The W-calculus: A Synchronous Framework for the Verified Modelling of Digital Signal Processing Algorithms

Emilio Jesús Gallego Arias, Pierre Jouvelot, Sylvain Ribstein, Dorian Desblancs

The W-calculus: A Synchronous Framework for the Verified Modelling of Digital Signal Processing Algorithms

Emilio Jesús Gallego Arias
Université Paris Cité, CNRS, Inria, IRIF, F-75013
Paris, France
emilio-jesus.gallego-arias@inria.fr

Pierre Jouvelot
MINES ParisTech, PSL University
Paris, France
pierre.jouvelot@mines-paristech.fr

Sylvain Ribstein
France
sylvain.ribstein@gmail.com

Dorian Desblancs
École normale supérieure Paris-Saclay
France
dorian.desblancs@ens-paris-saclay.fr

Abstract

We introduce the W-calculus, an extension of the call-by-value λ-calculus with synchronous semantics, designed to be flexible enough to capture different implementation forms of Digital Signal Processing algorithms, while permitting a direct embedding into the Coq proof assistant for mechanized formal verification. In particular, we are interested in the different implementations of classical DSP algorithms such as audio filters and resonators, and their associated high-level properties such as Linear Time-invariance.

We describe the syntax and denotational semantics of the W-calculus, providing a Coq implementation. As a first application of the mechanized semantics, we prove that every program expressed in a restricted syntactic subset of *W* is linear time-invariant, by means of a characterization of the property using logical relations. This first semantics, while convenient for mechanized reasoning, is still not useful in practice as it requires re-computation of previous steps. To improve on that, we develop an imperative version of the semantics that avoids recomputation of prior stream states. We empirically evaluate the performance of the imperative semantics using a *staged* interpreter written in OCaml, which, for an input program in *W*, produces a specialized OCaml program, which is then fed to the optimizing OCaml compiler. The approach provides a convenient path from the high-level semantical description to low-level efficient code.

FARM '21, August 27, 2021, Virtual, Republic of Korea

ACM ISBN 978-1-4503-8613-5/21/08…$15.00
https://doi.org/10.1145/3471872.3472970

CCS Concepts: • **Computer systems organization → Real-time languages**; • **Software and its engineering → Formal software verification**.

Keywords: Digital Signal Processing, Programming Language Semantics, Synchronous Programming, Formal Verification, Linear Time-invariance

ACM Reference Format:
Emilio Jesús Gallego Arias, Pierre Jouvelot, Sylvain Ribstein, and Dorian Desblancs. 2021. The W-calculus: A Synchronous Framework for the Verified Modelling of Digital Signal Processing Algorithms. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Functional Art, Music, Modelling, and Design (FARM '21), August 27, 2021, Virtual, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3471872.3472970

1 Introduction

Real-time Digital Signal Processing (DSP) lies at the frontier between the physical reality and the digital world. Mobile applications, data acquisition, wireless radio,...: real-time DSP is pervasive and used in every digital device.

One key domain where DSP thrives is music, where all digital audio formats can be interpreted as clocked "streams" of discrete "samples". which can be tweaked and processed via programming. The realm of computer music has thus been a steady provider of music-oriented DSP programming languages that offer a wide variety of approaches to handle domain-specific applications.

DSP programming is however notoriously known for posing difficult challenges. For instance, crucial properties are often not preserved under composition, making modular reasoning difficult; stream, buffer, timing, and memory handling is costly and error prone if done manually.

There is a wide spectrum of research trying to provide definitive answers to all or some of the previous problems. In particular, some *domain-specific languages (DSL) for DSP* aim to provide more convenient programming models, allowing programmers to produce code closer to the mathematical

Corrected Version of Record. V.1.1. Published June 20, 2022.

specification of the process they intend to implement, while cumbersome or repetitive details are taken care of by the toolchain. On the other side, *program verification techniques* aim to support rigorous code analysis, pointing out possible defects or problems, and constructing *correctness certificates* that can provide guarantees in high-assurance environments.

However, practical, provably-correct development of DSP systems is still an open problem: the use — due to efficiency and compatibility — of low-level implementation languages does not mix well with the mathematical nature of the specifications; existing state-of-the-art verification techniques either incur an insurmountable overheard, or are severely limited in what they can prove.

1.1 Design Goals

The core goal of this work is to provide a unified core mechanism to bridge performance and verification concerns for DSP programming.

With the introduction of the W-calculus, we start an experiment in DSL design with a triple objective: a) to have a good basis for a declarative DSP language amenable to state-of-the-art interactive verification, able to handle complex properties, such as linearity, bound properties, or filter equivalence; b) to ensure the language can accommodate the performance requirements of real-time processing, in particular in terms of memory allocation; and c) to provide a formal basis upon which to develop a full-fledged usable front-end and compiler, amenable to use by DSP experts.

We have been careful to ensure that the language remains low-level enough to distinguish among different implementation strategies, such as the different forms for filters, which are extremely important when reasoning about numerical properties. While we don't address the verification of floating-point numerical properties of filters in this paper, it is important that our semantics remain compatible with this future line of work; thus approaches to program interpretation that may involve non-stable numerical transformations have been ruled out.

1.2 Key Contributions and Structure of the Paper

We present the work in two main parts.

1. Section 2 describes the W-calculus, including syntax, design discussion, examples, and formal denotational semantics, defined using the Coq integrative proof assistant. Section 2.4 develops a concrete use case of the mechanized semantics: a proof that every well-typed program in the W-calculus is a linear function, in the sense of Linear Time-Invariant (LTI) systems.
2. Section 3 introduces an imperative semantics for the W-calculus programs, provided these programs perform only bounded accesses on streams' past values. This allows for buffers to be statically allocated. We implement the imperative semantics as a staged interpreter, which produces low-level, efficient code. Experimental run-time performance evaluation is done comparing the efficiency of the generated code versus hand-written versions of the same programs.

We conclude the paper with discussion on related (Section 4) and future (Section 5) work.

The source code for programs and proofs referenced in this paper can be found at https://github.com/ejgallego/mini-wagner-coq/.

1.3 Mathematical Preliminaries

We assume the reader to be familiar with basic functional programming notation and typing judgments. Unless otherwise specified, we use as our mathematical and logical universe Coq's type theory, and in particular the mathematical objects such as matrices and number structures provided by the Mathematical Components library. While we do use LaTeX-improved notation in the paper, all the definitions have a direct correspondence with their Coq counterpart. In the second part of the paper, we will assume familiarity with OCaml's syntax.

In particular, we rely on the definition of a numeric type $\mathbb{R}$, or `R`, which in our case is assumed to be an integral domain (basically a non-zero commutative ring), n-ary tuples of elements, written as `n.-tuple A` for elements of type `A`, with elements written using list-like notation `[:: e1, ..., en]`, and one-column matrices of dimension a, written as `'cV[R]_a`. All these types are already equipped with their corresponding operations, such as `nth` for accessing the n-th element of a tuple or list.

