Reconstruction of FIRs and IIRs from Faust signals

Yann Orlarey

EMERAUDE

13 fevrier 2024

Introduction

Motivations

- Improve code generation for SYFALA by recreating loops
- Facilitate WCPG calculation for IIRs

Agenda

- Faust signals and their properties
- FIR and IIR extensions
- Rewriting rules
- Code Generation

PART 1: Faust Signals and their properties

Faust Signals Terms $\mathbb{S}$

The set $\mathbb{S}$ is inductively generated by the following grammar:

Signal Terms

$$s \in \mathbb{S} ::= k \mid u \mid \mathtt{I}_n \mid \star(s_1, s_2, ...) \mid s_1 @ s_2 \mid X_i$$

Where:

$r1 : k$: constant number $\in \mathbb{R}$

$r2 : u$: user interface element (slider, button, etc.)

$r3 : \mathtt{I}_n$: audio input channel n

$r4 : \star(s_1, ...)$: numerical expressions ($\star \in \{+, *, \cos, \ldots\}$)

$r5 : s_1 @ s_2$: signal s_1 delayed by s_2 samples

$r6 : X$: symbol representing a tuple of mutually recursive signals

$r7 : X_i$: the i-th projection of X

Moreover, $\mathsf{def}[\![X]\!] \in \mathbb{S}^n$ represent the definition associated with X, and by extension $\mathsf{def}[\![X_i]\!] = \mathsf{def}[\![X]\!][i]$.

Semantics of Faust Signals

A Faust signal $s \in \mathbb{S}$ denotes a function of time, notated $[\![s]\!] : \mathbb{Z} \to \mathbb{R}$. Time can be negative, but computation really starts at $t = 0$. By definition, we have:

$t \geq 0$

$$p1 : [\![k]\!](t) = \text{the constant signal of value } k$$
$$p2 : [\![u]\!] = \text{the signal produced by acting on widget } u$$
$$p3 : [\![\mathtt{I}_n]\!] = \text{the signal on audio input channel } n$$
$$p4 : [\![\star(s_1, ...)]\!](t) = \star([\![s_1]\!](t), ...)$$
$$p5 : [\![s_1 @ s_2]\!](t) = [\![s_1]\!](t - [\![s_2]\!](t))$$
$$p6 : [\![X_i]\!] = [\![\mathsf{def}[\![X_i]\!]]\!]$$

$t < 0$

$$n1 : [\![s]\!](t) = 0$$

Additional Definitions

Constant Signals

A signal s is said *constant* if and only if: $\forall t_1, t_2 \in \mathbb{N}, [\![s]\!](t1) = [\![s]\!](t2)$.

Positive Signals

A signal s is said *positive* if and only if: $\forall t \in \mathbb{N}, [\![s]\!](t) \geq 0$.

Causal Delays

A delay $s_1@s_2$ is said *causal* if and only if s_2 is *positive*. Non-causal delays would imply looking at the future of a signal and are not allowed in Faust. Therefore, for every delay expression $s_1@s_2$, $[\![s_2]\!](t) \geq 0$.

Mutual Recursions

Mutual recursions are only allowed within the same recursive group X, but not between recursive groups. Therefore, dependencies between recursive groups form a DAG.

Equivalence Principle

Equivalence of Signals

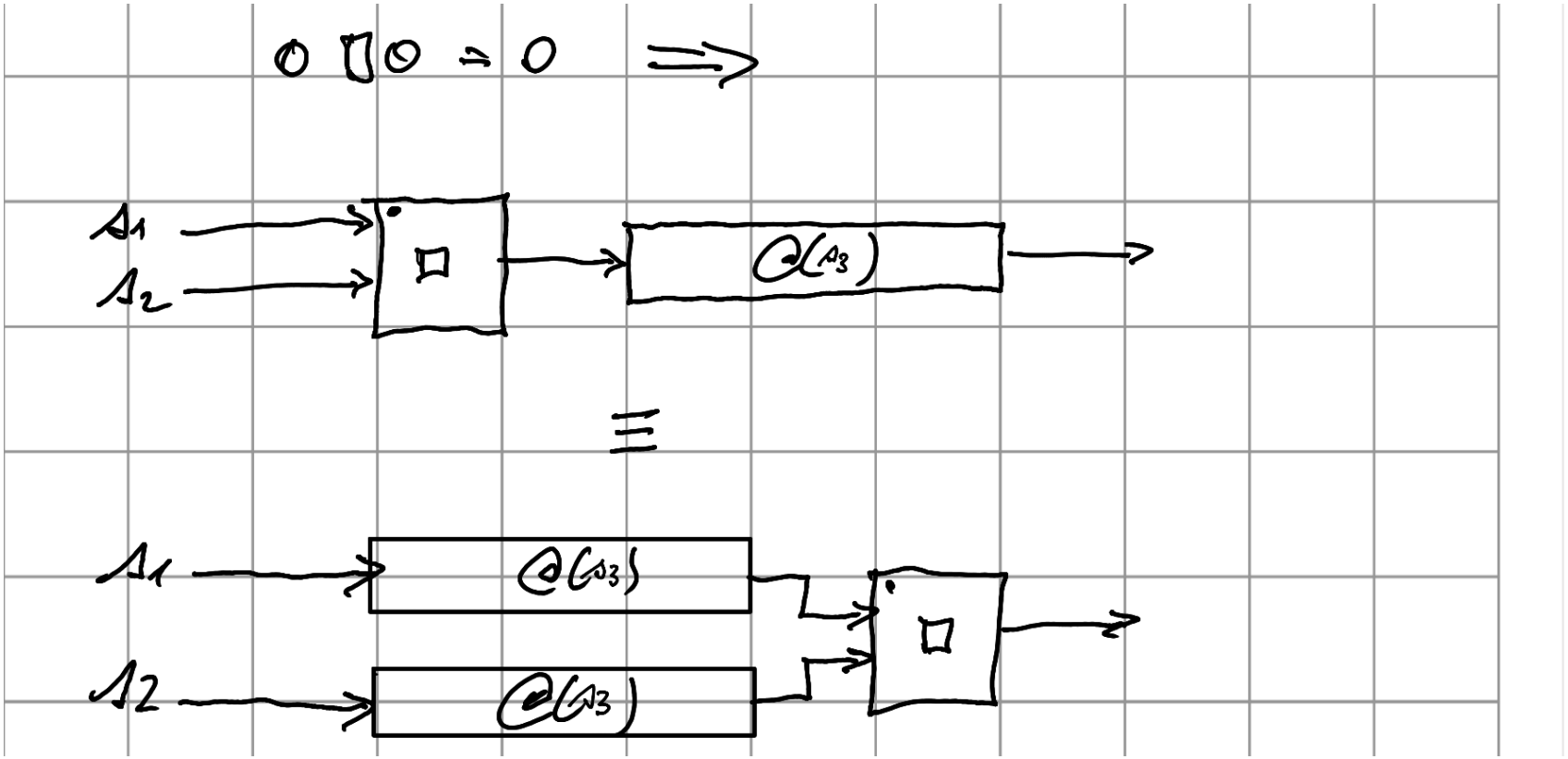
Two signals $s_1, s_2 \in \mathbb{S}$ are said *equivalent*, notated $s_1 \equiv s_2$, if and only if they have the same semantics

$$s_1 \equiv s_2 \Leftrightarrow [\![s_1]\!] = [\![s_2]\!]$$

Distributivity of the delay operation

The delay operation @ is distributive over *numerical operations that are null at origin* ($0 + 0 = 0$, $0 * 0 = 0$, etc.).

