Foundations of Total Functional Data-Flow Programming

Baltasar Trancón y Widemann
Ilmenau University of Technology, DE
baltasar.trancon@tu-ilmenau.de

Markus Lepper
<semantics/> GmbH, DE

The field of declarative stream programming (discrete time, clocked synchronous, modular, data-centric) is divided between the data-flow graph paradigm favored by domain experts, and the functional reactive paradigm favored by academics. In this paper, we describe the foundations of a framework for unifying functional and data-flow styles that differs from FRP proper in significant ways: It is based on set theory to match the expectations of domain experts, and the two paradigms are reduced symmetrically to a low-level middle ground, with strongly compositional semantics. The design of the framework is derived from mathematical first principles, in particular coalgebraic coinduction and a standard relational model of stateful computation. The abstract syntax and semantics introduced here constitute the full core of a novel stream programming language.

Keywords: coinduction; data flow; stream programming; total functions

1 Introduction and Related Paradigms

For many computation problems a static mapping from input to output values is not sufficient; outputs need to change over time according to corresponding changes in inputs and/or internal state. Applications range from reactive systems, embedded in a context with user input devices, sensors or communication channels, to processors and generators of time-series data, such as audio signals or dynamic simulations in numerous branches of science and engineering.

1.1 Data-Flow Graphs

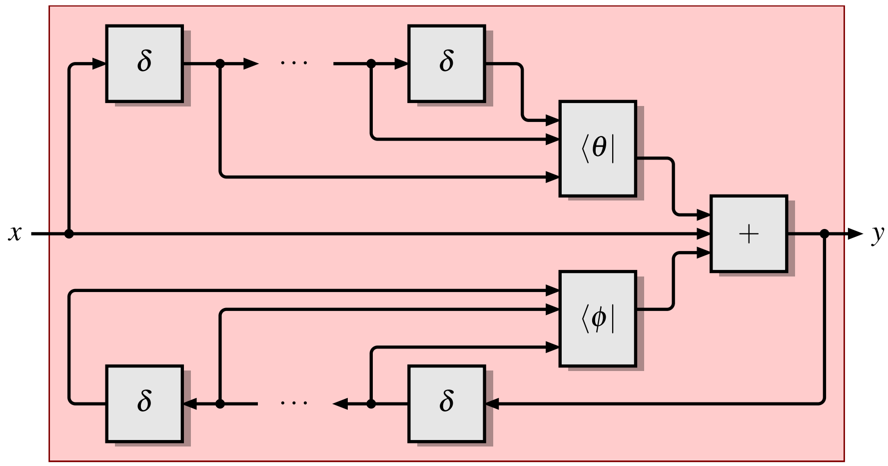
A single paradigm has dominated the landscape of available tools for the declarative construction of such computational systems, and is apparently greatly favoured by domain experts without formal training in programming or software engineering: the visual description in terms of *data-flow graphs*, made up of computational component "boxes" (operations) connected by "wires" (variables). A characteristic feature is that computations intended to act on whole streams are written *as if* they were to act on one element at a time. Memory of previous elements is retained semi-implicitly by *delay* components (either boxes or wires) which, unlike other components, do not denote instantaneous data-flow.

A real-world example is depicted in Figure 1. It specifies the basic structure of the auto-regressive–moving-average (ARMA) model class [2], which is tremendously influential in many empirical sciences, for reasons that should be evident from the shown example model output. The model is classically presented in the shown algebraic form, using formal power series of stream operators. While that form certainly has its merits for the analytic treatment of abstract model properties, it is clear that the data-flow graph of the model, as we draw it here, is more illuminating with respect to concrete algorithmic details.

The data-flow graph approach, in spite of its intuitive appeal, is afflicted with several annoying weaknesses of expressivity:

1. Regular, inverse tree-shaped data flow is far more clumsy than in term notations, in particular when noncommutative operators are concerned.

To appear in EPTCS.



$$\left(1-\sum \phi_n B^n\right) y = \left(1+\sum \theta_n B^n\right) x$$

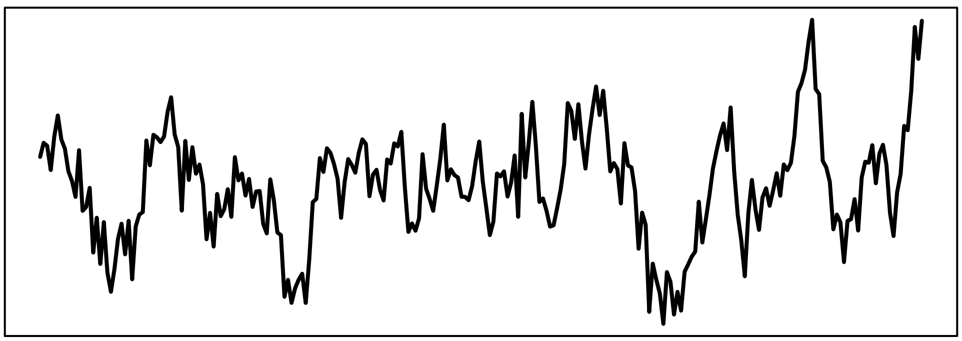


Figure 1: Auto-regressive moving-average (ARMA) model. *Top* – data-flow graph: δ is one-cycle delay; $\langle\phi|$ is auto-regressive linear form with coefficients $\phi_1,\ldots,\phi_p$; $\langle\theta|$ is moving-average linear form with coefficients $\theta_1,\ldots,\theta_q$. *Center* – Classical equational presentation in terms of formal power series of backshift operator B. *Bottom* – simulated output from 250 cycles of white noise input, with coefficients $\langle\phi| = (0.4, 0.3, 0.2, -0.1)$ and $\langle\theta| = (0.3, 0.2, -0.1)$.

2. There is no canonical concept for control issues, such as branching, mode transition, self-configuration, initialization. In terms of data structures, *product* shape is well-supported, whereas *coproduct* shape is not. It is no coincidence that practical systems often provide a distinct, not fully integrated language layer for this purpose; for instance see the Simulink–Stateflow pair [16].
3. The view is heavily biased towards *numeric* data and operations; their *symbolic* counterparts, essential for complex and high-level applications, are treated quite poorly.
4. Static type discipline is rough or entirely missing. A fortiori, software engineering issues, such as *totality* of system functions or specification *refinement* relationships, cannot be reasoned about.

To compound these problems, reference implementations on all levels, from artistic tools [18, 22] to industry-standard modeling frameworks [16], are notorious for their awkward behavior, caused by the conflated treatment of imperative and declarative features.[1]

[1]For instance, see a complex blog discussion on parameter initialization in Simulink at [13].

Some of these problems are slightly improved by switching to a textual representation of data-flow graphs, such as in the Lustre core language [5] of the graphical Scade environment [15]. Others are apparently consequences of the typical low-level machine paradigm underlying the data-flow graph approach.

1.2 Functional (Reactive) Programming

Functional programming eliminates many of the shortcomings discussed above directly, by virtue of its elementary features, such as sound computational purity, static typing, algebraic data types and pattern matching.

A real-world example of these principles in action is depicted in Figure 2. It shows the attack–decay–sustain–release (ADSR) model, which is used in musical synthesis as an amplitude envelope for the basic soundwave of virtual instruments. The model can be described in continous time as a hybrid automaton [7], with four eponymous states. An output signal x ranging over the real unit interval $[0,1]$ is computed from a gate input g over the binary range $\{0,1\}$. The shape of x is controlled by a first-order differential equation per state. Discrete transitions are triggered either extrinsically by the gate, or intrinsically by attainment of threshold values. An implementation as a Haskell (infinite) list program, involving both discretization in time and algebraic encoding of the state space, is a fairly straightforward exercise, as shown.

The paradigm of functional reactive programming (FRP) aims at a uniform functional style for very general time-varying data (signals), no matter whether change is continuous, asynchronously discrete (event-based) or synchronously discrete (clocked). Stream processing arises as a special case, see for instance [21]. FRP algorithms are typically formulated in embedded domain-specific notations hosted in a general-purpose functional language. A main achievement of FRP is a semantic framework that keeps the notoriously difficult enforcement of causality under the hood, by means of a carefully selected set of programming constructs, prescribed by the theory of *arrows* [8].

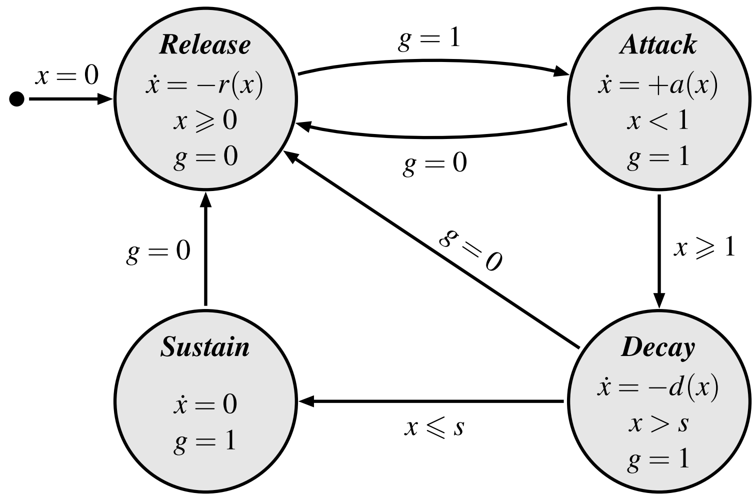
The Haskell ADSR implementation of Figure 2, for all its elegance *as code*, is theoretically, pragmatically and philosophically troubling *as a model*: its *complete* behavior certainly entails the *intended* behavior, but also a great number of additional cases, effected by the ubiquitous possibility of bottom values in the semantic CPOs. We do not believe that domain experts are generally willing, able or well-advised to study domain theory (pardon the pun). There is hardly ever reason for practical stream functions to be anything else than plain old mathematical functions. The phenomenon of bottom values is of course a fundamental feature from the perspective of Turing-complete host environments for stream programming, as well as abstract algebraic calculi that aim for a general recursive theory, such as the work of BROY [3, 4]. But from the perspective of a mathematically educated user, whether an engineer, scientific modeler, or digital artist, is appears more like a defect. We consider this a likely contributing factor to the generally low esteem that computational practitioners have for semantics.

2 Discussion and Outline

2.1 Vision

We approach the problem domain from a different angle, with simultaneously more modest and more ambitious requirements.