On the DSP side, we mainly follow the conventions in [54]. In this setting, streams are time-indexed functions, usually returning samples; negative-time access is assumed to return zero, so one can write $y(n) = x_1(n-1) + x_2(n-2)$ to define a delay y that takes two streams x_1 and x_2 and delays them by different amounts, 1 and 2, before mixing them. We also use a shift operator for streams defined as $x_{|k}(n) = x(n-k)$. It is also common to write x' for $x'(n) = x(n-1)$, that is to say, a one-sample delay. We restrict ourselves to causal definitions, that is to say, filters cannot refer to values in the future or introduce ill-defined definitional loops. In this setting, the definition of a linear time-invariant filter f is such that

$$f(x_{|k}) = f(x)_{|k} \text{ and}$$
$$f(c(x_1 + x_2)) = cf(x_1) + cf(x_2).$$

2 The W-calculus

2.1 Syntax

The syntax of W expressions e, f and types τ is given in Figure 1. Given a base sample type R, W's types are either n-tuples or functions from one tuple to another. For example, a stereo-to-mono program will have type $\mathsf{R}_2 \to \mathsf{R}_1$, etc. We write R for R_1 when it is clear from the context.

types

$$\tau ::= \mathrm{R}_a \quad a \in \mathbb{N} \mid \mathrm{R}_a \to \mathrm{R}_b \quad a, b \in \mathbb{N}$$

expressions

$$\begin{aligned} e, f ::= {} & x_k && k \in \mathbb{N},\ x \in \mathbb{V} \\ \mid {} & \lambda x.e \\ \mid {} & f\ e \\ \mid {} & c * e && c \in \mathrm{R} \\ \mid {} & e_1 + e_2 \\ \mid {} & \pi_i(e) && i \in \mathbb{N}, i > 0 \\ \mid {} & (e_1, e_2) \\ \mid {} & \text{feed } x.e \end{aligned}$$

Figure 1. Syntax of W expressions and types

Expressions, which denote discrete "streams" of quantified samples, are standard, except for two cases: variables and feedback. A variable x_j denotes x's value at time step $n - j$, where n is the current time step; note that we write x for x_0 when it is clear from the context, and let $x = e_1$ in e_2 for $(\lambda x.e_2)\ e_1$. The second interesting case is the *feedback* expression feed $x.e$, which implements causal self-referential expressions. For example, assuming the addition to W of straightforward syntactic extensions such as infix operators, constants and the like, feed $x.x + 1$ will implement a counter, with value n, where n is again the global time. References to x inside e are always assumed to start in the previous time step, to avoid causality problems; thus the previous example can be read in mathematical form (*not* W's syntax) as $x_{n+1} = x_n + 1$, assuming x at time step 0 or less is always zero, for any x.

Typing is also standard in our system, with rules shown in Figure 2. Note that products are the concatenated tuples of their projections, instead of the usual structural approach.

2.2 Examples

This calculus, while simple, can already be used to implement some core DSP primitives, as shown in Figure 3.

The examples have been taken from [55], and show a typical second-order infinite impulse response filter, in its two implementation forms; note how W does actually distinguish these two filters which, modulo floating-point arithmetic, are extensionally equivalent. We use the dot product $\vec{b} \cdot x$ as an abbreviation for $b_0 \times x_0 + \ldots + b_{a-1} \times x_{a-1}$, assuming $\vec{b}$ is the constant tuple $[b_0; \ldots; b_{a-1}]$ (in mathematical notation).

The third example is a *waveguide resonator*, a simple circuit implementing the wave equation. We write the example first using a notation natural to the DSP expert, which we then desugar to the corresponding W expression, using a feedback over a single variable of the product.

$$\frac{\Gamma\ x = \mathrm{R}_a}{\Gamma \vdash x_k : \mathrm{R}_a}\ \textsc{Var} \qquad \frac{\Gamma, x : \mathrm{R}_a \vdash e : \mathrm{R}_b}{\Gamma \vdash \lambda x.e : \mathrm{R}_a \to \mathrm{R}_b}\ \textsc{Lam}$$

$$\frac{\Gamma \vdash f : \mathrm{R}_a \to \mathrm{R}_b \qquad \Gamma \vdash e : \mathrm{R}_a}{\Gamma \vdash f\ e : \mathrm{R}_b}\ \textsc{App}$$

$$\frac{c \in \mathrm{R} \qquad \Gamma \vdash e : \mathrm{R}_a}{\Gamma \vdash c * e : \mathrm{R}_a}\ \textsc{Scale}$$

$$\frac{\Gamma \vdash e_1 : \mathrm{R}_a \qquad \Gamma \vdash e_2 : \mathrm{R}_a}{\Gamma \vdash e_1 + e_2 : \mathrm{R}_a}\ \textsc{Add}$$

$$\frac{\Gamma \vdash e : \mathrm{R}_a \qquad 1 \le i \le a}{\Gamma \vdash \pi_i(e) : \mathrm{R}_1}\ \textsc{Proj}$$

$$\frac{\Gamma \vdash e_1 : \mathrm{R}_a \qquad \Gamma \vdash e_2 : \mathrm{R}_b}{\Gamma \vdash (e_1, e_2) : \mathrm{R}_{a+b}}\ \textsc{Prod}$$

$$\frac{\Gamma, x : \mathrm{R}_a \vdash e : \mathrm{R}_a}{\Gamma \vdash \text{feed } x.e : \mathrm{R}_a}\ \textsc{Feed}$$

Figure 2. W typing rules

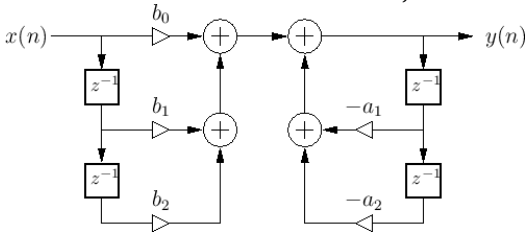
2.3 Semantics

We present the semantics for W programs as time-indexed functions mapping syntactic W objects to objects in the mathematical universe of Coq. Our use of indexing allows for by-construction causal semantical definition of streams, while relying only on induction (no co-induction on infinite streams).

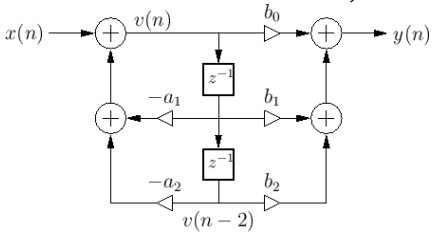
The interpretation $[\![\tau]\!]^k_{\mathsf{T}}$ for a type τ at time step k is defined in Figure 4. The reader can observe that the index k is only relevant for functions, but, indeed, here is the core idea of our interpretation for stream transformers: a stream transformer from $\tau \to \sigma$ at time k is a function that will take $k + 1$ values of the input τ, for the past k steps plus the actual value, producing the value for the current time step. Note that, in this model, functions at time k can access all past values of their arguments, without a bound.

The semantics for well-typed expressions is defined in Figure 5. An interpretation function up to time n, $\mathsf{I}_{\{n\}}$, for a well-typed expression $\Gamma \vdash e : \tau$ will take a bounded time step $k < n$, a well-formed environment for time step k, env Γ k, and will produce an element of the interpretation of its type τ. Environments are defined as heterogeneous lists, mapping each typed variable to its history of values. Recall that in our context, values in environments cannot be functional ones. The initial interpretation $\mathsf{I}_{\{0\}}$ will map every expression to the corresponding zero value for the type.

Second-order recursive filter, DF-I



Second-order recursive filter, DF-II

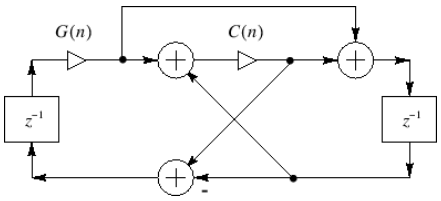


$\mathsf{df1} \equiv \lambda x.\ \mathsf{feed}\ y.\ \vec{b} \cdot x + \vec{a} \cdot y$

$\vec{a} \cdot y = \langle a_0, a_1 \rangle \cdot \langle y_0, y_1 \rangle \quad \vec{b} = \langle b_0, b_1, b_2 \rangle$

$\mathsf{df2} \equiv \lambda x.\ \mathsf{let}\ v = \mathsf{feed}\ v.\ x + \vec{a} \cdot v\ \mathsf{in}\ \vec{b} \cdot v$

Waveguide resonator



$$
\begin{aligned}
\mathsf{feed}\ x &= C \cdot (G \cdot x' + y') - y' \\
y &= C \cdot (G \cdot x' + y') + G \cdot x'
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{wgr} = {} & \mathsf{feed}\ z.\mathsf{let}\ x = \pi_1(z)\ \mathsf{in}\ \mathsf{let}\ y = \pi_2(z)\ \mathsf{in} \\
& (C \cdot (G \cdot x + y) - y \\
& , C \cdot (G \cdot x + y) + G \cdot x)
\end{aligned}
$$

