# The Future of Faust

## Ondemand and Co.

Yann Orlarey, Stéphane Letz

IFC 2024

EMERAUDE (INRIA/INSA/GRAME)

# Part 1 : A brief History of Multirate in Faust

# 2009: Semantics of multirate Faust

The always-active monorate model is simple, but not always sufficient.

HAL open science

**Semantics for multirate Faust**

Pierre Jouvelot, Yann Orlarey



## Multirate

### Vectorize

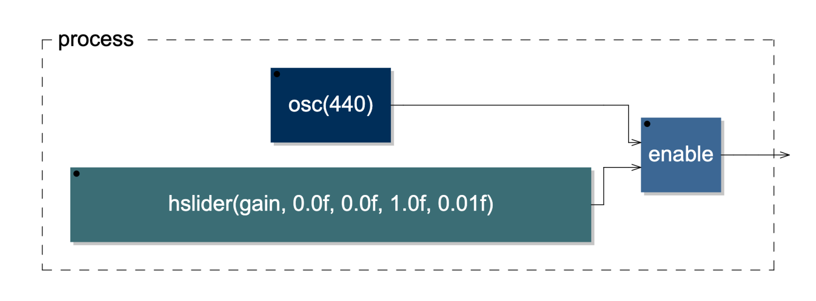

vectorize : $T^r \times n \to [n]T^{r/n}$

## Multirate

### Serialize

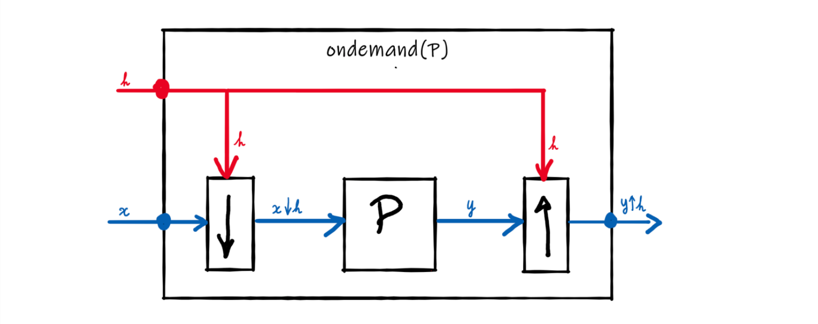

serialize : $[n]T^{r/n} \to T^r$

# 2015: Mute, Enable and Control




- 2015: `mute(x,y)` like `x*y` but the computation of `x` can be suspend when `y` is 0.
- Later, `mute` was renamed to `enable`, and a `control` variant was added.
- 2021: extended to `-vec` mode.

# 2020: Ondemand




- 2020: Till Bovermann asks for *demand-rate computations*
- 2020: Specification of *ondemand*
- 2022: Proof of concept presented at IFC-22
- 2024: *Ondemand* officially introduced at IFC-24

# Part 2 : Ondemand

# Introduction

### Objective

Provide *multirate* and *call-by-need* computation while preserving *efficiency* and *simple semantics*

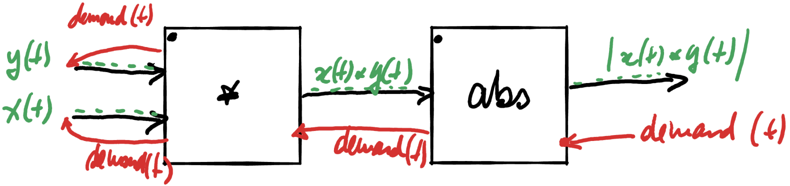
### Multirate Computation

- Frequency domain
- Upsampling
- Downsampling

### call-by-need

- Pay for what you use
- Controlling when computations occur
- Music composition-style computation

