow h = \lambda t.\, x(h^*(t))$$

For example if $x = 0.0, -0.1, -0.2, -0.3, -0.4, -0.5, -0.6, -0.7, \ldots$ and $h^* = 0, 3, 7, \ldots$, then $x \downarrow h = 0.0, -0.3, -0.7, \ldots$

3.3. Upsampling

The reverse *upsampling* operation, notated $\uparrow h$, expands the input signal by repeating the missing values. For a signal x, the upsampled signal $x \uparrow h$ representes the internal signal x as seen from outside $\texttt{ondemand}(P)$. It is defined as:

$$x \uparrow h = \lambda t.\, x(h^+(t))$$

For example if $x = 0.0, -0.3, -0.7, \ldots$ and $h^+ = 0, 0, 0, 1, 1, 1, 1, 2, 2, \ldots$ then $x \uparrow h = 0.0, 0.0, 0.0, -0.3, -0.3, -0.3, -0.3, -0.7, \ldots$

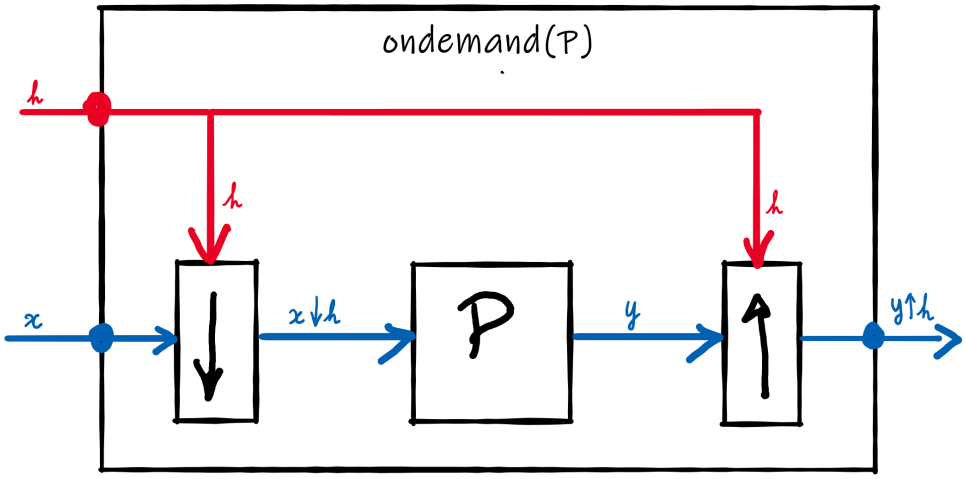
> *NOTE: please note that $\uparrow h : \downarrow h$ is the identity function, but that is not true for $\downarrow h : \uparrow h$.*

3.4. Semantics of $\texttt{ondemand}(P)$

We now have all the elements to define the semantics of $\texttt{ondemand}(P)$. Let's P be a signal processor with n inputs and m outputs. The semantics of $\texttt{ondemand}(P)$ is defined by the following rule:

$$\frac{[\![P]\!](x_1 \downarrow h, \ldots, x_n \downarrow h) = (y_1, \ldots, y_m)}{[\![\texttt{ondemand}(P)]\!](h, x_1, \ldots, x_n) = (y_1 \uparrow h, \ldots, y_m \uparrow h)}$$

As we can see, $[\![\texttt{ondemand}(P)]\!]$ is basically $[\![P]\!]$ applied to downsampled versions of the input signals: $x_i \downarrow h$. The downsampling depends on the demand clock h. Intuitively this corresponds to the fact that the values of the input signals are lost between two computation demands. Symmetrically the y_i signals returned by P have to be upsampled: $y_i \uparrow h$. This is illustrated by the following block-diagram



4. Combining on-demands

What happens when we combine on-demands? Can we factorize on-demands? For example, is the sequential composition of two on-demands with the same clock equivalent to the on-demand of the sequential composition of the inner processors? We need to be able to answer these questions in order to normalize Faust expressions and generate the most efficient code.

4.1. Notation

Let's start by defining some additional notation. Instead of writing the on-demand version of P controlled by clock h as the partial application: $\texttt{ondemand}(P)(h)$, we will simply write $P \downarrow h$.

Let's also notate $1_h = 1, 1, 1, \ldots$ the clock signal that contains only 1s, that is a demand every tick, and $0_h = 0, 0, 0, \ldots$ the clock signal that contains only 0s and therefore no computation demands at all.

4.2. Combining Clocks

We are interested in understanding what happens when we write something like: $(P \downarrow h_0) \downarrow h_1)$. There is, of course, an equivalent clock h_3 such that $(P \downarrow h_0) \downarrow h_1) = P \downarrow h_3$. But how do we compute it?

Let's call $\otimes$ the operation that combines two clock signals and such that:

$$(P \downarrow h_0) \downarrow h_1) = P \downarrow (h_0 \otimes h_1)$$

Let's see what the properties of $\otimes$ are.