Figure 3. Example W programs

$$
\begin{aligned}
[\![\mathsf{R}_a]\!]^k_{\mathsf{T}} &\triangleq \mathsf{tuple}_a\ \mathbb{R} \\
[\![\mathsf{R}_a \to \mathsf{R}_b]\!]^k_{\mathsf{T}} &\triangleq \mathsf{tuple}_{k+1}\ [\![\mathsf{R}_a]\!]_{\mathsf{T}} \to [\![\mathsf{R}_b]\!]_{\mathsf{T}}
\end{aligned}
$$

```
Definition tyI k t : Type :=
  match t with
  | tpair a ⇒ 'cV[R]_a
  | tfun a b ⇒ k.+1.-tuple 'cV[R]_a → 'cV[R]_b
  end.
```

Figure 4. Semantics of W types

Once we have defined what an interpretation is, the core of the semantics is given by the interpretation transformer $\mathcal{S}$. We write $\mathsf{P}, \Gamma \vdash e : \tau$, $k < n+1$, and Θ_k, for the first arguments (after unfolding the definition of $\mathsf{I}_{\{n+1\}}$), such that $\mathcal{S}_{\mathsf{P}}[\![\Gamma \vdash e : \tau]\!]^{k<n+1}_{\Theta_k}$ has type $[\![\tau]\!]^k_{\mathsf{T}}$ as required. The interpretation transformer is then defined structurally on expressions, so that, in particular, there is no problem regarding the termination requirements imposed by Coq, and termination is detected by Coq's regular guard checker.

There are a few interesting cases in this definition. For variables, the environment will contain the history of all the previous values of this variable; so we can just access it, or return the 0-interpretation if the access is out of bounds. Functions are interpreted as regular mathematical functions; function application builds the full history of the argument prior to passing it to the interpretation of the function. The feedback case is, however, a bit more tricky, as we now need to interpret the feedback expression itself, but over previous time steps. While we could strengthen our termination condition in Coq, to include both the structural and time-based orders to define a well-founded recursion, this usually leads to a complex proof setup, so we chose to untie the recursive call by moving the previous-time interpretation as a parameter $\mathsf{P} : \mathsf{I}_{\{n\}}$.

We can thus define $\mathcal{S}$ by induction on the terms, and then define the absolute interpretation function $\mathsf{I}_{\{n\}}$ by a simple induction on time.

2.4 Linearity

While the previous semantics may seem naturally correct, details are subtle in our context, and validation is required (the whole point, in fact, of encoding it inside a theorem prover).

We have proved without too much hassle a few basic properties such as that expected equations and program equivalences do hold, as well as checking inside Coq that the output is what is expected, as we can run the semantics, albeit quite slowly.

For this section, we focus however on what we think is a more interesting property that applies to the full set of well-typed W programs: linearity.

The avid reader may have noticed by now that our core syntax lacks constants, and indeed this omission is not a coincidence. We will fix this problem in the next section as we move towards more practical uses, but for now, we will forget about constants and proceed to prove a theorem that states that "all well-typed programs in the W-calculus are linear".

$$\mathcal{S} : \mathsf{I}_{\{n\}} \to \mathsf{I}_{\{n+1\}} \qquad \mathsf{I}_{\{n\}} \equiv \mathsf{expr}\ \Gamma\ \tau \to \forall\, k, k < n \to \mathsf{env}\ \Gamma\ k \to [\![\tau]\!]^k_{\mathsf{T}}$$

$$\Theta_k : \mathsf{env}\ \Gamma\ k \equiv (x \in \mathbb{V}) \to \mathsf{tuple}_{k+1}\ [\![\Gamma\ x]\!]_{\mathsf{T}}$$

$$\mathcal{S}_{\mathsf{P}}[\![\Gamma \vdash x_m : \mathsf{R}_a]\!]^{k<n+1}_{\Theta_k} \triangleq \mathsf{nth}\ m\ (\Theta_k\ x)\ 0_a, \qquad \text{where } 0_a \equiv \mathsf{init}\ a\ (\mathsf{fun}\ \_ \to 0)$$

$$\mathcal{S}_{\mathsf{P}}[\![\Gamma \vdash \lambda x.e : \mathsf{R}_a \to \mathsf{R}_b]\!]^{k<n+1}_{\Theta_k} \triangleq \mathsf{fun\ hist} \to \mathcal{S}_{\mathsf{P}}[\![\Gamma, x : \mathsf{R}_a \vdash e : \mathsf{R}_b]\!]^{k<n+1}_{\Theta_k \uplus \{x \leftarrow \mathsf{hist}\}}$$

$$\begin{aligned}\mathcal{S}_{\mathsf{P}}[\![\Gamma \vdash f\ e : \mathsf{R}_b]\!]^{k<n+1}_{\Theta_k} \triangleq\ & \mathcal{S}_{\mathsf{P}}[\![\Gamma \vdash f : \mathsf{R}_a \to \mathsf{R}_b]\!]^{k<n+1}_{\Theta_k}\ \mathit{hist} \\ & \text{where } \mathit{hist} \equiv [\mathcal{S}_{\mathsf{P}}[\![\widehat{e}]\!]^{k<n+1}_{\Theta_k}; \mathcal{S}_{\mathsf{P}}[\![\widehat{e}]\!]^{k-1<n+1}_{\Theta_{k-1}}; \ldots; \mathcal{S}_{\mathsf{P}}[\![\widehat{e}]\!]^{0<n+1}_{\Theta_0}] \\ & \text{and } \widehat{e} \equiv \Gamma \vdash e : \mathsf{R}_a\end{aligned}$$

$$\mathcal{S}_{\mathsf{P}}[\![\Gamma \vdash c * e : \mathsf{R}_a]\!]^{k<n+1}_{\Theta_k} \triangleq c * \mathcal{S}_{\mathsf{P}}[\![\Gamma \vdash e : \mathsf{R}_a]\!]^{k<n+1}_{\Theta_k}$$

$$\mathcal{S}_{\mathsf{P}}[\![\Gamma \vdash e_1 + e_2 : \mathsf{R}_a]\!]^{k<n+1}_{\Theta_k} \triangleq \mathcal{S}_{\mathsf{P}}[\![\Gamma \vdash e_1 : \mathsf{R}_a]\!]^{k<n+1}_{\Theta_k} + \mathcal{S}_{\mathsf{P}}[\![\Gamma \vdash e_2 : \mathsf{R}_a]\!]^{k<n+1}_{\Theta_k}$$

$$\mathcal{S}_{\mathsf{P}}[\![\Gamma \vdash \pi_i(e) : \mathsf{R}_1]\!]^{k<n+1}_{\Theta_k} \triangleq [\mathsf{nth}\ i\ (\mathcal{S}_{\mathsf{P}}[\![\Gamma \vdash e : \mathsf{R}_a]\!]^{k<n+1}_{\Theta_k})\ 0]$$

$$\mathcal{S}_{\mathsf{P}}[\![\Gamma \vdash (e_1, e_2) : \mathsf{R}_c]\!]^{k<n+1}_{\Theta_k} \triangleq (\mathcal{S}_{\mathsf{P}}[\![\Gamma \vdash e_1 : \mathsf{R}_a]\!]^{k<n+1}_{\Theta_k})\ ++\ (\mathcal{S}_{\mathsf{P}}[\![\Gamma \vdash e_2 : \mathsf{R}_b]\!]^{k<n+1}_{\Theta_k})$$

$$\begin{aligned}\mathcal{S}_{\mathsf{P}}[\![\Gamma \vdash \mathsf{let}\ x = e_1 \mathsf{in}\ e_2 : \mathsf{R}_b]\!]^{k<n+1}_{\Theta_k} \triangleq\ & \mathcal{S}_{\mathsf{P}}[\![\Gamma, x : \mathsf{R}_a \vdash e_2 : \mathsf{R}_b]\!]^{k<n+1}_{\Theta_k \uplus \{x \leftarrow \mathit{hist}\}} \\ & \text{where } \mathit{hist} \equiv [\mathcal{S}_{\mathsf{P}}[\![\widehat{e}]\!]^{k<n+1}_{\Theta_k}; \mathcal{S}_{\mathsf{P}}[\![\widehat{e}]\!]^{k-1<n+1}_{\Theta_{k-1}}; \ldots; \mathcal{S}_{\mathsf{P}}[\![\widehat{e}]\!]^{0<n+1}_{\Theta_0}] \\ & \text{and } \widehat{e} \equiv \Gamma \vdash e_1 : \mathsf{R}_a\end{aligned}$$

