A Taste of Sound Reasoning in Faust

Emilio Jesús Gallego Arias, Olivier Hermant, Pierre Jouvelot

A Taste of Sound Reasoning in Faust

Emilio Jesús Gallego Arias, Olivier Hermant, Pierre Jouvelot

MINES ParisTech, PSL Research University, France
{emilio.gallego_arias, olivier.hermant, pierre.jouvelot}@mines-paristech.fr

Abstract

We address the question of what software verification can do for the audio community by showcasing some preliminary design ideas and tools for a new framework dedicated to the formal reasoning about Faust programs. We use as a foundation one of the strongest current proof assistants, namely Coq combined with SSReflect. We illustrate the practical impact of our approach via a use case, namely the proof that the implementation of a simple low-pass filter written in the Faust audio programming language indeed meets one of its specification properties.

The paper thus serves three purposes: (1) to provide a gentle introduction to the use of formal tools to the audio community, (2) to put forward programming and formal reasoning paradigms we think are well suited to the audio domain and (3) to illustrate this approach on a simple yet practical audio signal processing example, a low-pass filter.

Keywords

DSP; audio; program verification; theorem proving

1 Introduction

Formal program verification is gaining strong support in the computer programming world with projects such as the CompCert certified C compiler [Leroy, 2009], with more and more tools such as the Coq[1] proof assistant striving to ease the development of correctness proofs for hopefully the every-day programmer.

While there has been some work in the mathematical correctness of DSP — see for instance [Krishnaswami, 2013; Brunel et al., 2014] for type-based techniques, and [Souari et al., 2014; Ghafari et al., 2011] for approaches using theorem proving — formal verification is still largely absent in the DSP and Computer Music (CM) communities. Yet, users and musicians are always striving for ever better sound experience and audio realism. Thus, ensuring the adequacy between an intended audio specification, for instance some sort of a limited-bandwidth filtering, and its actual implementation, along with other key properties such as robustness [Chaudhuri et al., 2012] or Bounded-Input Bounded-Output (BIBO) stability, is warranted. Too often, the only correctness test performed is to check that "it sounds about right", which is a methodology that clearly deserves some improvements.

The overall goal of our paper is to provide a case for the introduction of tools and best practices dedicated to the formal mathematical reasoning about audio/sound application programs. We illustrate this vision via the use of Coq/SSReflect for the Faust audio language[2]. We introduce a new framework for mathematically reasoning about Faust audio programs. We show how simple properties can be already proven for some Faust filtering applications using such an approach, thus paving the way to the future introduction of dedicated proof techniques for audio processing systems.

This paper is structured as follows. In Section 2, we introduce the Coq proof assistant and its SSReflect extension, which we use all along. Section 3 describes the Faust language core, using a low-pass filter as an example, while Section 4 provides a Coq implementation. We introduce in Section 5 a specific logic that will help reasoning about Faust programs. We finally put these tools into good use in Section 6, where we show how the low-pass filter can be proven equivalent to its specification. We discuss future work and conclude in Section 7.

2 A Coq/SSReflect Tour

Coq [Bertot and Castéran, 2004] is a software development environment in which programmers can write functional programs and prove properties about them. Coq's programming language Gallina is very similar to other functional programming languages like Ocaml, but with some added restrictions (in particular, all Coq pro-

[1] `coq.inria.fr`

[2] `http://faust.grame.fr`

grams must terminate) and a more advanced type system. Coq is a *strongly-typed* language; in particular, a type is understood as a property P, and an expression e of type P, written `e : P`, is the proof for P.

SSReflect [Gonthier et al., 2008] is a proof language and extensive library built on top of Coq. It promotes a structured style of programming and facilitates proof development by profiting from the fact that many properties of interest can be expressed as programs of type boolean.

Recursive Definitions We will illustrate some basics of the SSReflect's proof language by proving a toy property over a toy programming language. We first load some required libraries:

```
Require Import ssrfun ssrbool eqtype ssrnat.
```

The Abstract Syntax Tree (AST) of our toy programming language is defined in Coq with the `Inductive` command:

```
Inductive exp : Type :=
| cst : nat → exp
| mul : exp → exp → exp.
Notation "' n" := (cst n).
Notation "a ×b" := (mul a b).
```

which declares the recursive (so-called inductive) type `exp` with two constructors, `cst` for embedding natural integer constants, and `mul` for the multiplication of two expressions. Coq provides support to declare convenient notations such as × for the multiplication operation.

