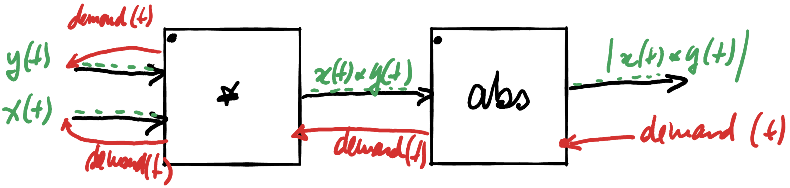
Compiling Faust with Ondemand (https://github.com/orlarey/ondemand-ifc22-slides/blob/master/slides.pdf)

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

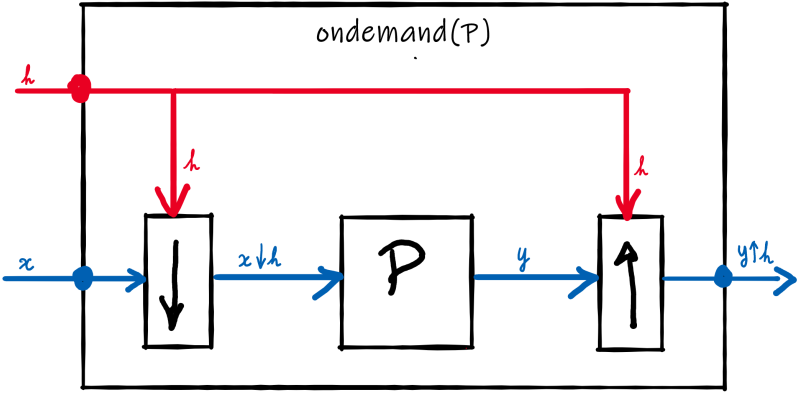
GRAME
CENTRE NATIONAL
DE CRÉATION
MUSICALE, LYON

June 2022

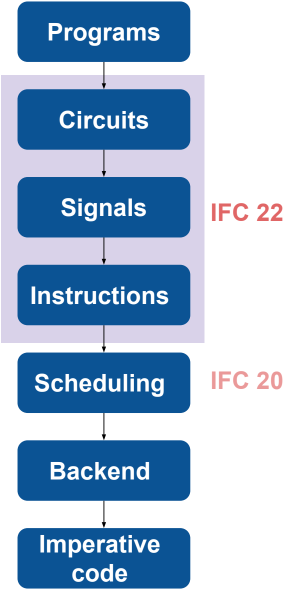
Ondemand : Traditional Approach



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

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

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;
- $\texttt{od}(C)$ *ondemand* version of C.

Well Formed Circuits

Number of Inputs and Outputs

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

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

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

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

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

$$(\text{od})\ \frac{\mathsf{io}[\![C]\!] : m \to n}{\mathsf{io}[\![\mathtt{od}(C)]\!] : m + 1 \to n}$$

Semantics of Well Formed Circuits

Signal Processor Semantics

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

Notation

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

Primitives Semantics (1)

Constant

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

$$\text{(num)}\ \frac{}{[\![k]\!]() = (k)}$$

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.

$$\text{(ctrl)}\ \frac{}{[\![u]\!]() = (u)}$$

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.

$$(\text{nop})\,\frac{}{[\![\star]\!](S_1, S_2, ...) = (\star(S_1, S_2, ...))}$$

Delay

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

$$(\text{delay})\,\frac{}{[\![@]\!](S_1, S_2) = (S_1 @ S_2)}$$

Primitives Semantics (3)

Cable

The cable has one input and one output.

$$\text{(cable)}\,\frac{}{[\![\_]\!](S) = (S)}$$

Cut

The cut has one input and no output.

$$\text{(cut)}\,\frac{}{[\![!]\!](S) = ()}$$

Circuit Compositions Semantics (1)

Sequential Composition Semantics

$$\text{(seq)}\;\frac{\begin{array}{c}[\![C_1]\!](S_1, ..., S_n) = (Y_1, ..., Y_m)\\ [\![C_2]\!](Y_1, ..., Y_m) = (Z_1, ..., Z_p)\end{array}}{[\![C_1 : C_2]\!](S_1, ..., S_n) = (Z_1, ..., Z_p)}$$