$$\begin{aligned}\mathcal{S}_{\mathsf{P}}[\![\Gamma \vdash \mathsf{feed}\ x.e : \mathsf{R}_a]\!]^{k<n+1}_{\Theta_k} \triangleq\ & \mathcal{S}_{\mathsf{P}}[\![\Gamma, x : \mathsf{R}_a \vdash e : \mathsf{R}_a]\!]^{k<n+1}_{\Theta_k \uplus \{x \leftarrow \mathit{hist}\}} \\ & \text{where } \mathit{hist} \equiv [\mathsf{P}[\![\widehat{fb}]\!]^{k-1<n}_{\Theta_{k-1}}; \ldots; \mathsf{P}[\![\widehat{fb}]\!]^{0<n}_{\Theta_0}] \text{ with } \widehat{fb} \equiv \Gamma \vdash \mathsf{feed}\ x.e : \mathsf{R}_a\end{aligned}$$

Figure 5. W single-step semantics of expressions, with $\mathsf{P} : \mathsf{I}_{\{n\}}$

From Section 1.3, we know that a function $f : \mathsf{R} \to \mathsf{R}$ is linear iff $f(x + y) = f(x) + f(y)$. However, this notion is too weak to be of direct use in our programming language context as a) not all expressions are functions, and b) program expressions are indeed open, so the property has to be extended as to be meaningful under an environment Γ.

As is often the case in interactive theorem proving, finding the right theorem and definition statements does amount to the large majority of the work the proof engineer or researcher has to do.

In our case, we will take advantage of the well-known technique of "logical relations": relations over values indexed by program types, and closed over functional values in such a way that functions must send related inputs to related outputs. We first specify what linearity means for values of W types using the ternary, type-indexed relation rel_additive:

```
Fixpoint rel_additive k t {struct t}
  : tyI k t → tyI k t → tyI k t → Prop :=
  match t with
  | tpair m   ⇒ fun x1 x2 x3 ⇒ x1 = x2 − x3
  | tfun  m n ⇒ fun f1 f2 f3 ⇒
      forall (x1 x2 x3 : k.+1.−tuple (tyI k m)),
        x1 = x2 − x3 → f1 x1 = f2 x2 − f3 x3
end.
```

In the above code, we have two cases. For basic values, we just require that the first element of the relation is the subtraction of the third from the second. For functional values, we do require that the functional values behave in a linear way. Note that, in the first versions of this work, we were able to get away with a unary relation, as one can substitute over the equalities; however, we felt that this presentation is more standard and allows us to define the upcoming fundamental lemma in a way more accessible for readers familiar with logical relations.

Note also that the use of the additive property $f(x - y) = f(x) - f(y)$ makes it more convenient to integrate our code with the standard linear algebra libraries of Coq's SSReflect Mathematical Components library, as it captures both linearity and preservation of negation.

The fundamental lemma, rel_additive_fund, does generalize the relation over values to a relation over open expressions:

```
Lemma rel_additive_fund
  Γ t (e : expr Γ t) n (I : I n) (HI : hlinP' I) :
  forall k (hk : k < n.+1) (Θ1 Θ2 Θ3 : env k Γ) ,
    env_additive Θ1 Θ2 Θ3      →
    let: v1 := exprI I hk e Θ1 in
    let: v2 := exprI I hk e Θ2 in
    let: v3 := exprI I hk e Θ3 in
    rel_additive v1 v2 v3.
```

`env_additive` states that $\Theta_1 = \Theta_2 - \Theta_3$, and the HI assumption ensures that the interpretation I is, for the feedback case, linear with respect to environments, which is a sub-case of `rel_additive`, but specialized to non-functional types; this is enough due to our current typing restrictions on the arguments of feedbacks and functions.

As a corollary of the fundamental lemma, we obtain:

```
Corollary funD n k (p : k <= n) a b
  (f : expr [::]  (tfun a b)) :
 let (f : k.+1.-tuple 'cV_a → 'cV_b) := exprIn p f tt in
 additive f.
```

which reads as "given an arbitrary time step n and a well-typed closed function f, the interpretation of f at k is additive".

Note that this process can be repeated for scaling, that is to say $f(cx) = cf(x)$, this time obtaining:

```
Corollary funP n k (p : (k <= n)%nat) a b
   (f : expr [::]  (tfun a b)) : lmorphism (exprIn p f tt).
```

where `lmorphism` is the Coq definition of a "linear morphism", which implies both additivity and scalability.

3 Running Ahead: Imperative Semantics

Up to this point, we have used the formal semantics in Figure 5 to reason about W programs and their properties; while this is nice, and it works reasonably well, we are still far from something usable as a programming framework.

Indeed, the semantics of Section 2.3 recompute the full time history of arguments at every function application node. This is simple to understand mathematically, and the locality (or referential transparency) of the definitions helps in formal reasoning, but it is not going to work in actual uses, as this argument re-computation is highly impractical and requires an unbounded amount of memory.

In this section, we focus on the concrete case when access to variables' history is bounded. This way, we can allocate a buffer at every application point — and feedback, of course — that will store the argument's past values up to the inferred bound in the buffer to avoid recomputing.

We claim that this produces a reasonable execution model, and will proceed to experimentally benchmark our programs to show so.

$$
\begin{array}{rcll}
\tau & ::= & \cdots \mid I_n & \\
e, f & ::= & x \qquad x \in \mathbb{V} & \\
& \mid & \cdots & \\
& \mid & e[e_{\text{idx}}] & \\
& \mid & f\ [\text{ptr} \leftarrowtail e] & \text{ptr} \in \textbf{Ptr} \\
& \mid & [\text{ptr} \leftarrowtail \text{feed}\ x.\ e] & \text{ptr} \in \textbf{Ptr}
\end{array}
$$

Figure 6. Updated W syntax and types, with buffer allocation and array indexes

3.1 Language Extensions for Allocations and Variables-as-Arrays

Before moving ahead with the imperative semantics, we slightly tweak and extend the syntax from Figure 1 to provide a more comfortable programming experience, at the cost of adding non-linear constructions; in particular:

- variables now lose the subindex and denote arrays – we also introduce a type of ordinals I_n (integers less than n), and a safe array-access operator $x[i]$;
- feedback and application expressions now store a pointer to a heap-allocated circular buffer, where past values for the history-tracking expressions are stored.

Figure 6 shows the new indexing operator $e[e_{\text{idx}}]$, with typing rule $[] : \mathsf{R}_n \to I_n \to \mathsf{R}$, and the extra pointer in both application and feedback nodes. We write W_{ptr} for this updated version when required.

Bounded Access. The new syntax implements bounded access by default, as the type I_n is only inhabited by integers $\{0, \ldots, n-1\}$; however, on the raw machine we will present soon, type information is erased and indeed an out-of-bounds access will lead to undefined behavior. This is just a particular design choice, but any method that provides, for a functional expression e, the bounds on access to their argument would work for us.

The idea of using bounded access to improve our machine can be understood by looking at Figure 7. There, we can see how the bound allows us to avoid allocating a new memory cell for the value of the argument in the $n+1$ step, and we can instead use a standard circular buffer.

Note that this goes beyond memoization (which is anyhow impractical for DSP), as memoization does avoid the multiple computations of all the previous values for arguments of functions, but allocation is still necessary for the new ones.

