Verification of Faust Signal Processing Programs in COQ

Emilio Jesús Gallego Arias Olivier Hermant Pierre Jouvelot
MINES ParisTech, PSL Research University

Abstract

We report on our ongoing work to formalize and prove properties of FAUST programs using COQ.

FAUST (Functional Audio Stream) is a functional programming language specifically designed for real-time digital signal processing (DSP) and synthesis. This Domain-Specific Language targets high-performance audio DSP applications and plug-ins for a variety of platforms and standards.

Faust programs are highly declarative and provide a reasonable set of static guarantees, but far from full correctness. In FAUST's domain, when errors occur, one will typically experience problems by hearing audio glitches or, even worse, by suffering imperceptible distortion, which can accumulate and become audible when more components are connected. To detect such cases *a priori*, manual reasoning is far from trivial: arithmetic errors, feedback loops, and the large size of the DSP circuits involved make it tedious and error-prone. Thus, automated verification is highly desirable in the growing ecosystem of FAUST users and libraries.

We intend to use COQ as the basis for a specification and automated verification platform for FAUST programs. We plan to use the resulting tool to certify library components, as well as to experiment with new language features or to enhance the FaustWorks IDE with proof automation features. Our choice of COQ instead of a custom-purpose tool is both pragmatic and philosophical: we believe that we can greatly profit from the existing tools and libraries, and that others may do so too from our effort.

The anticipated two main challenges: useful and modular automation, and integration of a quite diverse set of existing libraries.

Keywords DSP; audio; program verification; theorem proving

1. FAUST

FAUST (Functional Audio Stream) [11] is a functional programming language specifically designed for real-time audio signal processing and synthesis. A quick summary of FAUST main characteristics follows.

- FAUST is a specification language aiming to provide an adequate notation to describe signal processors from a mathematical point of view.
- A FAUST program describes a signal processor, transforming a group of (possibly empty) input signals to a group of (possibly empty) output signals. Most audio equipments can be modeled as signal processors.
- It works at the signal sample level. It is therefore suited to implement low-level DSP functions like recursive filters.
- FAUST programs are compiled to C++ programs with an special emphasis on performance; thanks to the notion of architecture, FAUST programs can be easily deployed on a large variety of audio platforms and plugin formats.
- FAUST combines two approaches: functional programming and algebraic block-diagrams, viewing block-diagram construction as function composition. FAUST defines a block-diagram algebra of five composition operations ($:, \sim <::>$).

FAUST has been used with success in the domain of specification of highly-performant audio processors. However, it is not uncommon that programs suffer from artifacts derived from the distance between the mathematical semantics and the actual implementation of the processors.

Given the domains involved, informal reasoning by the user is often difficult. There is very little support for formal reasoning over programs. Proposed extensions to FAUST such as multi-rate support [12] will make informal reasoning even more difficult for the programmer.

On the other hand, verification of interesting properties like robustness [5] or BIBO (Bounded-Input Bounded-Output) stability will allow the compiler to generate better code by taking the appropriate assumptions while providing users with stronger claims regarding program correctness.

2. Verification of FAUST programs

We have chosen to build our automatic verification efforts on top of COQ. Recent work [18, 6] has further pushed the barrier of automatic verification inside it. The tradeoffs of using COQ vs developing a custom-purpose tool are much in favor of the former. In our view, some of the strong points of COQ are:

- ease of play with different approaches;
- interest in doing a verified compiler;
- strong support for automation;
- ability to catch unsoundness/mistakes in techniques/tools;
- growing user community.

2.1 Goals

Our main goal is to build a FAUST-specific COQ environment allowing the user to prove particular programs correct with a high-degree of automation.

Examples of some properties we are interested in include:

- bounds in space, buffer, error, execution time...;
- normalization, i.e. absence of distortion in the output;
- BIBO stability;
- relational properties, relating two execution of the processors;
- temporal properties;
- equivalence properties, like memoization.

We are working on defining more properties in collaboration with FAUST users — musicians and sound designers.

Ultimately, we aim for our tool to be made accessible to regular users, to the point they can use it to express and prove application-specific properties in an automatic manner without (a lot) of our help.

Automation is of key importance when targeting a DSL such as FAUST, since its users will have little background in theorem proving. Obviously, we won't be able to achieve full automation, but we see our efforts as a way to introduce the ideas of formal verification into that particular programming community. Even if users are just able to formally specify some properties — without doing any proof —, that will be a huge gain from the current status quo, where correctness of FAUST programs is determined by heuristic methods [19, Sec. 4].

Particular emphasis will be placed in the usability of the library, with techniques like [1, 9] serving as inspiration, trying to produce small, reusable components, which to the best of our knowledge is still quite hard to do.

As a side effect, we would like to reuse our infrastructure for the development of a certified compiler and type-checker. Technologies like WebAudio [16] suppose a paradigm shift for the FAUST domain, where efficient execution of arbitrary programs is shifted to the browser. Even a partly-certified compiler would greatly help in that scenario, and it would be easy for us to profit from extraction and `js_of_ocaml` [20] to deliver the compiler to the browser.

2.2 Design Philosophy and Challenges

The main challenges for FAUST program verification comes from feedback loops — pervasive in synchronous systems —, static interval reasoning, multiple data rates [4], and machine-level integers and floats. We plan to leverage as much as possible existing tools and techniques inside our framework; a few examples are listed below.

