Proceedings of the 1st International Faust Conference (IFC-18), Mainz, Germany, July 17–18, 2018

OUTSIDE THE BLOCK SYNDICATE: TRANSLATING FAUST'S ALGEBRA OF BLOCKS TO THE ARROWS FRAMEWORK

Benedict R. Gaster
benedict.gaster@uwe.ac.uk

Nathan Renney
nathen.renney@uwe.ac.uk

Tom Mitchell
tom.mitchell@uwe.ac.uk

University of West of England (UWE)
Bristol, UK

ABSTRACT

Folklore has it that Faust's algebra of blocks can be represented in Hughes' algebra of Arrows. In this paper we formalise this understanding, showing that blocks can indeed be encoded with Causal Commutative Arrows.

Whilst an interesting finding in itself, we believe that this formal translation opens up new avenues of research. For instance, recent work in functional reactive programming on well typed clocks, could provide an alternative to the dependent type approach proposed for multi-rate Faust.

1. INTRODUCTION

Faust is a domain specific programming language for Digital Signal Processing (DSP) based on the Algebra of Blocks [1, 2]. Hughes' Arrows framework provides a generalization of function arrows to computations [3]. It too can be used as a language for DSP computations. What, if any, is the relationship between them? This is the subject of the paper you are reading.

Arrows generalize monads, retaining composition, while relaxing the stringent linearity they impose. Arrows have been used to express a wide variety of applications, often in the context of Embedded Domain Specific Languages (EDSL) [4] within the host Language Haskell [5]. A key application domain for arrows is Functional Reactive Programming (FRP). Originally proposed by Elliott and Hudak [6] in the context of functional animation, FRP is a programming paradigm for reactive programming, based on continuous signals. Signals (originally called behaviours by Elliot) are defined as:

$$Signal \approx Time \to A$$

where A is the type of values carried by the signal. Taking

$$A = \mathbb{R}$$

we get Faust's signal function [2]. Unfortunately, unrestricted access to signals makes it too easy to generate both time and space leaks and to address this Hudak et al proposed using arrows to structure FRP and in doing so introduced the world to Yampa, an arrow framework for FRP [7].

Hughes' arrows framework is not limited to DSP, which is at the same time both an advantage and disadvantage. On the plus side the use of arrows eliminates a subtle but paralizing form of a *space leak*, as described by Lui et al [8], that is found in earlier encodings of Functional Reactive Programming [6]. Additionally, arrows introduce a meta level of computation that aids in reasoning about program correctness, transformation, and optimization. Below, when we consider causality and Faust's 1-sample delay, we will see some of arrows' disadvantages.

Arrows, analogous to other forms of computation, such as monads [9], define a class of computations that are constructed to conform to a given structure:

$$\begin{array}{lcl} arr & : & (A \to B) \to A \rightsquigarrow B \\ (>\!\!>\!\!>) & : & (A \rightsquigarrow B) \to (B \rightsquigarrow C) \to (A \rightsquigarrow C) \\ first & : & (A \rightsquigarrow B) \to (A \times C \rightsquigarrow B \times C) \end{array}$$

We write $A \rightsquigarrow B$ for an arrow that consumes input of type A and produces values of type B. The combinator arr lifts a function from A to B to an arrow from A to B. The operator $(>\!\!>\!\!>)$ is sequential composition of two arrows, analogous to Faust's : operator, while $first$ is used in the context of parallel composition, lifting an arrow from A to B to an arrow that uses this arrow to transform its first argument, while leaving its second untouched.

The observant reader might be asking why does the above definition not include parallel composition? One of the key insights of Hughes', when defining arrows, was the observation that the above combinators were enough to define all "interesting" combinators on arrows. For example, parallel composition is defined in terms of *first*, *second*, and $>\!\!>\!\!>$[1]:

$$\begin{array}{lcl} (***) & : & (A \rightsquigarrow B) \to (C \rightsquigarrow D) \to (A \times C \rightsquigarrow B \times D) \\ f *** g & = & first\ f >\!\!>\!\!>\ second\ g \end{array}$$

Arrows, as specified above and in conjunction with a set of laws (given later), define an abstract algebra for a set of structured computations; to specialize to the domain of DSP, arrows become computations on signals[2]:

$$\mathsf{DSP} \approx \mathsf{Signal}\ A \rightsquigarrow \mathsf{Signal}\ B$$

When a value of type DSP is applied to an input signal of type $\mathsf{Signal}\ A$, it produces an output signal of type $\mathsf{Signal}\ B$. Given this definition we can define a stream of silence as:

$$silence = arr\ (\lambda().\ 0.0)$$

and is of type:

$$\mathsf{Signal}\ () \rightsquigarrow \mathsf{Signal}\ float$$

In Faust we might simply write:

$$silence = 0.0$$

[1] The definition of $second$ is omitted here as it is easily defined in terms of $first$ and is instead given in Figure 3.

[2] This is Yampa's stream function, which is in turn an instance of arrow for function arrows of type $\mathsf{Signal} \to \mathsf{Signal}$ [7].

Another example is the addition stream in Faust:

$$process = +$$

defined in arrow DSP as:

$$process = arr\ (\lambda(x, y).\ x + y)$$

which simply lifts the curried version of the addition operator to a DSP arrow.

In Faust, say we want to connect the output of the $+$ operator to the input of abs, to compute the absolute value of the output signal, then using sequential composition:

$$process = + : abs$$

and in arrow DSP:

$$process = arr\ (\lambda(x, y).\ x + y)\ >>>\ arr\ abs$$

The translation of Faust's parallel composition operator is slightly more involved, but still straightforward. Consider the following (stereo cable) example:

$$process = \_, \_$$

Faust's identity, _, is represented as an arrow simply by lifting the identity of the λ-calculus, $arr\ (\lambda x.\ x)$, and given this we define the above as:

$$process = arr\ (\lambda x.\ x)\ ***\ arr\ (\lambda x.\ x)$$

What about Faust's delay (mem), how is this encoded in Hughes' arrows? As defined it is not possible! This is a limitation of arrows; as originally defined they are not strong enough to capture all computations expressed by Faust—more laws are needed to constrain the computation space and the introduction of a side-effecting arrow is necessary. In particular, Hughes' original definition of arrows lacks laws for causality, future values must depend only on values of the past. Functional Reactive frameworks, such as Yampa [7], often specify that an arrow instance must have causality, but leave rules for enforcement implicit. More constrained forms of computation, e.g. monads [9] and applicative functors [10], are not general enough, and lack certain operators, e.g. delay and associated laws, needed by Faust. Instead what is needed is a specific form of arrow computations, called *causal commutative arrows* (CCA) [11].