Allocation and Heaps. Once we have a bound for our functional terms, we have to allocate a buffer in the heap and update the pointer for each expression. Note that it is crucial that buffers are not overlapping obviously, and initialized to zero, as is common in DSP (though we could provide pragmas for different initialization setups, usually from some probability distribution, which is a common pattern for physically-based sound processing).

$$
\begin{array}{llll}
[\![f\ a]\!]^n & = [\![f]\!]^n & [:: & [\![a]\!]^n; \ldots; [\![a]\!]^{n-d+1}; [\![a]\!]^{n-d}; \ldots; [\![a]\!]^0 \quad] \\
[\![f\ a]\!]^{n+1} & = [\![f]\!]^{n+1} & [:: [\![a]\!]^{n+1}; & [\![a]\!]^n; \ldots; [\![a]\!]^{n-d+1}; [\![a]\!]^{n-d}; \ldots; [\![a]\!]^0 \quad] \qquad \text{unbounded history access} \\
\\
[\![f\ a]\!]^n & = [\![f]\!]^n & [:: & [\![a]\!]^n; \ldots; [\![a]\!]^{n-d+1}; [\![a]\!]^{n-d} \quad] \\
[\![f\ a]\!]^{n+1} & = [\![f]\!]^{n+1} & [:: [\![a]\!]^{n+1}; & [\![a]\!]^n; \ldots; [\![a]\!]^{n-d+1} \quad] \qquad \text{bounded access, fixed-size circular buffer}
\end{array}
$$

Figure 7. History dynamics ($[\![e]\!]^n$ is, informally, the value of expression e at time step n; [:: ...; ...] is a list)

An important remark to keep in mind regarding allocation of history buffers is that the following code has to work fine. Assume fir : $\mathsf{R}_n \longrightarrow \mathsf{R}_m$ and $x, y : \mathsf{R}_n$, then

$$(\mathsf{fir}\ x, \mathsf{fir}\ y)$$

has to work properly, while sharing the implementation of fir; the above expression will allocate two buffers. Other languages do actually unfold fir and insert the buffers at delay time; however this creates cache locality and code size issues. Moreover, it makes it hard to remain in a call-by-value setting. We tried some different approaches to this problem, which took us quite a while, but have settled with this solution for now.

3.2 An OCaml-Based Interpreter

After the introduction of buffers to store the history of function arguments, we can define an interpreter or compilation scheme for W_{ptr} that should be capable of real-time DSP. Even if the history space and time consumption problem has been solved by the use of buffers, our language still features first-class functions, and thus in particular closures; thus the compilation scheme is not trivial, and we must eliminate or manage allocations introduced by λ-expressions, e.g., in $(\lambda x.x_0 + x_1)\ e$.

We thus face the tension between the need for rapid prototyping and experimentation, and the need for an optimizing compiler that can produce code competing with hand-written DSP routines.

We have found a middle ground by taking advantage of the fact that the only overhead in the imperative version of the semantics is interpretation and closure allocation. We can put to use three key components to write a toy W_{ptr} compiler producing performant, allocation-free code:

1. first, we define an OCaml interpreter for W_{ptr} (Figure 8) that takes advantage of Generalized Algebraic Data Types (GADTs) to represent W_{ptr} closures as OCaml closures – this step is critical, as we will see;
2. second, we instrument our W_{ptr} interpreter using MetaOCaml – this staged interpreter can take any W_{ptr} program as input and produce a specialized version of the interpreter for the particular input program;
3. last, we use the optimizing FLambda OCaml compiler, which is able to compile the code generated in the previous steps, eliminating all closures, and producing pretty fast machine code, allocation-free in all our examples.

We show the "denotational" and imperative interpreters in Figure 8 side-by-side, to help the reader spot the differences. The denotational version is a straight lifting of the interpretation function $\mathcal{S}_{\mathsf{P}}[\![\,]\!]$ defined above in Coq to OCaml.

The most interesting case for the imperative interpreter is the application case. As the reader can see, we first compute the value for the argument at the current time, then we push it to the buffer, passing the pointer to the interpretation of the function, which is an OCaml function thanks to our use of GADTs. The feedback node follows a similar scheme.

Once we have the interpreter, we proceed to stage it in a standard way — as shown in Figure 9 — so we can produce specialized code for each W_{ptr} input program. The generated code contains closures, but for our examples (see below), all of them will be eliminated by the OCaml optimizing compiler.

3.3 Experimental Evaluation

We perform an informal experimental evaluation of our compilation scheme. In particular, we compare 3 different implementations — imperative interpreter "basic", staged interpreter "gen", hand-made code "hm" — of 3 examples:

- fir, a simple finite-impulse response filter of 15th order;
- iir, a simple infinite impulse filter of 15th order;
- com, the composition of fir and iir.

Results can be seen in Table 1. The time column denotes time per run of 10 samples. The allocation columns "mWd" and "mjWd" refer to specifics of the OCaml's garbage collector, but we can interpret them as "minor" and "major" allocations, with the latter being very expensive, and the former being a lesser worry but still a cost in terms of CPU instructions.

The "basic" version provides a reasonable baseline. However we allocate a large amount of memory on each cycle

```
let rec machine_n :
  type a. int → a expr → env → a list =
  fun step e env →
    if step < 0 then []
    else machine step e env :: machine_n (step-1) e env
and machine : type a . int → a expr → env → a =
  fun step e env → match e with
  | Var (idx, id)   →
    lookup env id idx
  | Lam e →
    fun hist → machine step e (hist :: env)
  | App (ef, ea) →
    let a_hist = machine_n step ea env in
    machine step ef env a_hist
  | Feed e →
    let e_hist = machine_n (step-1) (Feed e) env in
    machine step e (e_hist::env)
  |...
```

```
let heap = ... (* pre-allocated *)
let rec imachine : expr → env → value =
  fun e env → match e with
  | Var (idx, id)   →
    let p = ptr_of id env in
    lookup heap p idx
  | Lam e →
    fun ptr → imachine e (ptr :: env)
  | App (p,ef,ea)  →
    let va = imachine ea env in
    shift_one heap ptr va;
    imachine ef env ptr
  | Feed (ptr,e)   →
    let v = imachine e (ptr::env) in
    shift_one heap ptr v;
    v
  |...

let rec repeat : int → (unit → 'a) → 'a = fun n c →
  if n = 0 then c ()
  else (c (); repeat (n-1) c)

let eval e env n = repeat n (fun () → imachine e env)
```

Figure 8. Denotational vs. imperative interpretations

Figure 9. Staged Interpreter, Selected Cases

```
let rec eval : type a. env → a expr → a code =
  fun g e → match e with
  | Cst r →
    let module M = Lift_array(Lift_float) in M.lift r
  | Var id →
    .< Array.(unsafe_get .~(heap) .~(List.nth g id)) >.
  | Idx (vec, id) →
    .< let v = .~(eval g vec) in
       let i = .~(eval g id) in
       Array.(unsafe_get v i) >.
  | Add (e1,e2) →
    .< let v1 = .~(eval g e1) in
       let v2 = .~(eval g e2) in
       WArray.map2 (+.) v1 v2 >.
  | Lam e →
    .<fun p → .~(eval (.<p>.::g) e)>.
  | App (p, f, e) →
    .< let v = .~(eval (.<p>.::g) e) in
       shift_one v Array.(unsafe_get .~(heap) p);
       .~(eval g f) p >.
  |...
```

for the composed example due to closures, plus the interpretation overhead; at some point we even hit the major heap, which is costly.

The generated code performs much better, surpassing the hand-written code in one instance. Keeping into account that the generated code was not optimized in any other way, we think the imperative execution model performs according to our goals, and that it has been validated in terms of potential real-time performance. Thus, writing a more refined compilation back-end seems to be worth it.

Table 1. Time (per sample) and memory allocation (per run) for each test; percentages are time with respect to the slowest case (com basic)

Name	Time (ns)	mWd (w)	mjWd (w)	Percentage
fir basic	4,982.48	789.01	0.11	47.54%
fir gen	188.42	21.00		1.80%
fir hm	85.21	2.00		0.81%
iir basic	4,765.38	701.01		45.47%
iir gen	184.08	43.00		1.76%
iir hm	212.46	2.00		2.03%
com basic	10,479.81	1,522.02	0.31	100.00%
com gen	394.78	83.00		3.77%
com hm	287.33	46.00		2.74%

Note that we wrote the hand-made code in OCaml, but we inspected the generated assembly closely, and indeed it seems to be competitive with C code. We did some more informal comparison of tuned C code generated from Faust vs. our hand-made setup, and while OCaml performs a bit worse than `gcc`, it is still below a 2x-overhead in the worst case. We believe our hand-made baseline to be a good reference point for now.

