

# Compiling Faust with Ondemand

(<https://github.com/orlarey/ondemand-ifc22-slides/blob/master/slides.pdf>)

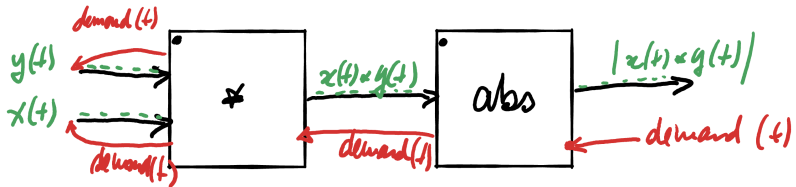
Yann Orlarey – Emeraude Team (INRIA-INSA-GAME)



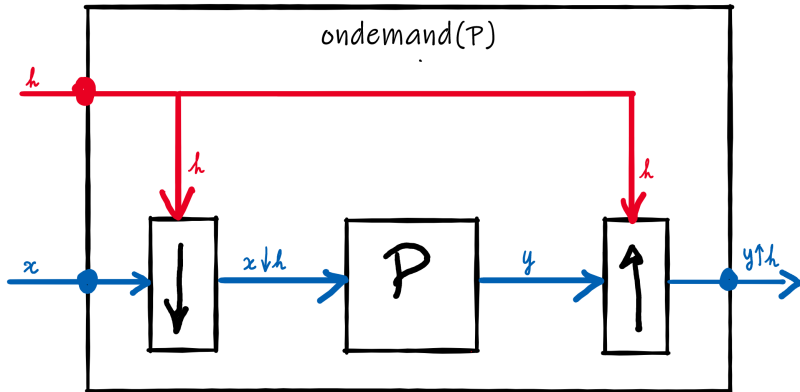
GRAME  
CENTRE NATIONAL  
DE CRÉATION  
MUSICALE, LYON

June 2022

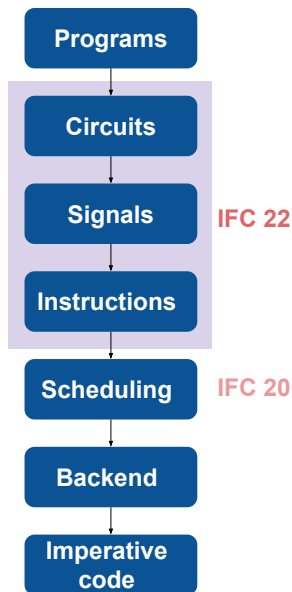
## On demand : Traditional Approach



## Ondemand : Faust Approach



# The Faust Compiler



# Faust Circuits as Formal Expressions

*Faust circuits* (evaluated Faust programs) are defined recursively by the following grammar:

## Circuits Definition

$$\begin{aligned} C \in \mathbb{C} ::= & k \mid u \mid \star \mid @ \mid ! \mid \_ \\ & \mid C_1 : C_2 \mid C_1, C_2 \\ & \mid C_1 <: C_2 \mid C_1 :> C_2 \\ & \mid C_1 \sim C_2 \mid \text{od}(C) \end{aligned}$$

## Primitives

- $k$  numbers (integer or real);
- $u$  user interface elements (sliders, buttons, etc.);
- $\star$  any numerical operation;
- $@$  the delay operation;
- $\_$  underscore, the identity circuit (a *perfect* cable);
- $!$  cut, the termination circuit.

# Faust Circuits as Formal Expressions

## Circuits Composition

- $C_1 <: C_2$  *split composition*, the outputs of  $C_1$  are distributed over the inputs of  $C_2$  ;
- $C_1 :> C_2$  *merge composition*, the outputs of  $C_1$  are summed to form the inputs of  $C_2$  ;
- $C_1 : C_2$  *sequential composition*, the outputs of  $C_1$  are propagated to the inputs of  $C_2$  ;
- $C_1, C_2$  *parallel composition*, the inputs are those of  $C_1$  and  $C_2$  and so are the outputs;
- $C_1 \sim C_2$  *recursive composition*, the outputs of  $C_1$  are fed back to the inputs of  $C_2$  and vice versa;
- $\text{od}(C)$  *ondemand* version of  $C$ .

# Well Formed Circuits

## Number of Inputs and Outputs

$$(\text{seq}) \frac{\text{io}[C_1] : m \rightarrow n \quad \text{io}[C_2] : n \rightarrow p}{\text{io}[C_1 : C_2] : m \rightarrow p}$$

$$(\text{par}) \frac{\text{io}[C_1] : m \rightarrow n \quad \text{io}[C_2] : p \rightarrow q}{\text{io}[C_1, C_2] : m + p \rightarrow n + q}$$

$$(\text{split}) \frac{\text{io}[C_1] : m \rightarrow n \quad \text{io}[C_2] : n.k \rightarrow p}{\text{io}[C_1 <: C_2] : m \rightarrow p}$$

$$(\text{merge}) \frac{\text{io}[C_1] : m \rightarrow k.n \quad \text{io}[C_2] : n \rightarrow p}{\text{io}[C_1 :> C_2] : m \rightarrow p}$$

$$(\text{rec}) \frac{\text{io}[C_1] : r + n \rightarrow q + m \quad \text{io}[C_2] : q \rightarrow r}{\text{io}[C_1 \sim C_2] : n \rightarrow q + m}$$

$$(\text{od}) \frac{\text{io}[C] : m \rightarrow n}{\text{io}[\text{od}(C)] : m + 1 \rightarrow n}$$

# Semantics of Well Formed Circuits

## Signal Processor Semantics

An audio circuit  $C \in \mathbb{C}$  denotes to a signal processor  $\llbracket C \rrbracket \in \mathbb{P} = \mathbb{S}^n \rightarrow \mathbb{S}^m$  that takes input signals and produces output signals.

