Higher-Order Causal Stream Functions in Sig from First Principles

Baltasar Trancón y Widemann [1,2] Markus Lepper [2]

[1] Technische Universität Ilmenau [2] semantics GmbH Berlin

Abstract

The Sig programming language is a total functional, clocked synchronous data-flow language. Its core has been designed to admit concise coalgebraic semantics. Universal coalgebra is an expressive theoretical framework for behavioral semantics, but traditionally phrased in abstract categorical language, and generally considered inaccessible. In the present paper, we rephrase the coalgebraic concepts relevant for the Sig language semantics in basic mathematical notation. We demonstrate how the language features characteristic of its paradigms, namely sequential and parallel *composition* for applicative style, *delay* for data flow, and *apply* for higher-order functional programming, are shaped naturally by the semantic structure. Thus the present paper serves two purposes, as a gentle, self-contained and applied introduction to coalgebraic semantics, and as an explication of the Sig core language denotational and operational design.

1 Introduction

Streams are a ubiquitous fundamental data structure, even more so in the realms of reactive and data flow-oriented computation. Semantic foundations in terms of universal coalgebra and coinduction [14] have been proposed almost twenty years ago [3, 15], but remain woefully obscure to the general programming community.

Here we set out to exploit the recent development of the Sig language as a use case for coalgebraic semantics. Unlike for previous technical papers, the priority here is not on the exposition of novel theoretical foundations or practical implementations, but on the clarification and justification of already established solutions. As such it is intended for a broader audience with a general background in programming languages. Consequently, the present paper is partly a self-contained educational summary of pre-existing isolated parts of technical work, and partly a report on novel work to connect the dots. In particular, there has been a considerable rhetorical gap between the theoretical framework [17] and the implementation strategy employed by the actual compiler [18]. While the design appears fairly obvious and grounded in standard compiler constructor wisdom, a rigorous reconstruction by calculation from first principles would obviously be more satisfactory. Notable novel contributions are:

- pseudorandom number generators as illuminating examples of coinductive stream computations (2.8), (5.6);
- a method to derive operational facts from abstract coalgebraic specifications (3.12) that qualifies as a basic form of coinductive program synthesis;
- its application to a number of basic concepts of functional stream programming, resulting in rigorous justification of the plausible implementation strategy by calculation (4.6), (4.7), (5.3), (6.3).

We introduce concepts of universal coalgebra as we go. Statements that have been specialized for the present discussion are marked with asterisks. For a more comprehensive exposition, see [14]. The material is organized into progressive sections which discuss streams, stream transformers, composition of stream transformers, delay, and higher order function application, respectively. Section 2 also summarizes the basic working tools of the colgebraic framework. These are completely standard as in [14], but contrarily to present custom, we spell them out in non-categorical language, and add novel and domain-specific examples. By analogy, the content of sections 3 and 4 is rephrased from [14] and our own theoretical work [17].

Since the present discussion is focused on calculation of operational semantics and space is limited, the front-end features and usage of the SIG language will be mentioned only in passing. We point to our recent work [18, 19] for detailed discussions.

2 Zeroth Order: Streams

Definition* 2.1 (Functor)**.** A *functor*[1] F is a pair of homonymous constructions: One that assigns to each set X a set $F(X)$, and another that assigns to each map $f : X \to Y$ a map $F(f) : F(X) \to F(Y)$.[2]

Definition 2.2 (Stream Functor)**.** Fix some nonempty set A and consider the Cartesian pairing operator $\mathcal{S}_A(X) = A \times X$. It can be extended to a functor, acting on maps as:

$$\mathcal{S}_A(f)(a, x) = \big(a, f(x)\big)$$

For any functor F, we can define the concepts of mathematical structures called F-coalgebras and their homomorphisms, or structure-preserving maps.

Definition* 2.3 (Coalgebra)**.** An *F-coalgebra* is a structure (X, f), where the set X is called the *carrier*, and the map $f : X \to F(X)$ is called the *operation*.

Definition 2.4 (Stream Coalgebra Operation)**.** Concretely, the $\mathcal{S}_A$-coalgebras are structures $(X, f : X \to A \times X)$. Usually it is more convenient to decompose the pair-valued map f equivalently as $f_0 : X \to A$ and $f_+ : X \to X$ with:

$$f(x) = \big(f_0(x), f_+(x)\big)$$

Algorithm 2.5. The $\mathcal{S}_A$-coalgebras are formal representations of enumeration procedures: The carrier X is the state space, and the operation components f_0 and f_+ specify, for each state, the *output* element from A and *transition* to the next state, respectively. Such a procedure is naturally executed by starting from an initial state x_0, and collecting the outputs $a_n = f_0(x_n)$ from successive states $x_{n+1} = f_+(x_n)$, as an infinite stream. In pseudocode:

```
exec(f0, f+)(x0):
  var x := x0
  forever
    yield f0(x)
    x := f+(x)
```

Example 2.6. The default textbook example of a stream enumeration procedure is the iterative Fibonacci algorithm:

$$A = \mathbb{N} \qquad X = \mathbb{N}^2 \qquad \mathit{fib}_0(a, b) = b \qquad \mathit{fib}_+(a, b) = (a + b, a)$$

Its state is the pair of the next and current Fibonacci number, respectively. ◇

Example 2.7. A procedure that is related in a non-obvious mathematical way, to be explained later in (2.19), is the following coalgebra,

$$A = \mathbb{R} \qquad X = \mathbb{N} \qquad \mathit{bif}_0(n) = \frac{\varphi^n}{\sqrt{5}} \quad \text{where } \varphi = \frac{1 + \sqrt{5}}{2} \qquad \mathit{bif}_+(n) = n + 1$$

which has simply a sequence index as its state, and thus qualifies as a closed-form generator; the projection bif_0 specifies each element of the sequence directly. ◇

[1] More precisely a *Set-endofunctor*, but no other kind is considered here.

[2] Functors are also required to preserve identity maps and compositions, but that is irrelevant for the present discussion.

Generating streams coalgebraically goes beyond the power of classical recursive sequence definitions; the state may contain relevant information that is not available from the observable output. Typical examples that exhibit this feature are pseudorandom number generators.

Example 2.8. The family of Marsaglia's four-word *xorshift* random number generators [8] is specified in coalgebraic form as

$$A = 2^\ell \quad X = (2^\ell)^4 \quad xsr_0(x,y,z,w) = r \quad xsr_+(x,y,z,w) = (y,z,w,r) \quad \text{where } r = (x \Leftarrow a \Rightarrow b) \oplus (w \Rightarrow c)$$

where ℓ is integer word length, a, b, c are small integer constants from a table of recommendations, $\oplus$ is bitwise exclusive or, and the characteristic operations are combinations of exclusive or and bit shifts:

$$(n \Rightarrow k) = n \oplus (n \gg k) \qquad (n \Leftarrow k) = n \oplus (n \ll k) \qquad \diamond$$

So far, we have argued only operationally what the given examples mean. But one can do much better in the coalgebraic framework. The intuitive execution model from (2.5) can be given a formal foundation.

Definition 2.9 (Coalgebra of Streams)**.** The set of infinite streams A^ω admits an $\mathcal{S}_A$-coalgebra operation:

$$s : A^\omega \to A \times A^\omega \qquad s_0(\alpha) = \alpha_0 \qquad s_+(\alpha)_n = \alpha_{n+1}$$

where s_0 and s_+ are commonly known as *head* and *tail*, respectively, and s^{-1} as *cons*. In order to let A vary, we write explicitly:

$$\begin{aligned} \mathrm{out}_A &: A^\omega \to A \times A^\omega & \mathrm{head}_A &: A^\omega \to A \\ \mathrm{cons}_A &: A \times A^\omega \to A^\omega & \mathrm{tail}_A &: A^\omega \to A^\omega \end{aligned}$$

We also abbreviate $\mathrm{cons}_A(a_0, \alpha)$ to $a_0 ; \alpha$, which saves many parentheses. Note that indices start always at zero.