Recursive functions are declared in Coq using the `Fixpoint` command. Then, the value of an expression in our toy language is defined by recursion over its structure:

```
Fixpoint eval (e : exp) : nat :=
  match e with
    | 'n => n
    | e1 × e2 => eval e1 * eval e2
end.
```

We can run our program with the `Eval` command:

```
Eval compute in eval ('2 ×('0 × '3)).
```

which will display, as expected, `0`. Going one step further, Ocaml code can be generated automatically from Coq programs, providing reasonably efficient and provably-correct implementations of the specified algorithms.

Proving Properties Following upon our previous simple test, assume we want now to *prove* the following, 0-absorbing property, named `eval0eB`: all expressions `e` that contain a `'0` subexpression evaluate to 0. We can write a boolean function `mem_exp` that checks if `'0` occurs in an expression `e` easily[3]:

```
Fixpoint mem_exp m e :=
  match e with
    | 'n => n == m
    | e1 × e2 => mem_exp m e1 || mem_exp m e2
  end.
```

Theorem `eval0eB` is rephrased now as: for all expressions `e`, if `mem_exp 0 e` is `true`, then `eval e` is `0`. Let's begin by proving an easier property for constant expressions; it is stated here in Coq as Lemma `eval0cB`, followed by its proof:

```
Lemma eval0cB :
  forall n, mem_exp 0 ('n) → eval ('n) == 0.
Proof. by move=> n /=; exact. Qed.
```

In addition to programming in Gallina, we can also use automated program building procedures called *tactics*. We can use an interactive proof editor such as Proof General of CoqIDE to step from `Proof` to `Qed`; after each tactic (terminated by a `.`), the current proof state is displayed. The proof state consists of a set of hypotheses $e : t$ (the "context") and a goal g which should be seen as a stack of properties p_0->....->p_n->g. Most SSReflect tactics `tac` can perform context manipulation operations before and after running. `tac`: `x` will remove a named hypotheses `x` : p from the context, pushing p to the goal before `tac` is run; `tac` => `x` will pop the top of the goal after `tac` is run, naming it `x`. Thus, if p_0->A is the current goal, `move=> x` will add `x` : p_0 to the context. Plenty of additional operations can be performed in addition to moving.

The previous proof started with the documentation-only `Proof` command, getting the goal:

```
forall n, mem_exp 0 ('n) → eval ('n) == 0.
```

with an empty context. The first step is to move n to the context with `move=> n /=`. The `/=` switch asks Coq to perform partial evaluation of the goal, resulting in a new goal `n == 0 → n == 0`, which the `exact` tactic solves. Finally, `Qed` checks that the proof is correct, and, as for all previous function and type definitions, `eval0cB` is added to the global context for further use.

Moving then to our full theorem, we state:

```
Theorem eval0eB :
  forall e, mem_exp 0 e → eval e == 0.
Proof.
  elim.
  – by apply: eval0cB.
  – move=> e1 H1 e2 H2 /=.
```

[3] Coq uses type inference to get the types of `m` and `e`.

```
    case/orP=> [/H1 | /H2] /eqP → .
    + by rewrite mul0n.
    + by rewrite muln0.
Qed.
```

Here, the proof starts by performing induction on the structure of expressions. The induction tactic `elim` operates on the top of the goal, which in this case is the expression e. The base and inductive subgoals are generated, displayed as:

```
subgoal 1 (ID 18) is:
forall n : nat, mem_exp 0 (' n) →
      eval (' n) == 0

subgoal 2 (ID 19) is:
forall e : exp,
 (mem_exp 0 e → eval e == 0) →
  forall e0 : exp,
   (mem_exp 0 e0 → eval e0 == 0) →
    mem_exp 0 (e ×e0) → eval (e ×e0) == 0
```

We must prove each goal separately (the `-` and `+` symbols indicate a new proof step). The base case is dealt with by first using Lemma `eval0cB` from the global context with the `apply` tactic (the `by` so-called "closing tactical" ensures that the current goal is finished). In the inductive case, the goal includes the facts that `eval0eB` is true on each subexpression, named here `e` and `e0` by Coq. We first move `e` and `e0`, renamed `e1` and `e2`, to the context, each followed by its induction hypotheses $\texttt{H}_i$, before performing a partial evaluation. After the move, we have the following proof state:

```
e1 : exp
H1 : mem_exp 0 e1 → eval e1 == 0
e2 : exp
H2 : mem_exp 0 e2 → eval e2 == 0
============================
mem_exp 0 e1 || mem_exp 0 e2 →
    eval e1 * eval e2 == 0
```

The top of the goal is a boolean disjunction, on which we can perform case analysis using the `case` tactic with the `orP` view. Each of the two resulting cases happen to be the premise of the induction hypotheses, `H1` and `H2`; thus the disjunctive pattern `[/H1|/H2]` will rewrite the goal to:

```
eval e1 == 0 → eval e1 * eval e2 == 0
```

and similarly for `e2`. The pattern `/eqP→` will rewrite `eval e1` to 0 in the goal obtaining a goal of `0 * eval e2 == 0`.