CCA extends arrows with a commutative law for parallel composition and with a general delay operator, i.e. one that introduces a necessary side effect. In the context of DSP arrows this operator provides a one sample delay, but is more general in other cases:

$$delay : A \to (A \rightsquigarrow A)$$

which in the case of DSP arrows the application $delay\ M$ is an arrow that produces M for any $t \in Time \wedge t \leq 1$, otherwise at time $t_{(i+1)}$ it generates the value passed at time t_i. With this, given a definition of an impulse in Faust:

$$impulse(x) = x - mem(x)$$

a translation into CCA is[3]:

$$impulse(x) = x - (x\ >>>\ delay\ 0)$$

[3]To aid reading we have used a slight abuse of notation to use a lifted version of the operator $-$.

The final Faust construct we consider is the recursion operator, $\sim$, that enables cycles and includes an implicit one-sample delay. To demonstrate the translation we use an example integrator that takes an input signal M and computes an output signal N such that $N(t) = M(t) + N(t-1)$:

$$process = + \sim \_$$

Hughes' original definition of arrows did not include a recursion operator, but later Patterson added a loop construct [12]:

$$\mathsf{loop} : (A \times C \rightsquigarrow B \times C) \to A \rightsquigarrow B$$

loop takes an arrow expecting an input argument (type A), and a feedback argument (type C), producing an output value (type B) and a value to feedback into the next iteration. The result of loop is an arrow from A to B, i.e. the feedback is constrained internally. loop is a variant of Curry's fixpoint combinator [13]. Unlike Faust's recursion operator loop′s feedback is not delayed and thus an explicit one sample delay must be inserted, thus, while Patterson did not require the additional constraints that CCA imposes on arrows, our translation must.

Given this discussion we can now give a translation of the Faust iterator given above:

$$\mathsf{loop}\ (arr\ swap\ >>>\ arr\ (\lambda(x, y).\ x + y)\ >>>\ arr\ (\lambda x.\ x)\ \&\&\&\ (delay\ 0.0\ >>>\ arr\ (\lambda x.\ x)))$$

The function $swap$ simply swaps the values of a tuple, while the arrow operator &&& is a specialized version of parallel composition that composes two arrows with the same input types concurrently, fanning out a single input to each arrow. The type of the above translation is analogous to the Faust recursion example, given above, but in the context of DSP arrows:

$$float \rightsquigarrow float$$

i.e, an arrow (stream transformer) that consumes values of type *float* and produces values of type *float*.

We have omitted discussion of the fan out (<:) and fan in (:>) operators of Faust here as they are easily handled in our translation. Worry not, these are not forgotten and will be presented in Section 3 along with the complete translation.

Clearly the syntactic noise of arrows is considerable, at least in comparison to Faust. However, we believe this is a function of Hughes choice of defining them within the context of Haskell, rather than some inherent limitation of arrows themselves. In fact it was in part our background in both functional programming and audio processing that lead us to develop the translation proposed in this paper. For the most part, we work with Hughes' original notation as it does not affect the translation. However, in later sections, we return to the question of both an alternative syntax for arrows, inspired by Faust, and an alternative system for type checking arrow DSP.

It is folklore that Faust's semantic model is very close to Hughes' arrows, indeed Faust's Wikipedia page [14] includes an informal translation, but to our knowledge there does not exist a formal analysis of this relationship. In this paper we provide a formal translation from Faust's Algebra of Blocks into a DSP instance of arrows.

It is important to note that while Hughes' original definition of arrows and much of the later work has focused on an embedding in Haskell, there is no fundemential connection between arrows and

$$Syntax$$

$$\begin{array}{lllll} Types & A, B, C & := & \mathit{float} \mid A \times B \mid () \mid A \to B \\ Terms & L, M, N & := & X \mid x \mid (M, N) \mid () \mid \mathit{fst}\ L \mid \mathit{snd}\ L \mid \lambda x.\ N \mid L\ M \mid \\ Environment`s & \Gamma, \Delta & := & x_1 : A_1, ..., x_n : A_n \end{array}$$

$$Types$$

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma, x : A \vdash N : B}{\Gamma \vdash \lambda x.\ N : A \to B} \qquad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}$$

$$\frac{}{\Gamma \vdash () : ()} \qquad \frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash (M, N) : A \times B} \qquad \frac{\Gamma \vdash L : A \times B}{\Gamma \vdash \mathit{fst}\ L : A} \qquad \frac{\Gamma \vdash L : A \times B}{\Gamma \vdash \mathit{snd}\ L : B}$$

$$Laws$$

$$\begin{array}{lrcl} (\beta_1^x) & \mathit{fst}\ (M, N) & = & M \\ (\beta_2^x) & \mathit{snd}\ (M, N) & = & N \\ (\eta^x) & (\mathit{fst}\ L, \mathit{snd}\ L) & = & L \\ (\beta^{\to}) & (\lambda x.\ N)\ M & = & N[M/x] \\ (\eta^{\to}) & \lambda x.\ (L\ x) & = & L \end{array}$$

Figure 1: Lambda Calculus

Haskell. In particular, there exists a categorical model, in terms of monoidal categories, for arrows [15, 16] and it this definition that drives the relation presented in this paper. It is not our intention to imply that Faust could or should be translated into a Haskell implementation of arrows. Rather, arrows provide an alternative formal model for Faust, which might provide new an interesting directions for future work.

The remainder of this paper is structured as follows:

- Section 2 captures the formal semantics of Faust's algebra of blocks and Lui et al's causal commutative arrows;
- Section 3 provides details of the translation from the algebra of blocks to the stream transformer instance of causal commutative arrows;
- Section 4 takes a more informal look at recent developments in the application of monads within the context of state transformers, the IO monad, and its potential use for providing stronger types for Faust's foreign function interface; and
- Section 5 concludes with pointers to possible fruitful directions for the development of DSP languages based on Faust and causal commutative arrows.