Definition* 2.10 (Homomorphism)**.** Let (X, f) and (Y, g) be F-coalgebras. A map $h : X \to Y$ is a *homomorphism* from (X, f) to (Y, g) if and only if:

$$g(h(x)) = F(h)(f(x))$$

Proposition 2.11. *For $\mathcal{S}_A$-coalgebras the homomorphism property for h from (X, f) to (Y, g) simplifies to:*

$$g_0\big(h(x)\big) = f_0(x) \qquad g_+\big(h(x)\big) = h\big(f_+(x)\big)$$

The $\mathcal{S}_A$-coalgebra homomorphisms map computations by one enumeration procedure to equivalent computations by another procedure. The coalgebra $(A^\omega, \mathrm{out}_A)$ is special, firstly in the sense that it abstracts from any actual computation by *reading* elements from a stream that is assumed to be given beforehands, and secondly due to the fact that it admits a particular map from any other $\mathcal{S}_A$-coalgebra.

Proposition 2.12. *Let (X, f) be an $\mathcal{S}_A$-coalgebra. Then there is a map:*

$$[\![f]\!] : X \to A^\omega \qquad [\![f]\!](x)_n = f_0\big(f_+^n(x)\big)$$

Proposition 2.13. *The homomorphism property (2.11) specializes for $[\![f]\!]$ to:*

$$\mathrm{head}_A\big([\![f]\!](x)\big) = f_0(x) \qquad \mathrm{tail}_A\big([\![f]\!](x)\big) = [\![f]\!]\big(f_+(x)\big)$$

It is easy to see that $[\![f]\!]$ is indeed a homomorphism.

Proof.

$$\begin{aligned} \mathrm{head}_A\big([\![f]\!](x)\big) &= [\![f]\!](x)_0 \\ &\overset{(2.12)}{=} f_0\big(f_+^0(x)\big) \\ &= f_0(x) \end{aligned} \qquad \begin{aligned} \mathrm{tail}_A\big([\![f]\!](x)\big)_n &= [\![f]\!](x)_{n+1} \\ &\overset{(2.12)}{=} f_0\big(f_+^{n+1}(x)\big) \\ &= f_0\Big(f_+^n\big(f_+(x)\big)\Big) \\ &\overset{(2.12)}{=} [\![f]\!]\big(f_+(x)\big)_n \end{aligned}$$

□

Proposition 2.14. *The two equations of* (2.13) *can be combined – the recursive equivalent of the closed form* (2.12)*, and declarative equivalent of imperative algorithm* (2.5)*:*

$$[\![f]\!](x) = f_0(x)\,;\,[\![f]\!]\big(f_+(x)\big)$$

The homomorphism $[\![f]\!] : X \to A^\omega$ gives the *intended* denotational semantics of the pseudocode (2.5), mapping each initial state to the stream of ensuing outputs. It is *natural* in the sense that it is the unique homomorphism of its type.

Proposition 2.15. *Any homomorphism $h : X \to A^\omega$ between $\mathcal{S}_A$-coalgebras (X, f) and $(A^\omega, \mathrm{out}_A)$, that is each map of that type satisfying* (2.11)*, is in fact equivalent to* (2.12)*.*

Proof. By induction in n, with induction hypothesis IH: $h(x)_n = f_0\big(f_+^n(x)\big)$.

$$\begin{aligned} h(x)_0 &= \mathrm{head}_A\big(h(x)\big) \\ &\overset{(2.11)}{=} f_0(x) \\ &= f_0\big(f_+^0(x)\big) \end{aligned} \qquad\qquad \begin{aligned} h(x)_{n+1} &= \mathrm{tail}_A\big(h(x)\big)_n \\ &\overset{(2.11)}{=} h(f_+(x))_n \\ &\overset{\text{IH}}{=} f_0\Big(f_+^n\big(f_+(x)\big)\Big) \\ &= f_0\big(f_+^{n+1}(x)\big) \end{aligned}$$

□

This uniqueness is an instance of an important general pattern.

Definition* 2.16. An F-coalgebra (Y, g) is called *final* if and only if there is a unique homomorphism from any other F-coalgebra (X, f).

Proposition* 2.17. *Final F-coalgebras are essentially unique; any pair of them is connected by a unique bijective homomorphism.*

In that terminology, streams are "the" final $\mathcal{S}_A$-coalgebra, and the *final semantics* of enumeration algorithms. It has several nice properties, of which we can state only the first in terms of the framework established so far.

Remark 2.18. The uniqueness proof given above concisely embodies the relationship of the coalgebraic approach to streams to traditional index-subscript-based notations: The proof is the only invocation of the induction principle necessary, and only at the meta level. It establishes a *coinductive* definition principle: any *small-step* behavior map $f : X \to A \times X$ gives rise to a unique *big-step* behavior map $[\![f]\!] : X \to A^\omega$. Note that in coinduction, foundation via terminating base cases is neither required nor particularly useful.

Example 2.19. Now we can state the intended use and precise meaning of the above examples.

- The standard Fibonacci sequence is obtained from (2.6) as $[\![\mathit{fib}]\!](1, 0)$.
- A near-Fibonacci sequence is obtained from (2.7) as $[\![\mathit{bif}]\!](0)$. It has the funny property that rounding $[\![\mathit{bif}]\!](0)_n$ yields $[\![\mathit{fib}]\!](1, 0)_n$, which can be derived from Binet's formula.
- A stream of reasonably random-looking integer words (that pass standard statistical tests [8]) is obtained from (2.8) by applying $[\![\mathit{xsr}]\!]$ to four *seed* values. ◇

In this section, we have introduced the concepts of stream spaces as final coalgebras, and of coinductively defined streams (as a generalization of classical recursively defined sequences) as the corresponding unique homomorphisms. This idea is well established, see for instance [7]. So far, the benefits of the theoretical framework are rather modest. Straightforward intuitions are confirmed, but little further insight is produced. Things shall get more illuminating now, when input is added.

3 First Order: Stream Transforms

Definition 3.1. Fix some nonempty sets A and B, and consider the functor:

$$\mathcal{T}_{AB}(X) = (A \to B \times X) \qquad\qquad \mathcal{T}_{AB}(f)(k)(a) = \big(b, f(x)\big) \quad \text{where } k(a) = (b, x)$$

Algorithmically, the $\mathcal{T}_{AB}$-coalgebras represent the category of Mealy-type stream transducers. Their operations $f : X \to A \to B \times X$, by Currying equivalent to $\overline{f} : X \times A \to B \times X$,[3] specify the output element from B and transition to next state, given the current state and input element from A. Again, it is sometimes convenient to consider instead the component maps $f_0 : X \to A \to B$ and $f_+ : X \to A \to X$, with $f(x)(a) = \big(f_0(x)(a), f_+(x)(a)\big)$.