$$\star(s_1, s_2, \ldots)@s_3 \equiv \star(s_1@s_3, s_2@s_3, \ldots)$$



Distributivity of the delay operation

We need to prove that $[\![\star(s_1, s_2, \ldots)@s_3]\!] = [\![\star(s_1@s_3, s_2@s_3, \ldots)]\!]$

Case 1: $t \geq [\![s_3]\!](t) \geq 0$

$$\begin{aligned}
[\![\star(s_1, s_2, \ldots)@s_3]\!](t) &= [\![\star(s_1, s_2, \ldots)]\!](t - [\![s_3]\!](t)) \\
&= \star([\![s_1]\!](t - [\![s_3]\!](t)), [\![s_2]\!](t - [\![s_3]\!](t)), \ldots) \\
&= \star([\![s_1@s_3]\!](t), [\![s_2@s_3]\!](t), \ldots) \\
&= [\![\star(s_1@s_3, s_2@s_3, \ldots)]\!](t) \qquad \square
\end{aligned}$$

Distributivity of the delay operation

We need to prove that $[\![\star(s_1, s_2, \ldots)@s_3]\!] = [\![\star(s_1@s_3, s_2@s_3, \ldots)]\!]$

Case 2: $[\![s_3]\!](t) > t \geq 0$

$$
\begin{aligned}
[\![\star(s_1, s_2, \ldots)@s_3]\!](t) &= [\![\star(s_1, s_2, \ldots)]\!](t - [\![s_3]\!](t)) \\
&= 0 \\
[\![\star(s_1@s_3, s_2@s_3, \ldots)]\!](t) &= \star([\![s_1]\!](t - [\![s_3]\!](t)), [\![s_2]\!](t - [\![s_3]\!](t)), \ldots) \\
&= \star(0, 0, \ldots) \\
&= 0
\end{aligned}
$$

□

Other Equivalence Properties of Faust Signals

The following equivalence properties are useful for the reconstruction of FIRs and IIRs in Faust signals:

Equivalence 1

$$(s_1 @ s_2) @ s_3 \equiv s_1 @ (s_2 @ s_3 + s_3)$$

Equivalence 2

$$(s_1 @ k) @ s_3 \equiv s_1 @ (k + s_3)$$

Equivalence 3

$$(s_1 @ s_2) * k \equiv (s_1 * k) @ s_2$$

Proof of Equivalence 1 : $(s_1@s_2)@s_3 \equiv s_1@(s_2@s_3 + s_3)$

We prove that $\forall t, [\![(s_1@s_2)@s_3]\!](t) = [\![s_1@(s_2@s_3 + s_3)]\!](t)$:

Proof

$$
\begin{aligned}
[\![(s_1@s_2)@s_3]\!](t) &= [\![s_1@s_2]\!](t - [\![s_3]\!](t)) \\
&= [\![s_1]\!](t - [\![s_3]\!](t) - [\![s_2]\!](t - [\![s_3]\!](t))) \\
&= [\![s_1]\!](t - [\![s_3]\!](t) - [\![s_2@s_3]\!](t)) \\
&= [\![s_1]\!](t - [\![s_3 + s_2@s_3]\!](t)) \\
&= [\![s_1@(s_3 + s_2@s_3)]\!](t) \\
&= [\![s_1@(s_2@s_3 + s_3)]\!](t) \qquad \square
\end{aligned}
$$

Proof of Equivalence 2 : $(s_1@k)@s_3 \equiv s_1@(k + s_3)$

We prove that $\forall t, [\![(s_1@k)@s_3]\!](t) = [\![s_1@(k + s_3)]\!](t)$ in two cases, when $t - [\![s_3]\!](t) \geq 0$ and when $t - [\![s_3]\!](t) < 0$

Case 1: $t - [\![s_3]\!](t) \geq 0$

$$\begin{aligned}
[\![(s_1@k)@s_3]\!](t) &= [\![s_1@k]\!](t - [\![s_3]\!](t)) \\
&= [\![s_1]\!](t - [\![s_3]\!](t) - [\![k]\!](t - [\![s_3]\!](t))) \\
&= [\![s_1]\!](t - [\![s_3]\!](t) - k) \\
&= [\![s_1]\!](t - [\![s_3]\!](t) - [\![k]\!](t)) \\
&= [\![s_1]\!](t - [\![s_3 + k]\!](t)) \\
&= [\![s_1@(s_3 + k)]\!](t) \qquad \square
\end{aligned}$$

Proof of Equivalence 2 : $(s_1@k)@s_3 \equiv s_1@(k + s_3)$

We prove that, when $t - [\![s_3]\!](t) < 0$, $[\![(s_1@k)@s_3]\!](t) = 0$ and $[\![s_1@(k + s_3)]\!](t) = 0$

Case 2a: $t - [\![s_3]\!](t) < 0 \Rightarrow [\![(s_1@k)@s_3]\!](t) = 0$

$$\begin{aligned}
[\![(s_1@k)@s_3]\!](t) &= [\![s_1@k]\!](t - [\![s_3]\!](t)) \\
&= [\![s_1]\!](t - [\![s_3]\!](t) - [\![k]\!](t - [\![s_3]\!](t))) \\
&= [\![s_1]\!](t - [\![s_3]\!](t) - 0) \\
&= [\![s_1]\!](t - [\![s_3]\!](t)) \\
&= 0
\end{aligned}$$

□

Case 2b: $t - [\![s_3]\!](t) < 0 \Rightarrow [\![s_1@(k + s_3)]\!](t) = 0$

$$\begin{aligned}
[\![s_1@(k + s_3)]\!](t) &= [\![s_1]\!](t - [\![k + s_3]\!](t)) \\
&= [\![s_1]\!](t - [\![k]\!](t) - [\![s_3]\!](t)) \\
&= 0
\end{aligned}$$