# call-by-need strategy

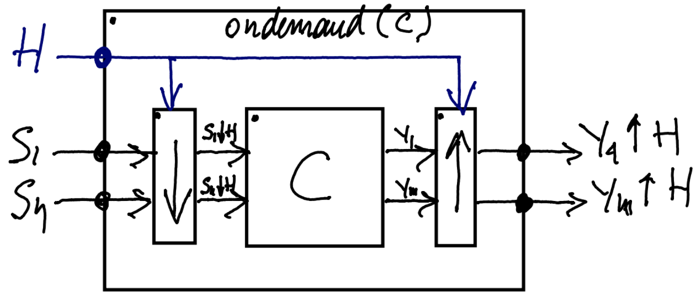



## Computations are only performed when explicitly required

- The demand (red arrow) is propagated backwards, starting from the outputs and moving towards the inputs.
- In response, the computed values (green arrows) are propagated forwards, moving from the inputs to the outputs.
- The output values remain constant until the next demand.

# Ondemand Semantics

`ondemand(C)` applies `C` to downsampled input signals ($S_i \downarrow H$), producing upsampled results ($Y_j \uparrow H$). Here, $H$ is the clock signal.




**Semantic rule**

$$(\text{od})\ \frac{[\![C]\!](S_1 \downarrow H, ..., S_n \downarrow H) = (Y_1, ..., Y_m)}{[\![\texttt{ondemand}(C)]\!](H, S_1, ..., S_n) = (Y_1 \uparrow H, ..., Y_m \uparrow H)}$$

# Downsampling

The downsampled $S_i \downarrow H$ is computed from $S_i$, based on the clock signal $H$. $t$ is the time observed outside C, and $t'$ inside.

| $t$ | $S_i$ | $H$ | $S_i \downarrow H$ | down$[\![H]\!]$ | $t'$ |
|---|---|---|---|---|---|
| 0 | a | 1 | a | 0 | 0 |
| 1 | b | 0 | . | . | . |
| 2 | c | 0 | . | . | . |
| 3 | d | 1 | d | 3 | 1 |
| 4 | f | 1 | f | 4 | 2 |
| 5 | g | 0 | . | . | . |

**Table 1:** Example of downsampling

**Semantic rule**

$$\text{(down)}\,\frac{\text{down}[\![H]\!] = \{n \in \mathbb{N} \mid [\![H]\!](n) = 1\}}{[\![S_i \downarrow H]\!](t) = [\![S_i]\!](\text{down}[\![H]\!](t))}$$

# Upsampling

$S_i \uparrow H$ is the upsampling of $S_i$ according to clock signal $H$. $t$ is the time observed outside `C`, and $t'$ inside.

| $t'$ | $S_i$ | $H$ | $S_i \uparrow H$ | $\text{up}[\![H]\!]$ | $t$ |
|---|---|---|---|---|---|
| 0 | a | 1 | a | 0 | 0 |
| 1 | d | 0 | a | 0 | 1 |
| 2 | f | 0 | a | 0 | 2 |
| . | . | 1 | d | 1 | 3 |
| . | . | 1 | f | 2 | 4 |
| . | . | 0 | f | 2 | 5 |

**Table 2:** Example of upsampling

## Semantic rule

$$(\text{up})\ \frac{\text{up}[\![H]\!](t) = \sum_{i=0}^{t}[\![H]\!](i) - 1}{[\![S_i \uparrow H]\!](t) = [\![S_i]\!](\text{up}[\![H]\!](t))}$$

# Example 1: Sample and Hold

`ondemand` simplifies the implementation of a *Sample and Hold* (SH)circuit. It is directly expressed as the `ondemand` version of the identity function `_`.