Parallel Composition Semantics

$$\text{(par)}\;\frac{\begin{array}{c}[\![C_1]\!](S_1, ..., S_n) = (U_1, ..., U_m)\\ [\![C_2]\!](Y_1, ..., Y_p) = (V_1, ..., V_q)\end{array}}{[\![C_1, C_2]\!](S_1, ..., S_n, Y_1, ..., Y_p) = (U_1, ..., U_m, V_1, ..., V_q)}$$

Circuit Compositions Semantics (2)

Split Composition Semantics

$$\text{(split)}\ \frac{[\![C_1]\!](S_1, ..., S_n) = (Y_1, ..., Y_m) \quad [\![C_2]\!](Y_1, ..., Y_m, **k) = (Z_1, ..., Z_p)}{[\![C_1 <: C_2]\!](S_1, ..., S_n) = (Z_1, ..., Z_p)}$$

Merge Composition Semantics

$$\text{(merge)}\ \frac{[\![C_1]\!](S_1, ..., S_n) = (Y_{1,1}, ..., Y_{1,m}, ..., Y_{k,1}, , ..., Y_{k,m}) \quad [\![C_2]\!](Y_{1,1} + ... + Y_{k,1}, ..., Y_{1,m} + ... + Y_{k,m}) = (Z_1, ..., Z_p)}{[\![C_1 :> C_2]\!](S_1, ..., S_n) = (Z_1, ..., Z_p)}$$

Circuit Compositions Semantics (3)

Recursive Composition Semantics

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

with $\text{def}[\![W]\!] = (Y_1, ..., Y_q)$.

Ondemand Semantics

Ondemand

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

Faust Signals as Formal Expressions

Faust signals are defined by the following grammar:

$$S \in \mathbb{S} ::= k \mid u \mid \mathtt{I}_c \mid X_i \mid \star(S_1, S_2, ...) \mid S_1@S_2 \mid S_1\downarrow S_2 \mid S_1\uparrow S_2$$

- k is a number (integer or real)
- u is a user interface element (slider, button, etc.)
- $\mathtt{I}_c$ is the input channel c
- $\star(S_1, S_2, ...)$ 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 $[\![S]\!] : \mathbb{Z} \rightarrow \mathbb{R}$. The value of this function at time t is notated $[\![S]\!](t)$.

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

$$\forall S \in \mathbb{S}, \forall t < 0, [\![S]\!](t) = 0$$

The semantics of Faust Signals as a function of time

For $t \geq 0$ we have:

- $[\![k]\!](t) = k$
- $[\![u]\!](t) =$ value of the user interface controller u at time t
- $[\![\mathtt{I}_c]\!](t) =$ value of the audio input channel c at time t
- $[\![X_i]\!](t) = [\![S_i]\!](t)$ with definitions $\mathsf{def}[\![X]\!] = (S_1, .., S_i, .., S_n)$
- $[\![\star(S_1, S_2, ...)]\!](t) = \star([\![S_1]\!](t), [\![S_2]\!](t), ...)$
- $[\![S_1 @ S_2]\!](t) = [\![S_1]\!](t - [\![S_2]\!](t))$
- $[\![S_1 \downarrow S_2]\!](t) = [\![S_1]\!](\mathsf{down}[\![S_2]\!](t))$
- $[\![S_1 \uparrow S_2]\!](t) = [\![S_1]\!](\mathsf{up}[\![S_2]\!](t))$

Signal Downsampling

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

S_1	S_2	$S_1 \downarrow S_2$	down$[\![S_2]\!]$
a	1	a	0
b	0		
c	0		
d	1	d	3
f	1	f	4
g	0		

Table 1: Example of downsampling

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

Signal Upsampling

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

S_1	S_2	$S_1 \uparrow S_2$	$\mathsf{up}[\![S_2]\!]$
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

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

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

$$M \in \mathbb{M} ::= k \mid u \mid \mathtt{I}_c \mid \star(M_1, M_2, ...) \mid t \mid m \mid v[M1, M2]$$

Where

- k is a number (integer or real)
- u is a user interface element (slider, button, etc.)
- $\mathtt{I}_c$ is the input channel c
- $\star(M_1, M_2, ...)$ 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

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

where $\mathtt{O}_n$ represents the audio buffer of the nth output channel, `t`, `m` and `v` are identifiers allocated at compile time.