Proposition 3.2. *For $\mathcal{T}_{AB}$-coalgebras the homomorphism property for h from (X, f) and (Y, g) simplifies to:*

$$g_0\big(h(x)\big)(a) = f_0(x)(a) \qquad\qquad g_+\big(h(x)\big)(a) = h\big(f_+(x)(a)\big)$$

Algorithm 3.3. Execution yields the transformed output stream as a synchronous function of the input stream:

```
exec(f0, f+)(x0):
  var x := x0
  forever
    var a := receive
    yield f0(x)(a)
    x := f+(x)(a)
```

Example 3.4. The following two similar coalgebra operations compute the (cumulative) sum and the (backward) difference transform of a stream of numbers, respectively.

$$A = \mathbb{R} \qquad B = \mathbb{R} \qquad X = \mathbb{R} \qquad \begin{array}{ll} sum_0(x, a) = a + x & sum_+(x, a) = a + x \\ dif_0(x, a) = a - x & dif_+(x, a) = a \end{array}$$

◇

However, there is a catch; the general function space $A^\omega \to B^\omega$ is too large to be the adequate semantic domain! Namely, it turns out that for a suitable $\mathcal{T}_{AB}$-coalgebra operation f, we would have to give a component of type $f_0 : (A^\omega \to B^\omega) \to A \to B$, which is not meaningful generically, without additional knowledge of A and B. On second thought, the $\mathcal{T}_{AB}$-coalgebras specify *sequential* elementwise computation on streams, and the internal state X can be used to transport information forwards in time, but not backwards. As a result, all stream transforms implemented by $\mathcal{T}_{AB}$-coalgebras have the property of *causality*: output up to each step is determined by input only up to the same step.

Definition 3.5 (Causality)**.** Let $\alpha_{<n}$ denote the sequence of the first n elements of the infinite stream α. Then we can define the space of *causal stream functions*:

$$A \overset{\omega}{\rightsquigarrow} B = \{k : A^\omega \to B^\omega \mid \forall\, n.\, \alpha_{<n} = \alpha'_{<n} \Longrightarrow k(\alpha)_{<n} = k(\alpha')_{<n}\}$$

The space of causal stream functions can be endowed with a $\mathcal{T}_{AB}$-coalgebra operation, by a not entirely trivial construction [17].

Proposition 3.6. *For all $k : A \overset{\omega}{\rightsquigarrow} B$ we have in particular, at the first step:*

$$\mathrm{head}_A(\alpha) = \mathrm{head}_A(\alpha') \implies \mathrm{head}_B\big(k(\alpha)\big) = \mathrm{head}_B\big(k(\alpha')\big)$$

Proposition 3.7. *The previous property ensures that there is a generic map satisfying*

$$\mathrm{head}_{AB} : (A \overset{\omega}{\rightsquigarrow} B) \to A \to B \qquad\qquad \mathrm{head}_{AB}(k)\big(\mathrm{head}_A(\alpha)\big) = \mathrm{head}_B\big(k(\alpha)\big)$$

with no reference to particular elements of A or B.

Definition 3.8. Now consider a related map defined by the equation:

$$\mathrm{tail}_{AB}(k)(a_0)(\alpha) = \mathrm{tail}_B\big(k(a_0 ; \alpha)\big)$$

Proposition 3.9. *This map can be specified in analogy to (3.7):*

$$\mathrm{tail}_{AB} : (A \overset{\omega}{\rightsquigarrow} B) \to A \to (A \overset{\omega}{\rightsquigarrow} B) \qquad\qquad \mathrm{tail}_{AB}(k)\big(\mathrm{head}_A(\alpha)\big)\big(\mathrm{tail}_A(\alpha)\big) = \mathrm{tail}_B\big(k(\alpha)\big)$$

[3] We denote both the currying and uncurrying operations by overlining.

Proposition 3.10. *The pair of equations* (3.7), (3.9) *can be combined equivalently:*

$$k(a_0 ; \alpha) = \mathrm{head}_{AB}(k)(a_0) ; \mathrm{tail}_{AB}(k)(a_0)(\alpha)$$

The causal function space $A \overset{\omega}{\rightsquigarrow} B$ admits a $\mathcal{T}_{AB}$-coalgebra operation t, namely $t_0 = \mathrm{head}_{AB}$ and $t_+ = \mathrm{tail}_{AB}$. This is again a final $\mathcal{T}_{AB}$-coalgebra: from any other $\mathcal{T}_{AB}$-coalgebra (X, f) there is a unique homomorphism $[\![f]\!]$, which however can not be written down in closed form quite as neatly as in (2.12).

Definition 3.11. Let (X, f) be a $\mathcal{T}_{AB}$-coalgebra. Then there is a map:

$$[\![f]\!] : X \to (A \overset{\omega}{\rightsquigarrow} B) \qquad\qquad [\![f]\!](x)(\alpha)_n = f_0\Big(f_+\big(\ldots\big(f_+(x)(\alpha_0)\big)\ldots\big)(\alpha_{n-1})\Big)(\alpha_n)$$

This makes the causality property explicit: the n-th element of $[\![f]\!](x)(\alpha)$ makes only use of up to the n-th element of the given, infinite input stream α. Causal stream computations are particularly well-behaved in online, reactive or real-time context: Assume that the elements of input are being made available at successive points in time. Once the first n elements have been obtained, the n-th output element can be computed, with a latency that depends only on the computation itself and can be determined by standard worst-case execution time analyses, but not on the environment. Thus *productiveness* (elementwise termination, [15]) of computations becomes a local property.

Proposition 3.12. *The homomorphism property* (3.2) *specializes for the final case to:*

$$\mathrm{head}_{AB}\big([\![f]\!](x)\big)(a_0) = f_0(x)(a_0) \qquad\qquad \mathrm{tail}_{AB}\big([\![f]\!](x)\big)(a_0)(\alpha) = [\![f]\!]\big(f_+(x)(a_0)\big)(\alpha)$$

The proof of validity and uniqueness of $[\![f]\!]$ is completely analogous to the $\mathcal{S}_A$-case; refer to (2.13) and (2.15), respectively.

Proposition 3.13. *Specialization of* (3.10) *with* $k = [\![f]\!](x)$ *and* (3.12) *gives the recursive equivalent of the closed form* (3.11)*:*

$$[\![f]\!](x)(a_0 ; \alpha) = f_0(x)(a_0) ; [\![f]\!]\big(f_+(x)(a_0)\big)(\alpha)$$

Algorithm 3.14. The following literal translation into Haskell syntax is a valid and useful, purely functional program:

```
trans :: (x -> a -> (b, x)) -> x -> [a] -> [b]
trans f x (a0 : as) = f0 x a0 : trans f (fp x a0) as
  where f0 x a = fst (f x a)
        fp x a = snd (f x a)
```

Example 3.15. The stream transforms of (3.4) have the desired semantics:

$$[\![sum]\!](c)(\alpha)_n = c + \sum_{i=0}^{n} \alpha_i \qquad \begin{aligned} [\![dif]\!](d)(\alpha)_0 &= \alpha_0 - d \\ [\![dif]\!](d)(\alpha)_{n+1} &= \alpha_{n+1} - \alpha_n \end{aligned} \qquad \text{that is} \qquad [\![dif]\!](d)(\alpha)_n = \alpha_n - (d ; \alpha)_n$$

◇

Remark 3.16. The *forward* difference transform, $\nabla(\alpha)_n = \alpha_n - \alpha_{n+1}$, is not causal.

In this section, we have introduced coinductively defined, causal stream transformers as a coalgebraic perspective on Mealy automata, in strong analogy to plain coinductive streams. The coalgebraic approach suggests a *head–tail* structure for transformers, which has incidentally also been exploited for continuation-based functional reactive programming [10].

The closed-form notation which has worked well for coinductive streams breaks down in the transformer case. This is quite typical for coalgebraic structures, which are often ill-suited to classical syntax-oriented formal representation, and therefore usually phrased in the more abstract language of category theory.