## Notation

- $(S_1, \dots, S_n)$  a tuple of signals,
- $()$  the empty tuple and
- $(S_1, \dots, S_n, **k)$  an  $n * k$  tuple  $(S_1, \dots, S_n, S_1, \dots, S_n, \dots)$  obtained by concatenating  $k$  times the tuple  $(S_1, \dots, S_n)$ .



# Primitives Semantics (1)

## Constant

A number  $k$  denotes an elementary circuit with no input, that produces a constant signal  $k$ .

$$(\text{num}) \frac{}{\llbracket k \rrbracket () = (k)}$$

## Control

A user interface element  $u$  denotes an elementary circuit with no input and one output, the signal  $u$  produced by the user interface element.

$$(\text{ctrl}) \frac{}{\llbracket u \rrbracket () = (u)}$$

## Primitives Semantics (2)

### Numeric operation

The  $\star$  symbol denotes a *generic* numerical operation on signals. It represents a circuit with  $n$  inputs (typically 1 or 2 depending on the nature of the operation) and one output.

$$\text{(nop)} \frac{}{\llbracket \star \rrbracket(S_1, S_2, \dots) = (\star(S_1, S_2, \dots))}$$

### Delay

A delay primitive  $@$  denotes a circuit with two inputs and one output.

$$\text{(delay)} \frac{}{\llbracket @ \rrbracket(S_1, S_2) = (S_1 @ S_2)}$$

## Primitives Semantics (3)

### Cable

The cable has one input and one output.

$$\text{(cable)} \frac{}{\llbracket \_ \rrbracket (S) = (S)}$$

### Cut

The cut has one input and no output.

$$\text{(cut)} \frac{}{\llbracket ! \rrbracket (S) = ()}$$

# Circuit Compositions Semantics (1)

## Sequential Composition Semantics

$$\text{(seq)} \frac{\begin{array}{l} \llbracket C_1 \rrbracket(S_1, \dots, S_n) = (Y_1, \dots, Y_m) \\ \llbracket C_2 \rrbracket(Y_1, \dots, Y_m) = (Z_1, \dots, Z_p) \end{array}}{\llbracket C_1 : C_2 \rrbracket(S_1, \dots, S_n) = (Z_1, \dots, Z_p)}$$

## Parallel Composition Semantics

$$\text{(par)} \frac{\begin{array}{l} \llbracket C_1 \rrbracket(S_1, \dots, S_n) = (U_1, \dots, U_m) \\ \llbracket C_2 \rrbracket(Y_1, \dots, Y_p) = (V_1, \dots, V_q) \end{array}}{\llbracket C_1, C_2 \rrbracket(S_1, \dots, S_n, Y_1, \dots, Y_p) = (U_1, \dots, U_m, V_1, \dots, V_q)}$$

# Circuit Compositions Semantics (2)

## Split Composition Semantics

$$\text{(split)} \frac{\begin{array}{l} \llbracket C_1 \rrbracket(S_1, \dots, S_n) = (Y_1, \dots, Y_m) \\ \llbracket C_2 \rrbracket(Y_1, \dots, Y_m, * * k) = (Z_1, \dots, Z_p) \end{array}}{\llbracket C_1 <: C_2 \rrbracket(S_1, \dots, S_n) = (Z_1, \dots, Z_p)}$$

## Merge Composition Semantics

$$\text{(merge)} \frac{\begin{array}{l} \llbracket C_1 \rrbracket(S_1, \dots, S_n) = (Y_{1,1}, \dots, Y_{1,m}, \dots, Y_{k,1}, \dots, Y_{k,m}) \\ \llbracket C_2 \rrbracket(Y_{1,1} + \dots + Y_{k,1}, \dots, Y_{1,m} + \dots + Y_{k,m}) = (Z_1, \dots, Z_p) \end{array}}{\llbracket C_1 :=> C_2 \rrbracket(S_1, \dots, S_n) = (Z_1, \dots, Z_p)}$$

## Circuit Compositions Semantics (3)

### Recursive Composition Semantics

$$\begin{array}{c} W = \text{fresh recursive symbol} \\ \llbracket C_2 \rrbracket(W_1 @ 1, \dots, W_q @ 1) = (Z_1, \dots, Z_r) \\ \text{(rec)} \frac{\llbracket C_1 \rrbracket(Z_1, \dots, Z_r, S_1, \dots, S_n) = (Y_1, \dots, Y_q, Y_{q+1}, \dots, Y_{q+m})}{\llbracket C_1 \sim C_2 \rrbracket(S_1, \dots, S_n) = (Y_1, \dots, Y_q, Y_{q+1}, \dots, Y_{q+m})} \end{array}$$

with  $\text{def} \llbracket W \rrbracket = (Y_1, \dots, Y_q)$ .