On the pragmatical side, we do not abstract from the structure of time the way FRP does. Instead, we assume that the system runs in a clocked synchronous manner. That is, all input and output data are equidistant time series, computations are performed each at a fixed, known clock rate, and data flow is




```
data State = Attack | Decay | Sustain | Release                         -- State space
adsr :: (Num a, Ord a) ⇒ (a → a) → (a → a) → a → (a → a) →              -- a, d, s, r
                    [Bool] → [a]                                        -- gate, out
adsr a_rate d_rate s_level r_rate = loop Release 0                      -- Initialization
  where loop state out (gate : gates) = out′ : loop state′ out′ gates   -- Coinduction
          where out′ = case state of                                    -- Output
                  Attack  → min 1       $ out + a_rate out
                  Decay   → max s_level $ out − d_rate out
                  Sustain →               out
                  Release → max 0       $ out − r_rate out
                state′ = case state of                                  -- Transition
                  Release |   gate                 → Attack
                  Attack  |   gate ∧ out′ ⩾ 1      → Decay
                  Decay   |   gate ∧ out′ ⩽ s_level → Sustain
                  _       | ¬ gate                 → Release
                  _                                → state
```

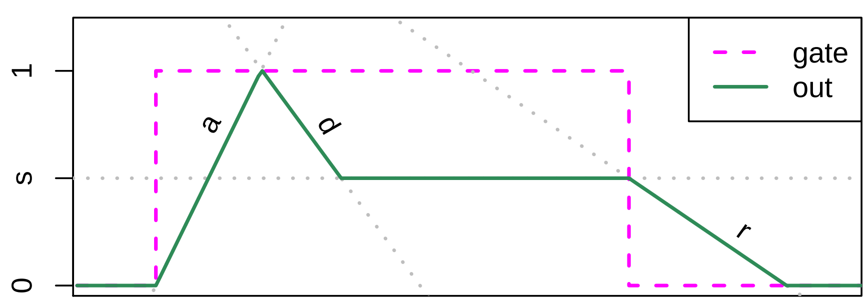


Figure 2: Attack–decay–sustain–release (ADSR) device. *Top* – Hybrid automaton; implicit constraints: $a(x), d(x), r(x) > 0$ except $r(0) = 0$; $0 \leqslant s \leqslant 1$. *Center* – Discrete-time Haskell implementation; adds guards (before $) against overshooting. *Bottom* – Example input/output signal pair; rates a, d, r constant.

instantaneous unless *delayed* explicitly. We allow for feedback loops by delayed circular data flow, but no other forms of recursion. Note that this rules out some paradoxes of concurrent programming, such as the *merge* anomaly; confer [3]. We envisage that subsystems operating at different clock rates may coexist in a single system description, but can be sliced and considered independently for semantic concerns; inter-rate data-flow is only through dedicated resampling connectors. Some clock is nevertheless assumed for every stream; this distinguishes our approach from the recent coalgebraic treatment of index-manipulating (such as sequential splitting and merging) operations on streams [11].

On the theoretical side, we assume a set-theoretical framework. Functions realized by components are total, and observable data never contain bottom values, confer [19]. We require semantics to be fully compositional, such that any syntactical fragment has a meaning in the same domain as the whole. We also require semantics to support logical hierarchies of abstraction and refinement of system descriptions.

On the technical side, we strive for operational compatibility with existing back-end technology, both in software, such as numeric and signal-processing algorithm libraries, and in hardware, such as DSP or FPGA chips. Note that the actual implementation of the semantic framework discussed here on such back-ends is future work.

2.2 Mathematical Tools

We employ the following tools in the construction of our semantical framework:

- TURNER's notion of total functional programming [20] and the corresponding dual modes of inductive reasoning for data, and coinductive reasoning for codata [19];
- RUTTEN's categorial universal coalgebra [14] as a highly abstract foundation of coinduction;
- the relational model of the Z notation [17] for stateful and possibly nondeterministic computation;
- PARDO's monadic coinduction for the unification of the preceding two;
- Static single-assignment (SSA) form [6] for the declarative core representation of programs.

2.3 This Paper

It is not the purpose of this paper to belittle or criticize the efforts of FRP. Instead, we pursue a dialectical programme, deriving an antithesis from distinct and complementary real-world requirements. Since our approach is still very much work in progress, we shall have to leave attempts at a possible synthesis of FRP and our own approach for later. Hence we do not currently draw a more elaborate technical comparison either.

The goal of the present paper is to demonstrate how the semantic ingredients work together, from the perspective of fundamental programming language design. It should not be understood as a recommendation on front-end notation and programming style, or as the description of an implementation strategy or actual compiler. The ongoing work on those aspects shall be discussed in due time in dedicated companion papers. However, the basic design and paradigm of a future, complete and practically useful programming language is specified here as completely as possible within reasonable space limits.

The derivation of our semantic framework from the purely mathematical perspective shall be described in greater detail, including proofs, in a forthcoming companion paper. The present paper is organized as follows: Section 3 introduces the mathematical ingredients of semantic objects and their properties. Section 4 outlines the idea of semantics assignment to compositional descriptions of stream computations, in terms of a novel extension to the well-known SSA form of intermediate languages. Section 5, the focus and main contribution of the present paper, deals with the syntax of a core language and corresponding inductive semantic translation and analysis rules.

3 Mathematical Foundations

3.1 Streams, Set-Theoretically

On behalf of domain experts with classical mathematical training, we agree wholeheartedly with TURNER [20]:

> *The driving idea of functional programming is to make programming more closely related to mathematics.* [...] *The existing model of functional programming, although elegant and powerful, is compromised to a greater extent than is commonly recognised by the presence of partial functions.*

TURNER proposes to distinguish sharply between *data* and *codata* with associated inductive and coinductive models of computation, respectively. In this terminology, streams are the paradigmatic example of a codata structure [19].

A set-theoretic account of streams (that is data of the infinite sequence form A^ω for some set A) and stream functions (that is, total functions of type $A^\omega \to B^\omega$ which obey some form of causality and can be evaluated online) is most elegantly given in terms of categorial coalgebra; confer the seminal work of Rutten [14]. The resulting theory is rather simpler and more regular than its counterparts over CPOs for partial and/or non-strict functional programming.

The endofunctor $\mathbf{S}_A(X) = A \times X$, or $\mathbf{S}_A = A \times -$ for short, on the category **Set** has coalgebras of the form $(S, \langle h,t \rangle)$ with *carrier* or *state space* S and *operations* $h : S \to A$ and $t : S \to S$. The coalgebra $(A^\omega, \langle head_A, tail_A \rangle)$, with $head_A(a) = a_0$ and $tail_A(a)_n = a_{n+1}$ is final. That is,

- the operation $\langle head_A, tail_A \rangle$ is bijective, with inverse $cons_A : A \times A^\omega \to A^\omega$;
- for every coalgebra $(S, \langle h,t \rangle)$ there is a unique *coiteration* homomorphism (or *anamorphism*) $[\![h,t]\!]_A : S \to A^\omega$, given by $[\![h,t]\!]_A(x)_n = h\big(\underbrace{t(\dots(t}_{n}(x))\dots)\big)$.

The access operations *head* and *tail* generalize inductively to $take_A(n) : A^\omega \to A^n$ and $drop_A(n) : A^\omega \to A^\omega$ in the obvious way, with $head = take(1)$ and $tail = drop(1)$. Note that, unlike for the familiar Haskell list counterparts, there are no spurious corner cases; the range of $take(n)$ is exactly A^n. Thus we obtain an *unrolling* similarity transform ${}^{(n)}$ for $n > 0$ on stream functions:

$$f : A^\omega \to B^\omega \implies f^{(n)} : (A^n)^\omega \to (B^n)^\omega \qquad f^{(n)} = chop_B(n) \circ f \circ chop_A^{-1}$$
$$\text{where} \quad chop_A(n) = [\![take_A(n), drop_A(n)]\!]_{A^n}$$

In the same vein, the operation $zip_{AB} : A^\omega \times B^\omega \to (A \times B)^\omega$ is a natural bijection. This is quite useful, because it allows us to reduce stream functions with multiple arguments and/or results to the unary base case. In particular, it gives rise to a tensor-product-like operation $\otimes$ on stream functions, by a similarity transform:

$$\begin{matrix} f : A^\omega \to B^\omega \\ g : C^\omega \to D^\omega \end{matrix} \implies f \otimes g : (A \times C)^\omega \to (B \times D)^\omega \qquad f \otimes g = zip_{BD} \circ (f \times g) \circ zip_{AC}^{-1}$$

Causal stream functions can be defined inductively in the classical way, as precisely the functions $f : A^\omega \to B^\omega$ such that $take_A(n)(x) = take_A(n)(y)$ implies $take_B(n)\big(f(x)\big) = take_B(n)\big(f(y)\big)$ for all n. We write $A^\omega \stackrel{!}{\to} B^\omega$ for the set of all such functions. The causality property implies the existence of lifted operations $take_{AB}(n) : (A^\omega \stackrel{!}{\to} B^\omega) \to A^n \to B^n$ and $drop_{AB}(n) : (A^\omega \stackrel{!}{\to} B^\omega) \to A^n \to A^\omega \stackrel{!}{\to} B^\omega$, such that

$$take_{AB}(n)(f) \circ take_A(n) = take_B(n) \circ f$$
$$uncurry\big(drop_{AB}(n)(f)\big) \circ \langle take_A(n), drop_A(n)\rangle = drop_B(n) \circ f$$

for all n. We abbreviate $head_{AB} = take_{AB}(1)$ and $tail_{AB} = drop_{AB}(1)$ as obvious.

The endofunctor $\mathbf{T}_{AB} = \mathrm{Hom}(A, -) \circ \mathbf{S}_B = (A \to B \times -)$ on the category **Set** has coalgebras of the form $(S, f : S \to A \to B \times S)$, where operations f are in one-to-one correspondence with pairs $h : S \to A \to B$ and $t : S \to A \to S$, namely by $uncurry(f) = \langle uncurry(h), uncurry(t)\rangle$, which we abbreviate to $f = \langle\langle h,t\rangle\rangle$. The coalgebra $(A^\omega \stackrel{!}{\to} B^\omega, \langle\langle head_{AB}, tail_{AB}\rangle\rangle)$ is final. That is,

- the operation $\langle\langle head_{AB}, tail_{AB}\rangle\rangle$ is bijective, with inverse $cons_{AB} : \big(A \to B \times (A^\omega \stackrel{!}{\to} B^\omega)\big) \to A^\omega \stackrel{!}{\to} B^\omega$;
- for every coalgebra $(S, \langle\langle h,t\rangle\rangle)$ there is a unique *coiteration* homomorphism $[\![h,t]\!]_A : S \to A^\omega$, given by $[\![h,t]\!]_{AB}(x)(a)_n = h\big(\underbrace{t(\dots(t}_{n}(x)(a_0))\dots)(a_{n-1})\big)(a_n)$.