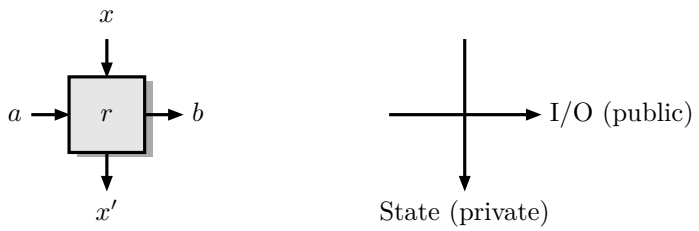

Figure 1: $\mathcal{T}_{AB}$-coalgebra $(X, \overline{r})$ in quadrilateral form, from [17].

4 Compositional Stream Networks

The uncurried, quaternary form of $\mathcal{T}_{AB}$-coalgebra operations, $r : X \times A \to B \times X$, henceforth abbreviated to $r : A \xrightarrow{X} B$, plays a central role in the semantics of the Sig language. Any data-flow network definition in Sig is brought to this canonical operational form via a series of program transformations described in [17]. The same form also suggests a visual, quadrilateral presentation, as illustrated in Figure 1. The dichotomy of public I/O and private state suggests an object-oriented approach to execution.[4] Alluding to the visual form, and for lack of a more fitting name, we shall call the elements of a function space $A \xrightarrow{X} B$ *quadrilaterals*.

Algorithm 4.1. Execution is reified as objects that consume one input element and produce one output element on each synchronous call of a public method. Control is transferred back after each element; the stream is held together by side-effect updates on the private state. The missing pieces r_0, r_+, x_0 are modeled as abstract methods.

```
class quad:
  private var x := x0()
  step(a):
    var b
    (b, x) := r(x, a)
    return b
  abstract x0()
  abstract r(x, a)
```

Definition 4.2. The denotational semantics $[\![r \vdash x]\!]$ of a quadrilateral form $r : A \xrightarrow{X} B$ with initial state x is given by (homomorphisms to) the final $\mathcal{T}_{AB}$-coalgebra.

$$[\![r \vdash x]\!] = [\![\overline{r}]\!](x)$$

Proposition 4.3. *The unique homomorphism for these semantics from (3.12) is specified by:*

$$\mathrm{head}_{AB}\big([\![r \vdash x]\!]\big)(a_0) = r_0(x, a_0) \qquad \mathrm{tail}_{AB}\big([\![r \vdash x]\!]\big)(a_0) = [\![r \vdash r_+(x, a_0)]\!]$$

Semantically, stream functions can be composed meaningfully along several axes, namely parallel and (data)-sequential.

Definition 4.4. Parallel and sequential composition of stream functions are defined in the obvious way:

$$\frac{k : A \overset{\omega}{\rightsquigarrow} B \quad l : C \overset{\omega}{\rightsquigarrow} D}{k \otimes l : A \times C \overset{\omega}{\rightsquigarrow} B \times D} \qquad (k \otimes l)(\alpha, \gamma)_n = \big(k(\alpha)_n, l(\gamma)_n\big)$$

$$\frac{k : A \overset{\omega}{\rightsquigarrow} B \quad l : B \overset{\omega}{\rightsquigarrow} C}{l \circ k : A \overset{\omega}{\rightsquigarrow} C} \qquad (l \circ k)(\alpha)_n = l\big(k(\alpha)\big)_n$$

Proposition 4.5. *The definition of parallel composition has the equivalent recursive form:*

$$\mathrm{head}_{(A\times C)(B\times D)}(k \otimes l)(a_0, c_0) = \big(\mathrm{head}_{AB}(k)(a_0), \mathrm{head}_{CD}(l)(c_0)\big)$$
$$\mathrm{tail}_{(A\times C)(B\times D)}(k \otimes l)(a_0, c_0) = \mathrm{tail}_{AB}(k)(a_0) \otimes \mathrm{tail}_{CD}(l)(c_0)$$

[4] Indeed, the current Sig compiler targets the Java VM [18].

Proof. Implement $k \otimes l = [\![f]\!](k,l)$ coalgebraically as $\big((A \overset{\omega}{\rightsquigarrow} C) \times (B \overset{\omega}{\rightsquigarrow} D), f\big)$ with:

$$f_0(k,l)(a_0,c_0) = \big(\mathrm{head}_{AB}(k)(a_0), \mathrm{head}_{CD}(l)(c_0)\big)$$
$$f_+(k,l)(a_0,c_0) = \big(\mathrm{tail}_{AB}(k)(a_0), \mathrm{tail}_{CD}(l)(c_0)\big)$$

The proof of correctness of this solution and transformation to the above form are left as an exercise to the reader. □

These abstract semantic concepts of composition can be pulled back to the concrete quadrilateral form.

Proposition 4.6. *For any quadrilaterals $q : A \xrightarrow{X} B$ and $r : C \xrightarrow{Y} D$, there is a quadrilateral $q \parallel r$ such that:*

$$q \parallel r : A \times C \xrightarrow{X \times Y} B \times D \qquad\qquad [\![q \parallel r \vdash x, y]\!] = [\![q \vdash x]\!] \otimes [\![r \vdash y]\!]$$

Instead of educated guesses as presented ad-hoc in [17], we can now give a formal derivation.

Proof. Take the composite $s = q \parallel r$ as unknown and to be solved for. Assume the goal

$$G\colon\ [\![s \vdash x, y]\!] = [\![q \vdash x]\!] \otimes [\![p \vdash y]\!]$$

and work out the homomorphism property according to (4.3):

$$\begin{aligned}
\overbrace{s_0\big((x,y),(a_0,c_0)\big)}^{\dagger} &\overset{(4.3)}{=} \mathrm{head}_{(A\times C)(B\times D)}\big([\![s \vdash x, y]\!]\big)(a_0,c_0) \\
&\overset{(4.2)}{=} \mathrm{head}_{(A\times C)(B\times D)}\big([\![q \vdash x]\!] \otimes [\![r \vdash y]\!]\big)(a_0,c_0) \\
&\overset{G}{=} \mathrm{head}_{(A\times C)(B\times D)}\big([\![\overline{q}]\!](x) \otimes [\![\overline{r}]\!](y)\big)(a_0,c_0) \\
&\overset{(4.5)}{=} \Big(\mathrm{head}_{AB}\big([\![\overline{q}]\!](x)\big)(a_0), \mathrm{head}_{CD}\big([\![\overline{r}]\!](y)\big)(c_0)\Big) \\
&\overset{(4.3)}{=} \underbrace{\big(q_0(x,a_0), r_0(y,c_0)\big)}_{\dagger}
\end{aligned}$$

$$\begin{aligned}
[\![s \vdash \overbrace{s_+\big((x,y),(a_0,c_0)\big)}^{\ddagger}]\!] &\overset{(4.2)}{=} [\![\overline{s}]\!]\big(\overline{s}_+(x,y)(a_0,c_0)\big) \\
&\overset{(3.12)}{=} \mathrm{tail}_{(A\times C)(B\times D)}\big([\![\overline{s}]\!](x,y)\big)(a_0,c_0) \\
&\overset{G}{=} \mathrm{tail}_{(A\times C)(B\times D)}\big([\![\overline{q}]\!](x) \otimes [\![\overline{r}]\!](y)\big)(a_0,c_0) \\
&\overset{(4.5)}{=} \mathrm{tail}_{AB}\big([\![\overline{q}]\!](x)\big)(a_0) \otimes \mathrm{tail}_{CD}\big([\![\overline{r}]\!](y)\big)(c_0) \\
&\overset{(3.12)}{=} [\![\overline{q}]\!]\big(\overline{q}_+(x)(a_0)\big) \otimes [\![\overline{r}]\!]\big(\overline{r}_+(y)(c_0)\big) \\
&= [\![\overline{q}]\!]\big(q_+(x,a_0)\big) \otimes [\![\overline{r}]\!]\big(r_+(y,c_0)\big) \\
&\overset{G}{=} [\![\overline{s}]\!]\big(q_+(x,a_0), r_+(y,c_0)\big) \\
&\overset{(4.2)}{=} [\![s \vdash \underbrace{\big(q_+(x,a_0), r_+(y,c_0)\big)}_{\ddagger}]\!]
\end{aligned}$$