2. FORMALIZATION

Our definition of arrows, and in fact the corresponding definition of the algebra of blocks, extends the core lambda calculus. We first give a standard definition of lambda calculus, including typing rules and the conventional rewrite laws, in Figure 1. We write X for the set of constant terms, e.g. $+ : \mathit{float} \to \mathit{float} \to \mathit{float}$, $10 : \mathit{float}$ and so on, and no additional typing rules are necessary.

The extended arrow calculus[4] is given in Figure 2. Our definition of arrows is extended with recursion (the loop combinator) and state (the $delay$ operator), with rules derived from Liu's causal commutative arrows [18]. Figure 3 specifies a number of useful definitions for both the lambda calculus and its extended arrow variant.

As described in the introduction, the type $A \rightsquigarrow B$ denotes a computation that accepts a value of type A and returns a value of type B, and in the presence of $delay$ can perform side effects. As is conventional we write a term typing judgment as $\Gamma \vdash M : A$ to mean that the term M has type A in environment Γ. A judgment $\Gamma \vdash M : A$ is valid under the rules defined in Figure 1 and for the arrow combinators, given in Figure 2, types are added as constants (i.e. $c \in X$).

While no additional typing rules are required for a type to be a valid instance of an arrow it must define implementations for each of these constants. It is not enough for an arrow definition to simply implement these constants, they must also satisfy the laws given at the bottom of Figure 2.

Figure 4 defines the algebra of blocks. We adopt a similar notation to Orlarey et al [2], in particular $A : I_M \to O_N$ to represent that a block-diagram A has I_M inputs and O_N outputs, defined:

$$\begin{array}{lcl} inputs(A) & = & A_{in}[0] \times A_{in}[1] \times ... \times A_{in}[I_{M-1}] \\ outputs(D) & = & A_{out}[0] \times A_{out}[1] \times ... \times A_{out}[I_{N-1}] \end{array}$$

Additionally to aid the translation, described in Section 3, we assume the following short hand for working with tuples:

$$\begin{array}{lcl} A_0 \times A_1 \times ... \times A_{n-1} & = & A_0 \times (A_1 ... (\times A_{n-1})) \\ (x_0, x_1 ..., x_{n-1}) & = & (x_0, (x_1, (..., x_{n-1}))) \end{array}$$

and fst_i and snd_i are defined pointwise in terms of fst and snd. It is straightforward to define operations to add $(+)$ and subtract $(-)$ both input and outputs and tuples values and types, for brevity we omit their definitions.

Unlike the arrows definition given in Figure 2 typing rules, following Orlarey et al [2], the meanings for the subset of lambda

[4] In the most part we retain Hughes' original definition of arrows and while we use the term arrow calculus we do not use the definitions and notation of Lindley [17], where the term originated.

$$Syntax$$

$$\begin{array}{llll} Types & A, B, C & := & \ldots \mid A \rightsquigarrow B \\ Terms & L, M, N & := & \ldots \mid delay \mid arr \mid (>>>) \mid *** \mid first \mid second \mid \mathsf{loop} \end{array}$$

$$Types$$

$$\begin{array}{l} arr : (A \to B) \to (A \rightsquigarrow B) \\ (>>>) : (A \rightsquigarrow B) \to (B \rightsquigarrow C) \to (A \rightsquigarrow C) \\ (***) : (A \rightsquigarrow B) \to (C \rightsquigarrow D) \to (A \times C \rightsquigarrow B \times D) \\ first : (A \rightsquigarrow B) \to (A \times C \rightsquigarrow B \times C) \\ loop : (A \times C \rightsquigarrow B \times C) \to A \rightsquigarrow B \\ delay : A \to A \rightsquigarrow A \end{array}$$

$$Laws$$

$$\begin{array}{lrcl} (\rightsquigarrow_1) & arr\ id\ >>>\ f & = & f \\ (\rightsquigarrow_2) & f\ >>>\ arr\ id & = & f \\ (\rightsquigarrow_3) & (f\ >>>\ g)\ >>>\ h & = & f\ >>>\ (g\ >>>\ h) \\ (\rightsquigarrow_4) & arr\ (g \cdot f) & = & arr\ f\ >>>\ arr\ g \\ (\rightsquigarrow_5) & first\ (arr\ f) & = & arr\ (f \times id) \\ (\rightsquigarrow_6) & first\ (f\ >>>\ g) & = & first\ f\ >>>\ first\ g \\ (\rightsquigarrow_7) & first\ f\ >>>\ arr\ (id \times g) & = & arr\ (id \times g)\ >>>\ first\ f \\ (\rightsquigarrow_8) & first\ f\ >>>\ arr\ fst & = & arr\ fst\ >>>\ f \\ (\rightsquigarrow_9) & first\ (first\ f)\ >>>\ arr\ assoc & = & arr\ assoc\ >>>\ \mathsf{first}\ f \\ (\rightsquigarrow_{10}) & loop\ (first\ h\ >>>\ f) & = & h\ >>>\ loop\ f \\ (\rightsquigarrow_{11}) & loop\ (f\ >>>\ first\ h) & = & loop\ f\ >>>\ h \\ (\rightsquigarrow_{12}) & loop\ (f\ >>>\ arr\ (id \times k)) & = & loop\ (arr\ (id \times k)\ >>>\ f) \\ (\rightsquigarrow_{13}) & loop\ (loop\ f) & = & loop\ (arr\ assoc^{-1}\ >>>\ f\ >>>\ arr\ assoc) \\ (\rightsquigarrow_{14}) & second\ (loop\ f) & = & loop\ (arr\ assoc\ >>>\ second\ f\ >>>\ arr\ assoc^{-1}) \\ (\rightsquigarrow_{15}) & loop\ (arr\ f) & = & arr\ (trace\ f) \\ (\rightsquigarrow_{16}) & first\ f\ >>>\ second\ g & = & second\ g\ >>>\ first\ f \\ (\rightsquigarrow_{17}) & delay\ i\ ***\ delay\ j & = & delay\ (i, j) \end{array}$$

Figure 2: Arrows

calculus terms remain the same, as defined in Figure 1. The intention here is the arrow calculus forms one extended variant of the lambda calculus, as does the blocks algebra, but at this point it is only the subset of lambda calculus terms that is valid in both. The translation from the algebra of blocks to the stream transformer instance of arrows, given in the following section enables one to consider any term of the algebra of blocks to be valid in the arrows variant of the lambda calculus, when arrows are considered as DSP signal transformers.