Note that the isomorphism $A^\omega \stackrel{!}{\to} B^\omega \cong A \to B \times (A^\omega \stackrel{!}{\to} B^\omega)$ also plays a central role in continuation-based implementations of FRP, see [10], which arrive at the same structure by different arguments.

3.2 Stateful Relations

As we have just seen, causal stream functions can be represented coalgebraically by a hidden state space S and an operation f with type $uncurry(f) : S \times A \to B \times S$, which takes pre-states and inputs to post-states and outputs. In stream programming, we generally wish to keep state under the hood. A traditional functional solution in the spirit of MOGGI [9] would be to massage the operation to obtain type $A \to S \to B \times S$, which can then be studied in the Kleisli category $\mathbf{Kl}(M)$ of a state monad $M = S \to - \times S$. However, that approach is orthogonal to the coalgebraic one, seeing as they fix complementary type parameters.

Hence we pursue a different direction, by noting the similarity to the representation of state-based systems in the formal notation Z [17], where pre-state, input, output and post-state variables are distinguished by decorating suffixes, namely nothing, ?, ! and ′, respectively. There, state is handled explicitly, which we shall emulate by inference en route from a front-end notation to the core calculus of our proposed framework. How the algebra of Z schemas relates to the theories of monads and arrows prevailing in FRP has, to our knowledge, not been established precisely. The former is certainly less abstract, and founded more on pragmatic utility and trusted first-order logic than axiomatic elegance. However, a framework of quaternary stateful relations can be put to good use in explicating the fundamentals of stream programming semantics, as we shall presently outline.

The algebra of stateful relations is best explained visually. Each relation can be depicted as a quadrilateral box, where the sides correspond to variable roles by convention: I/O and state transition are layed out left-to-right and top-to-bottom, respectively; see Figure 3. The four sides can in general each have product structure, corresponding to zero or more variables, and be connected in arbitrary ways.

Figure 3: Stateful relations: *Left* – general graphical layout. *Right* – single-step delay component δ.

For practical purposes, however, it is more desirable to consider building blocks with more disciplined behavior. For instance, stateless operations (the vast majority in many stream function definitions) have zero state variables and hence unit state. The most fundamental nontrivial stateful relation is the single-step delay, which transfers input to post-state and pre-state to output, respectively; see also Figure 3.

There are two sensible ways to encode relations,

1. as tuple subsets of $S \times A \times B \times S$, thereby inheriting the Boolean lattice structure of entailment that gives a hierarchy of behavioral descriptions, essential both a priori for nondeterministic specifications and refinement, and a posteriori for program abstraction and slicing;
2. equivalently and more conveniently for the following computational discussions, as set-valued maps of type $S \times A \to M(B \times S)$, where M is some variant of the powerset monad. These maps can be studied in the respective Kleisli category $\mathbf{Kl}(M)$, which yields general relations for the unrestricted powerset $M = \mathbf{P}$, and left-total/functional relations for the restrictions $M = \mathbf{P}_+/\mathbf{P}_1$ to nonempty/singular subsets, respectively. Note that the latter is merely a "disguised" equivalent of the identity monad; it helps to address functions as particular relations in the common relational language. For the differentiated use of all three candidates, see section 3.4 below.

Either way, we use the graphically evocative notation $s \,/\, a \mapsto b \,/\, s'$ for pointwise reasoning. It comes with two mnemonic aids:

1. s (top) above a (left) goes to b (right) above s' (bottom);
2. the I/O relation is $a \mapsto b$, embedded in the state context $s \,/\, \ldots \,/\, s'$.

In the visual data-flow representation, the boxes depicting relations are truly three-dimensional objects. There are three distinct composition operations on stateful relations that make intuitive sense. Each is associative and admits neutral elements, thus giving a monoidal structure (up to type compatibility). See Fig. 4. The rules become more evidently convincing when compared to the visual form. Note that ordinary relational composition, simultaneously affecting both state and I/O, has no place in this calculus.

With respect to stream computations, the state axis of our relations is special because, although flat on the element level, a feedback loop is implied on the stream level, with post-state of each step becoming pre-state of the following. Informally, an element-level relation R induces a stream-level function g as

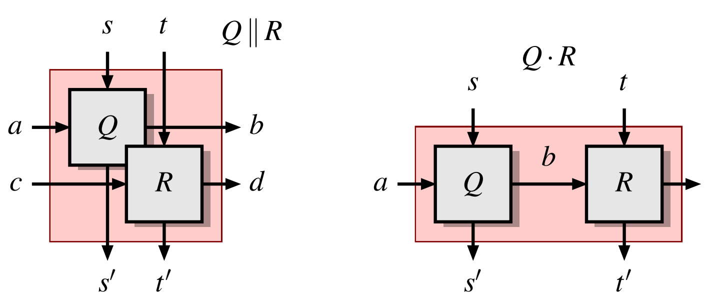
$$\text{“}\quad g = \lim_{n \to \omega} \underbrace{R \,;\, \ldots \,;\, R}_{n} \quad\text{”}$$

There is no obvious way, such as a topology, to take that limit literally. We shall demonstrate in the following subsection that the coalgebraic approach can be extended to solve the corresponding fixpoint equation $g \cong R \,;\, g$ in a canonical way.

$$\frac{s \,/\, a \xmapsto{Q} b \,/\, s' \qquad t \,/\, c \xmapsto{R} d \,/\, t'}{s,t \,/\, a,c \xmapsto{Q \| R} b,d \,/\, s',t'} \qquad\qquad / \xmapsto{I} /$$

$$\frac{s \,/\, a \xmapsto{Q} b \,/\, s' \qquad t \,/\, b \xmapsto{R} c \,/\, t'}{s,t \,/\, a \xmapsto{Q \cdot R} c \,/\, s',t'} \qquad\qquad /\, a \xmapsto{J_A} a \,/$$

$$\frac{s \,/\, a \xmapsto{Q} b \,/\, s' \qquad s' \,/\, c \xmapsto{R} d \,/\, s''}{s \,/\, a,c \xmapsto{Q ⨾ R} b,d \,/\, s''} \qquad\qquad s \,/ \xmapsto{K_S} /\, s$$



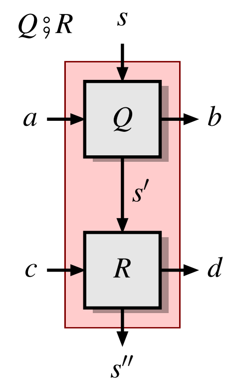


Figure 4: Stateful relational compositions, with inductive definitions and neutral elements

3.3 Relational Coinduction

The framework of monadic coinduction, although introduced specifically over categories of CPOs by PARDO [12], carries over nicely to our stateful relations over **Set**. All three candidate functors $M = \mathbf{P}, \mathbf{P}_+, \mathbf{P}_1$ admit the required structure:

- They are commutative strong monads with strength $\tau_{X,Y} : X \times M(Y) \to M(X \times Y)$. Note that τ_X for any fixed X is a distributive law of $\mathbf{S}_X$ over M.
- They come with distributive laws for the functors $\mathrm{Hom}(X,-)$ for any X, namely $\lambda_{X,Y} : \big(X \to M(Y)\big) \to M(X \to Y)$, where a function space $A \to B$ is seen as the Cartesian product $\prod_{a \in A} B$. Note that the case $M = \mathbf{P}_+$ corresponds to the axiom of choice. This law establishes a correspondence of stateful relations with $\mathbf{T}_{A,B}$-coalgebras over the Kleisli category $\mathbf{Kl}(M)$:

$$R : S \times A \to M(B \times S) \implies \underbrace{\lambda_{A,B\times S} \circ curry(R)}_{f_R} : S \to M(\underbrace{A \to B \times S}_{\mathbf{T}_{A,B}(S)})$$

- The two distributive laws composed yield a distributive law for $\mathbf{T}_{A,B}$ over M that satisfies suitable coherence laws to induce a lifting $\widehat{\mathbf{T}}_{A,B}$ of $\mathbf{T}_{A,B}$ from the category **Set** to $\mathbf{Kl}(M)$.

$$\widehat{\mathbf{T}}_{A,B}(X) = \mathbf{T}_{A,B}(X) \qquad\qquad f : X \to M(Y) \implies \widehat{\mathbf{T}}_{A,B}(f) : \widehat{\mathbf{T}}_{A,B}(X) \to M\big(\widehat{\mathbf{T}}_{A,B}(Y)\big)$$

Thus prepared, we can specify M-monadic $\mathbf{T}_{A,B}$-coinduction following [12]: To a $\mathbf{T}_{A,B}$-coalgebra (S, f) over $\mathbf{Kl}(M)$, and in particular to one arising from a stateful relation, we associate homomorphisms g such that the following diagram commutes, where $\_^\star$ denotes Kleisli extension:

$$\begin{array}{ccc} S & \xrightarrow{\quad g \quad} & M(A^\omega \xrightarrow{!} B^\omega) \\ {\scriptstyle f}\big\downarrow & & \big\downarrow{\scriptstyle M(cons_{AB}^{-1})} \\ M\big(\mathbf{T}_{A,B}(S)\big) & \xrightarrow[\widehat{\mathbf{T}}_{A,B}(g)^\star]{} & M\big(\mathbf{T}_{A,B}(A^\omega \xrightarrow{!} B^\omega)\big) \end{array}$$

This diagram does not generally have a unique solution, and [12] discusses the existence and possible choices over the category **CPO** in detail. We need to choose a canonical solution $g = (\!| f |\!)^M_{AB}$ over the category **Set** instead. Fortunately, the existence argument carries over from **CPO**. Noting that the right vertical arrow is the invertible operation of the final $\mathbf{T}_{A,B}$-coalgebra, we find the desired fixpoint equation:

$$g = \underbrace{M(cons_{AB}) \circ \widehat{\mathbf{T}}_{A,B}(g)^\star \circ f}_{\Phi(g)}$$

The candidate monad $M = \mathbf{P}$ carries a complete lattice structure that can be raised to the S-power. The operator Φ is easily seen to be monotone on this structure. Hence TARSKI's theorem guarantees a nonempty choice of solutions, including two canonical choices, namely the least and the greatest fixpoint. As a general rule, least fixpoints are preferred over **CPO** in general, and for induction over **Set**, but greatest fixpoints are preferred for coinduction over **Set**. This rule of thumb is confirmed here:

- the least fixpoint is the empty relation, a lifting of the fact the the initial $\mathbf{S}_A$-algebra is also empty;
- the more restricted candidate monad $M = \mathbf{P}_+$ has the same upper bounds, but generally no lower bounds.