□

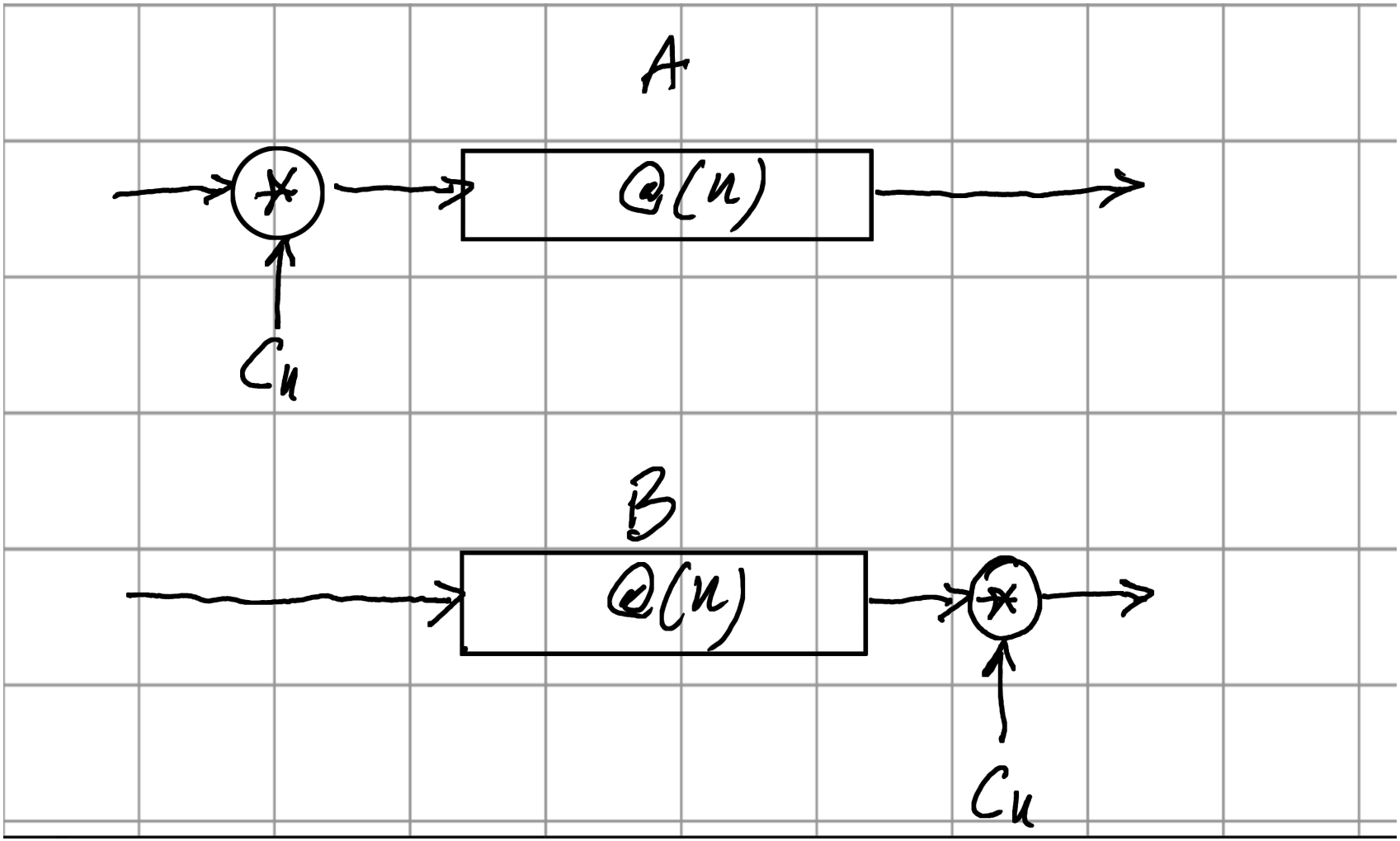
Proof of Equivalence 3 : $(s_1@s_2) * k \equiv (s_1 * k)@s_2$

We prove that $\forall t, [\![(s_1@s_2) * k]\!](t) = [\![(s_1 * k)@s_2]\!](t)$

Proof

$$
\begin{aligned}
[\![(s_1@s_2) * k]\!](t) &= [\![s_1@s_2]\!](t) * [\![k]\!](t) \\
&= [\![s_1@s_2]\!](t) * k \\
&= [\![s_1]\!](t - [\![s_2]\!](t)) * k \\
&= [\![s_1]\!](t - [\![s_2]\!](t)) * [\![k]\!](t - [\![s_2]\!](t)) \\
&= [\![s_1 * k]\!](t - [\![s_2]\!](t)) \\
&= [\![(s_1 * k)@s_2]\!](t) \qquad \square
\end{aligned}
$$

Proof of Equivalence 3 : $(s_1@n) * c_n \equiv (s_1 * c_n)@n$



Proof of Equivalence 3 : $(s_1@n) * c_n \equiv (s_1 * c_n)@n$

Relaxed equivalence

$$\begin{aligned} c_n \text{ constant} &\Rightarrow [\![A]\!] = [\![B]\!] \\ c_n \text{ variable} &\Rightarrow [\![A]\!] \neq [\![B]\!] \\ c_n \text{ slow and } n \text{ small} &\Rightarrow [\![A]\!] \approx [\![B]\!] \end{aligned}$$

PART 2: FIR and IIR Extensions

Extension of Faust Signals with IIRs and FIRs

We extend the signal grammar with two new terms: FIR and IIR.

Extended Signals $\mathbb{S}_e$

$$s, c \in \mathbb{S}_e ::= k \mid u \mid \mathtt{I}_n \mid \star(s_1, ...) \mid s_1 @ s_2 \mid X_i$$
$$\mid \mathsf{FIR}(s, c_0, c_1, ..., c_n)$$
$$\mid \mathsf{IIR}(X_i, s, c_0, c_1, ..., c_n)$$

Definition of FIR and IIR Signals

FIR and IIR are defined as follows:

FIR$(s, c_0, c_1, c_2 ...)$

$c_0 * s + c_1 * s@1 + c_2 * s@2 + ...$

IIR$(X_i, s, c_0, c_1, c_2, ...)$

def$[\![X_i]\!] = s + c_1 * X_i@1 + c_2 * X_i@2 + ...$ (usually $c_0 = 0$)

PART 3: Reconstruction Rewriting Rules

FIR Rewriting Rules

Goal

$$(c_0 * s + c_1 * s@1 + c_2 * s@2 + ...) \rightarrow \mathsf{FIR}(s, c_0, c_1, c_2...)$$

Rules

$w1 : s@k \rightarrow \mathsf{FIR}(s, 0, .\overset{k}{.}., 1)$

$w2 : \mathsf{FIR}(s, c_0, c_1, ...)@k \rightarrow \mathsf{FIR}(s, 0, .\overset{k}{.}., c_0, c_1, ...)$ $\quad c_i$ constant

$w3 : c * \mathsf{FIR}(s, c_0, c_1, ...) \rightarrow \mathsf{FIR}(s, c * c_0, c * c_1, ...)$

$w4 : \mathsf{FIR}(s, c_0, c_1, ...) + \mathsf{FIR}(s, c'_0, c'_1, ...) \rightarrow \mathsf{FIR}(s, c_0 + c'_0, c_1 + c'_1, ...)$

$w5 : \mathsf{FIR}(s_1, c_0, c_1, ...) + \mathsf{FIR}(s_2, c_0, c_1, ...) \rightarrow \mathsf{FIR}(s_1 + s_2, c_0, c_1, ...)$

$w6 : \mathsf{FIR}(s_1, c_0, c_1, ...) + \mathsf{FIR}(s_2, -c_0, -c_1, ...) \rightarrow \mathsf{FIR}(s_1 - s_2, c_0, c_1, ...)$

$w7 : c * s + \mathsf{FIR}(s, c_0, c_1, ...) \rightarrow \mathsf{FIR}(s, c + c_0, c_1, ...)$

Where $0, .\overset{k}{.}.$ means 0 repeated k times. For example: $0, .\overset{3}{.}. = 0, 0, 0$