Identifiers and Marking

Fresh identifier representing memory locations

- $\mathsf{id_s}[\![S.T]\!] = m$ unique scalar identifier for S in time context T ;
- $\mathsf{id_v}[\![S.T]\!] = v$ unique vector identifier for S in time context T ;
- $\mathsf{id_t}[\![T]\!] = t$ unique scalar identifier representing the current time in time context T.

Marking recursive definitions

- $\mathsf{mark}[\![Xi.T]\!] = \varnothing$: not yet marked ;
- $\mathsf{mark}[\![Xi.T]\!] \leftarrow v$: mark it with identifier v ;
- $\mathsf{mark}[\![Xi.T]\!] = v$: already marked with v.

Signal Compilation

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

Number

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

User interface

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

Inputs

$$(\mathsf{input})\,\frac{}{\mathsf{cs}[\![\mathtt{I}_c.T]\!] = \mathtt{I}_c \times \emptyset}$$

Signal Compilation

Numerical Operation

$$
\text{(nop)}\ \frac{\begin{array}{c} \mathsf{cs}[\![S_1.T]\!] = M_1 \times J_1 \\ \mathsf{cs}[\![S_2.T]\!] = M_1 \times J_2 \\ \vdots \\ \mathsf{id_v}[\![\star(M_1, M_2, ...)]\!] = m \end{array}}{\mathsf{cs}[\![\star(S_1, S_2, ...).T]\!] = m \times \{T \vdash m{:=} \star (M_1, M_2, ...)\} \bigcup_i J_i}
$$

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.

$$\text{(down)}\ \frac{\begin{array}{c}\mathsf{cs}[\![S_1.T]\!] = M_1 \times J_1 \\ \mathsf{cs}[\![S_2.T]\!] = M_2 \times J_2 \\ \mathsf{id_s}[\![M_1.T]\!] = m \\ J_3 = \{T \vdash m{:=}M_1\}\end{array}}{\mathsf{cs}[\![(S_1 \downarrow S_2).M_2.T]\!] = m \times J_3 \cup J_1 \cup J_2}$$

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.

$$
(\mathsf{up})\,\frac{\begin{array}{c}\mathsf{cs}[\![S_2.T]\!] = M_2 \times J_2 \\ \mathsf{cs}[\![S_1.M_2.T]\!] = M_1 \times J_1 \\ \mathsf{id_s}[\![M_1.M_2.T]\!] = m \\ J_3 = \{M_2.T \vdash m{:=}M_1\}\end{array}}{\mathsf{cs}[\![(S_1 \uparrow S_2).T]\!] = m \times J_1 \cup J_1 \cup J_2}
$$

Signal Compilation

Delay

$$\frac{\begin{array}{c}\mathsf{cs}[\![S_1.T]\!] = M_1 \times J_1 \\ \mathsf{cs}[\![S_2.T]\!] = M_2 \times J_2 \\ \mathsf{id_v}[\![M_1.T]\!] = v \\ \mathsf{id_t}[\![T]\!] = t \\ J_3 = \{T \vdash v[t,0]{:=}M_1\} \cup \{T \vdash t{:=}t+1\}\end{array}}{\mathsf{cs}[\![(S_1 @ S_2).T]\!] = v[t, M_2] \times J_3 \cup J_1 \cup J_2}\ (\mathsf{up})$$