To compile the code, we used OCaml's Flambda compiler, version 4.07.1, with the `-O3 - unbox-closures` options. For MetaOCaml, we used the 4.07.1+BER version, and the `core-bench` benchmarking framework generously open sourced by Jane Street.

3.4 Equivalence of Semantics

An open question at this point is whether one can show that the semantics of the denotational interpreter is equivalent to the semantics of the imperative version with buffers. The proof here becomes a bit more complex, and in this paper we just present a sketch of the idea and leave the fully formalized proof for future work.

The difficulties here are standard, and have been discussed in the context of verified compilation of synchronous programming languages [15]. Concretely, we may try to relate both interpreters using a logical or simulation relation, but in this case the types are not telling the whole story. For an expression $\Gamma \vdash e : \tau$, in the denotational case, Γ-compatible environments Θ will contain the full story for the binders, while in the imperative case, all they contain is a pointer on the buffer; so the buffer invariant cannot be easily stated.

There likely exist many possibilities to overcome this, but we intend to follow an idea similar to the one in Velus [15] and extend our definition of well-typed judgements to expose a *heap shape* Δ, so that $\Delta \mid \Gamma \vdash e : \tau$ denotes a well-typed expression, and pointers in application nodes refer to a unique location in the heap. This way we expect to be able to define a predicate relating heaps plus imperative environments to denotational environments, and prove that the one-step evaluation preserves this relation, thus making induction on the number of steps work.

4 Related Work

Domain-Specific Languages for DSP. The work presented here lies at the intersection of several domains, our influences and related work are quite varied. We briefly survey related work, classified in a non-mutually exclusive way.

W is directly inspired by the Faust programming language [48], and originally arose as a method to study / provide Faust with a mechanically specified operational semantics, including multi-rate [7, 33, 49]. Faust has a clear denotational model, but the particular implementation strategy is left open so its cost semantics will vary depending on the chosen compilation trade-offs. Moreover, Faust point-free presentation is harder to connect with standard programming languages techniques such as logical relations, which rely on a typical "functional calculus" presentation. Picking a particular evaluation order in *W* allowed us to reason about buffers / arrays, which is very convenient in DSP, so we ended up adding them as a first-order primitive for programmers' use. Another key goal of this work was to provide a semantics in which to interpret (future) multi-rate programs with a reasonable cost model (see Section 5 for more details).

A language that feels very similar in syntax to *W* is Arrp [37, 38], an array-based language. Arrp uses optimizing polyhedral compilation to produce very efficient code, while at the same time providing the programmer with more flexibility than *W* in terms of array manipulation. *W* exposes arrays, but only as primitive data-types, so at the moment array operations are compiled directly to the underlying array operations in the semantics. We haven't explored what kind of optimizations would be possible in this front.

Feldspar [5] is a DSL embedded in Haskell that can produce efficient code for DSP algorithms using a custom compilation strategy. CSound and Supercollider are music-processing languages that include DSP capabilities, usually based on the notion of UGens, custom C routines that process audio on a callback basis. Max/MSP and PureData are graphical programming languages allowing the definition of signal dataflow processing diagrams that are then interpreted. Some more recent languages aimed at music DSP are also Vult and Soul. All these languages were, in one way or another, considered during the design of the *W* framework, in particular when thinking about the requirements that the *W* calculus had to fulfil in order to ultimately become a possible compilation target for them, and thus deserve to be mentioned here. We also refer the reader to the survey on [6], which compares the different expressivities of music-oriented DSP languages.

Synchronous Programming. There exists a long tradition of research studying systems for real- and constrained-time programming, among which one of the most successful is the *Synchronous Paradigm*, in which programs are assumed to react instantaneously to changes of input, which are driven by some specific *temporal scale*.

Arising from the study of Kahn Proccess Networks [34], the field of synchronous programming has produced very important theoretical and practical contributions. As a very incomplete list of synchronous languages, we could mention Lustre [22], Esterel [10, 11], Signal [9, 28], Lucid Synchrone [21, 23], and Zelus [16, 17].

Particularly recent relevant work to us is the research on time refinement [42], causality analysis [8], constructive semantics [44], and the work of Adrien Guatto [30]. Indeed, Guatto's work is quite close in spirit to our proposal, as he introduces a synchronous language with integer clocks, while most of his metatheory is based on techniques developed in the functional programming community, such as step indexing [3, 12, 46]. A linear-based type system ensures that all well-typed programs are safe and can be compiled to finite circuits. *W* can be considered a simple synchronous language, with the main constraint that no clock information is required at run time; moreover *W* programs have a unique

typing derivation and the interpretation of programs doesn't depend on the typing derivation tree.

Data-Flow Programming. *Data-flow Programming* is particularly well suited to signal processing, and data-flow diagrams are commonly used to write and specify algorithms. Data-flow systems are typically characterized by their scheduling power, from synchronous data-flow [39] to more complex but less performant proposals that admit no static or cyclic schedules. A few remarkable systems that we looked at during W's design phase are Ptolemy [20, 50], StreamIt [58], and SDF3 [57].

Functional Reactive Programming. Arrows and Functional Reactive Programming [25, 31, 32, 40, 41, 47] provide an elegant framework to express many interesting programming patterns based on time. A more recent development [36] extends FRP to provide a leak-free full formal semantics; we have found a great deal of inspiration on this effort. Co-effects type systems [19, 26] provide a type-based method to quantify resource and buffering dependencies. A very relevant work on completeness of stream fusion is [35].

Formal Verification of DSP and Cyber-Physical Systems. An early effort for the deductive verification of a synchronous system is found in [13]. More recently, Cédric Auger's PhD thesis [4] made significant progress; it has been recently put to work to deliver the first certified Lustre compiler [14, 18], which ensures the correctness of the compilation of Lustre programs with respect to a denotational model. The VERIDRONE project [52] aims to verify the control engine of a drone with respect to safety conditions such as zoning. There exists a long line of work on the verification of mathematical properties for DSP by Tahar et al. [1, 2, 45, 53, 56], mainly using the Isabelle theorem prover.

Closest to our attempt to verify the imperative heap allocation pass is the work on the Velus verified compiler, as well as the techniques required in order to manage memory local to nodes.

5 Future Work

Throughout the paper we have already hinted at possible future work directions; in particular, we can distinguish 4 concrete lines of work that we are pursuing.

Multirate Processing. A key design goal of W was to accommodate multirate processing from the start. While the work presented here exclusively deals with the single-rate case, we have already developed a simple version of periodic regular clocks that allows to implement several multi-rate primitives in a simple way. In this setup, stream types are annotated with a rate parameter, so that R, representing a stream of samples in our setup, becomes R@1. Additional subtyping rules such as

$$\frac{\Gamma\ x = \mathsf{R}_a@r}{\Gamma \vdash x : \mathsf{R}_a[r]} \qquad \frac{\Gamma\ x = \mathsf{R}_a@r \qquad x \text{ is } k\text{-bounded}}{\Gamma \vdash x : \mathsf{R}_a[rk]}$$

implement conversion from rates to arrays (of type $\mathsf{R}_a[r]$) so they can be manipulated; the left side is the simple version, whereas the right version does allow "flattening" of a multi-rate stream that has been bounded up to k steps into an array. The introduction of stream rates now requires an *emit* operation, which we write $\{e_1, \ldots, e_n\}$ for emitting n elements at one concrete time step. In this discipline, we can now write several code examples (we note $[e_1, ..., e_n]$ an array constant with values e_i):

$$\begin{array}{ll} \mathsf{up} : \mathsf{R@1} \longrightarrow \mathsf{R@2} & \mathsf{down} : \mathsf{R@2} \longrightarrow \mathsf{R@1} \\ \mathsf{up} = \lambda x.\ \{x, 0\} & \mathsf{down} = \lambda x.\ \{x[0]\} \\ \\ \mathsf{pack} : \mathsf{R@2} \longrightarrow \mathsf{R}_2\mathsf{@1} & \mathsf{unpack} : \mathsf{R}_2\mathsf{@1} \longrightarrow \mathsf{R@2} \\ \mathsf{pack} = \lambda x.\{[x[0], x[1]]\} & \mathsf{unpack} = \lambda x.\{x[0], x[1]\} \end{array}$$

$$\begin{array}{l} \mathsf{FFT_w} : \mathsf{R@16} \longrightarrow \mathsf{R[128]@1} \\ \mathsf{FFT_w} = \lambda x.\ \{\mathsf{FFT}\ x\} \end{array}$$