IIR Rewriting Rules

Goal

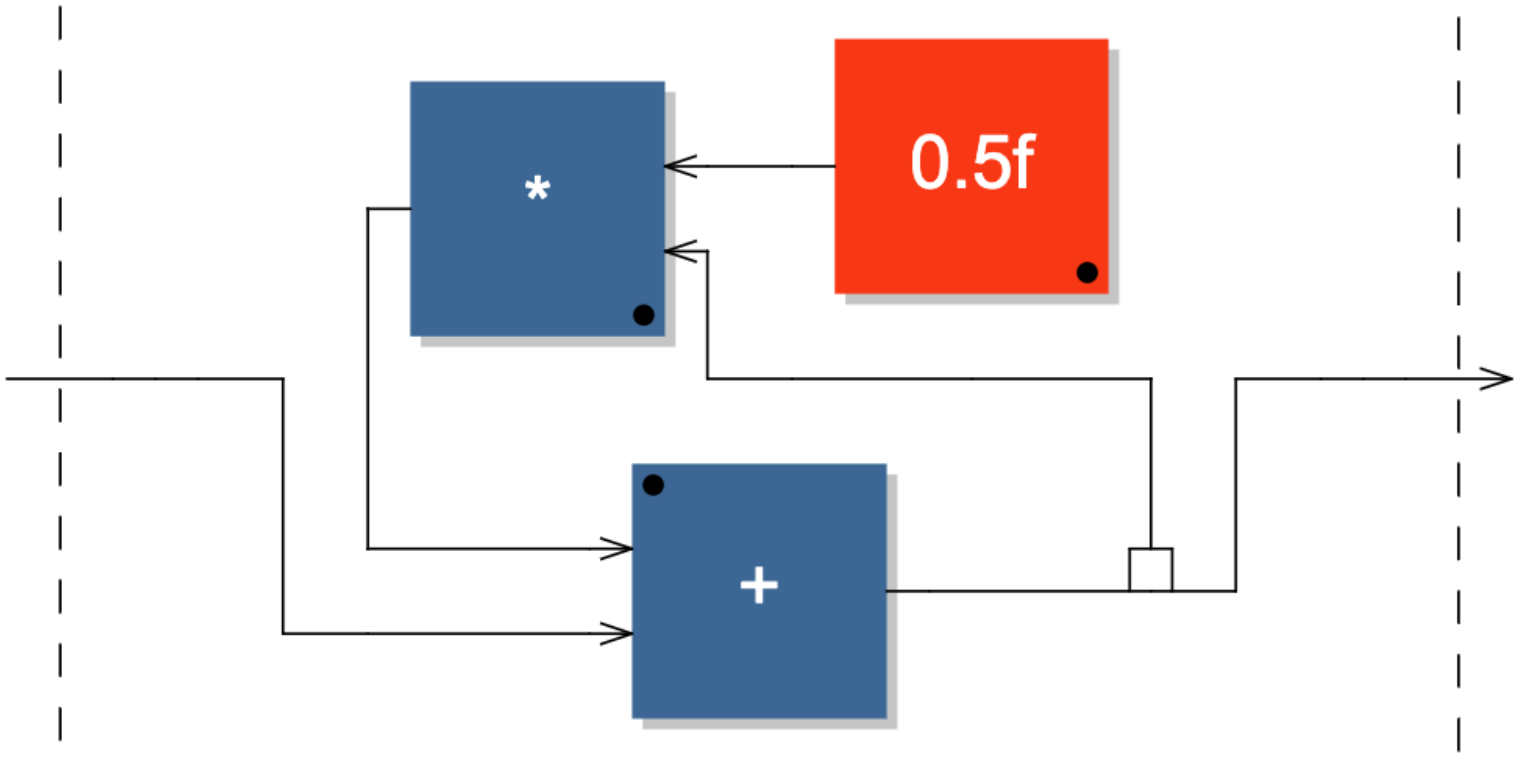
$$\mathsf{def}[\![X_i]\!] = s + c_1 * X_i@1 + c_2 * X_i@2 + ... \rightarrow \mathsf{IIR}(X_i, s, 0, c_1, c_2...)$$

Rules

$$w8 : s_1 + \mathsf{FIR}(X_i, 0, c_1, ...) \rightarrow \mathsf{IIR}(X_i, s_1, 0, c_1, ...) \quad (X_i \notin s_1)$$