4.3. Identities

There are two remarkable identities, related to the 1_h and 0_h clocks, that need to be taken into account:

$$(P \downarrow 1_h) \downarrow h) = P \downarrow h = (P \downarrow h) \downarrow 1_h)$$
$$(P \downarrow 0_h) \downarrow h) = P \downarrow 0_h = (P \downarrow h) \downarrow 0_h)$$

We therefore deduce that:

$$1_h \otimes h = h \otimes 1_h = h$$
$$0_h \otimes h = h \otimes 0_h = 0_h$$

4.4. A Manual Example

Let's first manually compute the demands for $(P \downarrow h_0) \downarrow h_1)$ with

$$h_0 = 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, \ldots$$
$$h_1 = 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, \ldots$$
$$tick = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \ldots$$

- at tick 0 we have a demand because $h_1(0) = 1$ and $h_0(0) = 1$;
- at tick 1 we have no demand because $h_1(1) = 1$, but $h_0(1) = 0$;
- at tick 2 we have no demand because $h_1(2) = 0$, moreover no demand is propagated to h_0;
- at tick 3 we have a demand because $h_1(3) = 1$ and $h_0(2) = 1$;
- Etc.

$$h_0 = 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, \ldots$$
$$h_1 = 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, \ldots$$
$$tick = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \ldots$$
$$h_0 \otimes h_1 = 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, \ldots$$

We can better understand how $h_0 \otimes h_1$ is computed by aligning h_0 values on top of h_1 demands:

$$h_0 = 1, 0, \quad 1, 0, \quad \ldots$$
$$h_1 = 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, \ldots$$
$$h_0 \otimes h_1 = 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, \ldots$$

4.5. General Rule

The property to observe from this manual example that one does not necessarily progress to all ticks in h_0. In reality, we only progress in h_0 when there are requests in h_1. In other words, when we are at the tick t in h_1 we are only at the tick $h_1^+(t) - 1$ in h_0. Furthermore, for a request to reach P it is necessary to have $h_1(t)$ and $h_0(h_1^+(t) - 1)$ at 1. We can now write down the general rule:

$$(h_0 \otimes h_1)(t) = h_1(t) * h_0(h_1^+(t))$$

It turns out that we've already met part of this formula with our upsampling function. So we can rewrite our definition as follows:

$$(h_0 \otimes h_1) = h_1 * (h_0 \uparrow h_1)$$

If we had chosen a more traditional upsampling function (without repetition, but with insertion of 0) we would have had simply: $\otimes = \uparrow$.

4.6. Commutativity

Is $\otimes$ commutative? Let's try with an example:

$$\begin{aligned} h_0 &= 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, \dots \\ h_1 &= 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, \dots \\ h_0 \otimes h_1 &= 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, \dots \\ h_1 \otimes h_0 &= 1, 0, 1, 0, 0, 0, 1, 0, 0, 0 \dots \end{aligned}$$

As we see $h_1 \otimes h_0 \neq h_0 \otimes h_1$, therefore $\otimes$ **is not a commutative operation** and, as a result:

$$(P \downarrow h_0) \downarrow h_1 \neq (P \downarrow h_1) \downarrow h_0$$

4.7. Associativity

Another interesting property is to check is associativity. This is an important one for the Faust compiler. It allows the normalizer to reorganize and factorize the code more easily. So, do we have: $((h_0 \otimes h_1) \otimes h_2) = (h_0 \otimes (h_1 \otimes h_2))$?

4.7.0.0.1. Notation

To lighten the notation during the proof, we will use capital letters A, B, C for clocks and write $A' = aA$ to indicate that $A'(0) = a$ and $A'(t \geq 1) = A(t-1)$.

Using our new notation, we can reformulate the clock composition operation $\otimes$ with two rewriting rules:

$$\begin{aligned} A \otimes 0B &\xrightarrow{\alpha} 0(A \otimes B) \\ aA \otimes 1B &\xrightarrow{\beta} a(A \otimes B) \end{aligned}$$

4.7.0.0.1. Associativity as a predicate

We want to check that $\otimes$ is associative for every possible clock. We can reformulate this property as a predicate $\mathcal{P}$ on triplets of clocks:

$$\mathcal{P}(A, B, C) \overset{\text{def}}{=} (A \otimes B) \otimes C = A \otimes (B \otimes C)$$

and check that P is true for every triplet of clocks.

4.7.0.0.1. Inductive set

To do that, we need an inductive definition of the set of triplets of clocks. For simplicity reasons, we are not considering arbitrary clocks but only clocks terminated with an infinite sequence of 0s.

Let's call $\mathbb{H}$ this specific set of clocks. Here is its inductive definition:

- $0_h \in \mathbb{H}$
- $A \in \mathbb{H} \implies 0A \in \mathbb{H}$
- $A \in \mathbb{H} \implies 1A \in \mathbb{H}$
- Nothing else is in $\mathbb{H}$.