3. TRANSLATIONS

We now consider the translation between the algebra of blocks into arrows at the instance of stream transformers, and show that the translation is sound.

The translation takes an algebra of blocks term M into an arrow $[\![M]\!]_{st}$:

$$[\![\Gamma \vdash M : A]\!]_{st} = [\![\Gamma]\!]_{st} \vdash [\![M]\!]_{st} : [\![A]\!]_{st}$$

The translation for environments (Γ) is defined pointwise and is straightforward. The translation for terms and judgments is given in Figure 7, while the translation for types is in Figure 6. This translation is similar to the one given, informally, on Faust's Wikipedia page[14], although it includes a complete translation, along with a modified, type correct(ed), version of the mapping for recursion, i.e. $\sim$.

The only complications in the translation arise from Faust's rules for fanning out and in. For fanning out we must duplicate the signal and in the case of fanning in, merge the signals using addition. As the standard definition of arrows relies only on the basic types of the simply typed λ-calculus, augmented with tuple types, it does not directly support composing arrows with differing numbers of input and output arguments. However, this is easily "emulated" with parallel composition ($***$), for example, consider the Faust expression:

$$\_ <: \_,_b\ f$$

A mono signal is duplicated and the resulting left and right channels are copies of the original carrier, the function f applied to the right channel, while the left channel is passed through. Applying the translation in Figure 7:

$$id_a\ >>>\ dup_2\ >>>\ id_a\ ***\ f_a$$

where $id_a = arr\ id$ and f_a is the arrow translation of f. The arrow dup_2 takes a single argument of type A and produces a tuple of type $A \times A$, duplicating the argument 2 times, implemented as:

$$dup_2 = arr\ (\lambda x.\ (x, x))$$

$$
\begin{array}{lcl}
id & : & A \to A \\
id & = & \lambda x.\ x \\
fst & : & A \times B \to A \\
fst & = & \lambda z.\ fst\ z \\
snd & : & A \times B \to B \\
snd & = & \lambda z.\ snd\ z \\
dup & : & A \to A \times A \\
dup & = & \lambda x.\ (x, x) \\
swap & : & A \times B \to B \times A \\
swap & = & \lambda z.\ (snd\ z, fst\ z) \\
(\times) & : & (A \to C) \to (B \to D) \to (A \times B \to C \times D) \\
(\times) & = & \lambda f.\ \lambda g.\ \lambda z.\ (f\ (fst\ z), g\ (snd\ z)) \\
assoc & : & (A \times B) \times C \to A \times (B \times C) \\
assoc & = & = \lambda z.\ (fst\ (fst\ z), (snd\ (fst\ z), snd\ z)) \\
(\cdot) & : & (B \to C) \to (A \to B) \to (A \to C) \\
(\cdot) & = & \lambda f.\ \lambda g.\ \lambda x.\ f\ (g\ x) \\
second & : & A \rightsquigarrow B \to (C \times A \rightsquigarrow C \times B) \\
second & = & \lambda f.\ arr\ swap >\!\!>\!\!>\ first\ f >\!\!>\!\!>\ arr\ swap \\
(***) & : & (A \rightsquigarrow B) \to (C \rightsquigarrow D) \to (A \times C \rightsquigarrow B \times D) \\
(***) & = & \lambda f.\ \lambda g.\ first\ f >\!\!>\!\!>\ second\ g \\
(\&\&\&) & : & (A \rightsquigarrow B) \to (A \rightsquigarrow C) \to (A \rightsquigarrow B \times C) \\
(\&\&\&) & = & \lambda f.\ \lambda g.\ arr\ dup >\!\!>\!\!>\ f *** g
\end{array}
$$

Figure 3: Lambda Calculus + Arrows Definitions

The arrow dup_i is implemented in terms of the helper arrow dup_aux_i, which is a family of arrows, indexed by $i \in \mathbb{N}$, that duplicate their arguments i times:

$$
\begin{array}{lcl}
dup\_aux_1\ x & = & x \\
dup\_aux_{i+1}\ x & = & (x, dup_i\ x)
\end{array}
$$

dup_i, used in the translation in Figure 7, is then defined:

$$dup_i = arr\ dup\_aux_i$$

Similarly par_i is a family of arrows, indexed by $i \in \mathbb{N}$, used in the translation, that forwards i inputs in parallel. We first define a helper:

$$
\begin{array}{lcl}
par\_aux_1 & = & id \\
par\_aux_{i+1} & = & id *** par_i
\end{array}
$$

and then:

$$par_i = arr\ par\_aux_i$$

Faust's fan out operator (<:) is easily defined in terms of arrows' sequential composition operator (>>>) and family of duplication arrows (dup_i). For the fan in operator (:>) we must define an additional family of merge functions ($merge_i$), that sums pairs of channels into a single channel. As above we define a helper function:

$$
\begin{array}{lcl}
merge\_aux_1\ s & = & fst\ s + snd\ s \\
merge\_aux_{i+1}\ s & = & fst\ s + merge\_aux_i\ (snd\ s)
\end{array}
$$

and then:

$$merge_i = arr\ merge\_aux_i$$

Although it is straightforward to emulate Faust's differing number of arguments in arrows, it is syntactically a limitation of arrows, as defined in Haskell. In Section 4 an alternative is discussed, based on an encoding of arrows in qualified types and removed from the restrictions of Haskell's type system.

To complete this section we state a soundness theorem for our translation.

Theorem 1. *If* $\Gamma \vdash M : I_M \rightsquigarrow_B O_M$ *and* $\Gamma \vdash N : I_M \rightsquigarrow_B O_M$ *are typing judgments in the algebra of blocks, such that* $M = N$ *under the laws give in Figure 4 and the semantics of blocks given in Orlarey et al [2]. Then there exists* $[\![\Gamma \vdash M : I_M \rightsquigarrow_B O_M]\!]_{st} = [\![\Gamma]\!]_{st} \vdash [\![M]\!]_{st} : [\![I_M \rightsquigarrow_B O_M]\!]_{st}$ *and* $[\![\Gamma \vdash N : I_N \rightsquigarrow_B O_N]\!]_{st} = [\![\Gamma]\!]_{st} \vdash [\![N]\!]_{st} : [\![I_N \rightsquigarrow_B O_N]\!]_{st}$*, such that* $[\![M]\!]_{st} = [\![N]\!]_{st}$ *under the laws given in Figure 1 and Figure 2.*

The proof follows by induction over M.