The most restricted candidate monad $M = \mathbf{P}_1$ has unique solutions by equivalence to the identity monad. Hence we can uniformly choose the greatest fixpoint in either case.

In summary, monadic coinduction takes each (general/left-total/functional) stateful relation canonically to a (general/left-total/functional) relation between initial states and possible observable behavior, that is, causal I/O stream functions.

$$R : S \times A \to M(B \times S) \implies [\![R]\!]^M = (\!| f_R |\!)^M : S \to M(A^\omega \xrightarrow{!} B^\omega)$$

To our knowledge, this is a novel application of monadic coinduction.

3.4 Compositional Denotational Semantics

Our approach to semantics poses a dialectical problem: On one hand, we intend to give total stream function semantics to box-like program components. On the other hand, we require semantics for program fragments to be compositional, including individual pattern matching rules, which are partial by their very essence. The synthesis that solves this problem involves a differentiation of several semantic meaning functions.

Let $\mathbb{V}[\![x]\!]$ denote the range of possible values for a variable x. This can be established by declared or inferred type information, or simply be the universe of terms in the untyped case. Note that, in a total functional setting, no undefined value $\bot$ is included by default. Instead, we write explicitly

$\mathbb{V}_\perp[\![x]\!] = \mathbb{V}[\![x]\!] \uplus \{\perp\}$ for the extension. Range assignment lifts to vectors of variables by taking the non-strict Cartesian product:

$$\mathbb{V}[\![x_1,\dots,x_n]\!] = \mathbb{V}[\![x_1]\!] \times \cdots \times \mathbb{V}[\![x_n]\!] \qquad \mathbb{V}_\perp[\![x_1,\dots,x_n]\!] = \mathbb{V}_\perp[\![x_1]\!] \times \cdots \times \mathbb{V}_\perp[\![x_n]\!]$$

It is possible for variables to have unit range $\mathbb{V}[\![x]\!] = \{\top\}$. While these have no effect on data flow in a total functional setting, the extension $\mathbb{V}_\perp[\![x]\!] = \{\top,\perp\}$ serves as a primitive Boolean type, which we shall exploit for reducing control flow to data flow. Hence we call variables *control* variables, if they have unit range, and *data* variables otherwise. In the data-flow graph notation, control wires are indicated by dashed arrows. See Figures 8 and 13 below for examples.

Then we can give a hierarchy of three element-wise semantics of (fragments of) stream programs, in terms of stateful relations. Each of these uses a different candidate monad; stream-wise semantics can be obtained by monadic coinduction at each level:

1. The compositional but partial, *internal* relational semantics of a program fragment with interface $s\,/\,x \to y\,/\,s'$, where all four meta-variables stand for vectors of object-level variables, is a left-total relation $R : S \times A \to \mathbf{P}_+(B \times S)$ with $A = \mathbb{V}_\perp[\![x]\!]$, $B = \mathbb{V}_\perp[\![y]\!]$ and $S = \mathbb{V}_\perp[\![s]\!] = \mathbb{V}_\perp[\![s']\!]$. Left-totality is appropriate because, in the general case that includes both partial fragments and nondeterministic specifications, each post-state and output variable can be undefined, or take one or more possible values, or both, but clearly not neither. By coinduction we obtain a nonempty set of possible stream functions for each initial state:

$$R : S \times A \to \mathbf{P}_+(B \times S) \implies [\![R]\!]^{\mathbf{P}_+} : S \to \mathbf{P}_+(A^\omega \xrightarrow{!} B^\omega)$$

   Note that the strongly synchronous model of computation which we have adopted avoids the well-known paradoxes of nondeterministic concurrent computation, such as the *merge* anomaly and the Brock–Ackermann anomaly, which could ruin compositionality in a more general setting.

2. At box boundaries, we additionally consider the *external* relational semantics, which are totalized along the I/O axis by component-wise exclusion of $\perp$. Clearly, the result is no longer generally left-total; hence we obtain a subrelation $R^\dagger : S \times A^\dagger \to \mathbf{P}(B^\dagger \times S)$ with $A^\dagger = \mathbb{V}[\![x]\!] \subseteq A$ and $B^\dagger = \mathbb{V}[\![y]\!] \subseteq B$.

   The transformation has two effects: Firstly, an output variable can no longer be considered undefined if it could also take some value; thus definedness is maximized individually. Secondly, all output variables need to be defined simultaneously for a valid output tuple.

   This is already the desired element-wise semantics for possibly nondeterministic specifications of whole components. Since $\perp$ is no valid data element in inter-component flow, components can communicate regardless of their respective error behavior. This is particularly beneficial in the presence of legacy code and/or hybrid hardware–software implementations. By coinduction we obtain a possibly empty set of possible stream functions for each initial state:

$$R^\dagger : S \times A^\dagger \to \mathbf{P}(B^\dagger \times S) \implies [\![R^\dagger]\!]^{\mathbf{P}} : S \to \mathbf{P}\big((A^\dagger)^\omega \xrightarrow{!} (B^\dagger)^\omega\big)$$

   Note that state is *not* totalized for compositionality reasons: the internal state variables of a box can be undefined at any time, otherwise we could not give semantics to local delayed data flow in pattern matching rules. However, undefined values may not leak to outputs.

3. At the level of a specification or abstraction of a program, it may be permitted for a functional block to mean zero or multiple causal stream functions. At the level of the concrete program, a deterministic meaning is required, which may constrain admissible pre- and post-states. This problem amounts to finding an acceptable state space $D \subseteq S$, typically by exclusion of $\bot$ from selected components, such that the corresponding domain-restricted subrelation is functional; namely $s \in D$ implies $R^\dagger(s \,/\, x) = \{y \,/\, s'\}$ with $s' \in D$ also. We write $R^{[D]} : S \times A^\dagger \to \mathbf{P}_1(B \times S)$. By coinduction we obtain a singleton set of possible stream functions for each initial state:

$$R^{[D]} : S \times A^\dagger \to \mathbf{P}_1(B^\dagger \times S) \implies [\![R^{[D]}]\!]^{\mathbf{P}_1} : S \to \mathbf{P}_1\big((A^\dagger)^\omega \xrightarrow{!} (B^\dagger)^\omega\big)$$

   The question can of course be hard to decide in general. We give no algorithm or guarantees for arbitrary, undisciplined use of partiality and nondeterminism. However, if these features are used in the sense of total functional programming, with partiality within each pattern matching rule, and harmless static nondeterminism between non-overlapping alternatives, then the problem reduces to standard analyses of pattern matching.

See Figure 9 below for a worked-out example.

To abstract from initial states is a typical tactic of the coalgebraic approach, in contrast to classical automata theory. In a concrete program, of course, an initial value needs to be supplied or inferred. We envisage both delay nodes annotated with their initial state, and thus equivalent to the traditional *followed-by* data-flow operator, and default values such as zero for numeric types.

3.5 Laws of Composition

The three axes of relational composition should correspond to basic operations on the stream function level:

$$\begin{aligned}
[\![R \parallel S]\!]^M(s,t) &= \{f : [\![S]\!]^M(s); g : [\![S]\!]^M(t) \bullet f \otimes g\} \\
[\![R \cdot S]\!]^M(s,t) &= \{f : [\![R]\!]^M(s); g : [\![S]\!]^M(t) \bullet g \circ f\} \\
[\![\underbrace{R \,;\, \cdots \,;\, R}_{n>0}]\!]^M(s) &= \{f : [\![R]\!]^M(s) \bullet f^{(n)}\}
\end{aligned}$$

While these laws are readily proven for $M = \mathbf{P}_1$ by equivalence to ordinary causal stream functions and coinduction, the situation for $M = \mathbf{P}, \mathbf{P}_+$ is more complicated. We leave these as conjectures for future validation, or falsification, of our theoretical framework. The simplest case is the most important for program structure in the large, anyway, as discussed in section 3.4 above.

Note that the last rule, although less general than the preceding two, is of great practical importance, because it enables the code optimization technique known as *loop unrolling*, which is essential for buffer-based high-performance implementations of stream functions.

4 Semantic Strategy

As we have just discussed, element-level relational semantics are the stepping stone for stream function semantics. Thus we need to assign stateful relations to programs, preferrably by induction over the expression structure. To this end, we shall give a core program calculus and suitable normalization

$$\frac{f(x_1,\dots,x_n) = y_1,\dots,y_m}{/\, x_1,\dots,x_n \overset{f}{\longmapsto} y_1,\dots,y_m \,/} \qquad \frac{f(x_1,\dots,x_n) = y}{/\, y \overset{f^{-1}}{\longmapsto} x_1,\dots,x_n,\top \,/} \qquad \frac{y \notin \operatorname{ran} f}{/\, y \overset{f^{-1}}{\longmapsto} \underbrace{\bot,\dots,\bot}_{n},\bot \,/}$$

$$\frac{}{s \,/\, x \overset{\delta}{\longmapsto} s \,/\, x}$$

Figure 5: Semantic building blocks (data flow)

algorithms. They reduce program expressions in either functional or data-flow style to a logical form, which essentially names interface variables and lists primitive computation operations as assignments to their respective target variables. The intended meaning is then the largest relation on the possible values of interface variables that satisfies the assignments as constraints.

Before the full formal details of inductive interpretation are given in the next section, we briefly introduce the constraint building blocks associated with primitive operations. References to other user-defined components are non-recursive. Hence they can be unfolded by inlining at the semantic level, at do not require special consideration.

4.1 Data Flow

The interpretation of data flow primitives is depicted in Figure 5. Delay operations (δ) are resolved as discussed above. Primitive operations on data values are stateless functions (f), and lifted to the stateful relational level in the obvious way. Data term constructors are special among these operations, as they are injective and have pairwise disjoint ranges. They have partial inverse functions (f^{-1}) with pairwise disjoint domains. In order to handle partiality, we lift inverse constructors to the relational level by adding a control output to indicate definedness. This is in particular necessary for nullary constructors.

For programs written in a declarative front-end formalism, variables of control type are not specified by the user, but arise only in this synthetic way. Whereas inverse constructors act as their sources, special control flow primitives are introduced to act as their sinks.

4.2 Control Flow

The interpretation of control flow primitives is depicted in Figure 6. As we have already criticized in the introduction, there is no canonical control flow construct in data-flow graph approaches to programming. Therefore we only consider pattern matching, in the form of a *case* expression, confer [1], as a universal control flow construct of non-recursive functional programming. To this end, we adapt the basic idea of the SSA form for imperative programming to a more data-centric scenario.