The first 4 examples perform straightforward rate conversion, with the particularity that both pack and unpack are semantically the identity function after type erasure, as the underlying pointers are the same. A more interesting example is the last one, windowed FFT. In this case, we assume a primitive $\mathsf{FFT} : \mathsf{R[128]} \longrightarrow \mathsf{R[128]}$; then, the subtyping rule will infer a bound for the input stream $k = 112$, so x will be given type R[128], with a buffer of that particular size, which gets shifted by 16 samples at each time step. Many other interesting examples are possible in this framework, such as filters with different control and data rates. Note that this discipline leads naturally to the introduction of two function types: a regular functional type for primitives not aware of streams (such as FFT above), and a stream processor type, where the arguments must be streams with rates.

Front End and Type Inference. As the reader may have noticed, the core calculus presented in this paper is not enough to compile programs, as we still need to compute access bounds on variables as to properly size the required buffers in the compilation scheme. To fill that gap, we have developed an extension of the type system to track variable usage using co-effects (see for example [19]). The idea is that every variable in the environment gets an annotation that reflects how many of their past values were used. This information is also carried over functional types, so we can then know a bound on the history access of each stream processor by just looking at their type. Note that typing rules now crucially rely on the particular evaluation strategy! In our case, we believe our current call-by-value discipline provides by far the better adapted scenario, but other approaches are

possible. The basic rules are now:

$$\frac{(x :_k \mathrm{R}_a) \in \Gamma}{\Gamma \vdash x : \mathrm{R}_a[k]} \qquad \frac{\Gamma, x :_k \tau \vdash e : \sigma}{\Gamma \vdash \lambda x.\, e : \tau \to_k \sigma}$$

so, for example, the FFT example becomes:

$$\mathsf{FFT}_\mathrm{w} \;:\; \mathrm{R}@16 \to_{112} \mathrm{R}[128]@1$$
$$\mathsf{FFT}_\mathrm{w} = \lambda x.\, \{\mathsf{FFT}\; x\}$$

We have implemented a full front end for this system, named "Wagner", which includes a parser and a bidirectional type-checker; the above approach seems to work fine in practice, with bidirectional type propagation being a key point in order to correctly compute bounds. We took many hints regarding type information dataflow from the approach implemented by the Mathematical Components library [29] in their matrix implementation.

Formally Verified Imperative Semantics. Work is underway to implement the strategy outlined in Section 3.4 as a formally verified Coq proof. In particular, we have defined the imperative machine inside Coq using a heap monad (a state monad where the state is the global heap), defined a typing relation for allocated programs that implies pointer independence, and are in the process of coding down the corresponding soundness relations. Additionally, we could consider implementing a program logic in the style of [27].

More Principled Code Generation. Additionally, more work in the back end is planned. The current solution using MetaOCaml was quick to implement and provided good results; however it is not practical as a daily driver for DSP development, where the compiler is expected to target low-level languages such as C or Rust. We thus plan to implement program generation targeting a simple imperative C-like format by specialization + defunctionalization [24, 51], which should suffice to generate reusable code.

The above are the main items in our work plan. However a few other more speculative possibilities do include:

a) higher-order streams are in general very difficult to implement efficiently (see for example [30]) and likely not very useful for our intended application. In the case of W, adding a *shift* index to closures would suffice to translate relative access to the environment, but would likely incur non-acceptable run-time costs;
b) dynamic dataflow graphs, since, so far, our calculus has a fixed-dataflow structure; it could be interesting to allow some dynamism here, but the way to do it is not yet clear;
c) modules, modular compilation and ML-style interfaces will prove quite useful in the construction of DSP systems and libraries, as well as to better structure components with a complex control and data input-output structure.

6 Conclusion

We have presented the W-calculus, a formal calculus for the encoding of real-time digital signal processors, and provided its mechanized denotational semantics and a mechanized proof that every program written in the calculus is linear, using logical relations.

Additionally, we have developed an efficient imperative interpreter, and performed an experimental evaluation of some selected examples using program specialization to validate our approach.

We believe that the presented work constitutes a good basis upon which to (1) keep studying formally verified semantics and properties of the language and programs written in it and (2) evolve towards a usable high-level DSP language that can be used to quickly and easily define efficient processing components.

Acknowledgments

This work has been supported by the ANR FEEVER project (ANR-13-BS02-0008). We thank Yann Orlarey, Adrien Guatto, and Timothy Bourke for interesting discussions and ideas.

References

[1] Behzad Akbarpour and Sofiène Tahar. 2004. A Methodology for the Formal Verification of FFT Algorithms in HOL. In *FMCAD 2004*. https://doi.org/10.1007/978-3-540-30494-4_4
[2] Behzad Akbarpour and Sofiène Tahar. 2006. An approach for the formal verification of DSP designs using Theorem proving. *IEEE Trans. on CAD of Integrated Circuits and Systems*. https://doi.org/10.1109/TCAD.2005.857314
[3] Andrew W. Appel and David A. McAllester. 2001. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*. https://doi.org/10.1145/504709.504712
[4] Cédric Auger. 2013. *Compilation Certifiée de SCADE/LUSTRE*. Thèse de Doctorat. Université Paris-Sud. http://tel.archives-ouvertes.fr/tel-00818169/.
[5] Emil Axelsson, Koen Claessen, Mary Sheeran, Josef Svenningsson, David Engdal, and Anders Persson. 2011. *The Design and Implementation of Feldspar*. Springer Berlin Heidelberg, Berlin, Heidelberg, 121–136. https://doi.org/10.1007/978-3-642-24276-2_8
[6] Karim Barkati and Pierre Jouvelot. 2013. Synchronous programming in audio processing: A lookup table oscillator case study. *ACM Comput. Surv.*. https://doi.org/10.1145/2543581.2543591
[7] Karim Barkati, Haisheng Wang, and Pierre Jouvelot. 2014. Faustine: A Vector Faust Interpreter Test Bed for Multimedia Signal Processing — System Description. In *FLOPS 2014*.
[8] Albert Benveniste, Timothy Bourke, Benoît Caillaud, Bruno Pagano, and Marc Pouzet. 2014. A type-based analysis of causality loops in hybrid systems modelers. In *HSCC'14*. https://doi.org/10.1145/2562059.2562125
[9] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. 1991. Synchronous Programming with Events and Relations: the SIGNAL Language and Its Semantics. *Sci. Comput. Program.*. https://doi.org/10.1016/0167-6423(91)90001-E
[10] Gérard Berry. 2000. The foundations of Esterel. In *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, Gordon D. Plotkin, Colin Stirling, and Mads Tofte (Eds.). The MIT Press, 425–454.
[11] Gérard Berry and Georges Gonthier. 1992. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Sci. Comput. Program.* 19, 2 (1992), 87–152. https://doi.org/10.1016/0167-6423(92)90005-V
[12] Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. 2012. First steps in synthetic guarded domain theory:

step-indexing in the topos of trees. *Logical Methods in Computer Science* 8, 4 (2012). https://doi.org/10.2168/LMCS-8(4:1)2012

[13] Sylvain Boulmé and Grégoire Hamon. 2001. Certifying Synchrony for Free. In *LPAR 2001*. https://doi.org/10.1007/3-540-45653-8_34

[14] Timothy Bourke, Lélio Brun, Pierre-Evariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. 2017. A Formally Verified Compiler for Lustre. In *PLDI 2017*.