4. A NEW APPROACH TO FOREIGN CALLS

Faust does not provide a general mechanism for side-effects and instead allows them only through its language for interfaces, such as the one used for describing GUIs for interaction with Faust programs, e.g. sliders, pots, etc. Faust also provides the ability to connect MIDI/OSC interfaces in a similar manner. Both of these capabilities provide limited, i.e. controlled, side-effects and in itself this is fine. However, Faust also provides the ability to call any C function through its foreign function interface:

$$
\begin{array}{l}
\mathsf{ffunction}(< \mathsf{function} - \mathsf{declaration} >, \\
\qquad\qquad < \mathsf{include} - \mathsf{file} >, < \mathsf{library} >)
\end{array}
$$

where the function-declaration must be of the form

$$< \mathsf{type} > \mathsf{fn}(< \mathsf{type} >);$$

where $<$ type $>$ is either int or *float*. In addition, the input type can be omitted, indicating no input argument. Thus,

$$< \mathsf{function} - \mathsf{declaration} >$$

means one of the following:

$$
\begin{array}{l}
int\ fn(int); \\
int\ fn(float); \\
float\ fn(int); \\
float\ fn(float); \\
int fn(); \\
float\ fn();
\end{array}
$$

Of course, the intention of the foreign interface is to provide Faust programs to access pre-existing DSP libraries or other highly optimized functions written in C, but, in general, additional side-effects can go unchecked! A consequence of this approach is that even though Faust itself is a (mostly) functional language there are no checks or design constraints placed within the type system to enforce (really control) this with the outside world.

An approach to structuring side-effects, in a type safe manner, was proposed by Wadler [19] using Moggi's monadic interpretation of denotational semantics [9]. Analogous to our earlier definitions for arrows, monads define a class of computations that are constructed to conform to a given structure[5]:

$$
\begin{array}{lcl}
(>\!\!>\!\!=) & : & m\ A \to (A \to m\ B) \to m\ B \\
\mathsf{return} & : & A \to m\ A
\end{array}
$$

[5]For conciseness we give only a minimal definition of monads, the interested reader can consult [19, 9].

$$Syntax$$

$$\begin{array}{llll} Types & A, B, C & := & ... \mid A \rightsquigarrow_B B \\ Terms & L, M, N & := & ... \mid ! \mid mem \mid \_ \mid ,_b \mid : \mid <: \mid :> \mid \sim \end{array}$$

$$Types$$

$$\frac{\Gamma \vdash M : I_M \rightsquigarrow_B O_M \qquad \Gamma \vdash N : I_N \rightsquigarrow_B O_N \qquad O_M = I_N}{\Gamma \vdash (M : N) : I_M \rightsquigarrow_B O_N}$$

$$\frac{\Gamma \vdash M : I_M \rightsquigarrow_B O_M \qquad \Gamma \vdash N : I_N \rightsquigarrow_B O_N}{\Gamma \vdash (M,_b N) : I_M + I_N \rightsquigarrow_B O_M + O_N}$$

$$\frac{\Gamma \vdash M : I_M \rightsquigarrow_B O_M \qquad \Gamma \vdash N : I_N \rightsquigarrow_B O_N \qquad O_N \leq I_M \qquad I_N \leq O_M}{\Gamma \vdash (M \sim N) : I_M - O_N \rightsquigarrow_B O_M}$$

$$\frac{\Gamma \vdash M : I_M \rightsquigarrow_B O_M \qquad \Gamma \vdash N : O_M * k \rightsquigarrow_B O_N \qquad O_M[i] = I_N[i + j * O_M] \qquad j < k}{\Gamma \vdash (M <: N) : I_M \rightsquigarrow_B O_N}$$

$$\frac{\Gamma \vdash M : I_M \rightsquigarrow_B I_N * k \qquad \Gamma \vdash N : I_N \rightsquigarrow_B O_N \qquad I_N[i] = O_M[i + j * I_N] \qquad j < k}{\Gamma \vdash (M >: N) : I_M \rightsquigarrow_B O_N}$$

$$\begin{array}{c} \_ : A \rightsquigarrow_B A \\ ! : A \rightsquigarrow_B () \\ mem : float \rightsquigarrow_B float \end{array}$$

$$Laws$$

$$\begin{array}{llcl} (ba_1) & ((A : B) : C) & = & (A : (B : C)) \\ (ba_2) & ((A,_b B),_b C) & = & (A,_b (B,_b C)) \\ (ba_3) & ((A <: B) <: C) & = & (A <: (B <: C)) \\ (ba_4) & ((A :> B) :> C) & = & (A :> (B :> C)) \end{array}$$

Figure 4: Block Diagram Algebra

where m is a monad and any instance must satisfy the following laws:

$$\begin{array}{lcl} \mathsf{return}\ M >\!\!>\!\!= N & = & M\ N \\ M >\!\!>\!\!= \mathsf{return} & = & M \\ M >\!\!>\!\!= \mathsf{return} \cdot f & = & map\ f\ M \\ M >\!\!>\!\!= (\lambda x.\ N\ x >\!\!>\!\!= H) & = & (M >\!\!>\!\!= N) >\!\!>\!\!= H \end{array}$$

But what do monads buy us? Monads bring modularity and by defining an operation monadically, it is possible to hide underlying machinery in a way that allows new features to be incorporated into the monad transparently. This enables the functional aspect of a language, for example, to remain functional and untouched by less pure, i.e. side-effecting, code. Walder gives numerous examples of useful monads, but maybe the most interesting in this context is the state monad [19]. The monad m for the state monad SM is defined as:

$$\mathsf{SM}\ A = \mathsf{SM}\ (S \rightarrow (A, S))$$

where A is the result type of a (potentially) stateful computation and S is the type of state. In other words SM is the type of computations that implicitly carries a state of type S. We omit the full definition of SM here, see Wadler's paper for an example implementation [19]. Combined with Haskell's do-notation[5] the state monad provides a powerful approach to implementating stateful computations that are combined with the pure in a type safe manner. The state monad is just one example of an interesting monad and like arrows they provide a powerful abstraction of computation.