This particular step is paradigmatic of the "proof by rewriting" technique: from a property `a = b`, we can replace all `a` terms occurring in the goal by `b`. The final step of the proof is again by rewriting, this time using the `mul0n` and `muln0` lemmas part of the `ssrnat` library, with type `mul0n : forall x, 0 * x = 0` and symmetrically.

Rewriting Magic SSReflect's emphasis on boolean expressions fosters proof by rewriting in arithmetic proofs. For instance, in the following lemma:

```
Lemma leq_2add :
  forall (x y : nat), x <= y → x + x <= y + y.
Proof. by move=> x y xy; rewrite leq_add. Qed.
```

the proof, where ";" combines proof steps, is done by rewriting with the `leq_add` lemma, provided by the `ssrnat` library, of type:

```
leq_add : forall (m1 m2 n1 n2 : nat),
   m1 <= n1 → m2 <= n2 → m1 + m2 <= n1 + n2
```

How could that happen? Equalities do not seem to occur in the conclusion of `leq_add`, but they have actually just been removed by the pretty printer. In fact, the exact conclusion is a boolean equality, namely `m1 + m2 <= n1 + n2 = true`, as the resulting type of the `<=` operator is boolean. Thus, `x + x <= y + y = true` can be easily matched and rewritten to true with proper bindings of `leq_add` variables, leading to the goal `true = true`.

Going Further It is obviously impossible to give a complete overview of SSReflect in two pages. In particular, SSReflect advocates particular conventions and coding styles that we did not fully follow for the sake of space and pedagogy. In particular, the idiomatic proof for Theorem `eval0eB` is:

```
by elim=> //= ? IH1 ? IH2 /orP
   [/ IH1|/IH2] /eqP → ; rewrite ?(mul0n, muln0).
```

The Mathematical Components library is a companion project to SSReflect, and includes extensive libraries about finite groups, algebra, number theory and more, which have been used to formally prove significant large theorems like the Four Color or Feit-Thompson theorems.

3 The Faust Audio Language

Faust— which stands for "Functional Audio Stream" — is a DSP language [Orlarey et al., 2004]. Faust's main focus is the fast development of efficient digital audio programs, and has been used in live performances and offline and online audio applications, such as FaustLive [Denoux et al., 2014] or moForte [4]. It has also been used in other signal processing contexts [Barkati et al., 2014].

[4] www.moforte.com/

Framework FAUST programs are structured in two layers. The high-level layer is a macro language corresponding to an untyped, full-fledged functional language. It is used to generate programs written in the *Core* FAUST syntax, where only a few primitives are available. In order to deal with the real-time constraints imposed by audio processing, FAUST programs are then optimized and compiled towards an efficient language; FAUST's main compiler generates `C++` code, while alternative backends can now generate other formats, such as `asm.js` code.

While the core language is unpractical for writing real applications, it contains enough primitives for the rest of this paper. Moreover, it enjoys a strong type system and operational semantics. For a more precise description of FAUST syntax and semantics, we refer the reader to [Jouvelot and Orlarey, 2011].

Semantics *Signal processors* are FAUST's key components. A signal is a (potentially infinite) stream of amplitude values. In the discrete case, we can represent signals as functions from (discrete) time to real numbers. Formally, we write $\mathbb{S}$ for the function space $\mathbb{N} \to \mathbb{R}$. We assume that signals have amplitude 0 when time is negative. Signal processors from i inputs to o outputs are functions in $\mathbb{S}^i \to \mathbb{S}^o$, and every valid FAUST program can be interpreted as a function of this type. We use "semantics brackets" to denote the function that maps a FAUST program to its mathematical interpretation. For instance, the *delay* processor, which delays its input signal by 1 audio sample, is denoted as:

$$[\![\texttt{delay}]\!](i)(n) = i(n-1).$$

The first argument i is the signal to be delayed, and n represents time. Then, `delay` will simply return the sample from i corresponding to the previous time value.

Core FAUST For the purposes of this paper, we focus on a minimal but functional subset of FAUST consisting of three particular signal processor-building blocks: primitives, composition, and feedback.

