Functional Stream Semantics for a Synchronous Block-Diagram Compiler

Timothy Bourke*†, Paul Jeanmaire*†, Marc Pouzet†*
*Inria Paris
†École normale supérieure, CNRS, PSL University

***Abstract*—Synchronous block-diagram languages have long been formalized as fixpoints of equations defining stream functions. We apply this approach to a compiler verified in an interactive theorem prover, allowing us to restate its end-to-end correctness theorem: if a program is accepted and has no runtime errors, its input/output behavior is preserved in the generated code. In a functional semantics, it is necessary to model all possible behaviors, including erroneous ones. We show that static typing and dependency analyses correctly rule out all errors except those arising from logical and arithmetic operators. Our definitions supplement existing formal ones, especially for the reset operator, which is both useful in itself and a basis of advanced control structures.**

***Index Terms*—Interactive Theorem Proving, Denotational semantics, Synchronous languages, Program correctness**

I. Introduction

Dataflow synchronous languages, notably Lustre [26] and Scade 6 [20], are used to specify safety-critical embedded control software. They are sometimes called "block-diagram" languages due to the graphical syntax often used in practice where components are represented by labelled rectangles and the connections between them by lines. They are, in fact, functional languages that define mappings from streams of inputs (values received from sensors or other components) to streams of outputs (values sent to other components or actuators), whose compilers produce code that executes in bounded time and memory. The Vélus project [6] formalizes a standard compilation scheme [2] and proves it correct in the Rocq interactive theorem prover [22]. It generates Clight programs that CompCert [3], [31] then compiles to assembly code. An overall correctness theorem states that the input/output behavior of the source stream semantics is reproduced step-by-step by the imperative semantics of the generated assembly code. It only applies, however, to well-behaved programs, defined implicitly as those that satisfy the semantic rules. Since these rules inherit the definitions of arithmetic and logical operations from Clight, some of which are only partially defined, a program is only well-behaved if it is free of run-time errors. In this article, we present an alternative stream semantics, for a core dataflow language with resettable function instances, where errors and fixpoints are explicitly defined.[1] This gives an explicit characterization of well-behaved programs allowing us to prove that the type systems and static analyses formalized in the compiler eliminate many erroneous behaviors, and giving a basis for reasoning about the remaining run-time errors. We show that the stream semantics satisfies the relational rules used in the compiler proof and thereby strengthen the overall correctness theorem.

[1]All definitions and proofs have been formalized in the Rocq interactive theorem prover. They are available at https://velus.inria.fr/lics2025/.

A. A Dataflow Synchronous Language

The dataflow synchronous language with resettable function instances that we study is introduced here via an example program. Its abstract syntax follows in section II-A and its formal semantics is the subject of the rest of the article.

The example is the *rising edge retrigger* [38, §1.2.2], which is typical of operators provided by libraries for control applications. It comprises three functions. The first, called `countdown`, maps two streams `n` and `r` to a stream `t`. The comments show the result for example inputs.

```
node countdown(n : int;   -- 4 4 4 4 4 5 5···
               r : bool)  -- F F F T F F T···
returns (t : int)         -- 4 3 2 4 3 2 5···
let
 t = case r               -- F F F T F F T
       (true => n)        -- 4 4 4 4 4 5 5
       (false =>
         n fby (t - 1));  -- 4 3 2 4 3 2 5
tel
```

A function, or *node*, is defined by a set of equations. Here there is only one. The stream `t` takes its value from `n` initially and whenever `r` is true, and otherwise decreases with rate 1. The `fby` ("followed-by") primitive [41] prepends the first value of the stream at left, `n`, to the stream at right, `t - 1`.