**1: without ondemand**

```
SH = (X,_:select2) ~ _ with { X = _,_ <: !,_,_,!; };
```

**2: with ondemand**

```
SH = ondemand(_);
```

# Example 1: Generated code

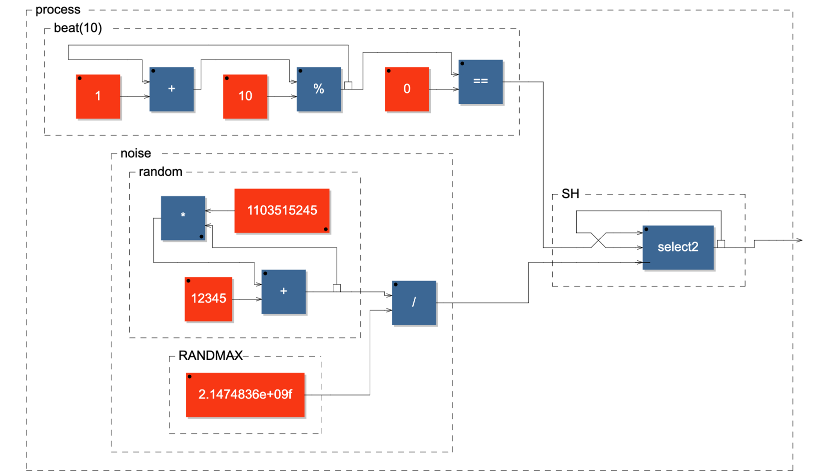
## 1: without ondemand

```
for (int i=0; i<count; i++) {
    fVec0SE[0] = ((int((float)input0[i])) ?
                  (float)input1[i] : fVec0SE[1]);
    output0[i] = (FAUSTFLOAT)(fVec0SE[0]);
    fVec0SE[1] = fVec0SE[0];
}
```

## 2: with ondemand

```
for (int i=0; i<count; i++) {
    fTemp0SE = (float)input1[i];
    if ((float)input0[i]) {
        fPermVar0SE = fTemp0SE;
    }
    output0[i] = (FAUSTFLOAT)(fPermVar0SE);
}
```

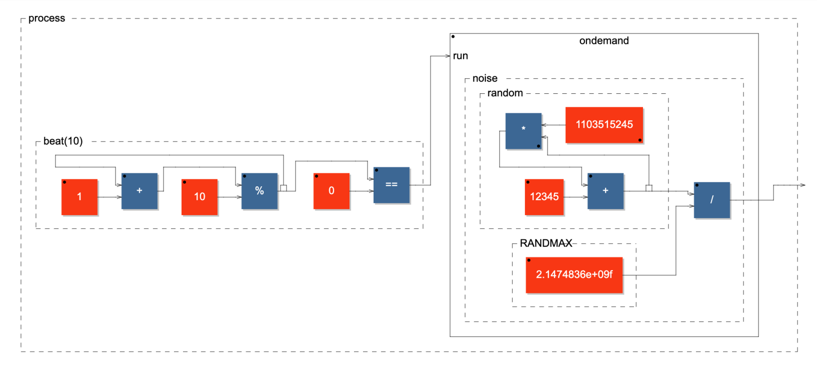
# Example 2: downsampled noise, without ondemand




## Faust code

```
process = ba.beat(100), no.noise : SH;
```

# Example 2: downsampled noise, with ondemand




## Faust code

```
process = ba.beat(100) : ondemand(no.noise);
```

# Example 2: Generated code, without ondemand

**Code generated for `ba.beat(100), no.noise : SH`**

```
for (int i=0; i<count; i++) {
    iVec0SI[0] = ((iVec0SI[1] + 1) % 100);
    iVec3SI[0] = ((1103515245 * iVec3SI[1]) + 12345);
    fVec2SI[0] = (((iVec0SI[0] == 0)) ?
                  (4.656613e-10f * float(iVec3SI[0]))
                  : fVec2SI[1]);
    output0[i] = (FAUSTFLOAT)(fVec2SI[0]);
    fVec2SI[1] = fVec2SI[0];
    iVec3SI[1] = iVec3SI[0];
    iVec0SI[1] = iVec0SI[0];
}
```

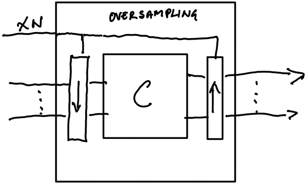
# Example 2: Generated code, with ondemand