Haskell, for example, approaches foreign function calls through a special IO monad [20]. We could easily recast Faust's function declarations for foreign calls within the IO monad, which would have types:

$$\begin{array}{c} fn : int \rightarrow IO\ int \\ fn : int \rightarrow IO\ float \\ fn : float \rightarrow IO\ int \\ fn : float \rightarrow IO\ float \\ fn : () \rightarrow IO\ int \\ fn : () \rightarrow IO\ float \end{array}$$

It seems likely that the intention of Faust's foreign function calls, is for the implementations to be stateless, at least in a way that would be visible to the calling program. Thus, enforcing that they all appear in the IO monad could place additional restrictions on the compiler that limit optimization. A less drastic approach would be to parametrize foreign function calls by a monad m. The programmer would then explicitly state what monad a particular foreign call was to be executed in.

$$\mathsf{type}\ SF\ a\ b \quad = \quad SF\ \{unSF\ :: a \to (b, SF\ a\ b)\}$$

$$\begin{array}{lll}
arr\ f & = SF\ h & \\
& \mathsf{where}\ h\ x & = \quad (f\ x, SF\ h) \\
first\ f & = SF\ (h\ f) & \\
& \mathsf{where}\ h\ f\ (x, z) & = \quad (\lambda s.\ (fst\ s, SF\ (h\ (snd\ s)))\ (unSF\ f\ x) \\
f >\!\!>\!\!> g & = SF\ (h\ f\ g) & \\
& \mathsf{where}\ h\ f\ (x, z) & = \quad (\lambda s.\ (\lambda s'.\ (fst\ s', SF\ (h\ (snd\ s)\ (snd\ s')))(unSF\ g\ (fst\ s)))\ (unSF\ f\ x) \\
\mathsf{loop}\ f & = SF\ (h\ f) & \\
& \mathsf{where}\ h\ f\ x & = \quad \begin{array}{l}\mathsf{let}\ ((y, z), f') = unSF\ f\ (x, z) \\ \mathsf{in}\ (y, SF\ (h\ f')))\end{array} \\
delay\ i & = SF\ (h\ i) & \\
& \mathsf{where}\ h\ i\ x & = \quad (i, SF\ (h\ x))
\end{array}$$

Figure 5: Causal Stream Transformer

$$\begin{array}{lcl}
[\![float]\!]_{st} & = & float \\
[\![A \times B]\!]_{st} & = & [\![A]\!]_{st} \times [\![B]\!]_{st} \\
[\![()]\!]_{st} & = & () \\
[\![A \to B]\!]_{st} & = & [\![A]\!]_{st} \to [\![B]\!]_{st} \\
[\![A \rightsquigarrow_B B]\!]_{st} & = & [\![A]\!]_{st} \rightsquigarrow [\![B]\!]_{st}
\end{array}$$

Figure 6: Block Algebra types to Stream Transformer Arrows

An immediate advantage of this is that we can return to Faust's current definition of foreign calls setting[6]

$$m = \mathsf{Identity},$$

i.e. the identity monad, which is easily optimized away.

Until now we have proposed the idea of introducing a monad interface for foreign function calls to Faust and could eventually consider extending Faust to allow use of additional monads. But for this to make sense, then Faust's semantics, both static and dynamic, must also be extended and this is were the translation to arrows can play a practical role as well as a theoretical one. Recent work by Perez et al introduce Monadic Stream Functions (MSF), a generalization of Yampa's stream transformer arrow [21]. Put simply a MSF is an arrow that can be computed within a given monad.

Perez et al define a MSF type and an evaluation function that applies an MSF to an input and returns, in a monadic context, an output and a continuation. We give only the type here:

$$\mathsf{MSF}\ m\ A\ B \quad = \quad A \to m\ (B, \mathsf{MSF}\ m\ A\ B)$$

The observant reader will note that the definition of MSF is the stream transformer given in Figure 5 parametrized by the monad m. Perez et al provide a proof that MSF is indeed an arrow and in fact it can be shown that it additionally satisfies the laws needed by the definition of CCA [21]. Thus, letting A and B be equal to $\mathbb{R}$ gives rise to a DSP arrow instance that is a target for the translation given in Section 3.

[6]We omit the definition of the identity monad as it is straightforward.

5. CONCLUSION

We have described a translation from Faust's Algebra of Blocks to a causal commutative variant of Hughes' arrows framework. While it is known that these languages are similar, to our knowledge, this is the first formal encoding. Our experience developing a DSP based DSL on arrows has benefited from developments in both Faust and Functional Reactive Programming, and to date our implementations works well in practice. By formalizing this relationship we open up a path to alternative approaches to composing control within Faust.

There are a number of areas for further work including:

- Faust is a excellent DSP programming language, with its concise syntax and powerful type system. However, it seems reasonable that the translation described in this paper could be used to develop and formalize new extensions, e.g. exposing embedded micro controller interfaces (such as GPIO), in a type safe manner. Monadic stream functions (DSP arrows) might be combined with a clock monad, providing an approach to resamping and scheduling.
- Our goal is to target embedded devices using a DSL called $\leftarrow AUDIO\prec$ that provides a research framework for building and playing with low-latency digital instruments and controllers. Inspired by the Bela platform [22], but using low-cost ARM Micro controllers for control and DSP processing, we are designing a type safe framework for exploring the composition of different DSLs for both control and DSP programming. Like Stride [23], $\leftarrow AUDIO\prec$ is designed with exposing the I/O pins, both analog and digital, in mind. Following similar arguments made in this paper we believe recent work on remote monads can provide a type safe and composable approach to connecting the low-level world of electronics to high-level abstractions for digital instrument design [24, 25].
- Faust's syntax is very concise and while the syntax of $\leftarrow AUDIO\prec$ is more refined than Hughes' embedding in Haskell, there is more that can be learned from Faust's approach. Future work could try to address the limitations of $\leftarrow AUDIO\prec's$ syntax by adopting one closer to Faust's, while remaining solely within the domain of DSP arrows. By staying with the arrow framework and more generally within a type system built on qualified types it is possible

$$\frac{}{[\![\Gamma \vdash x : A]\!]_{st} = [\![\Gamma]\!]_{st} \vdash x : [\![A]\!]_{st}}$$

$$\frac{}{[\![\Gamma \vdash \_ : A \rightsquigarrow_B A]\!]_{st} = [\![\Gamma]\!]_{st} \vdash arr\ id : [\![A \rightsquigarrow_B A]\!]_{st}}$$