[15] Timothy Bourke, Lélio Brun, and Marc Pouzet. 2020. Mechanized semantics and verified compilation for a dataflow synchronous language with reset. In *POPL 2020*. https://doi.org/10.1145/3371112

[16] Timothy Bourke, Jean-Louis Colaço, Bruno Pagano, Cédric Pasteur, and Marc Pouzet. 2015. A Synchronous-Based Code Generator for Explicit Hybrid Systems Languages. In *CC 2015*. https://doi.org/10.1007/978-3-662-46663-6_4

[17] Timothy Bourke and Marc Pouzet. 2013. Zélus: a synchronous language with ODEs. In *HSCC 2013*. https://doi.org/10.1145/2461328.2461348

[18] Timothy Bourke, Pierre Évariste Dagand, Marc Pouzet, and Lionel Rieg. 2017. Verifying clock-directed modular code generation for Lustre. In *JFLA'17*.

[19] Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. 2014. A Core Quantitative Coeffect Calculus. In *Programming Languages and Systems*. https://doi.org/10.1007/978-3-642-54833-8_19

[20] Joseph T. Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. 1994. Ptolemy: A Framework for Simulating and Prototyping Heterogenous Systems. *Int. Journal in Computer Simulation*.

[21] Paul Caspi, Grégoire Hamon, and Marc Pouzet. 2008. Synchronous functional programming: The Lucid Synchrone experiment. *Real-Time Systems: Description and Verification Techniques: Theory and Tools.*.

[22] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. 1987. Lustre: A Declarative Language for Programming Synchronous Systems. In *POPL 1987*. https://doi.org/10.1145/41625.41641

[23] Paul Caspi and Marc Pouzet. 1996. Synchronous Kahn Networks. In *ICFP '96*. https://doi.org/10.1145/232627.232651

[24] Olivier Danvy. 2008. Defunctionalized interpreters for programming languages. In *ICFP 2008*. https://doi.org/10.1145/1411204.1411206

[25] Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *ICFP '97*. https://doi.org/10.1145/258948.258973

[26] Marco Gaboardi, Shin-ya Katsumata, Dominic A. Orchard, Flavien Breuvart, and Tarmo Uustalu. 2016. Combining effects and coeffects via grading. In *ICFP 2016*. https://doi.org/10.1145/2951913.2951939

[27] Emilio Jesús Gallego Arias, Pierre Jouvelot, and Olivier Hermant. 2015. A Taste of Sound Reasoning in Faust. In *Proceedings of the 13th Linux Audio Conference*. https://github.com/ejgallego/mini-faust-coq

[28] Abdoulaye Gamatié, Thierry Gautier, Paul Le Guernic, and Jean-Pierre Talpin. 2007. Polychronous design of embedded real-time applications. *ACM Trans. Softw. Eng. Methodol.*. https://doi.org/10.1145/1217295.1217298

[29] Georges Gonthier. 2011. Point-Free, Set-Free Concrete Linear Algebra. In *ITP 2011*. https://doi.org/10.1007/978-3-642-22863-6_10

[30] Adrien Guatto. 2016. *A Synchronous Functional Language with Integer Clocks*. Ph.D. Dissertation. École Normale Supérieure.

[31] John Hughes. 2000. Generalising monads to arrows. *Sci. Comput. Program.*. https://doi.org/10.1016/S0167-6423(99)00023-4

[32] John Hughes. 2004. Programming with Arrows. In *Advanced Functional Programming, 5th International School, AFP 2004*. https://doi.org/10.1007/11546382_2

[33] Pierre Jouvelot and Yann Orlarey. 2011. Dependent vector types for data structuring in multirate Faust. *Computer Languages, Systems & Structures*. https://doi.org/10.1016/j.cl.2011.03.001

[34] Gilles Kahn. 1974. The Semantics of Simple Language for Parallel Programming. In *IFIP Congress*.

[35] Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. 2017. Stream Fusion, to Completeness. In *POPL 2017*. https://doi.org/10.1145/3009837.3009880

[36] Neelakantan R. Krishnaswami. 2013. Higher-order functional reactive programming without spacetime leaks. In *ICFP'13*. https://doi.org/10.1145/2500365.2500588

[37] Jakob Leben. 2016. Arrp: a functional language with multi-dimensional signals and recurrence equations. In *FARM@ICFP 2016*. https://doi.org/10.1145/2975980.2975983

[38] Jakob Leben and George Tzanetakis. 2019. Polyhedral Compilation for Multi-dimensional Stream Processing. *ACM Trans. Archit. Code Optim.* https://doi.org/10.1145/3330999

[39] Edward A. Lee and David G. Messerschmitt. 1987. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Trans. Computers*. https://doi.org/10.1109/TC.1987.5009446

[40] Hai Liu, Eric Cheng, and Paul Hudak. 2009. Causal commutative arrows and their optimization. In *ICFP 2009*. https://doi.org/10.1145/1596550.1596559

[41] Hai Liu, Eric Cheng, and Paul Hudak. 2011. Causal commutative arrows. *J. Funct. Program.*. https://doi.org/10.1017/S0956796811000153

[42] Louis Mandel, Cédric Pasteur, and Marc Pouzet. 2015. Time refinement in a functional synchronous language. *Sci. Comput. Program.*. https://doi.org/10.1016/j.scico.2015.07.002

[43] Louis Mandel, Florence Plateau, and Marc Pouzet. 2010. Lucy-n: a n-Synchronous Extension of Lustre. In *MPC 2010*. https://doi.org/10.1007/978-3-642-13321-3_17

[44] Michael Mendler, Thomas R. Shiple, and Gérard Berry. Constructive Boolean circuits and the exactness of timed ternary simulation. *Formal Methods in System Design*. https://doi.org/10.1007/s10703-012-0144-6

[45] Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar (Eds.). 2008. *TPHOLs 2008*.

[46] Hiroshi Nakano. 2000. A Modality for Recursion. In *LICS 2000*. https://doi.org/10.1109/LICS.2000.855774

[47] Henrik Nilsson, John Peterson, and Paul Hudak. 2003. Functional Hybrid Modeling. In *PADL 2003*. https://doi.org/10.1007/3-540-36388-2_25

[48] Yann Orlarey, Dominique Fober, and Stephane Letz. 2004. Syntactical and semantical aspects of Faust. *Soft Comput.*. https://doi.org/10.1007/s00500-004-0388-1

[49] Yann Orlarey and Pierre Jouvelot. 2016. Signal Rate Inference for Multidimensional Faust. In *IFL 2016*. https://doi.org/10.1145/3064899.3064902

[50] José Luis Pino, Soonhoi Ha, Edward A Lee, and Joseph T Buck. 1995. Software synthesis for DSP using Ptolemy.

[51] John C. Reynolds. 1998. Definitional Interpreters for Higher-Order Programming Languages. *High. Order Symb. Comput.*. https://doi.org/10.1023/A:1010027404223

[52] Daniel Ricketts, Gregory Malecha, and Sorin Lerner. 2016. Modular Deductive Verification of Sampled-Data Systems. In *EMSOFT*.

[53] Umair Siddique, Mohamed Yousri Mahmoud, and Sofiène Tahar. 2014. On the Formalization of Z-Transform in HOL. In *ITP 2014*. https://doi.org/10.1007/978-3-319-08970-6_31

[54] Julius Orion Smith III. 2007. *Introduction to Digital Filters: with Audio Applications*. W3K Publishing. https://ccrma.stanford.edu/~jos/filters/

[55] Julius Orion Smith III. 2007. *Mathematics of the Discrete Fourier Transform (DFT): with Audio Applications* (2nd ed.). W3K Publishing. https://ccrma.stanford.edu/~jos/mdft/

[56] Anis Souari, Amjad Gawanmeh, Sofiène Tahar, and Mohamed Lassaad Ammari. Design and verification of a frequency domain equalizer. *Microelectronics Journal* (2014). https://doi.org/10.1016/j.mejo.2013.10.012

[57] Sander Stuijk, Marc Geilen, and Twan Basten. 2006. SDF3: SDF For Free.. In *ACSD*.

[58] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. 2002. StreamIt: A Language for Streaming Applications. In *CC 2002*. https://doi.org/10.1007/3-540-45937-5_14