By matching the marked terms we find the most evident solution:

$$(q \parallel r)_0\big((x,y),(a,c)\big) = \big(q_0(x,a), r_0(y,c)\big) \qquad\qquad (q \parallel r)_+\big((x,y),(a,c)\big) = \big(q_+(x,a), r_+(y,c)\big)$$

□

Proposition 4.7. *For any quadrilaterals $q : A \xrightarrow{X} B$ and $r : B \xrightarrow{Y} C$, there is a quadrilateral $q \cdot r$ such that:*

$$q \cdot r : A \xrightarrow{X \times Y} C \qquad\qquad [\![(q \cdot r) \vdash (x,y)]\!] = [\![r \vdash y]\!] \circ [\![q \vdash x]\!]$$

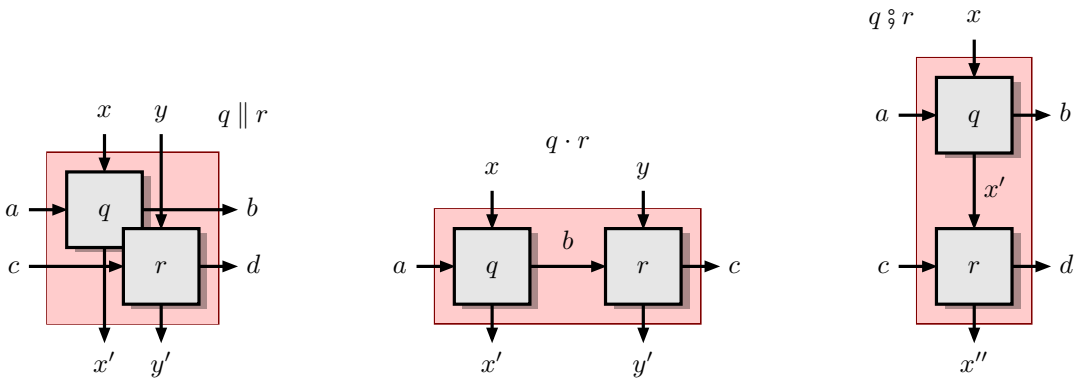

Figure 2: Legal composition axes for data flow networks [17].

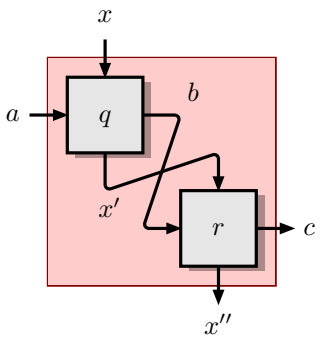

Figure 3: Illegal "monadic" composition of data flow networks.

Proof Sketch. By reasoning analogously to the preceding case, we find the solution:

$$(q \cdot r)_0\big((x,y),a\big) = r_0\big(y, q_0(x,a)\big) \qquad (q \cdot r)_+\big((x,y),a\big) = \Big(q_+(x,a), r_+\big(y, q_0(x,a)\big)\Big)$$

□

Definition 4.8. A third meaningful axis of quadrilateral composition can be defined, symmetrically to the preceding. For any quadrilaterals $q : A \xrightarrow{X} B$ and $r : C \xrightarrow{X} D$, there is $q \fatsemi r : A \times C \xrightarrow{X} B \times D$, defined as:

$$(q \fatsemi r)_0\big(x,(a,c)\big) = \Big(q_0(x,a), r_0\big(q_+(x,a),c\big)\Big) \qquad (q \fatsemi r)_+\big(x,(a,c)\big) = r_+\big(q_+(x,a),c\big)$$

All three compositions are depicted in Figure 2.

Remark 4.9. A "diagonal" composition of the form depicted in Figure 3, probably expected by any reader familiar with the *state monad* in functional programming [24], is conspicuously absent, as it breaks the synchronous paradigm where "horizontal" data flow is instantaneous. By constrast, such an operation makes perfect sense in sequential programming, where data dependency (b) is generally understood to entail state dependency (x'). A little formal detail can make a big difference: for the state monad form, quadrilaterals are simply curried the other way, to $\widetilde{r} : A \to (S \to B \times S)$, but the resulting change of perspective is enormous. As a side effect, comparison of coinductive and functional reactive programming styles is very difficult, due to the orthogonal attitudes.

Remark 4.10. Cyclic data flow cannot be constructed by the given composition operators. For the Sig approach to cycles, see the next section.

In this section, we have introduced the quadrilateral form of elementwise stream computation, which corresponds straightforwardly to either the state monad or the coinductive stream transformer approach on the theoretical side, and to an object-oriented implementation strategy on the practical side. We have calculated parallel and sequential composition operators on quadrilaterals, and confirmed the visual and formal intuitive solutions. The method used to obtain these calculations is of general notability: We have reworked the coalgebraic semantic equations according to (4.3), using the specification of the desired operation in the process, and

Figure 4: Delay operator; *left:* state-implicit bilateral form; *right:* state-explicit quadrilateral form.

read off the most evident solution by syntactic pattern matching. Despite the simple approach, we shall find this method of coinductive program synthesis just as useful for other primitive operations of stream programming language implementations.

5 State and Delay

The quadrilateral form, and the explicit management of state, exists only in the intermediate representation of Sig programs. The front-end language takes a more high-level approach, adapted to pervasive patterns of causal stream programming. On the one hand, element-wise computations where output β_n depends on input α_n only are extremely common. They are implemented for free, by simply ignoring the state axis.

On the other hand, dependency on past inputs often has relative and bounded form, with β_n depending on α_{n-i} for a few small delay lengths i only. To this end, Sig features a single-step *delay* operator. In [17] we have argued for its implementation in quadrilateral form by appeal to operational intuition. Actually we can do better in the coalgebraic framework, and derive the same result from a rigorous semantic specification.

Definition 5.1. The semantic delay operator takes an initial value and a stream to be delayed by one step.

$$\mathrm{delay}_A : A \to A \overset{\omega}{\rightsquigarrow} A \qquad \mathrm{delay}_A(a_0)(\alpha)_0 = a_0 \qquad \mathrm{delay}_A(a_0)(\alpha)_{n+1} = \alpha_n$$

In other words, delay_A is simply $\overline{\mathrm{cons}_A}$ with explicit causality; the initial value is *prepended*, and $\mathrm{delay}_A(\alpha)_{n+1}$ depends on α_n exactly. We abbreviate $\overline{\mathrm{cons}_A}(a_0)$ to $(a_0;)$.

Proposition 5.2. *The prepending operation as a class of final $\mathcal{T}_{AA}$-coalgebra elements obeys the recurrence:*

$$\mathrm{head}_{AA}(a_0;)(a_1) = a_0 \qquad \mathrm{tail}_{AA}(a_0;)(a_1) = (a_1;)$$

Armed with this auxiliary fact, we can formulate the problem.

Proposition 5.3. *There is a quadrilateral form $\delta : A \xrightarrow{A} A$ such that $[\![\delta \vdash a_0]\!] = (a_0;)$.*

The reader is encouraged to pause for an educated guess before reading on.