$$\frac{}{[\![\Gamma \vdash ! : A \rightsquigarrow_B ()]\!]_{st} = [\![\Gamma]\!]_{st} \vdash arr\ (\lambda x.\ ()) : [\![A]\!]_{st} \rightsquigarrow ()}$$

$$\frac{[\![\Gamma \vdash M : \mathit{float}]\!]_{st}}{[\![\Gamma \vdash mem\ M : \mathit{float}]\!]_{st} = [\![\Gamma]\!]_{st} \vdash [\![M]\!]_{st} >\!\!>\!\!> delay\ 0 : \mathit{float}}$$

$$\frac{\begin{array}{c}[\![\Gamma \vdash M : I_M \rightsquigarrow_B O_M]\!]_{st} = [\![\Gamma]\!]_{st} \vdash [\![M]\!]_{st} : [\![I_M \rightsquigarrow_B O_M]\!]_{st} \\ [\![\Gamma \vdash N : I_N \rightsquigarrow_B O_N]\!]_{st} = [\![\Gamma]\!]_{st} \vdash [\![N]\!]_{st} : [\![I_N \rightsquigarrow_B O_N]\!]_{st} \qquad O_M = I_N\end{array}}{[\![\Gamma \vdash (M : N) : I_M \rightsquigarrow_B O_N]\!]_{st} = \Gamma \vdash [\![M]\!]_{st} >\!\!>\!\!> [\![N]\!]_{st} : [\![I_M]\!]_{st} \rightsquigarrow [\![O_N]\!]_{st}}$$

$$\frac{\begin{array}{c}[\![\Gamma \vdash M : I_M \rightsquigarrow_B O_M]\!]_{st} = [\![\Gamma]\!]_{st} \vdash [\![I_M]\!]_{st} : [\![I_M \rightsquigarrow_B O_M]\!]_{st} \\ [\![\Gamma \vdash N : I_N \rightsquigarrow_B O_N]\!]_{st} = [\![\Gamma]\!]_{st} \vdash [\![I_N]\!]_{st} : [\![I_N \rightsquigarrow_B O_N]\!]_{st}\end{array}}{[\![\Gamma \vdash (M,_b N) : I_M + I_N \rightsquigarrow_B O_M + O_N]\!]_{st} = [\![\Gamma]\!]_{st} \vdash [\![M]\!]_{st} *\!*\!* [\![N]\!]_{st} : I_M + I_N \rightsquigarrow_B O_M + O_N}$$

$$\frac{\begin{array}{l}[\![\Gamma \vdash M : I_M \rightsquigarrow_B O_M]\!]_{st} = [\![\Gamma]\!]_{st} \vdash [\![M]\!]_{st} : [\![I_M \rightsquigarrow_B O_M]\!]_{st} \\ [\![\Gamma \vdash N : O_M * k \rightsquigarrow_B O_N]\!]_{st} = [\![\Gamma]\!]_{st} \vdash [\![N]\!]_{st} : [\![O_M * k \rightsquigarrow_B O_N]\!]_{st} \\ O_M[i] = I_N[i + j * O_M] \qquad j < k\end{array}}{[\![\Gamma \vdash (M <: N) : I_M \rightsquigarrow_B O_N]\!]_{st} = [\![\Gamma]\!]_{st} \vdash [\![M]\!]_{st} >\!\!>\!\!> dup_{(k-1)} >\!\!>\!\!> [\![N]\!]_{st} : [\![I_M \rightsquigarrow_B O_N]\!]_{st}}$$

$$\frac{\begin{array}{l}[\![\Gamma \vdash M : I_M \rightsquigarrow_B I_N * k]\!]_{st} = [\![\Gamma]\!]_{st} \vdash [\![M]\!]_{st} : [\![I_M \rightsquigarrow_B I_N * k]\!]_{st} \\ [\![\Gamma \vdash N : I_N \rightsquigarrow_B O_N]\!]_{st} = [\![\Gamma]\!]_{st} \vdash [\![N]\!]_{st} : [\![I_N \rightsquigarrow_B O_N]\!]_{st} \\ I_N[i] = O_M[i + j * I_N] \qquad j < k\end{array}}{[\![\Gamma \vdash (M >: N) : I_M \rightsquigarrow_B O_N]\!]_{st} = [\![\Gamma]\!]_{st} \vdash [\![M]\!]_{st} >\!\!>\!\!> merge_{(k-1)} >\!\!>\!\!> [\![N]\!]_{st} : [\![I_M \rightsquigarrow_B O_N]\!]_{st}}$$

$$\frac{\begin{array}{c}[\![\Gamma \vdash M : I_M \rightsquigarrow_B O_M]\!]_{st} = [\![\Gamma]\!]_{st} \vdash [\![M]\!]_{st} : [\![I_M \rightsquigarrow_B O_M]\!]_{st} \\ [\![\Gamma \vdash N : I_N \rightsquigarrow_B O_N]\!]_{st} = [\![\Gamma]\!]_{st} \vdash [\![N]\!]_{st} : [\![I_N \rightsquigarrow_B O_N]\!]_{st}\end{array}}{\begin{array}{l}[\![\Gamma \vdash (M \sim N) : I_M + I_N \rightsquigarrow_B O_M + O_N]\!]_{st} = \\ [\![\Gamma]\!]_{st} \vdash loop\ (arr\ swap >\!\!>\!\!> [\![M]\!]_{st} >\!\!>\!\!> arr\ id\ \&\!\&\!\&\ (delay\ 0 >\!\!>\!\!> [\![N]\!]_{st})) : [\![I_M + I_N \rightsquigarrow_B O_M + O_N]\!]_{st}\end{array}}$$

Figure 7: Block Algebra values to Stream Transformer Arrows

to apply many of the developments in monads, etc. without the need to develop additional heavyweight type systems.

6. ACKNOWLEDGMENTS

The authors would like to thank the reviewers for their constructive and useful feedback.

7. REFERENCES

[1] Y. Orlarey, D. Fober, and S. Letz, "An algebraic approach to block diagram constructions," in *Actes des Journées d'Informatique Musicale JIM2002, Marseille*, GMEM, Ed., 2002, pp. 151–158.

[2] Y. Orlarey, D. Fober, and S. Letz, "Syntactical and semantical aspects of faust," *Soft Computing*, vol. 8, no. 9, pp. 623–632, Sep 2004.