Results of alternative branches are joined using virtual “Phony” (ϕ) nodes, which are usually thought of as choosing the appropriate case nondeterministically in the static perspective, but deterministically in the dynamic perspective, because only one option is available at each instance. Unlike SSA, however, we do not assume alternative branches to be taken by single-threaded execution of conditional jumps, but instead we exploit the absence of side effects in declarative programs, and allow for speculative evaluation in parallel. This gives more freedom for implementation strategies, and may even make sense literally for hardware or near-hardware back-ends. In this view, ϕ nodes choose nondeterministically from their defined input, if any.

$$\frac{y \in \{x_1,\ldots,x_n\} \setminus \{\bot\}}{/\, x_1,\ldots,x_n \stackrel{\phi}{\longmapsto} y \,/} \qquad \frac{\{x_1,\ldots,x_n\} \subseteq \{\bot\}}{/\, x_1,\ldots,x_n \stackrel{\phi}{\longmapsto} \bot \,/}$$

$$\frac{\bot \notin \{c_1,\ldots,c_k\}}{/\, x, c_1,\ldots,c_k \stackrel{\gamma}{\longmapsto} x/} \qquad \frac{\bot \in \{c_1,\ldots,c_k\}}{/\, x, c_1,\ldots,c_k \stackrel{\gamma}{\longmapsto} \bot/}$$

Figure 6: Semantic building blocks (control flow)

$$\begin{aligned}
\mathit{Expr} &::= () \mid \mathit{Expr}\,,\mathit{Expr} \mid \mathit{Var} \mid \mathit{Val} \mid \mathit{Op}(\mathit{Expr}) \\
&\quad \mid \textbf{let}\ \mathit{Vars} := \mathit{Expr}\ \textbf{in}\ \mathit{Expr} \mid \textbf{case}\ \mathit{Expr}\ \textbf{of}\ \mathit{Rule} \\
\mathit{Vars} &::= () \mid \mathit{Vars}\,,\mathit{Vars} \mid \mathit{Var} \\
\mathit{Op} &::= \mathit{Fun} \mid \mathit{Cons} \mid \mathit{Cons}^{-1} \mid \delta \mid \gamma \mid \phi \\
\mathit{Rule} &::= \mathit{Pat} \to \mathit{Expr} \mid \mathit{Rule} \sqcup \mathit{Rule} \\
\mathit{Pat} &::= () \mid \mathit{Pat}\,,\mathit{Pat} \mid \mathit{Var} \mid \mathit{Cons}(\mathit{Pat}) \\
\mathit{Abs} &::= \lambda\,\mathit{Rule} \mid [\mathit{Face}\ \textbf{where}\ \mathit{Form}] \\
\mathit{Face} &::= \mathit{Vars}\ /\ \mathit{Vars} \to \mathit{Vars}\ /\ \mathit{Vars} \\
\mathit{Form} &::= \top \mid \bot \mid \mathit{Form} \wedge \mathit{Form} \mid \mathit{Form} \vee \mathit{Form} \\
&\quad \mid \exists\,\mathit{Var}\,\mathit{Form} \mid \mathit{Vars} := \mathit{Expr} \mid \mathit{Var} = \bot \mid \mathit{Var} \neq \bot
\end{aligned}$$

		1	**2**	**3**
Expr	$\mathit{Expr}\,,\mathit{Expr}$	•	○	○
	$\mathit{Op}(\mathit{Expr})$	•	○	○
	let ⋯ **in** ⋯	•	–	–
	case ⋯ **of** ⋯	•	–	–
Op	Cons^{-1}	–	•	•
	δ	•	–	–
	$\gamma \mid \phi$	–	•	–
Rule		•	–	–
Pat		•	–	–
Abs	$\lambda\,\mathit{Rule}$	•	–	–
Form	$\bot \mid \mathit{Form} \vee \mathit{Form}$	–	–	•
	$\mathit{Var} = \bot \mid \mathit{Var} \neq \bot$	–	–	•

Figure 7: Core calculus abstract syntax. *Left* – Grammar rules. *Right* – Special forms (synopsis): allowed •; restricted ○; forbidden –.

Choices from alternative values are then to be made from concurrent pattern matching rules, with the definedness of each rule conditional on the successful matching of *all* its pattern constructors. The control aspect of choice is reflected back to the data flow view by our novel extension to SSA: "Guard" (γ) nodes propagate a single data input, subject to strictness in one or more additional control inputs. They act as control flow sinks, corresponding to inverse constructors as sources, and perform the selection of (results of) alternative branches.

5 Core Calculus

The abstract syntax of the core calculus is depicted in Figure 7 (left). Many of the nonterminals and productions are self-explanatory.

- Issues of concrete syntax, such as operator precedence, are not considered. Various brackets are added in concrete examples for disambiguation.
- Sufficient and distinct stores of variables, defined function names, and constructor names are

referred to by *Var*, *Fun* and *Cons*, respectively.

- Literals *Val* denote some primitive values of interest, including the undefined value $\perp$ as discussed in sections 3.4 and 4.
- The nonterminals *Expr*, *Vars* and *Pat* are meant to form free monoids with associative operator $(,)$ and neutral element $()$.
- Operators of the form $Cons^{-1}$, γ, δ and ϕ are special, in the sense that they do not belong to the ordinary functional programming fragment of the language; they are explained in detail below.
- The rule combinator $\sqcup$ denotes general nondeterministic choice. It is nevertheless useful in deterministic programs to combine mutually exclusive rules. It follows that no resolution rule for overlapping patterns, such as first-fit or best-fit, is implied. An asymmetric first-fit combinator can be added with little difficulty, but is omitted here for simplicity.
- The square bracket form of function abstraction, reminiscent of data-flow boxes, is a symmetric variant of λ that names outputs as well as inputs. It is more suitable for the direct denotation of data-flow graphs, in particular where the flow is not tree-shaped or outputs are subject to delayed feedback.
- The notation for the nonterminal *Face*, specifying interface variables, mimics the tuple notation for stateful relations.

The notation is subject to a few static sanity conditions on the use of variables:

- *Linearity* – Variables left of the arrow of a *Rule* or *Face* are assumed to be pairwise distinct.
- *Barendregt convention* – Variables occuring in a *Pat* or bound by $\exists$ in a *Form* are assumed to be pairwise distinct, and disjoint from the free variables.
- *Single assignment* – Variables assigned to by $:=$ in an *Expr* or *Form* are assumed to be pairwise distinct.
- *Determinism* – In a concrete program, all variables in an *Abs*, except for specified pre-states and inputs, are assigned to by $:=$. Note that, in a loose specification, an unassigned variable is understood to take *all* possible values instead; hence we can simply take the intersection of overlapping partial specifications.

Note that **let** expression may be recursive at face value. That the corresponding, apparently circular data flow is properly causal shall be ensured after a suitable normalization, see 5.2 below.

The comma monoid structure of expressions, variables and patterns caters for the inherent parallel compositionality of data flow, both in terms of wires grouped into "buses", and boxes with varying arity or juxtaposed. This static *shape* information is considered distinct from (product) *type* information. The former is treated in the following semantic rule system (as usual; confer [3]), whereas the latter is ignored for simplicity:

Arities of all operations are assumed to be statically known, and combine associatively; for instance, given the arity information $f : 3 \to 2$, $g : 2 \to 1$ and $h : 0 \to 2$, the assignment $y,z := f\big(g(w,x),h()\big)$ is well-shaped. Drawing the corresponding data-flow graph is left as an exercise to the reader. Note that named variables always have arity 1.

By contrast, we do not assume a particular system of data types, but instead provide for generic term-shaped data, by expressions of the form $Cons(Expr)$. This data universe is general enough to accommodate most first-order data structures, and to support pattern matching as a universal control-flow construct.

The large-scale structure of programs beyond the expression level is assumed to be extremely simple:

- We assume that no higher-order or nested functions are present. If they are supported by a high-level front-end notation, they should be eliminated by lambda lifting and defunctionalization first. This simplifies the semantic discussion greatly, but is also supported by pragmatic considerations; for instance, it is far from clear what a stream function-valued stream function actually means.
- We do not allow for circular references among user-defined functions. This rules out all forms of explicit recursion. This is not as much of a loss as one might assume, since only *instantaneous* recursion within the computation of a single stream element is affected; a feature that stream algorithms rarely require, in particular in real-time environments.
- Thus the basic form of a closed program is a sequence of defining pairs $Fun = Abs$, where right-hand-side *Fun* references are only to preceding definitions.

We define three distinct special forms as inductive sublanguages of the full core language, depicted synoptically in Figure 7 (right), as stages of a program transformation that achieves the unification of the functional paradigm with the data-flow paradigm on one hand, and the front-end paradigms with the relational back-end paradigm on the other. The three forms focus on functional front-end, SSA or flat data-flow graph, and pure first-order logic, respectively. We give translation rules between the forms as syntax-directed rewriting algorithms.

As a running example of the forms and translations, we shall use the *sample-and-hold* component. This small stream processing component has a single signal input x and output y, and either feeds the current input through (*sample*), or retains the previous output (*hold*), depending on some switching condition, typically a trigger input t taking values from $\{\mathsf{S},\mathsf{H}\}$. The sample-and-hold component plays an important role in many low-level stream computations, in particular for audio signals. It has the charming property that each of the features that we propose to add to traditional data-flow programming, is used in a minimal but nontrivial way. The only downside is that the example makes only trivial use of pattern matching, but readers can surely extrapolate from their experience with more complex case distinctions.

To illustrate hierarchical specification, implicit state and compositionality, Figure 8 depicts a variety of views on the component: a loose nondeterministic specification that abstracts from the switching condition, with either implicit or explicit state, as well as a refined deterministic component fragment that presupposes switching, and the corresponding pattern matching front-end. The concrete program arises by composition of the latter two. It is given in each of the three special forms of our core calculus in turn, in a form that arises by the given canonical translations up to straightforward algebraic–logic simplifications.