Proof. Follow the derivation strategy of (4.6) with the goal G: $[\![\overline{\delta}]\!](a) = (a;)$.

$$\begin{aligned} \overbrace{\delta_0(a_0,a_1)}^{\dagger} &= \overline{\delta}_0(a_0)(a_1) \\ &\overset{(3.12)}{=} \mathrm{head}_{AA}\big([\![\overline{\delta}]\!](a_0)\big)(a_1) \\ &\overset{G}{=} \mathrm{head}_{AA}(a_0;)(a_1) \\ &\overset{(5.2)}{=} \underbrace{a_0}_{\dagger} \end{aligned} \qquad \begin{aligned} [\![\overline{\delta}]\!]\big(\overbrace{\delta_+(a_0,a_1)}^{\ddagger}\big) &= [\![\overline{\delta}]\!]\big(\overline{\delta}_+(a_0)(a_1)\big) \\ &\overset{(3.12)}{=} \mathrm{tail}_{AA}\big([\![\overline{\delta}]\!](a_0)\big)(a_1) \\ &\overset{G}{=} \mathrm{tail}_{AA}(a_0;)(a_1) \\ &\overset{(5.2)}{=} (a_1;) \\ &\overset{G}{=} [\![\overline{\delta}]\!]\underbrace{(a_1)}_{\ddagger} \end{aligned}$$

By matching the marked terms we find the now evident, but generally mildly surprising, solution (see Figure 4):

$$\delta(a_0, a_1) = (a_0, a_1) \qquad \square$$

This formal argument rigorously reconstructs the previous intuitive construction of δ:

> *Simultaneously route pre-state to output and input to post-state, respectively, such that the current input becomes pre-state and hence output in the next step, and so forth.*

Algorithm 5.4. With specialization to δ, the object-oriented implementation becomes:

```
class delay extends quad:
  step(a):
    var a' := x
    x := a
    return a'
```

A visualization of delay is depicted in Figure 4. Delay operators can be combined freely with stateless computations in a data flow network. The quadrilateral form is obtained by synthesizing a pair of pre- and post-state variable for each occurrence of delay, and wiring them in the way depicted on the right hand side. Since each δ node is replaced by a pair of independent identities, the network may be simplified afterwards by propagation.

Remark 5.5. Apparent data flow *through* the δ node is broken in the process. Hence delayed feedback loops no longer appear as cycles. The SIG language forbids instantaneous cyclic data flow; only networks that become acyclic by delay elimination are valid.

Since the state space $X_1 \times \cdots \times X_n$ is synthesized from the types of the n occurrences of delay operators in a network r, the corresponding initial value $\xi = (x_1, \ldots, x_n)$ need to be inferred as well. Then the semantics of the network is the single causal stream function $[\![r \vdash \xi]\!]$. The SIG front-end language provides a binary delay operator of the form `a;e` which translates to $\delta(e)$ with implied initial value a.

Example 5.6. The random number generators from (2.8) in delay-based state-implicit form are depicted in Figure 5, left hand side. The shift register structure is more clearly elucidated than in the coalgebraic form, or in the original imperative code, for that matter. The following pseudocode in simplified SIG describes this network, including seed values (s_0, s_1, s_2, s_3):

```
[ -> r
  where
    r := (x ^<< a ^>> b) ^ (w ^>> c)
    x := s0 ; s1 ; s2 ; w
    w := s3 ; r
]
```

The result of state synthesis from delay operators is depicted in Figure 5, right hand side. Note the absence of cycles. The object-oriented implementation gives the moral equivalent of the original presentation in the C language, in pseudocode:

```
class xsr(s0, s1, s2, s3):
  var x := s0; y := s1; z := s2; w := s3
  step():
    r := (x ^<< a ^>> b) ^ (w ^>> c)
    (x, y, z, w) := (y, z, w, r)
    return r
```

◇

In this section, we have introduced the single-step delay operator, the crucial ingredient in stream programming beyond memoryless elementwise computation and basic operation of the pervasive "shift register" algorithmic pattern. We have calculated the mildly confusing implementation of delay as the identical quadrilateral, and illustrated the transition from delay to explicit state in the front-to-core reduction of the SIG language.

6 Higher Order: Stream Function Application

Higher-order functions are notoriously a more complex subject in a stream context than in ordinary functional programming. However, they are of crucial importance for the high-level expressivity of a practical stream programming language.

On the one hand, if streams are the first-class citizens, then there are streams of stream functions, and functions on these, to deal with. An abstract semantics has been given in terms of comonads [22, 23]. All theoretical elegance notwithstanding, that theoretical framework is even less well-known to the general programming community than coalgebras. Apparently it is also less directly tied to practical matters; we are not aware of any relevant implementation it could have inspired.

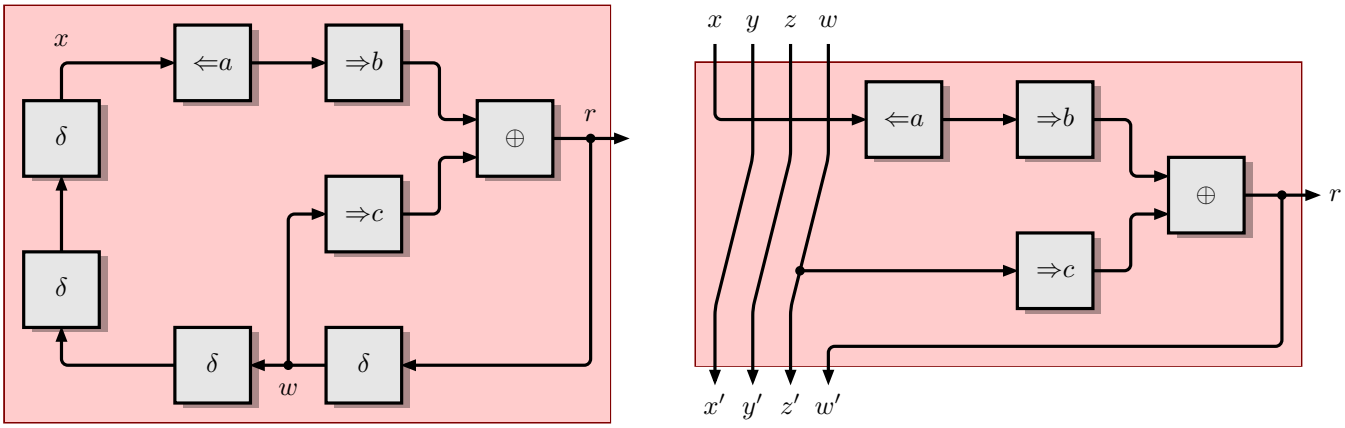

Figure 5: Marsaglia's four-word *xorshift* RNG; *left:* state-implicit form; *right:* state-explicit form.

On the other hand, there is the view on a single stream function as a stream of element functions, given by the head_{AB} and tail_{AB} operations on $A \overset{\omega}{\rightsquigarrow} B$. This view had already been explored, albeit in ad-hoc terms, and its practical relevance argued in [3]. We have given a systematic theoretical reconstruction in [17] and independently a practical implementation strategy in [18].

In the following section, in order to connect the two more precisely, we follow up on the method already applied in the two preceding section, and calculate a representation for first-class stream functions and its interpretation from a coalgebraic specification.

As a first approximation, assume a generic abstract datatype Z_{AB} with an interpretation map, app_{AB}, to represent *some* set of causal stream functions for each domain–range pair A, B:

$$\mathrm{app}_{AB} : Z_{AB} \to (A \overset{\omega}{\rightsquigarrow} B)$$

The type of the interpretation map, with the final coalgebra range, suggests a definition by homomorphism.

Proposition 6.1. *There is a quadrilateral* $@ : A \xrightarrow{Z_{AB}} B$ *such that* $[\![\overline{@}]\!] = \mathrm{app}_{AB}$.