A second node, `actdef`, illustrates the sampling operators `when` and `merge`. The declaration of `x` indicates that it is only available when `k` is true. In the comments, the example values of `x` are aligned with the true values of `k` as we consider a *synchronous semantics* in which a program is ultimately executed cyclically like a digital circuit.

```
node actdef(k : bool;          -- F F T F T T F
            x : int when k)    --     4   3 2
returns (y : int)              -- 0 0 4 0 3 2 0
let
 y = merge k                   -- F F T F T T F
       (true => x)             --     4   3 2
       (false => 0 when not k);-- 0 0   0     0
tel
```

The `merge` consumes a value from one stream, here `k`, to determine from which of the branches to consume a value. We will see that the synchronous semantics imposes a strict alignment on the input streams, which is why the `when not k`

in the second argument is necessary. In contrast, the `case` operator in the `countdown` function consumes values from all streams and uses the value from the first one to decide which of the others to propagate.

A third node, with shortened name `rer`, defines the operator itself: a rising edge on `i` (re)triggers `n` true values on `o`.

```
node rer(i : bool;                    -- F T T T F
         n : int)                     -- 2 2 2 2 2
returns (o : bool)                    -- F T T F F
var r, c : bool; d : int when c;
let
 r = i and (false fby (not i));       -- F T F F F
 c = r or (false fby o);              -- F T T T F
 d = countdown((n, r) when c);        --   2 1 0
 o = actdef(c, d) > 0;                -- F T T F F
tel
```

There are three local variables: `r`, `c`, and `d`. The equation for `r` detects a rising edge by comparing successive input values:

```
                    i   F  T  T  T  F  ···
 false fby (not i)      F  T  F  F  F  T  ···
-------------------------------------------
            . and .     F  T  F  F  F  F  ···
```

The stream `c` is true only when an edge is detected or the output was previously true. The equation for `d` uses an instance of `countdown` to count backward from `n` to 0 from each rising edge. The instance is applied to the streams for `n` and `r` filtered according to `c`. The result `d` is thus also synchronized with the instants when `c` is true. It is resynchronized with the input streams using `actdef` to insert a 0 value whenever `c` is false.

The possibility to restart the count down is provided explicitly by the interface and definition of `countdown`. This works well here, but, in libraries and larger programs, adding and propagating such signals is tedious and inefficient. This problem is solved in languages like Scade 6 by constructions for modular reinitialization. For instance, we could declare:

```
node countdown(n: int) returns (t: int)
let
  t = n fby (t - 1);
tel
```

and instantiate it within `rer` as a resettable node instance:

```
d = (restart countdown every (r when c))
                                  (n when c);
```

The `restart/every` applies only to `countdown` and not to its argument `n when c`. It transforms this stream function into another one that "restarts" whenever the associated expression, `r when c`, is true. Not only does this feature obviate the need to propagate reset signals throughout programs, it is also important for compiling state machines [19].

Resettable function instances can be programmed using higher-order recursive stream functions [13], [27], as we explain in section III-D. But as such functions cannot, in general, be executed in bounded time and space, they are rejected by compilers for embedded software. Instead, patterns like resettable function instances are provided as primitives that compilers treat specially. Specifying real-time systems in a functional programming language which can be compiled to embedded code is the essence of the dataflow synchronous approach and guides the Rocq formalization.

B. Compiler Correctness

The correctness of the Vélus compiler is stated in the following theorem [6], [9].

Theorem 1 (Vélus Compiler Correctness).

$$\begin{aligned} \textbf{if} \quad & \mathsf{compile}\ G\ f = \mathsf{OK}\ asm \\ \textbf{and} \quad & \mathsf{well\text{-}typed\text{-}inputs}\ G\ f\ xs \\ \textbf{and} \quad & G \vdash f(xs) \Downarrow ys \\ \textbf{then} \quad & asm \Downarrow \langle \mathsf{Load}(xs(i)) \cdot \mathsf{Store}(ys(i)) \rangle_{i=0}^{\infty} \end{aligned}$$

If the compilation of node f in program G successfully produces an assembly program asm and the node semantics associates a list of well-typed input streams xs to a list of output streams ys, then the assembly program admits a trace that endlessly alternates load and store events that correspond to the values of the input and output streams.