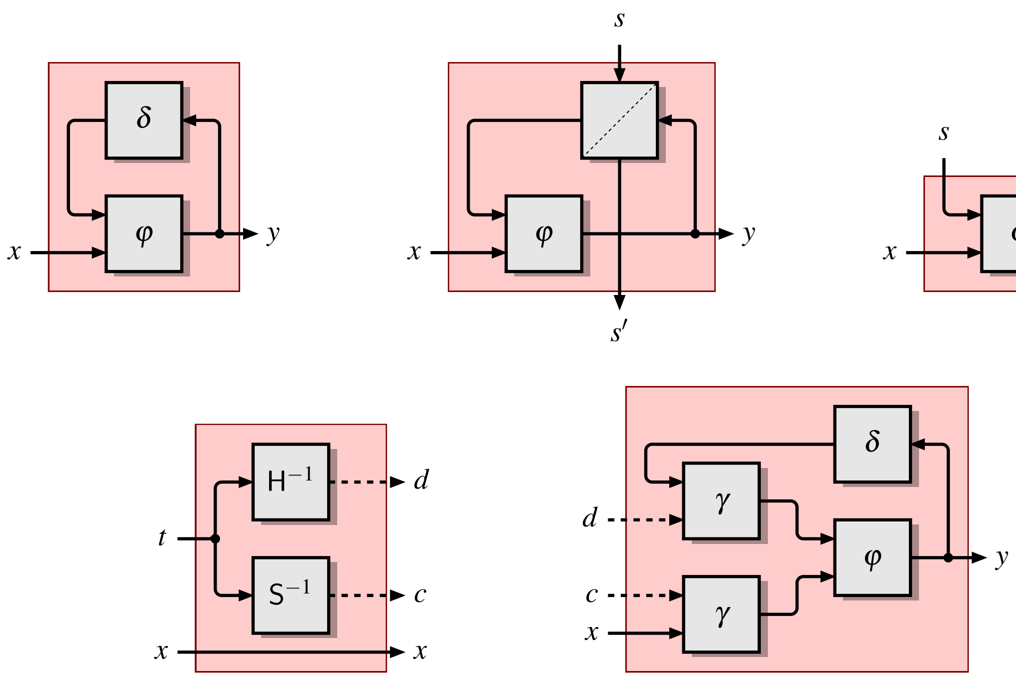
5.1 First Form

A program is in *first* form, if and only if the following constraints are satisfied:

- No operation symbol of the form $Cons^{-1}$, γ or ϕ occurs in expressions.
- Interface declarations *Face* are of the form *Vars*, and do not mention state variables.
- Logical formulas use the operators $\top$, $\wedge$, $:=$ and $\exists$ only.

This is the form of elementary functional and data-flow programming. It is already more low-level than one would like for a real programming or specification language front-end, although many more advanced features can be reduced to this form by standard program transformations, as discussed above. It already allows to mix the functional and data-flow paradigms compositionally.

A corresponding instance of the running example is depicted in Figure 8. It uses the data-flow box abstraction instead of functional λ, because of the very common pattern that uses delayed outputs as inputs in a feedback loop, which is slightly awkward to express in functional style due to anonymous



$$sah\_prog_1 = \left[x,t \to y \textbf{ where } y := \textbf{ case } t \textbf{ of } \{\mathsf{S}() \to x \sqcup \mathsf{H}() \to \delta(y)\}\right]$$

$$sah\_prog_2 = \left[s \,/\, x,t \to y \,/\, y \textbf{ where } \exists c,d,v,w \begin{pmatrix} c := \mathsf{S}^{-1}(t) & v := \gamma(x,c) & \\ & & y := \phi(v,w) \\ d := \mathsf{H}^{-1}(t) & w := \gamma(s,d) & \end{pmatrix} \right]$$

$$sah\_prog_3 = \left[s \,/\, x,t \to y \,/\, y \textbf{ where } \exists c,d,v,w \begin{pmatrix} c := \mathsf{S}^{-1}(t) & (c \neq \bot \wedge v := x) \vee (c = \bot \wedge v := \bot) \\ d := \mathsf{H}^{-1}(t) & (d \neq \bot \wedge w := s) \vee (d = \bot \wedge w := \bot) \\ & (y := v \vee y := w) \\ & (v = \bot \vee w = \bot \vee y \neq \bot) \end{pmatrix} \right]$$

Figure 8: Sample and hold. *Top* – loose specification with implicit (*left*) and explicit state and eliminated delay, before (*center*) and after (*right*) copy propagation. *Center* – functionally partial implementation with external control (*right*) and frontend for functionally complete implementation with internal control (*left*). *Bottom* – Concrete program in first, second and third form, respectively.

results. Internally the behavior of the box is defined in terms of functional pattern matching, which makes for a much more natural expression than any ad-hoc data-flow switching mechanism, and scales better to nontrivial case distinctions than if-then-else. State is still implicit, the appearance of element-wise computation is maintained.

5.2 Second Form

A program is in *second* form, if and only if the following constraints are satisfied:

- No **let** or **case** construct appears in expressions.

- No operation symbol of the form δ occurs in expressions.
- Interface declarations *Face* are of the form *Vars* / *Vars*, and do mention state variables.
- Logical formulas use the operators $\top$, $\wedge$, $:=$ and $\exists$ only.
- Expressions are flat: any term of the form $Op(Expr)$ is actually of the simpler form $Op(Var^*)$.
- Assignments are atomic: the expression operator $(,)$ does not occur in right operands of $:=$.
- Assignments are non-circular.

This is the basic exchange format for program analysis and code generation. State is made explicit, putting all computationally relevant information together. Structure is reduced to atomic assignments, either copying values or storing results of a single operation. The form is inspired by standard form, such as A-normal form of functional programs, or SSA form of imperative programs. It adopts the ϕ nodes for converging data flow from the latter, but uses a different encoding of control flow, to be explained below.

A corresponding instance of the running example is depicted in Figure 8. It has two inverse constructors as sources of control flow, two γ nodes as the corresponding sinks and one ϕ node for choice. It also corresponds directly to the horizontal composition of the two fragments depicted in the center row of Figure 8.

From the second form, data-flow graphs can be drawn, by reading the conjunction of assignments as a specification of adjacency, and element-level relational semantics can be read off by taking formulas as constraints in standard first-order logic.

For the running example, relational element-level semantics and their coinductive lifting to streams are depicted in Figure 9. The definition of relation R corresponds directly to the second form of the sample-and-hold program. It has not been simplified in any way, in order to make the structure stand out. The subsequent relation $R^\dagger$ shows the removal of bottom values. Primitive relations have been eliminated, by substituting their definitions, or alternatively by translation to third form; see below.

Additionally, stream function semantics are given, deterministically for the concrete program, and nondeterministically for the loose specification depicted in the top row of Figure 8, respectively. For the former, the state space is restricted, in the sense discussed in Section 3.4, by exclusion of $\bot$. Note that, from the strongly formal viewpoint, these self-referential presentations are a step backwards; to verify that thay are well-defined implicitly invokes coinduction.

5.3 Reduction from First to Second Form

A simple production-wise comparison of the first and second form reveals the tasks to be solved by a translation:

1. State must be made explicit by elimination of δ operations and introduction of state variables. The solution idea is already depicted in Figure 3. The data flow of the component must become non-circular in the process, otherwise the source program is ill-formed.
2. Composite applicative expressions must be decomposed into their operations, by explicitly binding all intermediate values to variables.
3. Variable-binding expressions and abstractions, and hence pattern matching rules, must be eliminated entirely. This entails a translation of patterns to applications by a reversal of data flow, replacing *Cons* with $Cons^{-1}$.

$$
\begin{aligned}
R = \big\{ & s,v,w,x,y : A \uplus \{\bot\}; t : \{\mathsf{S},\mathsf{H},\bot\}; c,d : \{\top,\bot\} \\
& \big|\ /\, t \overset{\mathsf{S}^{-1}}{\mapsto} c \,/ \wedge /\, t \overset{\mathsf{H}^{-1}}{\mapsto} d \,/ \wedge /\, x,c \overset{\gamma}{\mapsto} v \,/ \wedge /\, s,d \overset{\gamma}{\mapsto} w \,/ \wedge /\, v,w \overset{\phi}{\mapsto} y \,/ \\
& \bullet\, s \,/\, x,t \mapsto y \,/\, y \big\} \\
R^{\dagger} = \big\{ & s,x,y : A; t : \{\mathsf{S},\mathsf{H}\} \ \big|\ (t = \mathsf{S} \Rightarrow y = x) \wedge (t = \mathsf{H} \Rightarrow y = s) \bullet s \,/\, x,t \mapsto y \,/\, y \big\}
\end{aligned}
$$

$$
[\![R^{[A]}]\!]^{\mathbf{P}_1}(s) = \{h\} \quad \text{with} \quad cons\big((x,t),a\big) \overset{h}{\mapsto} cons\big(y, h(y)(a)\big) \qquad \text{where} \qquad y = \begin{cases} x & t = \mathsf{S} \\ s & t = \mathsf{H} \end{cases}
$$

$$
[\![Q^{\dagger}]\!]^{\mathbf{P}}(s) = \Big\{ h : A^{\omega} \overset{!}{\to} A^{\omega} \ \Big|\ \big(\forall a : A^{\omega}; n : \mathbb{N} \bullet h(a)_n \in \{a_n, a'_n\}\big) \Big\} \qquad \text{where} \qquad a' = cons(s, h(a))
$$

Figure 9: Relation semantics of sample-and-hold. *Top* – deterministic element-level; first partial (compositional), then total (boundary), simplified. *Bottom* – stream-level; first deterministic program, then nondeterministic specification.

Both first and second form share a simple structure of logical formulas. Since existential quantification can always be lifted from a conjunction, thanks to the Barendregt convention even without renaming, they admit a prenex normal form: A (commutative) sequence of existential quantifiers introducing local variables, followed by a (also commutative, thanks to single assignment) sequence of assignments.