Proof. Follow the same derivation strategy as before.

$$\begin{aligned}\overbrace{\overline{@}_0(z,a_0)}^{\dagger} &= \overline{@}_0(z)(a_0)\\ &\overset{(3.12)}{=} \mathrm{head}_{AB}\big([\![\overline{@}]\!](z)\big)(a_0)\\ &\overset{G}{=} \underbrace{\mathrm{head}_{AB}\big(\mathrm{app}_{AB}(z)\big)(a_0)}_{\dagger}\end{aligned}$$

$$\begin{aligned}\overbrace{\mathrm{app}_{AB}\big(@_+(z,a_0)\big)}^{\ddagger} &\overset{G}{=} [\![\overline{@}]\!]\big(@_+(z,a_0)\big)\\ &= [\![\overline{@}]\!]\big(\overline{@}_+(z)(a_0)\big)\\ &\overset{(3.12)}{=} \mathrm{tail}_{AB}\big([\![\overline{@}]\!](z)\big)(a_0)\\ &\overset{G}{=} \underbrace{\mathrm{tail}_{AB}\big(\mathrm{app}_{AB}(z)\big)(a_0)}_{\ddagger}\end{aligned}$$

By matching the marked terms, we find the specification up to interpretation:

$$@_0(z,a_0) = \mathrm{head}_{AB}\big(\mathrm{app}_{AB}(z)\big)(a_0) \qquad \mathrm{app}_{AB}\big(@_+(z,a_0)\big) = \mathrm{tail}_{AB}\big(\mathrm{app}_{AB}(z)\big)(a_0)$$

□

This account is, well, abstract. In particular, no help is given for making Z_{AB} closed in the sense of the right hand equation; for each element $z \in Z_{AB}$, we must also include, for each a_0, some sucessor element $@_+(z,a_0) \in Z_{AB}$ whose interpretation is the stream function tail of the interpretation of z.

The problem can be simplified substantially, if the representation is given at the same implementation level as its usage context, namely in terms of quadrilaterals. Assume a generic abstract datatype W_{ABX} with an interpretation map app'_{ABX} to represent some set $R_{ABX} \subseteq (A \xrightarrow{X} B)$ of quadrilaterals for each domain–range pair A, B and state space X:

$$\mathrm{app}'_{ABX} : W_{ABX} \to (A \xrightarrow{X} B)$$

Unlike in the previous attempt, the choice of represented functions, $\mathrm{Img}(\mathrm{app}'_{ABX})$, is completely arbitrary; we shall demonstrate that it need not be closed under any operation. Even finite W_{ABX} will do.

Any stream function may be implemented by various quadrilaterals: only the input–output relation is specified, but neither the state space parameter X, nor the initial state $x \in X$. Packing these information items into a dependent record type yields our improved datatype of stream functions:

Definition 6.2.

$$Z_{AB} = \bigcup_X \{X\} \times W_{ABX} \times X \qquad\qquad \mathrm{app}_{AB}(X, w, x) = [\![\mathrm{app}'_{ABX}(w) \vdash x]\!]$$

Proposition 6.3. *There is a quadrilateral* $@ : A \xrightarrow{Z_{AB}} B$*, such that* $[\![\overline{@}]\!] = \mathrm{app}_{AB}$.

Proof. Follow the usual derivation strategy.

$$\begin{aligned}
&\overbrace{\overline{@_0}((X,w,x),a_0)}^{\dagger} \\
&= \overline{@_0}(X,w,x)(a_0) \\
&\overset{G}{=} \mathrm{head}_{AB}\big(\mathrm{app}_{AB}(X,w,x)\big)(a_0) \\
&\overset{(6.2)}{=} \mathrm{head}_{AB}\big([\![\mathrm{app}'_{ABX}(w) \vdash x]\!]\big)(a_0) \\
&\overset{(4.2)}{=} \mathrm{head}_{AB}\big(\overline{[\![\mathrm{app}'_{ABX}(w)]\!]}(x)\big)(a_0) \\
&\overset{(3.12)}{=} \overline{\mathrm{app}'_{ABX}(w)_0}(x)(a_0) \\
&= \underbrace{\mathrm{app}'_{ABX}(w)_0(x,a_0)}_{\dagger}
\end{aligned}
\qquad
\begin{aligned}
&\mathrm{app}_{AB}\overbrace{\big(\overline{@_+}((X,w,x),a_0)\big)}^{\ddagger} \\
&\overset{G}{=} \mathrm{tail}_{AB}\big(\mathrm{app}_{AB}(X,w,x)\big)(a_0) \\
&\overset{(6.2)}{=} \mathrm{tail}_{AB}\big([\![\mathrm{app}'_{ABX}(w) \vdash x]\!]\big)(a_0) \\
&\overset{(4.2)}{=} \mathrm{tail}_{AB}\big(\overline{[\![\mathrm{app}'_{ABX}(w)]\!]}(x)\big)(a_0) \\
&\overset{(3.12)}{=} \overline{[\![\mathrm{app}'_{ABX}(w)]\!]}(\overline{\mathrm{app}'_{ABX}(w)_+}(x)(a_0)) \\
&\overset{(4.2)}{=} [\![\mathrm{app}'_{ABX}(w) \vdash \overline{\mathrm{app}'_{ABX}(w)_+}(x)(a_0)]\!] \\
&\overset{(6.2)}{=} \mathrm{app}_{AB}\underbrace{\big(X, w, \mathrm{app}'_{ABX}(w)_+(x,a_0)\big)}_{\ddagger}
\end{aligned}$$

By matching the marked terms, we find the most evident solution:

$$@_0\big((X,w,x),a_0\big) = \mathrm{app}'_{ABX}(w)_0(x,a_0) \quad @_+\big((X,w,x),a_0\big) = (X,w,x') \quad \text{where} \quad x' = \mathrm{app}'_{ABX}(w)_+(x,a_0)$$

□

The calculated solution keeps the first two fields of the dependent record, namely the state space X and the quadrilateral reference w constant, and updates only the transient state component x. Hence we have kept our promise that the choice of represented stream functions is completely arbitrary, and not subject to any closure constraints.

This solution also goes extremely well with the object-oriented approach to implementation, where X and w are realized statically as a field declaration and method binding, respectively, whereas as x, x' is the dynamically changing state variable typed by X. The return value and in-place update effect of $\mathrm{app}'_{AB}(w)$ can be understood as the invocation of the *step* update on a private instance of the class implementing w. Thus the application of stream functions is realized as the *aggregation* of objects.

Example 6.4. Consider the Euler method for numerical integration of autonomous ordinary differential equations:

$$\dot{y} = f(y) \qquad \text{discretized to} \qquad y_{n+1} = y_n + \delta t \cdot f(y_n)$$

Its implementation with output stream y, indepently of the stream function f given as a class w, is a simple matter of managing a private instance z:

```
class int(dt, w, y0):
  var y := y0
  const z := new w()
  step():
    var u := y
    y := y + dt * z.step(y)
    return u
```

Note how variables w and z have been chosen to allude to the above formal datatypes, how the *new* operation instantiates w to z, with unknown initial private state, and how the method invocation $z.step$ performs the computations of $@_0$ and $@_+$ as interface data flow and side effect, respectively.

The pictured code pattern, distilled from the preceding formal reasoning, is in line with what could be expected from a competent imperative programmer. On the one hand, this observation serves to illustrate that the calculation of higher-order function semantics is not so much of theoretical, but rather of practical interest, and relevant for efficient code generation. On the other hand, it embodies the aspiration of the SIG design to reconcile formal semantics with domain-specific low-level programming lore. ◇

In this section, we have introduced the dynamic application operator for coinductive stream functions, as the key ingredient for higher-order programming. We have given a formal representation in terms of dependent records, the mathematical metaphor for objects with hidden state and virtual step methods, and calculated the intuitive solution, namely application-as-aggregation, by coinductive synthesis.