Signal Compilation: recursion

First visit

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

$$\mathrm{(r1)}\ \frac{\begin{array}{c}\mathsf{id_v}[\![X_i.T]\!] = v \\ \mathsf{mark}[\![X_i.T]\!] \leftarrow v \\ \mathsf{def}[\![X]\!] = (..., S_i, ...) \\ \mathsf{cs}[\![S_i.T]\!] = M_i \times J_i \\ \mathsf{cs}[\![S_d.T]\!] = M_d \times J_d \\ \mathsf{id_t}[\![T]\!] = t \\ J_3 = \{T \vdash v[t, 0] := M_i\} \cup \{T \vdash t := t + 1\}\end{array}}{\mathsf{cs}[\![(X_i @ S_d).T]\!] = v[t, M_d] \times J_3 \cup J_i \cup J_d}$$

Signal Compilation: recursion

Next visits

If it is not the first visit, we have $\mathsf{mark}[\![X_i.T]\!] = v$:

$$\text{(r2)}\ \frac{\mathsf{cs}[\![S_d.T]\!] = M_d \times J_d \qquad \mathsf{id_t}[\![T]\!] = t}{\mathsf{cs}[\![(X_i@S_d).T]\!] = v[t, M_d] \times J_d}$$

Global compilation

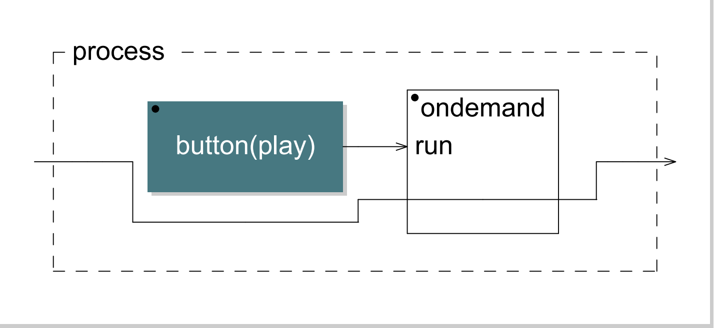
$\mathsf{comp}[\![.]\!] : \mathbb{S}^n \to \mathcal{P}(\mathbb{I})$

Global compilation

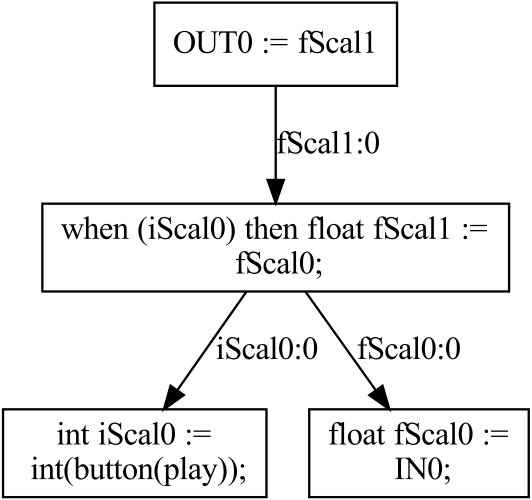
$$
(\mathsf{comp})\ \frac{\begin{array}{c}\vdots \\ \mathsf{cs}[\![S_i.1]\!] = M_i \times J_i \\ J_i' = \{1 \vdash \mathtt{0}_i{:=}M_i\} \cup J_i \\ \vdots\end{array}}{\mathsf{comp}[\![(\ldots, S_i, \ldots)]\!] = \ldots \cup J_i' \cup \ldots}
$$