# Ondemand Semantics

Ondemand

$$(\text{od}) \frac{\llbracket C \rrbracket (S_1 \downarrow H, \dots, S_n \downarrow H) = (Y_1, \dots, Y_m)}{\llbracket \text{od}(C) \rrbracket (H, S_1, \dots, S_n) = (Y_1 \uparrow H, \dots, Y_m \uparrow H)}$$

# Faust Signals as Formal Expressions

Faust signals are defined by the following grammar:

$$S \in \mathbb{S} ::= k \mid u \mid I_c \mid X_i \mid \star(S_1, S_2, \dots) \mid S_1 @ S_2 \mid S_1 \downarrow S_2 \mid S_1 \uparrow S_2$$

- $k$  is a number (integer or real)
- $u$  is a user interface element (slider, button, etc.)
- $I_c$  is the input channel  $c$
- $\star(S_1, S_2, \dots)$  is a numerical operation on signals
- $X_i$ : is the  $i$ -th signal of a group of mutually recursive signals associated to symbol  $X$
- $S_1 @ S_2$  is  $S_1$  delayed by  $S_2$
- $S_1 \downarrow S_2$  is  $S_1$  downsampled by  $S_2$
- $S_1 \uparrow S_2$  is  $S_1$  up-sampled by  $S_2$ .



# The semantics of Faust Signals as a function of time

A Faust signal  $S \in \mathbb{S}$  denotes a function of time, notated  $\llbracket S \rrbracket : \mathbb{Z} \rightarrow \mathbb{R}$ .  
The value of this function at time  $t$  is notated  $\llbracket S \rrbracket(t)$ .

By definition in Faust, the value of any signal before time 0 is always 0.  
Therefore we have:

$$\forall S \in \mathbb{S}, \forall t < 0, \llbracket S \rrbracket(t) = 0$$

# The semantics of Faust Signals as a function of time

For  $t \geq 0$  we have:

- $\llbracket k \rrbracket(t) = k$
- $\llbracket u \rrbracket(t) =$  value of the user interface controller  $u$  at time  $t$
- $\llbracket I_c \rrbracket(t) =$  value of the audio input channel  $c$  at time  $t$
- $\llbracket X_i \rrbracket(t) = \llbracket S_i \rrbracket(t)$  with definitions  $\text{def } \llbracket X \rrbracket = (S_1, \dots, S_i, \dots, S_n)$
- $\llbracket \star(S_1, S_2, \dots) \rrbracket(t) = \star(\llbracket S_1 \rrbracket(t), \llbracket S_2 \rrbracket(t), \dots)$
- $\llbracket S_1 @ S_2 \rrbracket(t) = \llbracket S_1 \rrbracket(t - \llbracket S_2 \rrbracket(t))$
- $\llbracket S_1 \downarrow S_2 \rrbracket(t) = \llbracket S_1 \rrbracket(\text{down} \llbracket S_2 \rrbracket(t))$
- $\llbracket S_1 \uparrow S_2 \rrbracket(t) = \llbracket S_1 \rrbracket(\text{up} \llbracket S_2 \rrbracket(t))$

# Signal Downsampling

$S_1 \downarrow S_2$  is the downsampling of  $S_1$ , based on the clock signal  $S_2$ .

$S_1$	$S_2$	$S_1 \downarrow S_2$	$\text{down}[\llbracket S_2 \rrbracket]$
a	1	a	0
b	0		
c	0		
d	1	d	3
f	1	f	4
g	0		

Table 1: Example of downsampling

$$\text{(down)} \frac{\text{down}[\llbracket S_2 \rrbracket] = \{n \in \mathbb{N} \mid \llbracket S_2 \rrbracket(n) = 1\}}{\llbracket S_1 \downarrow S_2 \rrbracket(t) = \llbracket S_1 \rrbracket(\text{down}[\llbracket S_2 \rrbracket](t))}$$

# Signal Upsampling

$S_1 \uparrow S_2$  is the upsampling of  $S_1$  according to clock signal  $S_2$ .

$S_1$	$S_2$	$S_1 \uparrow S_2$	$\text{up}[\llbracket S_2 \rrbracket]$
a	1	a	0
d	0	a	0
f	0	a	0
	1	d	1
	1	f	2
	0	f	2

Table 2: Example of upsampling

$$\text{(up)} \frac{\text{up}[\llbracket S_2 \rrbracket](t) = \sum_{i=0}^t \llbracket S_2 \rrbracket(i) - 1}{\llbracket S_1 \uparrow S_2 \rrbracket(t) = \llbracket S_1 \rrbracket(\text{up}[\llbracket S_2 \rrbracket](t))}$$

# Memory Signals

Memory signals are like signals seen before, but using *memory references* to implement delay lines, recursions, and sharing of common subexpressions. During the compilation signals are translated to *memory signals*.