The reduction algorithm specified below is given in terms of syntax-directed rewrite rules. It is slightly unconventional, in the sense that context information is collected strictly bottom-up. Hence contexts appear on the right rather than the left hand sides of rewrite rules, and specify variables contributed by the translation target, rather than depended upon by the translation source. A few details of the notation need explanation:

- Meta-level variables c and $s,\dots,z$ stand for vectors of zero or more pairwise distinct object-level variables. Individual variables are selected by subscripts if necessary. Following the tradition of linear algebra, subscripts i, j range from 1 to m,n, respectively. Other meta-variables stand for singular language fragments.
- Static checking of shape information boils down to comparisons of the length of variable vectors. We write $x : n$ to state that vector x contains n variables, and $x \sim y$ to state that vectors x and y contain the same number of variables.
- Judgements with bottom-up context, making up the main antecedents and the conclusions of rewrite rules, take the form $P \implies \Gamma \vdash Q$, where P, Q are fragments of the source/target form, respectively, possibly belonging to different nonterminals, and Γ is a collection of free variables of Q, sorted into different roles. In order to distinguish the roles, we adapt and extend the interface notation of stateful relations from section 3.2: $s \,/\, x \to y \,/\, s' \parallel c$ has variables s,x,y,s',c for pre-state, input, output, post-state and control, respectively. In order to reduce typographical clutter, we omit parts of this sequence, separators and all, if they contain zero variables, according to the following nesting, indicated by horizontal lines:
$$\overline{\underline{s\,/}\ x \to}\ y\ \underline{/\,s'}\ \underline{\parallel c}$$

The naive bottom-up rewriting strategy has the advantage that the problem of static shape inference is solved automatically. The minor downside is that, since no binding information is propagated downwards,

$$\frac{}{() \implies \vdash \top}\text{Unit} \qquad \frac{\begin{matrix} s \sim s' & & t \sim t' \\ a \implies s \mathbin{/\!\!\to} x \,/\, s' \vdash A & & b \implies t \mathbin{/\!\!\to} y \,/\, t' \vdash B \end{matrix}}{a,b \implies s,t \mathbin{/\!\!\to} x,y \,/\, s',t' \vdash A \wedge B}\text{Agg}$$

$$\frac{x \sim y : 1 \quad y \text{ fresh}}{x \implies y \vdash x := y}\text{Ref} \qquad \frac{\begin{matrix} s \sim s' \quad x : n \quad y : m \quad t \sim t' : k \quad f : k \,/\, n \to m \,/\, k \quad t,t',y \text{ fresh} \\ a \implies s \mathbin{/\!\!\to} x \,/\, s' \vdash A \end{matrix}}{f(a) \implies s,t \mathbin{/\!\!\to} y \,/\, s',t' \vdash \exists x\big(A \wedge y \,/\, t' := f(t \,/\, x)\big)}\text{App}$$

$$\frac{\begin{matrix} s \sim s' \quad x \sim y \sim t \sim t' : m \quad t,t',y \text{ fresh} \\ a \implies s \mathbin{/\!\!\to} x \,/\, s' \vdash A \quad B \equiv \bigwedge_i (y_i := t_i \wedge t'_i := x_i) \end{matrix}}{\delta(a) \implies s,t \mathbin{/\!\!\to} y \,/\, s',t' \vdash \exists x(A \wedge B)}\text{Delay}$$

$$\frac{\begin{matrix} s \sim s' & t \sim t' & x \sim y : n \\ a \implies s \mathbin{/\!\!\to} x \,/\, s' \vdash A & b \implies t \mathbin{/\!\!\to} z \,/\, t' \vdash B & C \equiv \bigwedge_j (y_j := x_j) \end{matrix}}{\textbf{let } y := a \textbf{ in } b \implies s,t \mathbin{/\!\!\to} z \,/\, s',t' \vdash \exists x,y(A \wedge B \wedge C)}\text{Let}$$

$$\frac{\begin{matrix} s \sim s' & t \sim t' & x \sim y : n \\ a \implies s \mathbin{/\!\!\to} x \,/\, s' \vdash A & r \implies t \,/\, y \to z \,/\, t' \vdash B & C \equiv \bigwedge_j (y_j := x_j) \end{matrix}}{\textbf{case } a \textbf{ of } r \implies s,t \mathbin{/\!\!\to} z \,/\, s',t' \vdash \exists x,y,c(A \wedge B \wedge C)}\text{Case}$$

Figure 10: Core calculus reduction from first to second form (Expressions)

the resulting algorithm tends to produce redundant copies of variables. Similar effects are known from naive SSA algorithms [6]. This is not an issue unless one actually builds a compiler. Results can of course be simplified with a standard copy propagation pass. To the contrary, the default strategy to create fresh variables bottom-up has nice properties from the perspective of algorithmic analysis, wich are easy to verify on a rule-wise basis:

1. The sanity conditions on variable use are preserved.
2. Variables introduced in translation are either $\exists$-quantified locally, or propagated upwards in the context.

The rewrite rules concerning expressions are depicted in Figure 10. Expressions are generally rewritten to logical formulas, with a bottom-up context of pre-states, outputs and post-states, but no inputs or controls.

- Rules (Unit) and (Agg) collect contexts and logical clauses. Associativity is easy to see.
- Rule (Ref) creates a fresh copy of a variable inherited from the source.
- Rules (App) and (Delay) decompose nested terms by introducing variables x for the intermediate values, and t,t' for the state of the referenced function. Many primitive functions, including constructors, are stateless with $k = 0$. m-ary delay is reduced to $2m$ copying assignments as depicted in Figure 3.
- Rules (Let) and (Case) are essentially data-flow sequential compositions.

The rewrite rules concerning patterns are depicted in Figure 11. Patterns are generally rewritten to logical formulas, with a bottom-up context of inputs and controls, but no states or outputs. Rules mimic

$$\frac{}{() \Longmapsto \;\vdash \top}\,\text{Unit'} \qquad \frac{\begin{array}{cc} s \sim s' & t \sim t' \\ p \Longmapsto x \rightarrow\!\| \; c \vdash A & q \Longmapsto y \rightarrow\!\| \; d \vdash B \end{array}}{p,q \Longmapsto x,y \rightarrow\!\| \; c,d \vdash A \wedge B}\,\text{Agg'}$$

$$\frac{x \sim y : 1 \qquad y \text{ fresh}}{x \Longmapsto y \rightarrow\!\vdash y := x}\,\text{Ref'} \qquad \frac{\begin{array}{c} x : m \qquad y \sim d : 1 \qquad f : m \rightarrow 1 \qquad y,d \text{ fresh} \\ p \Longmapsto x \rightarrow\!\| \; c \vdash A \end{array}}{f(p) \Longmapsto y \rightarrow\!\| \; c,d \vdash \exists x\big(A \wedge x,d := f^{-1}(y)\big)}\,\text{App'}$$

Figure 11: Core calculus reduction from first to second form (Patterns)

$$\frac{\begin{array}{cc} & s \sim s' \\ p \Longmapsto x \rightarrow\!\| \; c \vdash A & a \Longmapsto s \; / \!\rightarrow y \; / \; s' \vdash B \end{array}}{p \rightarrow a \Longmapsto s \; / \; x \rightarrow y \; / \; s' \; \| \; c \vdash A \wedge B}\,\text{Match}$$

$$\frac{\begin{array}{c} s \sim s' \qquad v : \ell \times n \qquad w \sim x : \ell \times m \qquad y : n \qquad z : m \qquad x,y,z \text{ fresh} \\ r_h \Longmapsto s_{h*} \; / \; v_{h*} \rightarrow w_{h*} \; / \; s'_{h*} \; \| \; c_{h*} \vdash A_h \quad \text{for all } h \\ B \equiv \bigwedge_{h,j}(v_{hj} := y_j) \qquad C \equiv \bigwedge_{h,i}\big(x_{hi} := \gamma(w_{hi}, c_{h*})\big) \qquad D \equiv \bigwedge_i\big(z_i := \phi(x_{*i})\big) \end{array}}{\bigsqcup_h r_h \Longmapsto s \; / \; y \rightarrow z \; / \; s' \vdash \exists c,v,w,x(A \wedge B \wedge C \wedge D)}\,\text{Choice}$$

Figure 12: Core calculus reduction from first to second form (Rules)

their expression counterparts closely, except for (App'), which introduces a control d to indicate the success of matching against constructor f, and exploits the statelessness of f for simplification.

The rewrite rules concerning pattern matching rules are depicted in Figure 12. Rules are generally rewritten to logical formulas; the bottom-up context is full, including controls for lower-level rule (Match), but excluding controls for higher-level rule (Choice).

- Rule (Match) plugs together the complementary context of left and right hand side.
- Rule (Choice) is by far the most complex part of the algorithm. It composes subrules speculatively in parallel, by distributing the common inputs (formula B), associating outputs with local control conditions (formula C), and collecting the alternative outputs in transposed grouping (formula D). Each output variable is guarded with the collected controls of the same subrule by a γ operation. The guarded outputs are joined by ϕ operations.

  The schema is depicted in Figure 13.[2] Note that meta-variables c, s, s', v, w, x stand for matrices of object-level variables. The additional subscript h ranges from 1 to ℓ. A subscript $*$ selects a whole row or column.

  Even though the choice operator $\sqcup$ is given as binary in the abstract syntax for simplicity, it is *not* associative; rewrite rule (Choice) is to be applied only once for a whole, flat sequence of zero or more pattern matching rules. Note that the degenerate case $\ell = 0$ will result in constantly undefined outputs z_i flowing from nullary ϕ nodes.

The rewrite rules concerning component abstractions and logic formulas are depicted in Figures 14 and 15, respectively. Abstractions, being independent top-level program constructs, are rewritten to

[2]This diagram also gives a drastic illustration of the limitations of the data-flow graph approach for control flow.

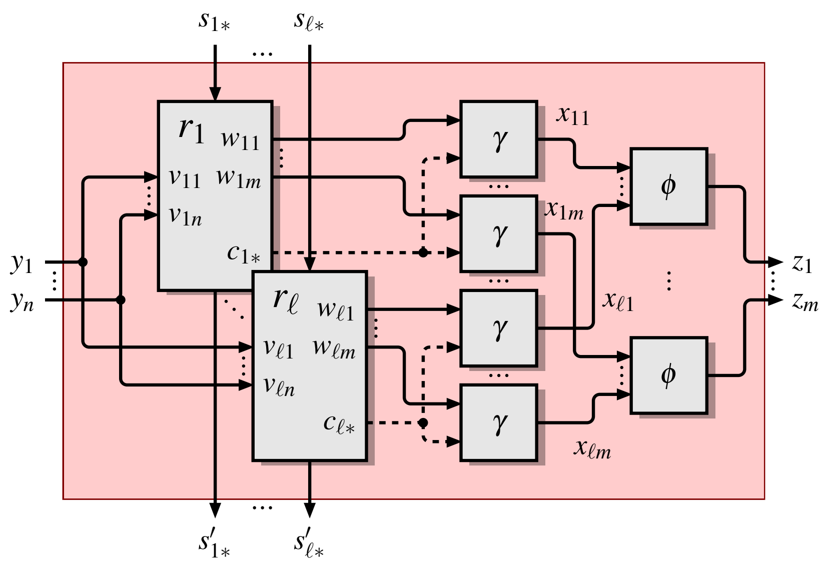


Figure 13: Effect of rule (Choice), graphically

$$\frac{\begin{array}{c} s \sim s' \\ r \implies s \,/\, x \to y \,/\, s' \vdash A \end{array}}{\lambda\, r \implies [s \,/\, x \to y \,/\, s' \textbf{ where } A]}\ \text{Lambda}$$

$$\frac{\begin{array}{ccc} s \sim s' & w \sim y : n & x \sim z : m \\ A \implies s \,/\, w \to x \,/\, s' \vdash A' & B \equiv \bigwedge_j (w_j := y_j) & C \equiv \bigwedge_i (z_i := x_i) \end{array}}{[y \to z \textbf{ where } A] \implies [s \,/\, y \to z \,/\, s' \textbf{ where } \exists w, x(A' \wedge B \wedge C)]}\ \text{Box}$$

Figure 14: Core calculus reduction from first to second form (Abstractions)

abstractions with no context. Formulas are rewritten to formulas with full bottom-up context except controls. All rules are fairly straightforward. Note that rule (Exists) exploits the Barendregt convention to skip checks for variable capture, since w is disjoint from s, s', x, y.