Example 1, block-diagram

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

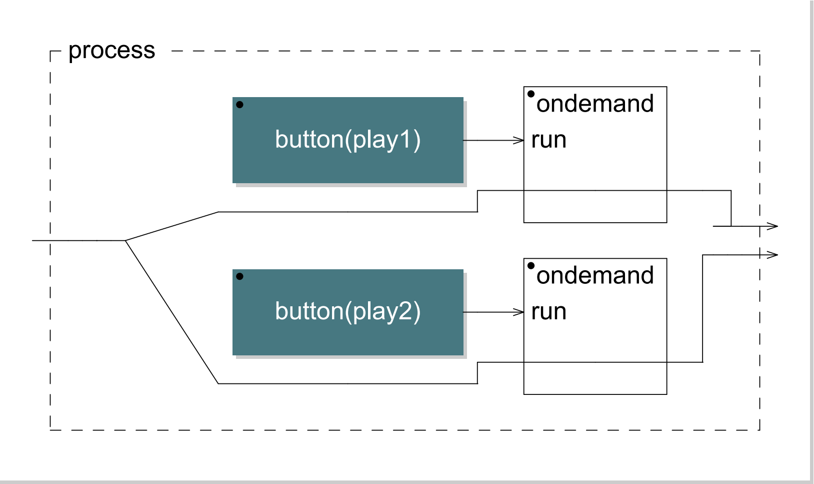


Example 1, instruction graph

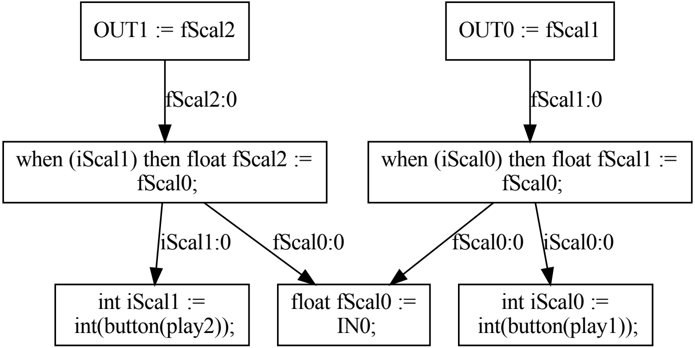


Example 2, block-diagram

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

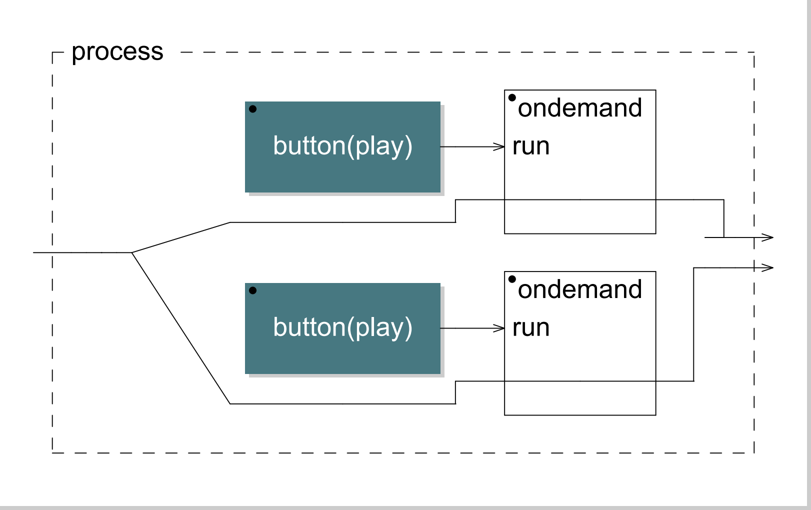


Example 2, instruction graph

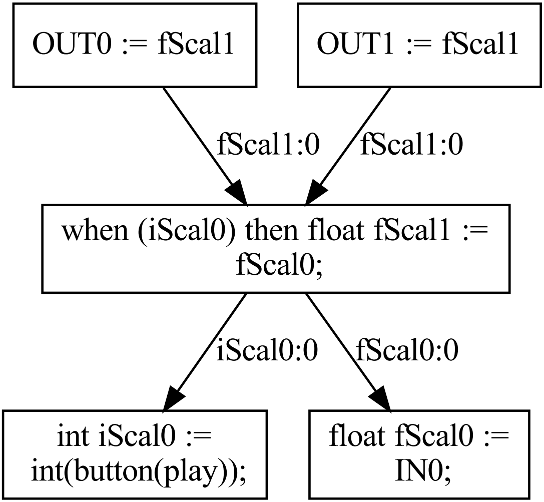


Example 3, block-diagram

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