7 Conclusion

7.1 Related Work

Coalgebraic approaches have been studied early on in the context of structural operational semantics [20], for their potential to reduce the conceptual distance between denotational and operational semantics considerably. Whereas algebraic approaches produce abstract *metaphors* of computational concepts, their coalgebraic duals lead to more concrete *descriptions* of hands-on computational business, but at the same level of formal rigor.

Coalgebraic semantics have been given for programming with objects and classes [6], and with abstract data types [5]. Whereas these deal with the general expressivity of the respective concepts, we are only interested in OO in a narrow, back-end sense; namely for the implementation of quadrilaterals, the common denominator of our operational semantics of stream programming, in terms of private state variables and public I/O interfaces.

Coinductive datatypes and their associated recursive functions play an important role in functional programming under the Curry–Howard correspondence: in dependently typed languages such as Agda [16], constructive theorem provers such as Coq [2], (both of which have worrying issues with coinduction [9]), categorical languages such as Charity [4], and "elementary" approaches to total functional programming [15, 21].

Coalgebraic streams have been considered as a foundation for synchronous data-flow programming already in Lucid Synchrone [3]. They remain a viable, if not very popular, alternative to streams-as-functions-of-time approaches, analogously to closed-form versus recursive mathematical definitions of sequences.

A notable high-level approach to coalgebraic streams and natural continuation of [14] is *stream calculus* [12], which extends the classical mathematical framework of infinitesimal calculus to streams, and presents computations as systems of differential equations. However, that approach is both more general than warranted for our present purpose, considering more than just causal stream functions [11], and more mathematical, with a particular emphasis on linear relationships [1, 13].

By contrast, we feel that the operational aspect, namely the adequate representation of low-level loop-based programming patterns that pervade real-world stream processing, has not been fully appreciated. We consider the design and implementation of SIG a well-arguable case in point.

7.2 Outlook

For the sake of accessibility, we have not pushed the coalgebraic framework to the limits of its expressivity. In particular, we have omitted accounts for semantic relations, embodied in the formal concepts of *behavioral equivalence* and *bisimulation*. These would be relevant to the calculation of semantics-preserving program transformations, such as featuring in code optimization passes of the SIG compiler, as well as in the study of alternative solutions to the ones we have calculated as the "most evident".

Nevertheless, we consider the method for the calculation of practically illuminating operational specifications, as applied repeatedly in each of the preceding sections, a valuable tool in the coalgebraic semantics box, as well as promising step towards more general coinductive program synthesis.

References

[1] H. Basold et al. "(Co)Algebraic Characterizations of Signal Flow Graphs". In: *Horizons of the Mind*. Vol. 8464. Lecture Notes in Computer Science. Springer, 2014, pp. 124–145. DOI: 10.1007/978-3-319-06880-0_6.

[2] Y. Bertot and E. Komendantskaya. "Inductive and Coinductive Components of Corecursive Functions in Coq". In: *Electronic Notes in Theoretical Computer Science* 203.5 (2008), pp. 25–47. DOI: 10.1016/j.entcs.2008.05.018.

[3] P. Caspi and M. Pouzet. "A Co-iterative Characterization of Synchronous Stream Functions". In: *Electronic Notes in Theoretical Computer Science* 11 (1998), pp. 1–21. DOI: 10.1016/S1571-0661(04)00050-7.

[4] R. Cockett and T. Fukushima. *About Charity*. Tech. rep. University of Calgary, 1992.

[5] M. Erwig. "Categorical Programming with Abstract Data Types". In: *Proc. AMAST*. Vol. 1548. Lecture Notes in Computer Science. Springer, 1998, pp. 406–421. DOI: 10.1007/3-540-49253-4_29.

[6] B. Jacobs. "Objects and Classes, Co-Algebraically". In: *Object Orientation with Parallelism and Persistence*. 1995, pp. 83–103.

[7] B. Jacobs and J. Rutten. "A Tutorial on (Co)Algebras and (Co)Induction". In: *EATCS Bulletin* 62 (1997), pp. 222–259.

[8] G. Marsaglia. "Xorshift RNGs". In: *Journal of Statistical Software* 8.14 (2003). DOI: 10.18637/jss.v008.i14.

[9] C. McBride. "Let's See How Things Unfold: Reconciling the Infinite with the Intensional (Extended Abstract)". In: *Proc. CALCO*. 2009, pp. 113–126. DOI: 10.1007/978-3-642-03741-2_9.

[10] H. Nilsson, A. Courtney, and J. Peterson. "Functional Reactive Programming, Continued". In: *Proc. Haskell Workshop*. ACM, 2002, pp. 51–64. DOI: 10.1145/581690.581695.

[11] M. Niqui and J. Rutten. "Sampling, Splitting and Merging in Coinductive Stream Calculus". In: *Proc. MPC*. Vol. 6120. Lecture Notes in Computer Science. Springer, 2010, pp. 310–330. DOI: 10.1007/978-3-642-13321-3_18.

[12] J. Rutten. "A coinductive calculus of streams". In: *Mathematical Structures in Computer Science* 15.1 (2005), pp. 93–147. DOI: 10.1017/S0960129504004517.

[13] J. Rutten. "Coalgebraic Foundations of Linear Systems". In: *Proc. CALCO*. Vol. 4624. Lecture Notes in Computer Science. Springer, 2007, pp. 425–446. DOI: 10.1007/978-3-540-73859-6_29.

[14] J. Rutten. "Universal coalgebra: a theory of systems". In: *Theoretical Computer Science* 249.1 (2000), pp. 3–80. DOI: 10.1016/S0304-3975(00)00056-6.

[15] A. Telford and D. Turner. "Ensuring Streams Flow". In: *Proc. AMAST*. 1997, pp. 509–523. DOI: 10.1007/BFb0000493.

[16] *The Agda Wiki*. The Agda Team. 2010. URL: http://wiki.portal.chalmers.se/agda/.

[17] B. Trancón y Widemann and M. Lepper. "Foundations of Total Functional Data-Flow Programming". In: *Proc. MSFP*. Vol. 154. Electronic Proceedings in Theoretical Computer Science. 2014, pp. 143–167. DOI: 10.4204/EPTCS.153.10.

[18] B. Trancón y Widemann and M. Lepper. "On-Line Synchronous Total Purely Functional Data-Flow Programming on the Java Virtual Machine with Sig". In: *Proc. PPPJ*. ACM, 2015, pp. 37–50. DOI: 10.1145/2807426.2807430.

[19] B. Trancón y Widemann and M. Lepper. "The Shepard Tone and Higher-Order Multi-Rate Synchronous Data-Flow Programming in Sig". In: *Proc. FARM*. ACM, 2015, pp. 6–14. DOI: 10.1145/2808083.2808086.

[20] D. Turi and G. D. Plotkin. "Towards a Mathematical Operational Semantics". In: *Proc. LICS*. IEEE, 1997, pp. 280–291. DOI: 10.1109/LICS.1997.614955.

[21] D. A. Turner. "Total Functional Programming". In: *Universal Computer Science* 10.7 (2004), pp. 751–768. DOI: 10.3217/jucs-010-07-0751.

[22] T. Uustalu and V. Vene. "Signals and Comonads". In: *Universal Computer Science* 11.7 (2005), pp. 1310–1326. DOI: 10.3217/jucs-011-07-1311.

[23] T. Uustalu and V. Vene. "The Essence of Dataflow Programming". In: *Proc. APLAS*. Ed. by K. Yi. Vol. 3780. Lecture Notes in Computer Science. Springer, 2005, pp. 2–18. DOI: 10.1007/11575467_2.

[24] P. Wadler. "Comprehending Monads". In: *Proc. LFP*. ACM, 1990, pp. 61–78. DOI: 10.1145/91556.91592.