$$w9 : s \rightarrow \varnothing$$

Example 1: `process = + ~ *(0.5);`

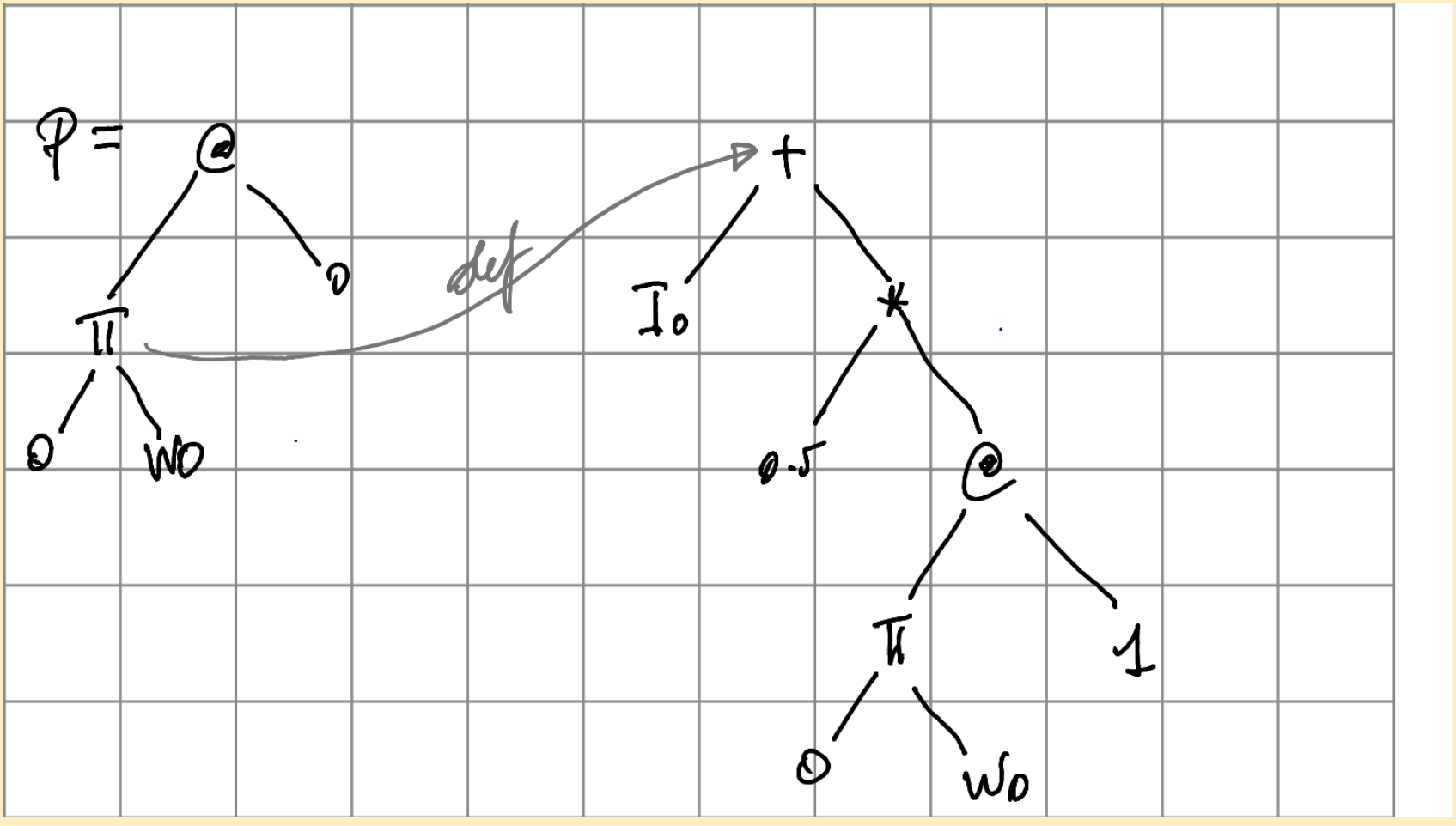


FIR and IIR Representation

```
IIR[VOID, IN[0], 0.0f, 0.5f]
```

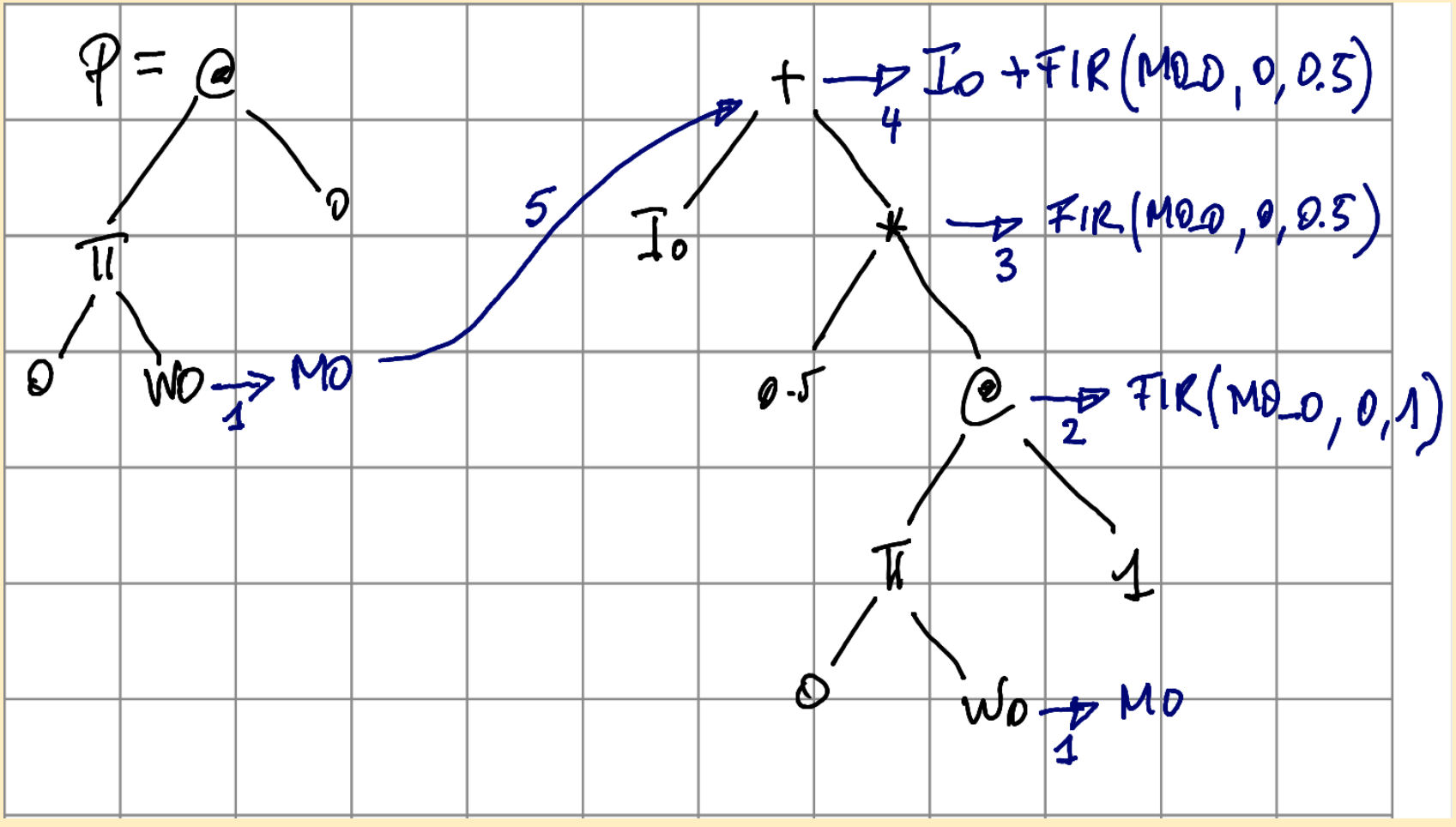
Example 1: `process = + ~ *(0.5);`

Signal Representation



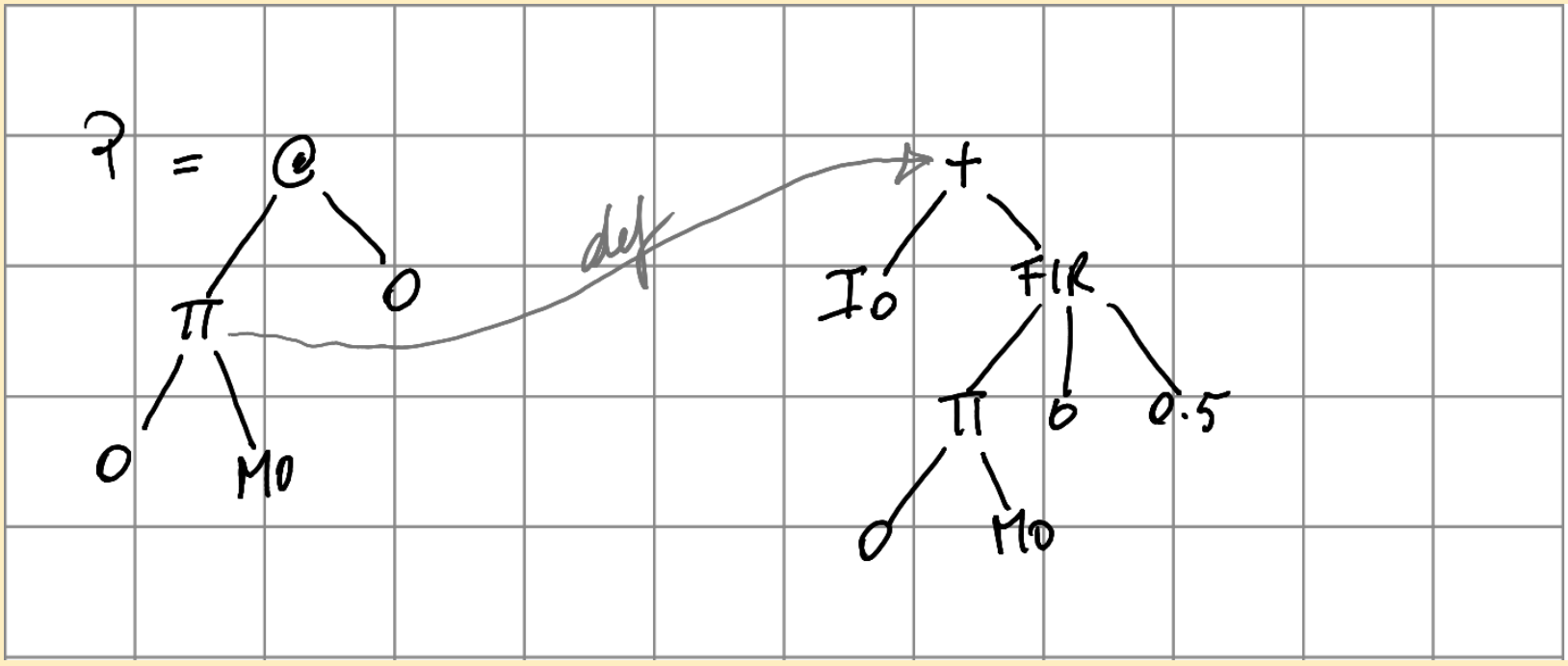
Example 1: `process = + ~ *(0.5);`

FIR Rewriting



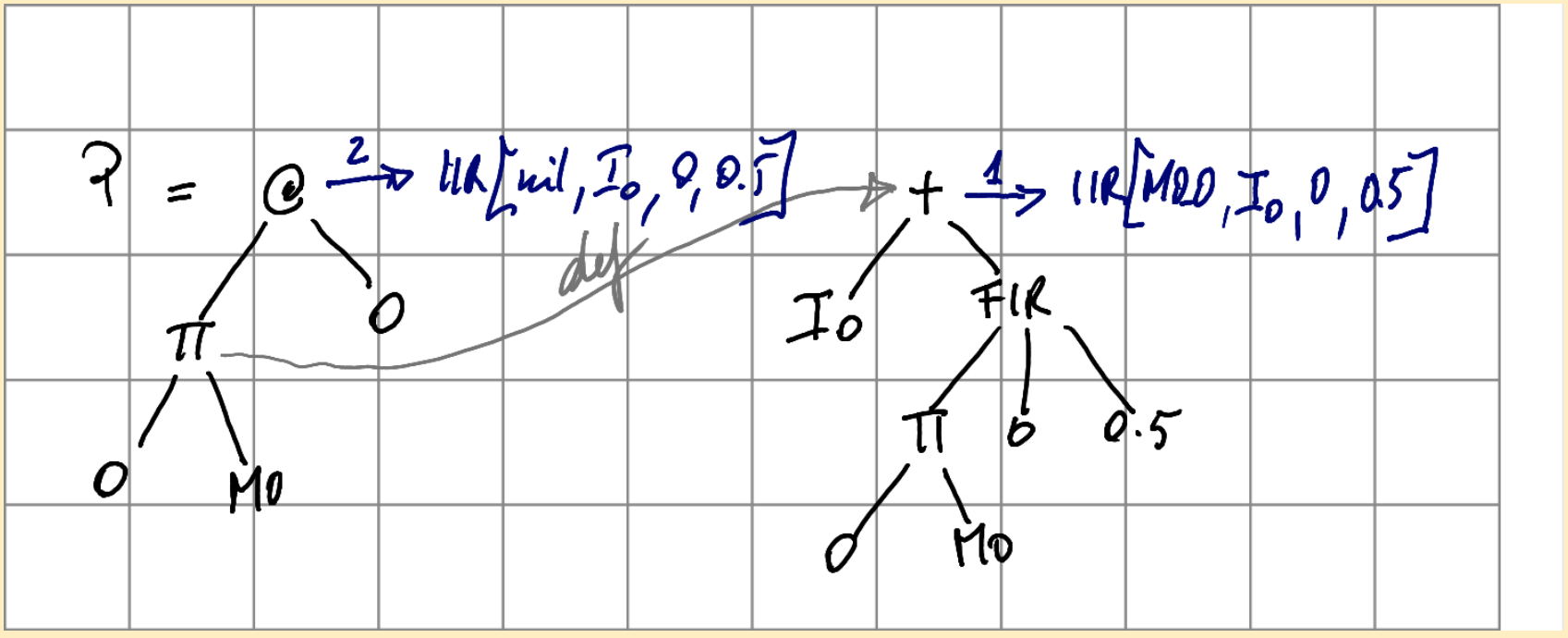