## Definition

$$M \in \mathbb{M} ::= k \mid u \mid \mathbb{I}_c \mid \star(M_1, M_2, \dots) \mid t \mid m \mid v[M_1, M_2]$$

## Where

- $k$  is a number (integer or real)
- $u$  is a user interface element (slider, button, etc.)
- $\mathbb{I}_c$  is the input channel  $c$
- $\star(M_1, M_2, \dots)$  is a numerical operation on signals
- $t$ : is a scalar memory reference corresponding to the current time
- $m$ : is a scalar memory reference corresponding to a signal
- $v[M_1, M_2]$  is a vector memory reference where  $M_1$  is the time and  $M_2$  the delay.

# Instructions

An *instruction* is an intermediate representation, of type SSA, for signals.

## Definition

$$I \in \mathbb{I} ::= T \vdash t := t + 1 \mid T \vdash d := M \mid T \vdash v[M_1, M_2] := M_3$$

## Where

- $T$  is a time reference indicates when this instruction must be executed ;
- $t$  is a memory reference used for the current value of the time reference ;
- $d$  and  $v$  are memory references ;
- $M$  is a signal in memory that is computed.

# Time reference

A *time reference* is a non empty list of clock signals that indicates when an instruction should be executed.

## Definition

$$T \in \mathbb{T} ::= 1 \mid S.T$$

## Where

- $S \in \mathbb{M}$  is a clock signal  $S : \mathbb{Z} \rightarrow \{0, 1\}$
- $1$  is the top level clock signal (execution every sample)

# Memory Destinations

A *memory destination* indicates where the writing of the result should take place. This can be an output buffer, a scalar variable, or a vector in the case of delay lines for example.

## Definition

$$D \in \mathbb{D} ::= \mathbf{0}_n \mid t \mid m \mid v[M, M]$$

where  $\mathbf{0}_n$  represents the audio buffer of the  $n$ th output channel,  $t$ ,  $m$  and  $v$  are identifiers allocated at compile time.



# Identifiers and Marking

## Fresh identifier representing memory locations

- $\text{id}_s\llbracket S.T \rrbracket = m$  unique scalar identifier for  $S$  in time context  $T$  ;
- $\text{id}_v\llbracket S.T \rrbracket = v$  unique vector identifier for  $S$  in time context  $T$  ;
- $\text{id}_t\llbracket T \rrbracket = t$  unique scalar identifier representing the current time in time context  $T$ .

## Marking recursive definitions

- $\text{mark}\llbracket Xi.T \rrbracket = \emptyset$ : not yet marked ;
- $\text{mark}\llbracket Xi.T \rrbracket \leftarrow v$ : mark it with identifier  $v$  ;
- $\text{mark}\llbracket Xi.T \rrbracket = v$ : already marked with  $v$ .

# Signal Compilation

Function  $\text{cs}[\![\cdot]\!] : \mathbb{S} \times \mathbb{T} \rightarrow \mathbb{M} \times \mathcal{P}(\mathbb{I})$

Number

$$(\text{num}) \frac{}{\text{cs}[\![k.T]\!] = k \times \emptyset}$$

User interface

$$(\text{ctrl}) \frac{}{\text{cs}[\![u.T]\!] = u \times \emptyset}$$

Inputs

$$(\text{input}) \frac{}{\text{cs}[\![I_c.T]\!] = I_c \times \emptyset}$$

# Signal Compilation

## Numerical Operation

$$\begin{array}{c} \text{cs}\llbracket S_1.T \rrbracket = M_1 \times J_1 \\ \text{cs}\llbracket S_2.T \rrbracket = M_1 \times J_2 \\ \vdots \\ \text{id}_v\llbracket \star(M_1, M_2, \dots) \rrbracket = m \end{array}$$

---

$$\text{(nop)} \frac{}{\text{cs}\llbracket \star(S_1, S_2, \dots).T \rrbracket = m \times \{T \vdash m := \star(M_1, M_2, \dots)\} \cup_i J_i}$$

# Signal Compilation

## Downsampling

The downsampling  $S_1 \downarrow S_2$  appears at the entrance of an ondemand. This means that compiling  $S_1 \downarrow S_2$  into the  $M_2.T$  time environment (where  $M_2$  is the compiled version of  $S_2$ ) is like compiling  $S_1$  into the  $T$  time environment and using a variable to do the downsampling.

$$\begin{array}{c} \text{cs}\llbracket S_1.T \rrbracket = M_1 \times J_1 \\ \text{cs}\llbracket S_2.T \rrbracket = M_2 \times J_2 \\ \text{id}_s\llbracket M_1.T \rrbracket = m \\ J_3 = \{T \vdash m := M_1\} \\ \text{(down)} \frac{}{\text{cs}\llbracket (S_1 \downarrow S_2).M_2.T \rrbracket = m \times J_3 \cup J_1 \cup J_2} \end{array}$$

# Signal Compilation

## Upsampling

The  $S_1 \uparrow S_2$  upsampling appears at the output of an ondemand. It is necessary to compile  $S_1$  into the clock time reference  $S_2$  (which is added to the current time reference). The signal  $S_1$  must also be stored in a variable to do the upsampling.