[3] John Hughes, "Generalising monads to arrows," *Sci. Comput. Program.*, vol. 37, no. 1-3, pp. 67–111, May 2000.

[4] Paul Hudak, "Building domain-specific embedded languages," *ACM Comput. Surv.*, vol. 28, no. 4es, pp. 196, 1996.

[5] Simon Marlow, "Haskell 2010 language report," 2010.

[6] Conal Elliott and Paul Hudak, "Functional reactive animation," in *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997.*, 1997, pp. 263–273.

[7] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson, "Arrows, robots, and functional reactive programming," in *Advanced Functional Programming, 4th International School, AFP 2002, Oxford, UK, August 19-24, 2002, Revised Lectures*, 2002, pp. 159–187.

[8] Hai Liu and Paul Hudak, "Plugging a space leak with an arrow," *Electr. Notes Theor. Comput. Sci.*, vol. 193, pp. 29–45, 2007.

[9] Eugenio Moggi, "Notions of computation and monads," *Inf. Comput.*, vol. 93, no. 1, pp. 55–92, July 1991.

[10] Conor Mcbride and Ross Paterson, "Applicative programming with effects," *J. Funct. Program.*, vol. 18, no. 1, pp. 1–13, Jan. 2008.

[11] Hai Liu, Eric Cheng, and Paul Hudak, "Causal commutative arrows," *J. Funct. Program.*, vol. 21, no. 4-5, pp. 467–496, 2011.

[12] Ross Paterson, "A new notation for arrows," in *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, 2001, pp. 229–240.

[13] Haskell B. Curry, Robert Feys, and William Craig, *Combinatory Logic*, North-Holland Publishing Company, 1958.

[14] Wikipedia contributors, "Faust (programming language)," 2018, [Online; accessed 15-Feburary-2018].

[15] Bart Jacobs, Chris Heunen, and Ichiro Hasuo, "Categorical semantics for arrows," *J. Funct. Program.*, vol. 19, no. 3-4, pp. 403–438, July 2009.

[16] EXEQUIEL RIVAS and MAURO JASKELIOFF, "Notions of computation as monoids," *Journal of Functional Programming*, vol. 27, pp. e21, 2017.

[17] Sam Lindley, Philip Wadler, and Jeremy Yallop, "The arrow calculus," *J. Funct. Program.*, vol. 20, no. 1, pp. 51–69, 2010.

[18] Hai Liu, *The Theory and Practice of Causal Commutative Arrows*, Ph.D. thesis, Yale University, 2011.

[19] Philip Wadler, "Monads for functional programming," in *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, London, UK, UK, 1995, pp. 24–52, Springer-Verlag.

[20] Simon L. Peyton Jones and Philip Wadler, "Imperative functional programming," in *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, New York, NY, USA, 1993, POPL '93, pp. 71–84, ACM.

[21] Ivan Perez, Manuel Bärenz, and Henrik Nilsson, "Functional reactive programming, refactored," in *Proceedings of the 9th International Symposium on Haskell*, New York, NY, USA, 2016, Haskell 2016, pp. 33–44, ACM.

[22] Andrew McPherson, Robert Jack, and Giulio Moro, "Action-sound latency: Are our tools fast enough?," in *Proceedings of the International Conference on New Interfaces for Musical Expression*, Brisbane, Australia, 2016, vol. 16 of *2220-4806*, pp. 20–25, Queensland Conservatorium Griffith University.

[23] Joseph Tilbian and Andres Cabrera, "Stride for interactive musical instrument design," in *Proceedings of the International Conference on New Interfaces for Musical Expression*, Copenhagen, Denmark, 2017, pp. 446–449, Aalborg University Copenhagen.

[24] Andy Gill, Neil Sculthorpe, Justin Dawson, Aleksander Eskilson, Andrew Farmer, Mark Grebe, Jeffrey Rosenbluth, Ryan Scott, and James Stanton, "The remote monad design pattern," in *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, New York, NY, USA, 2015, Haskell '15, pp. 59–70, ACM.

[25] Mark Grebe and Andy Gill, "Haskino: A remote monad for programming the Arduino," in *Practical Aspects of Declarative Languages*, Lecture Notes in Computer Science. 2016.

[26] Mark P. Jones, "Partial evaluation for dictionary-free overloading," Tech. Rep., 1993.

[27] P. Wadler and S. Blott, "How to make ad-hoc polymorphism less ad hoc," in *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, New York, NY, USA, 1989, POPL '89, pp. 60–76, ACM.

[28] Mark P. Jones, *Qualified Types: Theory and Practice*, Cambridge University Press, New York, NY, USA, 1995.

[29] Pierre Jouvelot, "Type inference in multirate faust is undecidable," FEEVER!Mee(ng, 2014.

[30] Benedict R. Gaster, "$\leftarrow AUDIO\prec$ or real time audio with arrows," In development, 2018.

[31] Manuel Barenz, "Rhine - frp with type-level clocks," See https://www.manuelbaerenz.de/sites/default/files/Rhine.pdf, 2018.

[32] Sheng Liang, Paul Hudak, and Mark Jones, "Monad transformers and modular interpreters," in *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, New York, NY, USA, 1995, POPL '95, pp. 333–343, ACM.

[33] Yann Orlarey and Pierre Jouvelot, "Signal rate inference for multidimensional faust," in *Proceedings of the 28th Symposium on the Implementation and Application of Functional Programming Languages*, New York, NY, USA, 2016, IFL 2016, pp. 1:1–1:12, ACM.

[34] Yann Orlarey, Stéphane Letz, and Dominique Fober, *New Computational Paradigms for Computer Music*, chapter "Faust: an Efficient Functional Approach to DSP Programming", Delatour, Paris, France, 2009.

[35] Julius O. Smith, "Signal processing libraries for Faust," in *Proceedings of Linux Audio Conference (LAC-12)*, Stanford, USA, May 2012.

[36] Albert Gräf, "pd-faust: An integrated environment for running Faust objects in Pd," in *Proceedings of the Linux Audio Conference (LAC-12)*, Stanford, USA, April 2012.

[37] Romain Michon and Julius O. Smith, "Faust-STK: a set of linear and nonlinear physical models for the Faust programming language," in *Proceedings of the 14th International Conference on Digital Audio Effects (DAFx-11)*, Paris, France, September 2011.