Example 1: `process = + ~ *(0.5);`

After FIR Rewriting



Example 1: `process = + ~ *(0.5);`

IIR Rewriting



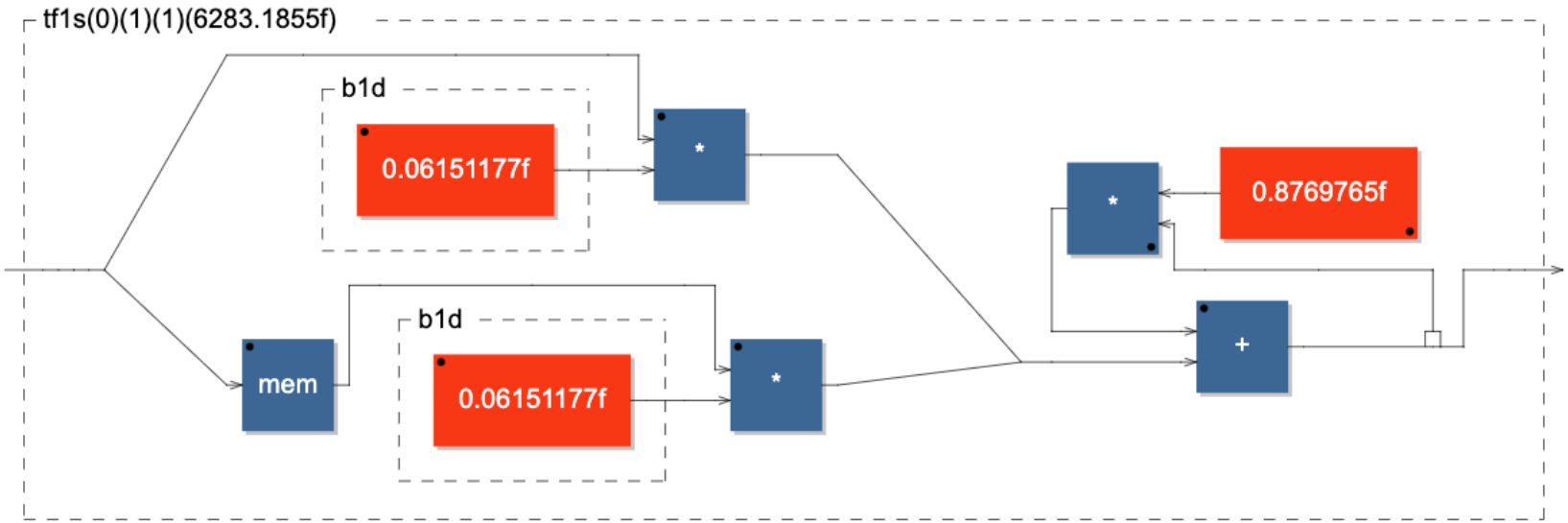
Example 2: `process = fi.lowpass(3, 1000);`

FIR and IIR Representation

```
s1 = FIR[IN[0], 0.06151177f, 0.06151177f];
s2 = IIR[VOID, s1, 0.0f, 0.8769765f];
s3 = IIR[VOID, s2, 0.0f, 1.8614085f, -0.8774705f]
s4 = FIR[s3, 0.004015505f, 0.00803101f, 0.004015505f]
```