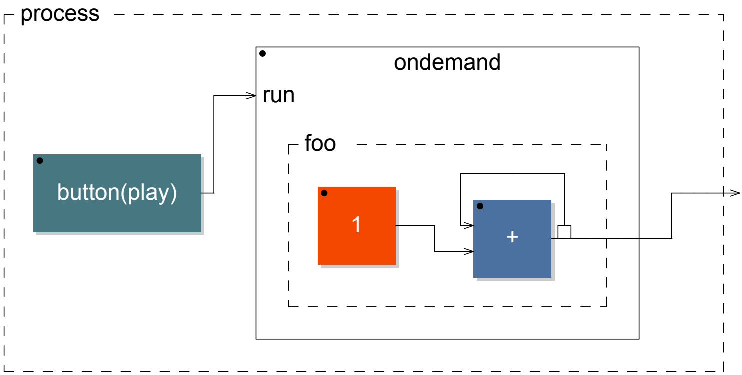


Example 3, instruction graph

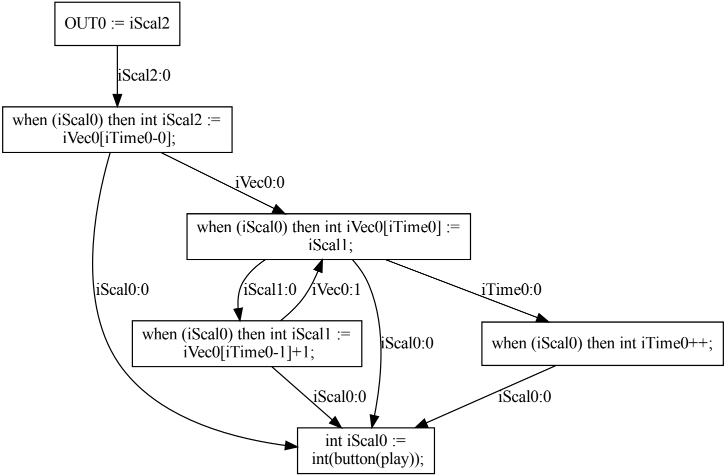


Example 4, block-diagram

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

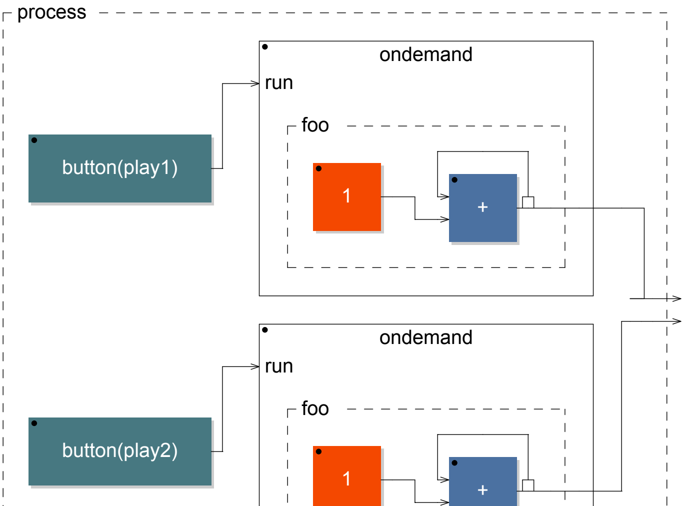


Example 4, instruction graph

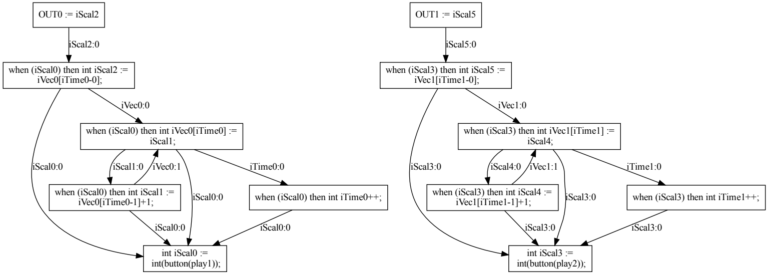


Example 5, block-diagram

```
foo = 1:+~_;
process = ondemand(foo)(button("play1")), ondemand(foo)(button("
```