Examples of primitive signal processors are $+$, which takes as input two signals and outputs a signal with amplitude the sum of the input signal amplitudes, or $*(\texttt{c})$, that scales the amplitude by a constant factor c. Formally:

$$\begin{aligned}[\![+]\!](i_1, i_2)(n) &= i_1(n) + i_2(n)\\ [\![*(c)]\!](i_1)(n) &= c * i_1(n).\end{aligned}$$

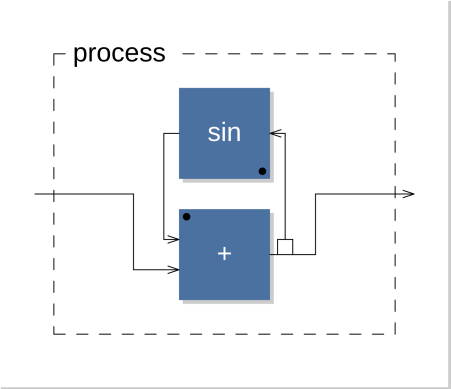

Figure 1: Simple Feedback in FAUST

We write $\texttt{f} : \texttt{g}$ for the composition of signal processors `f` and `g`:

$$[\![\texttt{f} \ : \ \texttt{g}]\!](i) \ = [\![\texttt{g}]\!]([\![\texttt{f}]\!](i)).$$

The interesting construction is feedback, written as $\texttt{f} \sim \texttt{g}$. In this case, `f` is assumed be a processor from two signals to one, and `g`, a unary signal processor. Then, $\texttt{f} \sim \texttt{g}$ represents the 1-delay feedback loop through `g`.

$$[\![\texttt{f} \sim \texttt{g}]\!](i) \ = [\![\texttt{f}]\!]([\![\texttt{g}]\!]([\![\texttt{f} \sim \texttt{g}]\!]([\![\texttt{delay}]\!](i))), i).$$

Note that the definition of the semantics of feedback is recursive. For example, the FAUST program $+ \sim \texttt{sin}$ is depicted in Figure 1.

A Simple Low-Pass Filter For the rest of the paper, we will work with a simple low-pass filter written in FAUST as:

$$\texttt{smooth(c)} = *(1 - \texttt{c}) :+ \sim *(\texttt{c}).$$

`smooth` is intended to be used with a coefficient c in the interval $[0, 1]$. If c is 0, then the filter has no effect, whereas as we increase c the cutoff frequency decreases, with a limit case of $c = 1$, that outputs a constant signal. The filter first multiplies the input amplitude by $1 - c$, then to perform 1-sample additive feedback with coefficient c. Its block diagram with $c = 0.9$ is drawn in Figure 2.

While this filter may not be very adequate for audio, due to its frequency response curve, it is useful for instance for smoothing control parameters, and for other applications with high-frequency components. A key property of filters is *stability*. That is to say, we expect `smooth`'s output amplitude to remain in bounds that depend on the input. An excessive amount of feedback could cause the filter to behave badly.

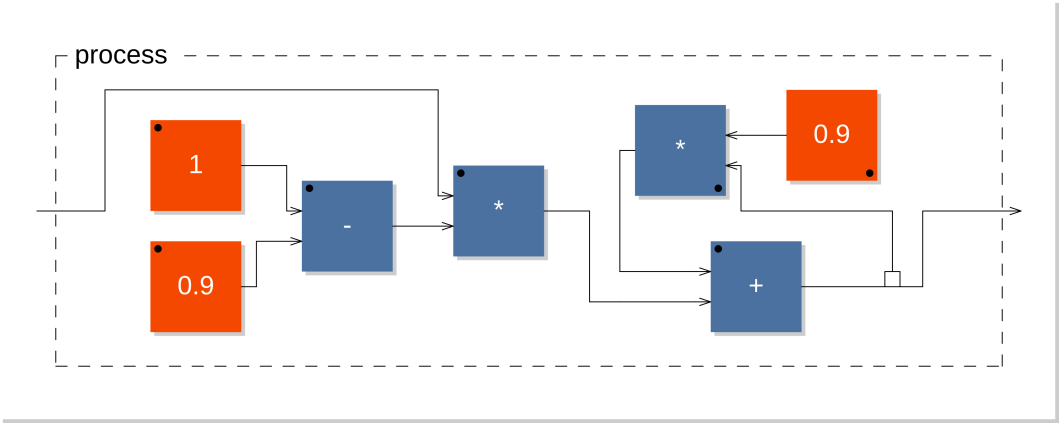

Figure 2: The `smooth` Low-Pass Filter

Stability also helps in compilation, as bounds known in the input signal can be propagated to the output, helping with buffer allocation and other issues.