Example 2: `process = fi.lowpass(3, 1000);`

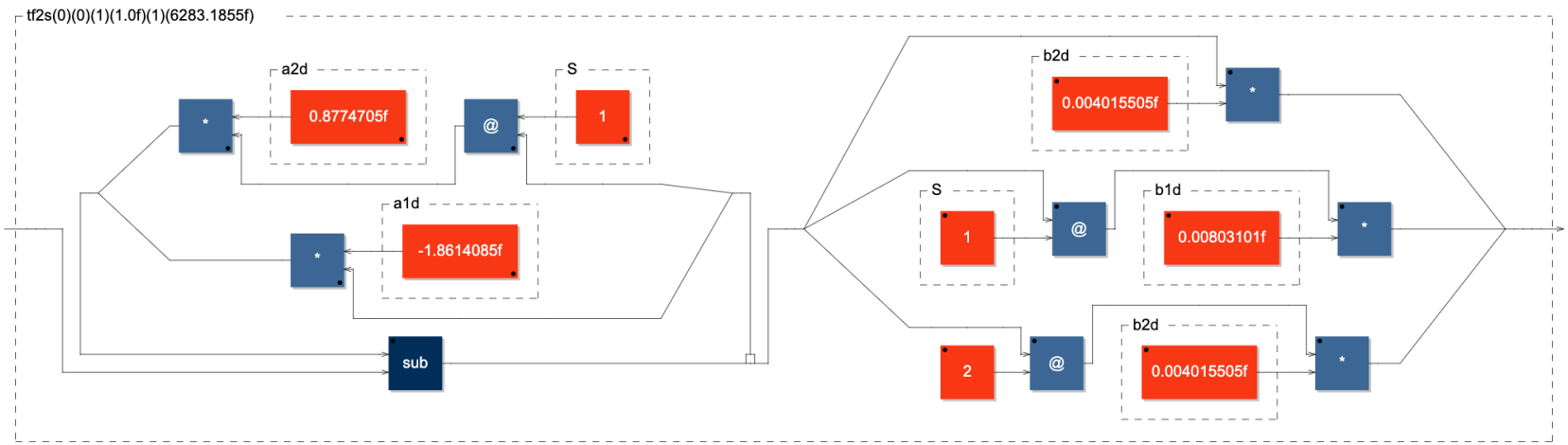
FIR and IIR Representation




```
s1 = FIR[IN[0], 0.06151177f, 0.06151177f];
s2 = IIR[VOID, s1, 0.0f, 0.8769765f];
```

Example 2: `process = fi.lowpass(3, 1000);`

FIR and IIR Representation




```
s3 = IIR[VOID, s2, 0.0f, 1.8614085f, -0.8774705f]
s4 = FIR[s3, 0.004015505f, 0.00803101f, 0.004015505f]
```

PART 4: Code Generation

Comparing Current Compilation Strategies

Comparing the current compilation strategies for various IIRs and FIRs examples, in particular vector and scalar modes.

Test cases

- iir4
- fir16
- fir16:fir16
- iir4:fir16
- fir16:iir4

Compilation modes

- vector loop variant 0
- vector loop variant 1
- scalar
- experimental scalar mode

Source Code

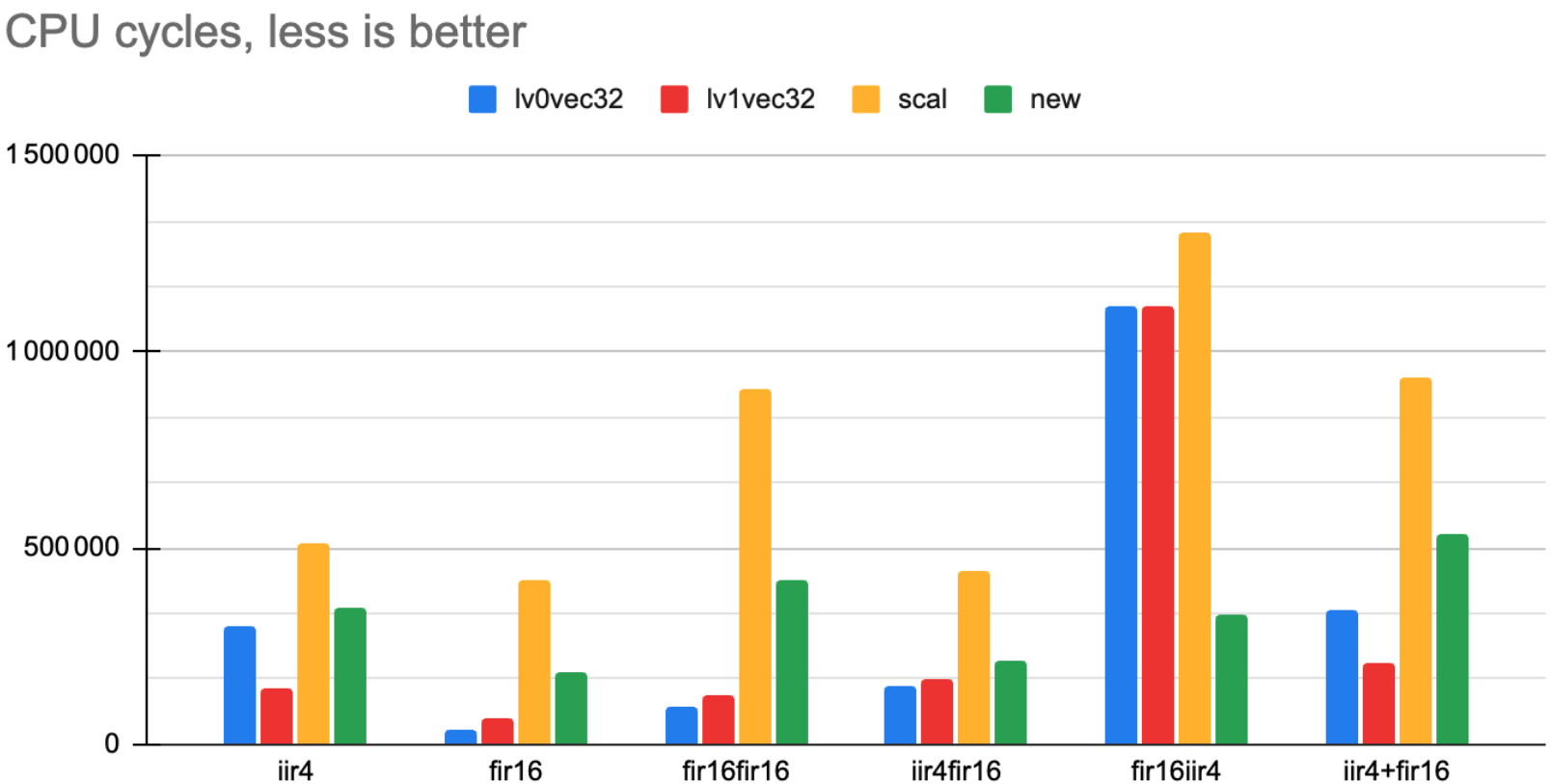
```
import("stdfaust.lib");

coeffs(N) = par(i, N, 1.0/float(i+2));
coeffsb(N) = par(i, N, 1.0/float(i+2.1));

iir4 = +~fi.fir(coeffs(4));
fir16 = fi.fir(coeffs(16));
fir16fir16 = fi.fir(coeffs(16)) : fi.fir(coeffsb(16));
iir4fir16 = +~fi.fir(coeffs(4)) : fi.fir(coeffs(16));
fir16iir4 = fi.fir(coeffs(16)) : +~fi.fir(coeffs(4));
```