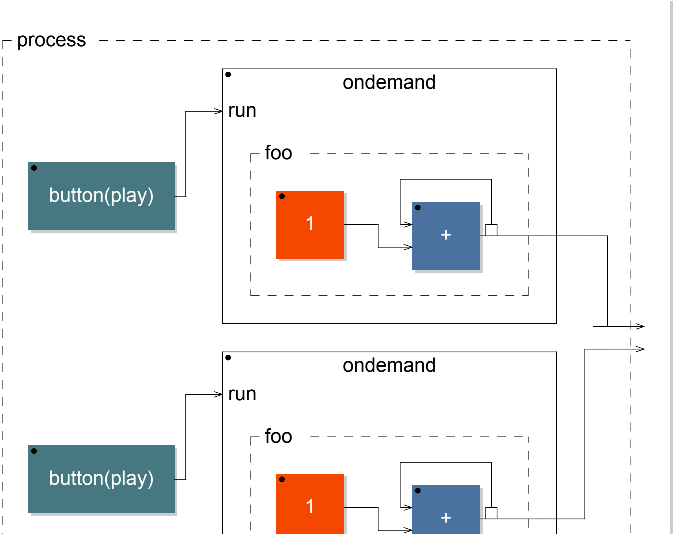


Example 5, instruction graph

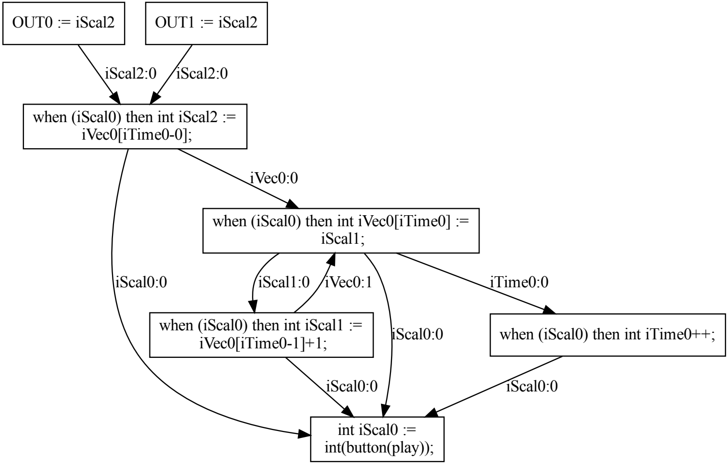


Example 6, block-diagram

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

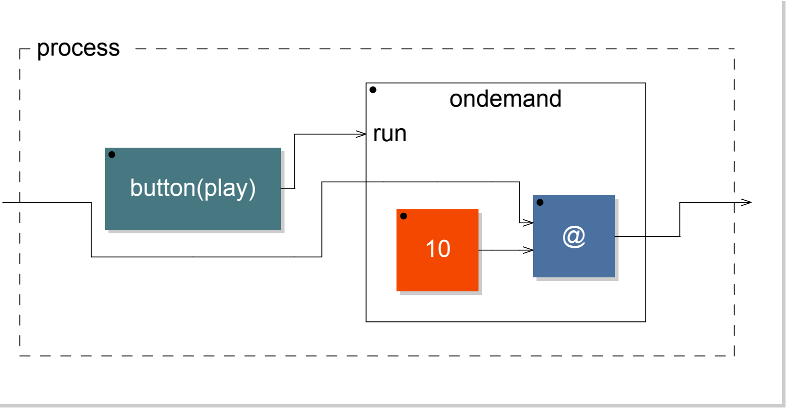


Example 6, instruction graph

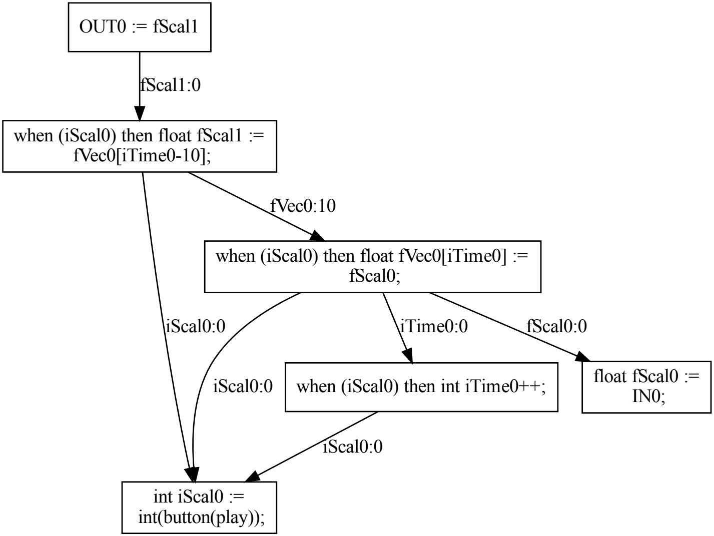


Example 7, block-diagram

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

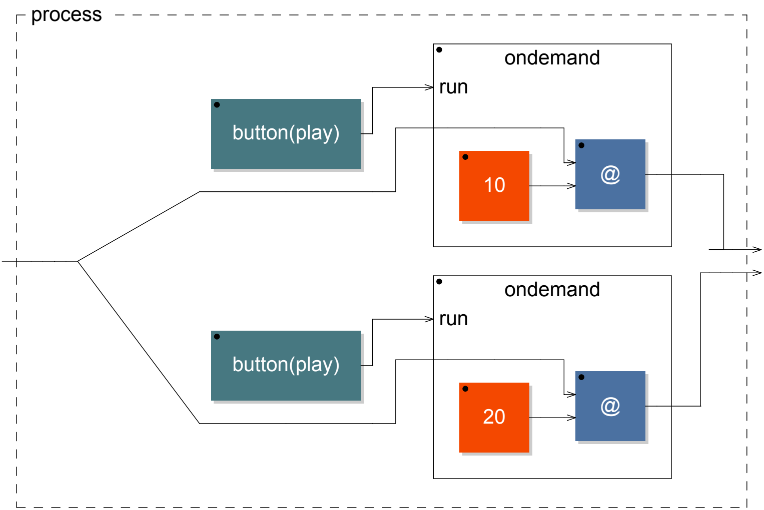


Example 7, instruction graph