5.4 Third Form

A program is in *third* form, if and only if the following constraints are satisfied:

- No **let** or **case** construct appears in expressions.
- No operation symbol of the form δ, γ, ϕ occurs in expressions.
- Interface declarations *Face* are of the form *Vars* / *Vars*, and do mention state variables.
- Expressions are flat: any term of the form $Op(Expr)$ is actually of the simpler form $Op(Var^*)$.
- Assignments are atomic: the expression operator $(,)$ does not occur in right operands of $:=$.

This is the fundamental logical form of component behavior. It contains no operations except for primitive functions and their inverses; all administrative nodes have been eliminated. Hence, on the

$$\frac{}{\top \Longmapsto \;\vdash \top}\text{True} \qquad \frac{\begin{array}{cc} s \sim s' & t \sim t' \\ A \Longmapsto s \,/\, w \to x \,/\, s' \vdash A' & B \Longmapsto t \,/\, y \to z \,/\, t' \vdash B' \end{array}}{A \wedge B \Longmapsto s,t \,/\, w,y \to x,z \,/\, t,t' \vdash A' \wedge B'}\text{And}$$

$$\frac{\begin{array}{cc} s \sim s' & x \sim y : n \\ a \Longmapsto s \,/\!\to y \,/\, s' \vdash A & B \equiv \bigwedge_j (x_j := y_j) \end{array}}{x := a \Longmapsto s \,/\!\to x \,/\, s' \vdash \exists y(A \wedge B)}\text{Assign} \qquad \frac{\begin{array}{c} s \sim s' \\ A \Longmapsto s \,/\, x \to y \,/\, s' \vdash A' \end{array}}{\exists w A \Longmapsto s \,/\, x \to y \,/\, s' \vdash A'}\text{Exists}$$

Figure 15: Core calculus reduction from first to second form (Formulas)

$$\frac{\begin{array}{cc} x,y : 1 & c : m \\ C \equiv (\bigwedge_{i=1}^m c_i \neq \bot) & D \equiv (\bigvee_{i=1}^m c_i = \bot) \end{array}}{y := \gamma(x,c) \Longmapsto (C \wedge y := x) \vee (D \wedge y := \bot)}\text{Guard}$$

$$\frac{\begin{array}{cc} x : n & y : 1 \\ C \equiv (\bigvee_{j=1}^n y := x_j) & D \equiv (\bigvee_{j=1}^n x_j = \bot) \end{array}}{y := \phi(x) \Longmapsto (C \vee y := \bot) \wedge (D \vee y \neq \bot)}\text{Phony}$$

Figure 16: Core calculus reduction from second to third form

theoretical level, it serves as a demonstration of the "virtual" nature of the synthetic nodes we have introduced.

On the practical level, the third form is obvious much less suited to analyses and transformation that aim at program *execution*, because properties such as single assignment are obscured. On the other hand, it may be found useful for other kind of analysis. For instance, the problem of well-defined, that is, complete and non-overlapping, pattern matching rules can be solved in this form using standard SAT techniques.

5.5 Reduction from Second to Third Form

A comparison of the second and third form yields a single translation task, namely the elimination of γ and ϕ nodes. In return, disjunction and tests for $\bot$ can be added to logical formulas.

The translation rules are depicted in Figure 16. This translation is algorithmically much simpler than the preceding one, because it involves no rearrangement of variables, and can hence be presented in context-free form. It is a straightforward realization of the specification in Figure 6. Additional, omitted rules for deep rewriting are trivial.

Note that the laws of propositional logic give two equivalent views on the result of rule (Guard). For simplicity consider the single-control case $m = 1$. The given semantics of the assignment $y := \gamma(x,c)$ can be transformed to a more "guard-like form", using the expansion of the primitive $c \neq \bot$ to $\neg(c = \bot)$:

$$(c \neq \bot \wedge y := x) \vee (c = \bot \wedge y := \bot) \iff (c \neq \bot \Rightarrow y := x) \wedge (c = \bot \Rightarrow y := \bot)$$

6 Conclusion

Starting from domain-specific and rigidly mathematical requirements, we have outlined an inter-paradigm design that unifies data-flow and total functional styles for stream programming. We have given a formal semantical framework in terms of established concepts, namely (monadic) coinduction over set-theoretic coalgebras, and Z-style stateful step relations. These concepts have been combined in a novel way, and mapped to a full-fledged core programming calculus, as the fundamental step of programming language design.

The calculus has the expressivity to accomodate pure programs in either paradigm, as well as pragmatic and fine-grained mixtures. Transformation rules have been given to reduce programs to a constraint form where their relational semantics can be read off directly. We have demonstrated how to embed the data-flow view straightforwardly into a state-of-the-art declarative intermediate representation of programming languages (SSA). Thereby we inherit a large body of knowledge concerning both reduction of front-end language features, and compilation to back-end platforms.

A core calculus with precise semantics may be a major and essential step towards the realization of a paradigm, but of course there remains a lot of future work.

On the theoretical level, the finer implications of set-theoretic monadic coinduction are still poorly understood. We point to the unproven conjectures regarding the three axes of relational composition in Section 3.5 as an open example problem.

On the level of algorithmic expressivity, our strict prohibition of general recursion need to be reconsidered carefully. Experience shows that, for most applications in the domains of interest, non-recursive algorithms with single-step delay nodes suffice: the coinductive lifting from element-wise to stream calculation does all the rest. But in some illuminating corner cases, more powerful means of feedback may be required.

On the level of programming language engineering, our approach is readily compatible with many standard procedures, both regarding front-end issues such as type systems and diagnostics, and back-end code generation techniques. On the other hand, due to the rigidity of semantics, it steers clear of some notorious implementation difficulties, such as the non-compositionality of operation scheduling in the presence of arbitrary instantaneous feedback; confer [5]. That these hopes are indeed justified shall be demonstrated by a forthcoming compiler. Another challenging open problem is the design of a type system that enforces our srong assumptions in a way that is flexible, unobtrusive and practically useful.

Acknowledgments

Signal graphs for the ARMA and ADSR models in Figures 1 and 2, respectively, have been drawn using the free statistical software system R.

References

[1] L. Augustsson. "Compiling pattern matching". In: *Functional Programming Languages and Computer Architecture*. Vol. 201. Lecture Notes in Computer Science. Springer, 1985, pp. 368–381.

[2] G. E. Box and G. M. Jenkins. *Time series analysis: Forecasting and control*. San Francisco: Holden–Day, 1970. ISBN: 0816210942.

[3] M. Broy. "Nondeterministic data flow programs: how to avoid the merge anomaly". In: *Science of Computer Programming* 10 (1988), pp. 65–85.

[4] M. Broy and G. Ştefănescu. “The algebra of stream processing functions”. In: *Theoretical Computer Science* 258.1–2 (2001), pp. 99–129. ISSN: 0304-3975. DOI: 10.1016/S0304-3975(99)00322-9.

[5] P. Caspi et al. “Lustre: A Declarative Language for Programming Synchronous Systems”. In: *POPL*. ACM Press, pp. 178–188. DOI: 10.1145/41625.41641.

[6] R. Cytron et al. “Efficiently computing static single assignment form and the control dependence graph”. In: *TOPLAS* 13.4 (1991), pp. 451–490. DOI: 10.1145/115372.115320.

[7] T. A. Henzinger. “The Theory of Hybrid Automata”. In: *Proc. Logic in Computer Science (LICS '96)*. IEEE Computer Society, 1996, pp. 278–292. ISBN: 0-8186-7463-6. DOI: 10.1109/LICS.1996.561342.

[8] J. Hughes. “Generalising monads to arrows”. In: *Sci. Comput. Program.* 37.1-3 (2000), pp. 67–111. DOI: 10.1016/S0167-6423(99)00023-4.

[9] E. Moggi. “Notions of Computation and Monads”. In: *Information and Computation* 93.1 (1991), pp. 55–92. DOI: 10.1016/0890-5401(91)90052-4.

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

[11] M. Niqui and J. Rutten. “Stream processing coalgebraically”. In: *Science of Computer Programming* 78 (2013), pp. 2192–2215. DOI: 10.1016/j.scico.2012.07.013.

[12] A. Pardo. “Monadic Corecursion – Definition, Fusion Laws and Applications”. In: *ENTCS* 11 (1998), pp. 105–139. DOI: 10.1016/S1571-0661(04)00055-6.

[13] G. Rouleau and S. Popinchalk. *Initializing Parameters*. Matlab Central Blog. Retrieved 2013-12-31. 2008. URL: http://blogs.mathworks.com/seth/2008/12/25/initializing-parameters/.

[14] J. 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] *Scade Suite*. Esterel Technologies. 2013. URL: http://www.esterel-technologies.com/products/scade-suite/.

[16] *Simulink, Dynamic System Simulation for Matlab — Using Simulink*. The MathWorks. 2000. URL: http://www.mathworks.com.

[17] J. M. Spivey. *The Z Notation: a reference manual*. International Series in Computer Science. Prentice Hall, 1988. ISBN: 0139785299.

[18] *Supercollider Homepage*. 2011. URL: http://supercollider.sourceforge.net/.

[19] A. Telford and D. A. Turner. “Ensuring streams flow”. In: *Algebraic Methodology and Software Technology*. Ed. by M. Johnson. Vol. 1349. Lecture Notes in Computer Science. Springer, 1997, pp. 509–523. ISBN: 978-3-540-63888-9. DOI: 10.1007/BFb0000493.

[20] D. A. Turner. “Total Functional Programming”. In: *Univ. Comput. Sci.* 10.7 (2004), pp. 751–768. DOI: 10.3217/jucs-010-07-0751.

[21] Z. Wan and P. Hudak. “Functional reactive programming from first principles”. In: *SIGPLAN Not.* 35.5 (2000), pp. 242–252. ISSN: 0362-1340. DOI: 10.1145/358438.349331.

[22] S. Wilson, D. Cottle, and N. Collins. *The Supercollider Book*. The MIT Press, 2011. ISBN: 9780262232692. URL: http://supercolliderbook.net.