Results

	iir4	fir16	fir16fir16	iir4fir16	fir16iir4	iir4+fir16
lv0vec32	300 507	39 023	97 848	147 819	1 115 630	339 530
lv1vec32	145 199	64 206	123 549	168 292	1 116 570	209 405
scal	514 765	417 509	908 155	441 582	1 302 560	932 274
new	350 846	183 467	419 829	212 594	330 260	534 313



Vector Mode vs Handwritten Strategies for FIR16

Vector Mode (Loop Variant 0)

```
for (int i = 0; i < vsize; i = i + 1) {
    output0[i] = FAUSTFLOAT(0.5f * float(input0[i])
            + 0.33333334f * fYec0[i - 1]
            + 0.25f * fYec0[i - 2] + 0.2f * fYec0[i - 3]
            + 0.16666667f * fYec0[i - 4]
            + 0.14285715f * fYec0[i - 5]
            + 0.125f * fYec0[i - 6]
            + 0.11111111f * fYec0[i - 7]
            + 0.1f * fYec0[i - 8]
            + 0.09090909f * fYec0[i - 9]
            + 0.083333336f * fYec0[i - 10]
            + 0.07692308f * fYec0[i - 11]
            + 0.071428575f * fYec0[i - 12]
            + 0.06666667f * fYec0[i - 13]
            + 0.0625f * fYec0[i - 14]
            + 0.05882353f * fYec0[i - 15]);
}
```

Vector Mode vs Handwritten Strategies for FIR16

Coefficients Loop

```
float coef[] = {0.5f, 0.33333334f, 0.25f, 0.2f, 0.16666667f,
                0.14285715f, 0.125f, 0.11111111f, 0.1f,
                0.09090909f, 0.083333336f, 0.07692308f,
                0.071428575f, 0.06666667f,  0.0625f,
                0.05882353f};

for (int i = 0; i < vsize; i = i + 1) {
    float sum = 0.0f;
    for (int c = 0; c < 16; c++) {
        sum += coef[c] * fYec0[i - c];
    }
    output0[i] = FAUSTFLOAT(sum);
}
```

Vector Mode vs Handwritten Strategies for FIR16

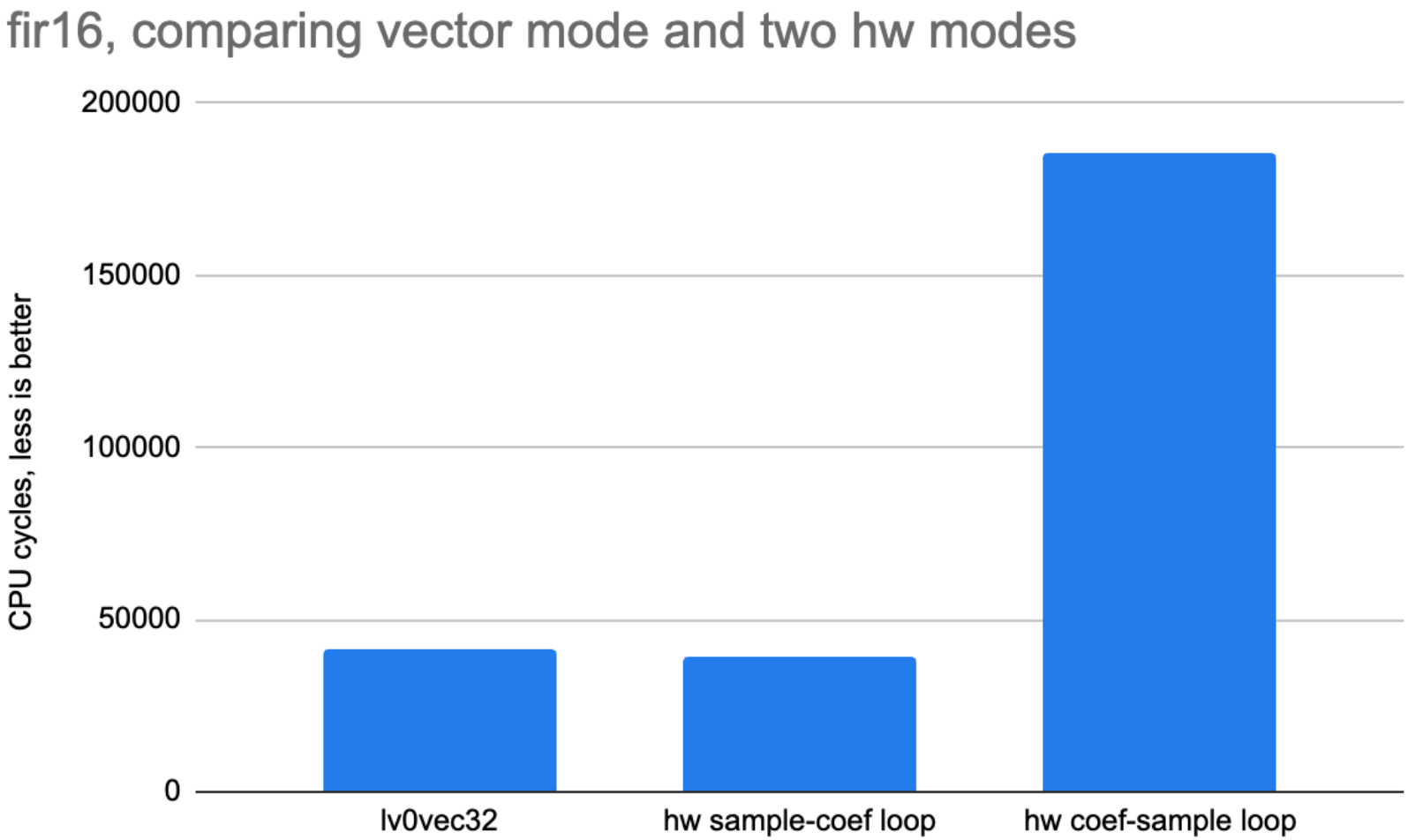
Switched Coefficients Samples Loop (à la SYFALA)

```
float coef[] = {0.5f, 0.33333334f, 0.25f, 0.2f, 0.16666667f,
                0.14285715f, 0.125f, 0.11111111f, 0.1f,
                0.09090909f, 0.083333336f, 0.07692308f,
                0.071428575f, 0.06666667f,  0.0625f,
                0.05882353f};

for (int i = 0; i < vsize; i = i + 1) {
    output0[i] = 0.0f;
}
for (int c = 0; c < 16; c++) {
    float* src = &fYec0[-c];
    for (int i = 0; i < vsize; i = i + 1) {
        output0[i] += FAUSTFLOAT(coef[c] * src[i]);
    }
}
```

Results

	fir16
lv0vec32	41648
hw sample-coef	38920
hw coef-sample	185299



PART 5: What's remaining?

What's remaining?

- Code generation;
- WCPG calculation;
- Tests on FPGA;
- Grouping of parallel filters.