Reflection: Feedback-free circuits have a good set of decidable properties [17]. We thus will make extensive use of reflection [10] and decision procedures.

Outside COQ: For complex properties, we may follow [13, 7] and use some external decision procedures, verifying in COQ that their output is correct. Interval analysis, abstract interpretation, invariant generation and floating-point reasoning are likely candidates.

External tools and libraries: Some interesting external libraries for us — apart from SSREFLECT— are [3], for real analysis, Gappa [8] and Flocq [2], for floating-point arithmetic, and YNot [15].

Tactics: Our plan is to make use of tactics as a connecting tool between the several internal and external components, not as a basic proving tool. While recent advances like MTac [21] or MirrorShard [14] promise better composition of tactics, we still fear maintenance problems due the experimental nature of our language.

We believe our development may serve as a good stress test for how well several different components can interact, and provide some insight on the use of COQ as an automated verification tool.

How successful the automatic approach will be in this particular setting and how much we can reuse from other efforts are open questions. Our methodological results will hopefully help pave the way for introducing more proof-assistant-based tools within existing DSL environments or, conversely, help make future DSLs more amenable to proof handling.

3. Current Status and Goals for the Workshop

We have a prototype specification of FAUST semantics in COQ/-SSREFLECT, as well as some proof of concepts of automation. Current work is focused on defining and proving properties.

For the workshop timeline, we expect to showcase a quite complete tool, to report on our experience working with COQ and associated libraries, and to assess how many bugs we found in FAUST programs.

References

[1] Y. Bertot et al. “Canonical Big Operators”. In: *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*. Vol. 5170. Springer, 2008, pp. 86–101.

[2] S. Boldo and G. Melquiond. “Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq”. In: *Computer Arithmetic (ARITH), 2011 20th IEEE Symposium on*. July 2011, pp. 243–252.

[3] S. Boldo, C. Lelay, and G. Melquiond. “Coquelicot: A User-Friendly Library of Real Analysis for Coq”. English. In: *Mathematics in Computer Science* (2014), pp. 1–22.

[4] S. Boulmé and G. Hamon. “Certifying Synchrony for Free”. In: *Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conference, LPAR 2001, Havana, Cuba, December 3-7, 2001, Proceedings*. Vol. 2250. Springer, 2001, pp. 495–506.

[5] S. Chaudhuri, S. Gulwani, and R. Lublinerman. “Continuity and robustness of programs”. In: *Commun. ACM* 55.8 (2012), pp. 107–115.

[6] A. Chlipala. “From Network Interface to Multithreaded Web Applications: A Case Study in Modular Program Verification”. In: *POPL 2015*. 2015.

[7] G. Claret et al. “Lightweight Proof by Reflection Using a Posteriori Simulation of Effectful Computation”. English. In: *Interactive Theorem Proving*. Vol. 7998. Springer Berlin Heidelberg, 2013, pp. 67–83.

[8] F. de Dinechin, C. Q. Lauter, and G. Melquiond. “Certifying the Floating-Point Implementation of an Elementary Function Using Gappa”. In: *IEEE Trans. Computers* 60.2 (2011), pp. 242–253.

[9] F. Garillot et al. “Packaging Mathematical Structures”. In: *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*. Vol. 5674. Springer, 2009, pp. 327–342.

[10] G. Gonthier, A. Mahboubi, and E. Tassi. *A Small Scale Reflection Extension for the Coq system*. Research Report RR-6455. 2008.

[11] GRAME. *The Faust Project*. 2014. URL: `http://faust.grame.fr/` (visited on 10/12/2014).

[12] P. Jouvelot and Y. Orlarey. “Dependent vector types for data structuring in multirate Faust”. In: *Computer Languages, Systems & Structures* 37.3 (2011), pp. 113–131.

[13] X. Leroy. “Formal verification of a realistic compiler”. In: *Commun. ACM* 52.7 (2009), pp. 107–115.

[14] G. Malecha, A. Chlipala, and T. Braibant. “Compositional Computational Reflection”. In: *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*. Vol. 8558. Springer, 2014, pp. 374–389.

[15] A. Nanevski et al. “Ynot: dependent types for imperative programs”. In: *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*. ACM, 2008, pp. 229–240.

[16] C. R. Paul Adenot Chris Wilson. *Web Audio API*. 2014. URL: `http://webaudio.github.io/web-audio-api/` (visited on 10/12/2014).

[17] C. Paulin-Mohring. "Circuits as Streams in Coq: Verification of a Sequential Multiplier". In: *Types for Proofs and Programs, International Workshop TYPES'95, Torino, Italy, June 5-8, 1995, Selected Papers*. Vol. 1158. Springer, 1995, pp. 216–230.

[18] D. Ricketts et al. "Automating formal proofs for reactive systems". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. ACM, 2014, p. 47.

[19] J. O. Smith III. *Audio signal processing in Faust*. Tech. rep. Stanford University, 2010.

[20] J. Vouillon and V. Balat. "From bytecode to JavaScript: the Js_of_ocaml compiler". In: *Softw., Pract. Exper.* 44.8 (2014), pp. 951–972.

[21] B. Ziliani et al. "Mtac: a monad for typed tactic programming in Coq". In: *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. ACM, 2013, pp. 87–100.