4 Formalizing Faust in Coq

In the previous section, we described a core subset of Faust *on paper*. In this section, our goal will be to replicate this description inside a mechanized framework, Coq. This will enable us to later *mechanically reason* about Faust programs, avoiding a lot of potential error sources and getting strong confidence on the soundness of our reasoning.

Overview Mechanized reasoning about programs requires to encode their behavior or semantics in the particular theorem prover of choice. In a sense, this is equivalent to defining an *interpreter*; however the idiosyncrasy of theorem proving often makes the process quite different from writing a regular interpreter or compiler for our language.

Once the semantic representation is in place, we can wonder whether two programs have the same behavior (read: semantics), what happens when the input does not meet certain criteria, etc.

The concrete tasks we need to complete in order to start reasoning about Faust programs in Coq are: a) define a representation of Faust syntax in Coq; b) define a representation for signals, or streams, in Coq; c) write a function that takes a program's AST and returns a stream processor.

Once this is done, we can start proving! However, a key point in theorem proving is *how convenient* will it be to write proofs. We would have a hard time justifying formalized reasoning if we needed thousand of hours and lines of proof to perform trivial reasoning. We will address this point in Section 5, while devoting the rest of this one to explain how Faust is embedded into Coq. All the Coq code and examples can be downloaded from `https://github.com/ejgallego/mini-faust-coq/`.

Faust AST in Coq We will encode the *syntax* of the program using an *algebraic* or *inductive data type*. ADT (Abstract Data Types) are one of the most powerful features of Coq, allowing the user to define new richly-typed datatypes. In Section 2, we saw an example of an ADT for expressions. Here, we will make use of an extra feature known as *indexed* or dependently-typed ADT. Thus, we will encode Faust expressions using the `fterm` datatype, which carries additional information about the number of input and output signals of the program:

```
Inductive fterm : nat → nat → Type :=
| mult : num → 1 ⇝1
| plus : 2 ⇝ 1
| comp : 1 ⇝ 1 → 1 ⇝1 → 1 ⇝1
| feed : 2 ⇝ 1 → 1 ⇝1 → 1 ⇝1
where "i ⇝o" := (fterm i o).

Notation "'+" := plus.
Notation "'*(c)" := (mult c).
Notation "f : g" := (comp f g).
Notation "f ~ g" := (feed f g).
```

Thus, a program of type `i ⇝ o` will exactly go from i to o channels. Note that the constructors that correspond to the primitives enforce this requirement. In particular two signal processors can be composed only if they have the right types; thus no ill-typed Faust program can be built. We also define some notations to make display nicer.

Now, we can define our simple low-pass filter as follows:

```
Def smooth c : 1 ⇝1 := '*(1 - c) : '+ ~ '*(c).
```

Streams in Coq Once we have defined the syntax of our Faust programs, the next step is to define their semantics. We will encode signals as finite-length sequences of reals. We could have used several other representations, but it is beyond the scope of this paper to discuss the advantages of this particular definition.

We will index signals by their length, using the SSReflect type `n.-tuple R`, the type of sequences of exactly n reals. We write `'S_n` for `n.-tuple R` to shorten notation. Signal processors are encoded using regular Coq functions:

```
Notation "'S_n" := (n.-tuple R).
Notation "'SP(i,o)" := (∀ n, 'S_n^i → 'S_n^o).
```

For instance, the type for signals with three samples is `'S_3`. A signal processor (of type `"'SP(i,o)"`) must be able to process signals of arbitrary length; thus we quantify the second definition on all lengths n. We write `'S_n^k` for `k` copies of `'S_n`, e.g., `'S_n^2` is (`'S_n * 'S_n`).

Interpretation Function With both syntax and semantics in place, we can define a function linking the two worlds. In our case, a Faust expression of type `i ⇝ o` will be interpreted by a Coq function of type `∀ n, 'S_n^i → 'S_n^o`. Our interpreter I will thus have type:

```
I : i ⇝ o → 'SP(i, o)
```

Given a program `f`, the resulting function `I f` is *effective*, that is to say, given input signals, it computes the corresponding output ones. In particular, `I f n` formally corresponds to the semantic brackets introduced in Section 3, restricted to the first n samples of the signal. We write $[\![f]\!]_n$ for this truncated semantics.

How is I defined? Definition for primitives and composition is straightforward; the most interesting case is the feedback:

```
Definition I_feedback f g n i :=
  iter n (fun fb => f (g (x0 :: fb), i)) [::].
```

where `x0` is the initial value for the feedback loop, usually 0, and `iter n f x` is the function that applies `f n` times to `x`.