$$\begin{array}{c} \text{cs}\llbracket S_2.T \rrbracket = M_2 \times J_2 \\ \text{cs}\llbracket S_1.M_2.T \rrbracket = M_1 \times J_1 \\ \text{id}_s\llbracket M_1.M_2.T \rrbracket = m \\ J_3 = \{M_2.T \vdash m := M_1\} \\ \text{(up)} \frac{}{\text{cs}\llbracket (S_1 \uparrow S_2).T \rrbracket = m \times J_1 \cup J_1 \cup J_2} \end{array}$$

# Signal Compilation

## Delay

$$\text{cs}\llbracket S_1.T \rrbracket = M_1 \times J_1$$

$$\text{cs}\llbracket S_2.T \rrbracket = M_2 \times J_2$$

$$\text{id}_v\llbracket M_1.T \rrbracket = v$$

$$\text{id}_t\llbracket T \rrbracket = t$$

$$\text{(up)} \frac{J_3 = \{T \vdash v[t, 0] := M_1\} \cup \{T \vdash t := t + 1\}}{\text{cs}\llbracket (S_1 @ S_2).T \rrbracket = v[t, M_2] \times J_3 \cup J_1 \cup J_2}$$

# Signal Compilation: recursion

## First visit

If it is the first visit, we have  $\text{mark}[[X_i.T]] = 0$ :

$$\begin{array}{l} \text{id}_v[[X_i.T]] = v \\ \text{mark}[[X_i.T]] \leftarrow v \\ \text{def}[[X]] = (... , S_i, ...) \\ \text{cs}[[S_i.T]] = M_i \times J_i \\ \text{cs}[[S_d.T]] = M_d \times J_d \\ \text{id}_t[[T]] = t \\ \text{(r1)} \frac{J_3 = \{T \vdash v[t, 0] := M_i\} \cup \{T \vdash t := t + 1\}}{\text{cs}[(X_i @ S_d).T] = v[t, M_d] \times J_3 \cup J_i \cup J_d} \end{array}$$

# Signal Compilation: recursion

## Next visits

If it is not the first visit, we have  $\text{mark}\llbracket X_i.T \rrbracket = v$ :

$$\text{(r2)} \frac{\begin{array}{c} \text{cs}\llbracket S_d.T \rrbracket = M_d \times J_d \\ \text{id}_t\llbracket T \rrbracket = t \end{array}}{\text{cs}\llbracket (X_i @ S_d).T \rrbracket = v[t, M_d] \times J_d}$$



# Global compilation

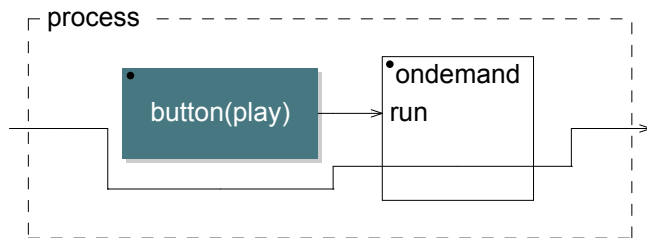
$$\text{comp}[\![\cdot]\!] : \mathbb{S}^n \rightarrow \mathcal{P}(\mathbb{I})$$

Global compilation

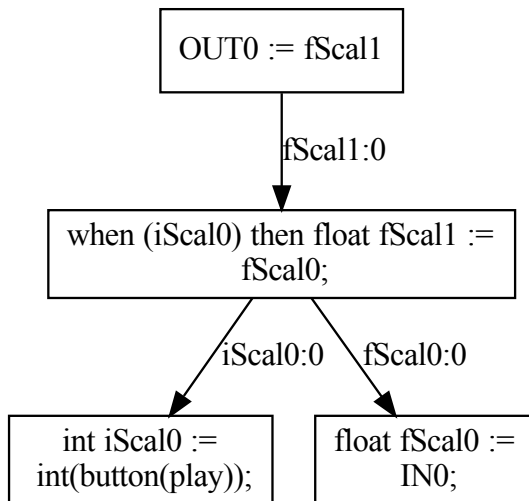
$$\text{(comp)} \frac{\begin{array}{c} \vdots \\ \text{cs}[S_i.1] = M_i \times J_i \\ J'_i = \{1 \vdash 0_i := M_i\} \cup J_i \\ \vdots \end{array}}{\text{comp}[\![\dots, S_i, \dots]\!] = \dots \cup J'_i \cup \dots}$$