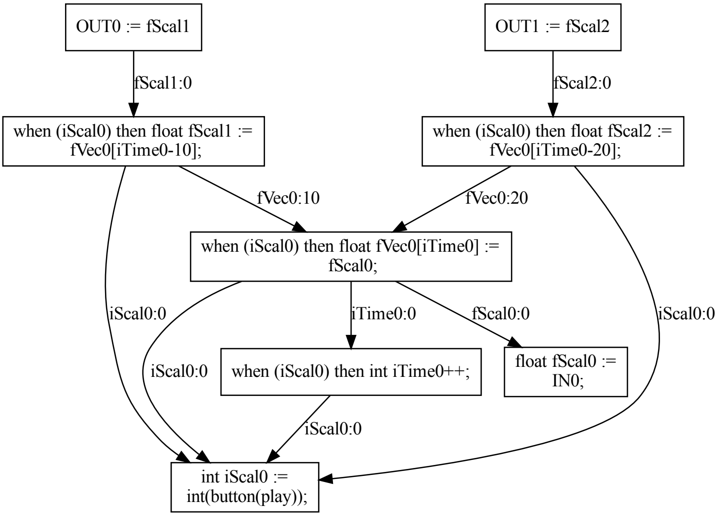


Example 8, block-diagram

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



Example 8, instruction graph



Conclusion

What's missing ?

- Several primitives like tables, waveforms, etc. are missing
- Replace current interval computation
- Proper C++ code generation
- Merge with dev branch
- Future extensions:
  - interleave(P)
  - upsample(N,P)
  - downsample(N,P)
  - modulation
- Code génération improvements:
  - Data Parallelism