The set of triplets of clocks we are interested in is $\mathbb{H} \times \mathbb{H} \times \mathbb{H}$. But we need an inductive definition of it. Here is one:

- Base case: $\forall A, B \in \mathbb{H}, (A, B, 0_h) \in \mathbb{H}^3$
- Induction step 1: $(A, B, C) \in \mathbb{H}^3 \implies (A, B, 0C) \in \mathbb{H}^3$
- Induction step 2: $(A, B, C) \in \mathbb{H}^3 \implies (A, 0B, 1C) \in \mathbb{H}^3$
- Induction step 3: $(A, B, C) \in \mathbb{H}^3 \implies (0A, 1B, 1C) \in \mathbb{H}^3$
- Induction step 4: $(A, B, C) \in \mathbb{H}^3 \implies (1A, 1B, 1C) \in \mathbb{H}^3$
- Nothing else is in $\mathbb{H}^3$.

To be sure that our inductive definition is correct, we need first to prove that the resulting set $\mathbb{H}^3$ is equivalent to $\mathbb{H} \times \mathbb{H} \times \mathbb{H}$, in other words, that :

- a) $\forall (A, B, C) \in \mathbb{H}^3 \implies A, B, C \in \mathbb{H}$
- b) $\forall A, B, C \in \mathbb{H} \implies (A, B, C) \in \mathbb{H}^3$

Proof:

- a) trivial
- b) We provide a recursive proof that is guaranteed to end on a base case $(A, B, 0_h) \in \mathbb{H}^3$. This is because one element of C is removed at each iteration, leading to 0_h after a finite number of iterations.
    - to prove $(0A, 0B, 0C) \in \mathbb{H}^3$, prove $(0A, 0B, C) \in \mathbb{H}^3$ and use induction step 1
    - to prove $(0A, 0B, 1C) \in \mathbb{H}^3$, prove $(0A, B, C) \in \mathbb{H}^3$ and use induction step 2
    - to prove $(0A, 1B, 0C) \in \mathbb{H}^3$, prove $(0A, 1B, C) \in \mathbb{H}^3$ and use induction step 1
    - to prove $(0A, 1B, 1C) \in \mathbb{H}^3$, prove $(A, B, C) \in \mathbb{H}^3$ and use induction step 3
    - to prove $(1A, 0B, 0C) \in \mathbb{H}^3$, prove $(0A, 0B, C) \in \mathbb{H}^3$ and use induction step 1
    - to prove $(1A, 0B, 1C) \in \mathbb{H}^3$, prove $(1A, B, C) \in \mathbb{H}^3$ and use induction step 2
    - to prove $(1A, 1B, 0C) \in \mathbb{H}^3$, prove $(1A, 1B, C) \in \mathbb{H}^3$ and use induction step 1
    - to prove $(1A, 1B, 1C) \in \mathbb{H}^3$, prove $(A, B, C) \in \mathbb{H}^3$ and use induction step 4

4.7.1. Proof of Associativity

To prove $\mathcal{P}$ for all elements of $\mathbb{H}^3$, we have to prove it for the base cases and the four induction steps.

4.7.1.0.1. Base case: $\mathcal{P}(A, B, 0_h)$ is true.

Proof:

$$(A \otimes B) \otimes 0_h \to 0_h \leftarrow A \otimes 0_h \leftarrow A \otimes (B \otimes 0_h)$$

4.7.1.0.1. Induction step 1: $P(A, B, C) \implies P(A, B, 0C)$

Proof:

$$(A \otimes B) \otimes 0C \to 0((A \otimes B) \otimes C) = 0(A \otimes (B \otimes C)) \leftarrow A \otimes 0(B \otimes C) \leftarrow A \otimes (B \otimes 0C)$$

4.7.1.0.1. Induction step 2: $P(A, B, C) \implies P(A, 0B, 1C)$

Proof:

$$(A \otimes 0B) \otimes 1C \to 0(A \otimes B) \otimes 1C \to 0((A \otimes B) \otimes C) = 0(A \otimes (B \otimes C)) \leftarrow A \otimes 0(B \otimes C) \letfarrow A$$

4.7.1.0.1. Induction step 3: $P(A, B, C) \implies P(0A, 1B, 1C)$

Proof:

$$(0A \otimes 1B) \otimes 1C \to 0(A \otimes B) \otimes 1C \to 0((A \otimes B) \otimes C) = 0(A \otimes (B \otimes C)) \leftarrow 0A \otimes 1(B \otimes C) \leftarrow 0A \otimes$$

4.7.1.0.1. Induction step 4: $P(A, B, C) \implies P(1A, 1B, 1C)$

Proof:

$$(1A \otimes 1B) \otimes 1C \to 1(A \otimes B) \otimes 1C \to 1(A \otimes B) \otimes C = 1(A \otimes (B \otimes C)) \leftarrow 1A \otimes 1(B \otimes C) \leftarrow 1A \otimes (1$$

□