## Example 1, block-diagram

```
process = button("play"), _ : ondemand(_);
```

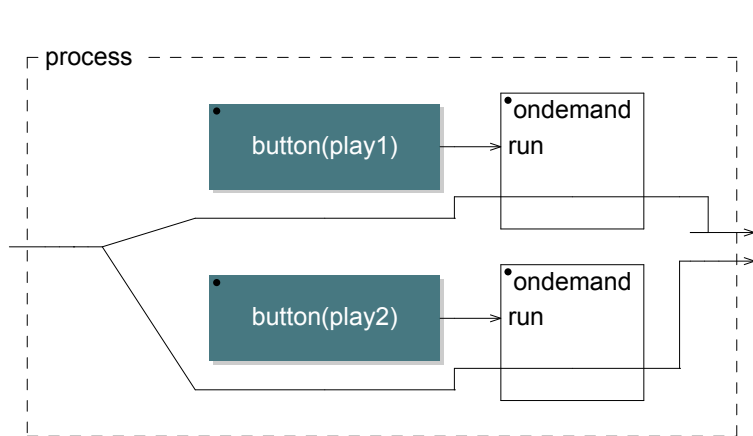


## Example 1, instruction graph

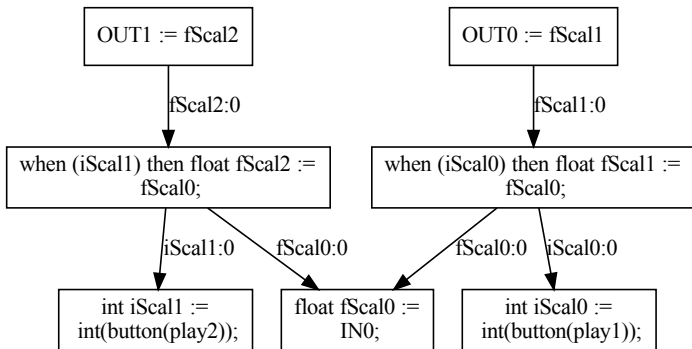


## Example 2, block-diagram

```
process = _ <: ondemand(_)(button("play1")), ondemand(_)(button("play2"))
```

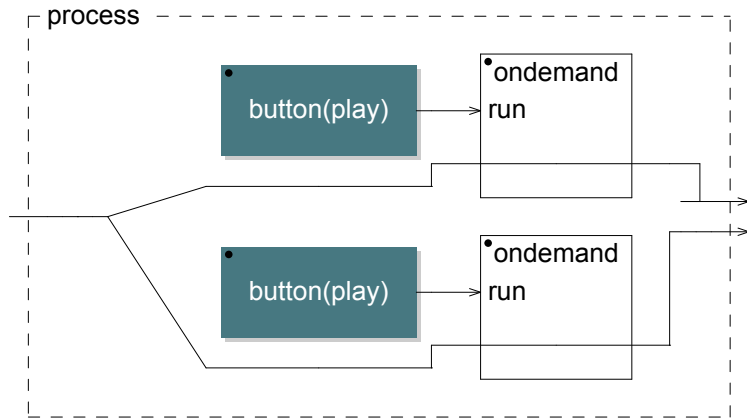


## Example 2, instruction graph

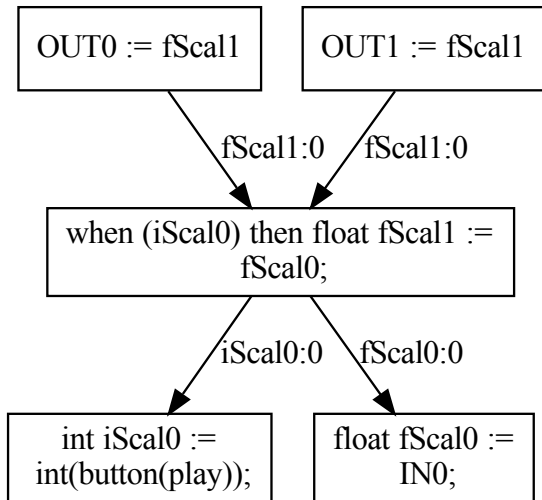


## Example 3, block-diagram

```
process = _ <: ondemand(_)(button("play")), ondemand(_)(button("
```

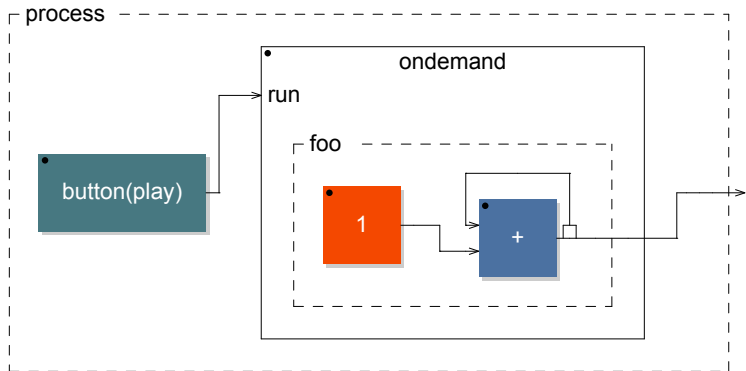


## Example 3, instruction graph



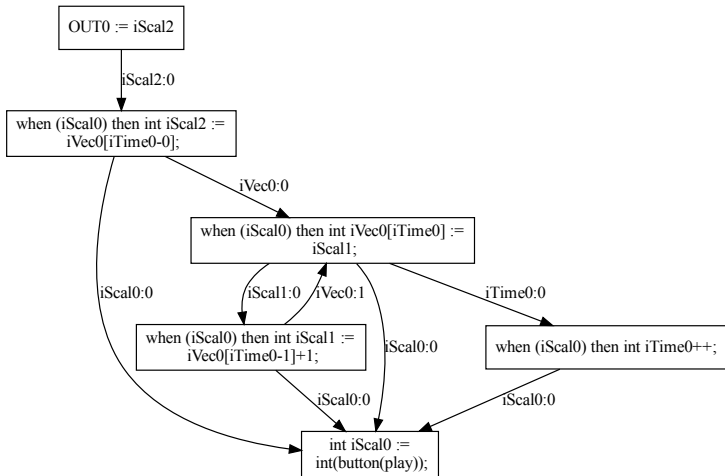
## Example 4, block-diagram

```
foo = 1:+~_;  
process = ondemand(foo)(button("play"));
```