Note that this function outputs a signal of size n when the input is of size n, and does so by computing the feedback for n steps. The reader familiar with signal processing will notice that this implementation is extremely inefficient, as it may take quadratic time even for simple programs! Indeed, the goal of our interpretation is not to achieve efficiency, but to have a convenient representation of the semantics in order to reason about it. Usually, efficient implementations are not very well-suited for reasoning and vice-versa. The usual remedy when we care about efficiency inside Coq is to define two implementations, and prove them equivalent [Dénès et al., 2012].

First Steps in Proving With those ingredients, we can start reasoning about programs. For instance, a proof of the fusion property of the multiplication stream processors is:

```
(* Fusion of mult *)
Lemma multF : I ('*(a) : '*(b)) = I ('*(a*b)).
Proof.
  apply: val_inj; case: i => s /= _.
  by elim: s => //= x s → ; rewrite mulrCA mulrA.
Qed.
```

The proof is straightforward, by induction, associativity, and commutativity of the multiplication operator of the real numbers. However, some amount of boilerplate is necessary to set up the induction, task that can get tricky with more complex programs. Indeed, this inductive proof method is common to most proofs; thus we will identify common patterns and will define higher-level reasoning principles that allow us to prove things with less effort in the next section.

5 Structured Reasoning: A Sample Logic

As we just saw in Section 4, the Coq semantics allow us to state — and attempt to prove — almost any property imaginable about Faust programs. However, in most cases, reasoning can be repetitive, long and error-prone. That is the price we have to pay for accessing such a power.

A key observation is that proofs of certain classes of properties share common parts, while only a minor part of the proof actually depends on the property. As we saw in the previous fusion case, the relevant part of the proof is less than 10% of its total code.

Imagine a property φ, supposed to hold for all samples of a signal. Then, it is enough to define φ as a predicate over one sample, and we can "automatically" lift the predicate over signals, checking that φ holds for all time.

Indeed, to illustrate the principles of high-level structured reasoning over programs, we will focus on such "sample-level" properties in this section. While we will sacrifice quite a bit of expressivity, by limiting our language to one-sample statements, this will still be enough to carry out proofs of stability and will significantly

facilitate our proofs, allowing us to proceed in a short and structured way.

Sample-level Properties For our purposes, a predicate over a sample is a function from reals to booleans, $\varphi, \psi \in \mathbb{R} \to \mathbb{B}$, or, in Coq, `P, Q : R → bool`. Then[5], we say a property φ holds for a signal s if $\forall n.\varphi(s[n])$; that is, for all time moments n, the sample meets φ. In Coq, we can use the `all` function for sequences, thus writing `all P s`. For instance, the property that a signal is bound by the interval $[a, b]$ is defined as $\varphi(x) = x \in [a, b]$, or in Coq as `P := fun x => x \in `[a,b].

It makes sense to extend our properties to signal processors. In this case, we would like to relate properties over input signals with properties over the output signals. Given sample-level properties (φ, ψ) and input and output signals (i, o), a reasonable statement could be: "if the input signal i satisfies φ, then o should satisfy ψ."

If we think of our previous "being in an interval" property, its extension to signal processors allows us to capture stability. Indeed, we can precisely state now: "if the input signal is bounded, then the output signal will be too."

Judging the Sampling The previous relation between input and output signals and their properties constitutes an instance of a "high-level" reasoning principle. It is highly convenient thus to encode the fact that a signal processor satisfies the property as a "judgment." Our judgments will be of the form, $\{\varphi\}\ f\ \{\psi\}$, with intended interpretation $[\![\{\varphi\}\ f\ \{\psi\}]\!]$ such that, for all input signals with samples satisfying φ, all the output samples of f satisfy ψ. Formally:

$$[\![\{\varphi\}\ f\ \{\psi\}]\!] \iff \forall i.(\forall t.\varphi(i(t))) \implies (\forall t, \psi([\![f]\!])(i)(t))$$

the Coq version is expressed in a slightly different way, using `all`:

```
Definition '[[ { P } f { Q } ]] :=
    ∀ n (i : 'S_n), all i P ==> all (I f n) Q.
```

In the case of i input and o output signals, judgements are extended pointwise to use one predicate per signal: $[\![\{\varphi_1, \ldots, \varphi_i\}\ f\ \{\varphi_1, \ldots, \varphi_o\}]\!]$.

Reasoning Rules Now, we'd like to have a system of rules to determine when a judgment is valid without resorting to analyzing its semantics.