**Code generated for `ba.beat(100) : ondemand(no.noise)`**

```
for (int i=0; i<count; i++) {
    iVec0SI[0] = ((iVec0SI[1] + 1) % 100);
    if ((iVec0SI[0] == 0)) {
        iVec2SI[0] = ((1103515245 * iVec2SI[1]) + 12345);
        fPermVar0SI = (4.656613e-10f * float(iVec2SI[0]));
        iVec2SI[1] = iVec2SI[0];
    }
    output0[i] = (FAUSTFLOAT)(fPermVar0SI);
    iVec0SI[1] = iVec0SI[0];
}
```

# Part 3 : ondemand variants

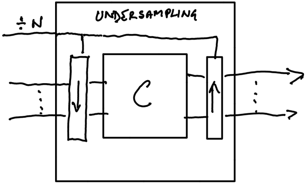
# Oversampling




**oversampling(C)**

Circuit C is run $N$ times faster than the surrounding circuit. The *sampling frequency* observed by C, is adjusted proportionally to the oversampling factor.

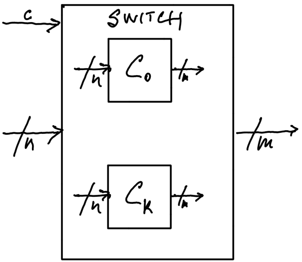
# Undersampling




**`undersampling(C)`**

Circuit `C` is run $N$ times slower than the surrounding circuit. The *sampling frequency* observed by `C`, is adjusted proportionally to the undersampling factor.

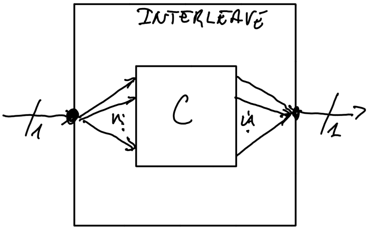
# Switch




**switch(C0,C1,...,Ck)**

Activate one of the `Ci` circuits according to the control input `c`. All the circuits must have the same type $n \to m$.

# Interleave




**interleave(C)**

Assuming `C` is of type $n \to n$, `interleave(C)` is of type $1 \to 1$ and operates as follows:

- The incoming samples are distributed sequentially to each of the $n$ inputs of `C`,
- `C` is then executed once, producing $n$ output values.
- These $n$ output values are interleaved back into a single output signal.

# Conclusion

**Ondemand and its variants introduce new perspectives**

- Frequency domain computation
- Oversampling and undersampling
- Composition-style, call-by-need computation

**While maintaining**

- Code efficiency
- Simple semantics
- Native integration as circuit primitives.

# Additional Examples

# Euclidian Rythms

```
euclidian(n) = vgroup("%n.EUCLID", er(pulses,steps)
    with {
        // UI: pulses < steps
        steps = vslider("steps[style:knob]", 16, 2, 16, 1)+0.5:i
        pulses = vslider("pulses[style:knob]", 1, 1, 16, 1)+0.5:

        // Implementation
        er(B,P,C) =
            C * ondemand (
                (+(1) : %(P)) ~ _
                : *(B)
                : %(P)
                : decr
               )(upfront(C));
        decr(x) = x < x';
        upfront(x) = x > x';
    }
```

# Loop

```
key(n) = vgroup("%n.KEY",
        trig : ondemand(irnd(k1,k2):loop(rn,ln):ba.midikey2hz) )
with {  random = +(12345) ~ *(1103515245);
        noise = random / 2147483647;
        irnd(x,y) = x+(noise+1)/2*(y-x);
        upfront(x) = x>x';
        loop(n,m) = select2(every(n)|for(m)) ~ @(m-1)
        with { every(n) = ((+(1):%(n))~_)' == 0;
               for(n) = 1-1@n; };
        k1 = vslider("[1]key[style:knob]", 60, 0, 127, 1);
        k2 = k1+vslider("[2]delta[style:knob]", 0, 0, 24, 1);
        ln = vslider("[3]len[style:knob]", 3, 2, 64, 1);
        rn = vslider("[4]renew[style:knob]", 11, 2, 127, 1);
        trig = button("[5]trig") : upfront;
    };
```