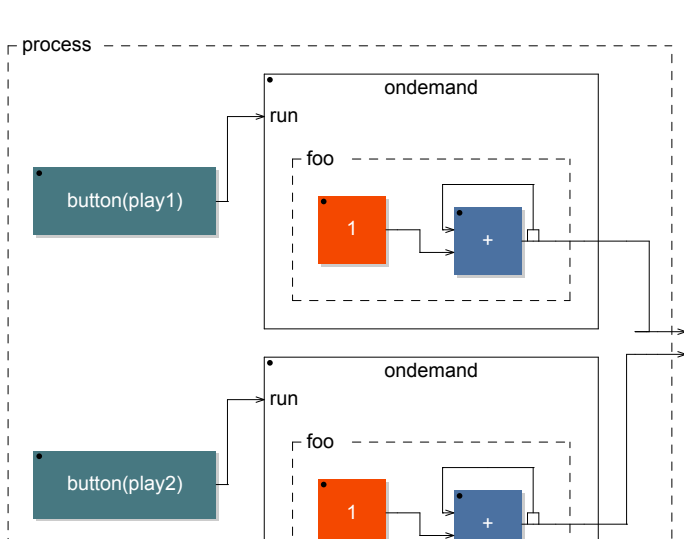
## Example 4, instruction graph



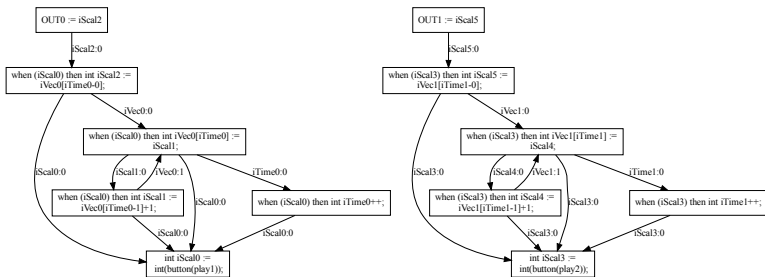
## Example 5, block-diagram

```
foo = 1:+~_;
```

```
process = ondemand(foo)(button("play1")), ondemand(foo)(button("play2"))
```



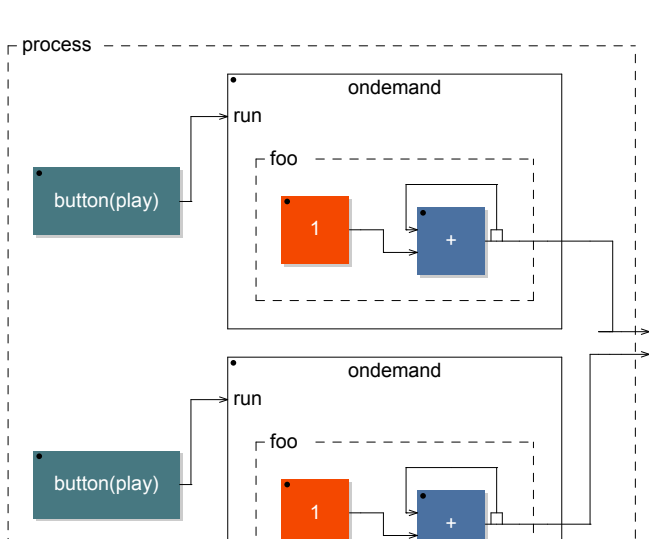
## Example 5, instruction graph



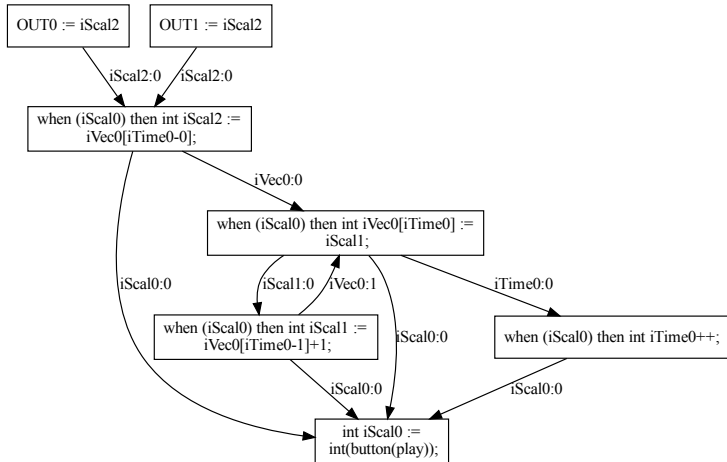
## Example 6, block-diagram

```
foo = 1:+~_;
```

```
process = ondemand(foo)(button("play")), ondemand(foo)(button("p
```