[5]From now on, we will interchangeably use Coq and mathematical notation where no confusion can arise, omitting double definitions.

The standard way to achieve this goal is to introduce a "logic", or a set of rules to infer validity of judgments, and, by extension, of their intended properties. The form of a rule is

$$\frac{A_1 \quad \ldots \quad A_n}{B}$$

meaning that, if $A_1, \ldots, A_n$ are valid, then B also is. This way, we can hopefully reduce validity checking for B to smaller problems.

The rules of our particular system for sample-level reasoning are shown in Figure 3. Rule *Prim* is an example of a base rule, stating that a judgment about a primitive is valid if its semantics is. Rule *Comp* allows to reduce the verification of composition to the verification of its individual parts; a judgment about composition is valid if there are valid judgments about the individual signal processors such that the property of the output of f implies the required property for the input stream of g.

The *Feed* rule is quite similar to the composition rule: the internal state of the feedback should obey an invariant θ, and samples from the feedback output should be compatible with the requirements of g's input. We also require that the initial value x_0 satisfies ψ.

Now, all that remains is to check that the rules are sound, that is to say that validity of the premises implies the validity of the conclusion, and we can reason using the newly defined logic:

Theorem 5.1 (Soundness)**.** *For any program f of type $i \rightsquigarrow o$, if $\{\varphi_1, \ldots, \varphi_i\}\ f\ \{\psi_i, \ldots, \psi_o\}$ is derivable, $[\![\{\varphi_1, \ldots, \varphi_i\}\ f\ \{\psi_i, \ldots, \psi_o\}]\!]$ is valid.*

Proof. We proceed by induction on the derivation. The base case is the *Prim* rule, and proof is immediate. For *Comp* soundness automatically follows by induction hypotheses. For *Feed*, we apply induction on the length of the input signal, plus induction hypotheses. □

6 Case Study: Filter Stability

As a case study, we will verify that the `smooth` filter of Section 3 is stable, that is, if the input amplitude is bounded, the output amplitude is.

Assume a well-formed interval $[a, b]$, including 0, and $c \in [0, 1]$. In Coq:

```
Hypothesis (Hab : a <= b).
Hypothesis (H0ab : 0 \in `[a, b]).
Hypothesis (Hrc : c \in `[0, 1]).
```

then, we will prove:

$$\{i \in [a, b]\}\ \texttt{smooth(c)}\ \{o \in [a, b]\}$$

$$\frac{\forall i_1, i_2, (\forall t. \varphi_1(i_1(t)) \wedge \varphi_1(i_2(t))) \implies (\forall t. \psi(i_1(t) + i_2(t)))}{\{\varphi_1, \varphi_2\} + \{\psi\}}\,Prim$$

$$\frac{\{\varphi\}\ f\ \{\theta\} \qquad \{\theta\}\ g\ \{\psi\}}{\{\varphi\}\ f : g\ \{\psi\}}\,Comp \qquad \frac{\models \psi(x_0) \qquad \{\theta, \varphi\}\ f\ \{\psi\} \qquad \{\psi\}\ g\ \{\theta\}}{\{\varphi\}\ f \sim g\ \{\psi\}}\,Feed$$

Figure 3: A simple logic for FAUST program verification

$$\frac{\square \qquad \dfrac{\square \qquad \square}{\{I_{abc}, I_{ab\bar{c}}\} + \{I_{ab}\} \qquad \{I_{ab}\} *(c)\ \{I_{abc}\}}}{\{I_{ab}\} *(1-c)\ \{I_{ab\bar{c}}\} \qquad \{I_{ab\bar{c}}\} + \sim *(c)\ \{I_{ab}\}}}{\{i \in [a,b]\} *(1-c) : + \sim *(c)\ \{o \in [a,b]\}}$$

with:

$$I_{ab}(x) \equiv x \in [a,b] \qquad I_{abc}(x) \equiv x \in [a*c, b*c] \qquad I_{ab\bar{c}}(x) \equiv x \in [a*(1-c), b*(1-c)]$$

Figure 4: Derivation for `smooth`

Let us recall the definition of smooth:

```
smooth(c) = *(1-c) : + ~*(c)
```

then, we should apply rule *Comp*, with $\theta(s) = s \in [a*(1-c), b*(1-c)]$. Using *Prim* gets us to a first obligation, shown in COQ as:

```
Hi : i \in '[ a, b]
============================
i * (1 - c) \in '[(a * (1 - c)), (b * (1 - c))]
```

which can be proved using the libraries by:

```
by rewrite ?itv_boundlr /= ?ler_wpmul2r
     ?ler_subr_addr ?add0r ?Hrc ?(itvP Hi).
```

The next step is to apply *Feed*, choosing $\theta(s) = s \in [a*c, b*c]$. Then, we apply *Prim* twice to get the obligations for + and *(c):

```
H1 : i1 \in '[( a * c), (b * c)]
H2 : i2 \in '[( a * (1 - c)), (b * (1 - c))]
============================
i1 + i2 \in '[a, b]
```

solved by:

```
have Ha: a = a * c + a * (1 - c)
    by rewrite -mulrDr addrC addrNK mulr1.
have Hb: b = b * c + b * (1 - c)
    by rewrite -mulrDr addrC addrNK mulr1.
by rewrite itv_boundlr /= Ha Hb
        !ler_add ?(itvP H1) ?(itvP H2).
```

where `have` introduces a local lemma, and

```
Hi : i \in '[ a, b]
============================
i * c \in '[( a * c), (b * c)]
```

solved by:

```
by rewrite itv_boundlr /=
        ?ler_wpmul2r ?(itvP Hi) ?Hrc.
```

We have chosen to prove the arithmetic obligations manually, but we should remark that there exists tools that can prove this kind of results automatically. The full derivation is in Figure 4.

7 Conclusion

We presented a case for the use of developer-assisted formal reasoning tools in the field of computer music and, more generally, audio DSP. We gave a quick tour of the COQ/SSREFLECT environment, which we believe can be particularly well fitted to reach such a vision. We applied our approach to the FAUST audio signal processing, providing a formal semantics for its core and detailing how a property of a filter can be proven using a specific logic designed for FAUST.

Future work will tackle other applications in the audio processing domain to assess our tool, together with the development of specific DSP mechanisms within COQ/SSREFLECT (tactics, tacticals, or even a dedicated DSP library). We would also be interested in seeing how our system can be of help to prove more foundational theorems such as the Shannon Sampling Theorem.

8 Acknowledgements

We want to thank Yann Orlarey and Arnaud Spiwack for their insightful comments. Partial funding for this research was provided by the ANR FEEVER Project.

References

Karim Barkati, Haisheng Wang, and Pierre Jouvelot. 2014. Faustine: A Vector Faust Interpreter Test Bed for Multimedia Signal Processing - System Description. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, volume 8475 of *Lecture Notes in Computer Science*, pages 69–85. Springer.

Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer.

Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. 2014. A core quantitative coeffect calculus. In Zhong Shao, editor, *Programming Languages and Systems*, volume 8410 of *Lecture Notes in Computer Science*, pages 351–370. Springer Berlin Heidelberg.

Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. 2012. Continuity and Robustness of Programs. *Commun. ACM*, 55(8):107–115, August.

Maxime Dénès, Anders Mörtberg, and Vincent Siles. 2012. A refinement-based approach to computational algebra in COQ. In Lennart Beringer and Amy Felty, editors, *ITP - 3rd International Conference on Interactive Theorem Proving - 2012*, volume 7406 of *Lecture Notes In Computer Science*, pages 83–98, Princeton, USA. Springer, Springer.

Sarah Denoux, Stéphane Letz, Yann Orlarey, and Dominique Fober. 2014. FAUSTLIVE, Just-In-Time Faust Compiler... and much more. In *Linux Audio Conference*.

Naghmeh Ghafari, Ramana Kumar, Jeff Joyce, Bernd Dehning, and Christos Zamantzas. 2011. Formal Verification of Real-time Data Processing of the LHC Beam Loss Monitoring System: A Case Study. In *Proceedings of the 16th International Conference on Formal Methods for Industrial Critical Systems*, FMICS'11, pages 212–227, Berlin, Heidelberg. Springer-Verlag.

Georges Gonthier, Assia Mahboubi, and Enrico Tassi. 2008. A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, INRIA.

Pierre Jouvelot and Yann Orlarey. 2011. Dependent vector types for data structuring in multirate Faust. *Computer Languages, Systems & Structures*, 37(3):113–131.

Neelakantan R. Krishnaswami. 2013. Higher-order functional reactive programming without spacetime leaks. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 221–232. ACM.

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115.

Yann Orlarey, Dominique Fober, and Stephane Letz. 2004. Syntactical and semantical aspects of Faust. *Soft Comput.*, 8(9):623–632.

Anis Souari, Amjad Gawanmeh, Sofiène Tahar, and Mohamed Lassaad Ammari. 2014. Design and verification of a frequency domain equalizer. *Microelectronics Journal*, 45(2):167–178.