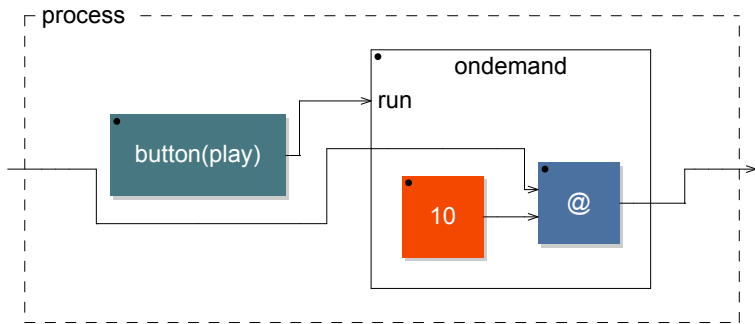


## Example 6, instruction graph

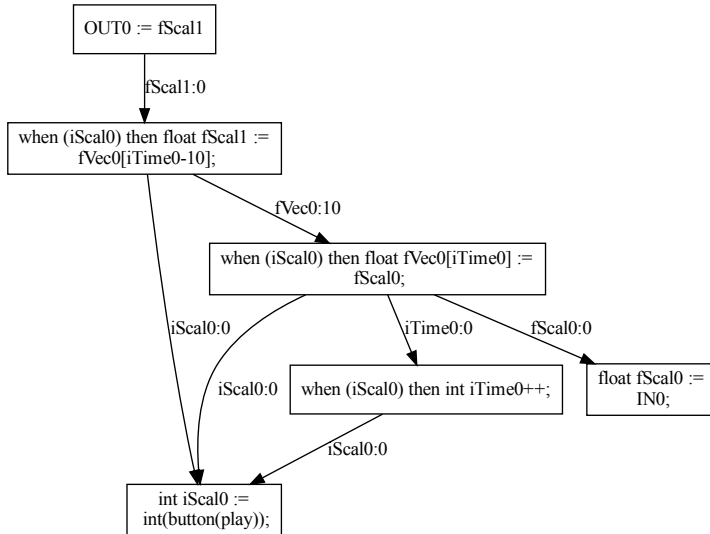


## Example 7, block-diagram

```
process = ondemand(@(10))(button("play"));
```

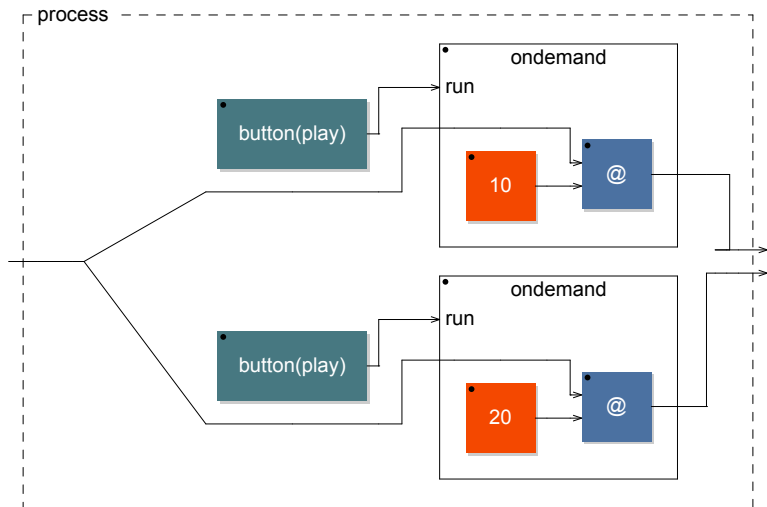


## Example 7, instruction graph



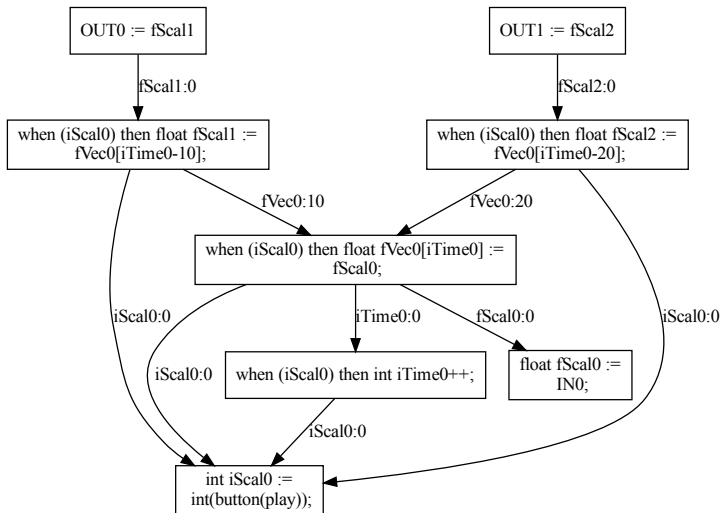
## Example 8, block-diagram

```
process = _ <: ondemand(@10))(button("play")), ondemand(@20))(
```





## Example 8, instruction graph



# Conclusion

What's missing ?

- Several primitives like tables, waveforms, soundfiles, etc. are not compiled yet
- Replace current interval computation
- Proper C++ code generation
- Merge with dev branch
- Future extensions :
  - Data Parallelism
  - `interleave(P)`