The Rocq development includes a proof that the semantic relation, $G \vdash f(xs) \Downarrow ys$, for a given G, f, and xs, is deterministic [37, §3.4], but not that it has a solution. In fact, there is no solution when the application of node f to streams xs involves the use of a type cast, or arithmetic or logical operator outside of its valid domain. Consider, for instance, the following node.

```
node mayfail(r : bool) returns (s : int)
let
  s = 42 / countdown(3, r);
tel
```

The semantic relation induced by this node is simply empty if `r` contains the subsequence $\cdots F \cdot F \cdot F \cdots$. This is one way of resolving the mismatch between a source semantics of functions mapping streams to streams, and generated code that can trigger a processor exception. It is a choice that simplifies (a) the semantic definitions, whose rules need not be complete; (b) the compiler, which need not preserve errors; and (c) the correctness proofs, which need only consider valid cases. It complicates, however, reasoning formally about error freedom or the applicability of the correctness theorem. Similar issues occur in any pure functional language. In Haskell, for example, the expression `42 `div` 0` does not reduce to a value, but rather results in the message:

```
*** Exception: divide by zero
```

Partial definitions can be eliminated by redefining operators. For example, integer division, `z = x / y`, could become

```
if (y == 0) { z = 0; } else { z = x / y; }
```

But we instead choose to define a new semantics that models errors explicitly, thereby giving a means to prove their absence prior to execution on an embedded platform. Note that we do not redefine, in any way, the compile function.

C. Contributions and Overview

In this article, we present a new semantics for the language introduced in the example program. The semantics is distinguished by its treatment of errors and fixpoints. The semantics and associated theorems are formalized in the Rocq interactive theorem prover. Such a formalization is difficult from a technical perspective, but the precision required by the tool yields (i) a new definition of the `fby` operator that treats the subtleties of properly handling errors in a fixpoint semantics, (ii) proofs that link two existing functional definitions [27], [25] for resettable node instances to the relational form introduced more recently [7], (iii) a refinement to the `merge` operator used in the functional definition of resettable node instances, (iv) a proof that the new functional semantics satisfies the semantic relations used to define compiler correctness for programs without run-time errors, (v) a corollary that strengthens the compiler correctness theorem, and (vi) a demonstration that our framework can be used to reason about the correctness of static analyses for excluding run-time errors. This work contributes to the study of the domain-specific languages that are used to program safety-critical embedded control software. It is limited in that it does not treat block-based control structures like hierarchical state machines.

Section II presents the syntax of the source language, its relational semantics, and an existing Rocq formalization of complete partial orders. Section III presents our functional semantics, comparing it directly with the corresponding rules of the relational semantics. The lemmas that formally relate the two semantics are proved under the assumption that errors cannot occur. We prove that all assumptions, except freedom from run-time errors, are justified by dependency and typing analyses. Section IV presents the improved compiler correctness theorem and demonstrates the feasibility of justifying the remaining assumption by a syntactic check. Section V discusses related work.

II. Background

Our goal is to express stream functions in an interactive theorem prover, and to reason about them and their compilation. This is done under two constraints: (i) the syntax of programs must be represented explicitly, since it is transformed by compilation functions, and (ii) recursive function definitions must respect rules that ensure consistency. Section II-A presents the abstract syntax of our language, which is defined in Rocq in the standard way. Section II-B introduces the existing relational semantics, though to facilitate comparison, most of the rules are presented later together with those of the new semantics. Section II-C presents a formalization of a fixpoint operator that we exploit in our recursive definitions.

A. Syntax

The language that we study is the subset of Lustre/Scade formalized in earlier articles on Vélus, with core dataflow operators [9, §2], enumeration types, and resettable node instances [7]. The example in the introduction contains all of these features. For precision, we define the four levels of the abstract syntax: expressions, equations, nodes, and programs.

$$\begin{aligned} e ::=\ & c \mid C \mid x \mid \diamond\ e \mid e \oplus e \mid e^+ \texttt{ fby } e^+ \mid e^+ \texttt{ when } C \texttt{ (} x \texttt{)} \\ \mid\ & \texttt{merge } x \texttt{ (} C \texttt{ => } e^+ \texttt{)}^+ \mid \texttt{case } e \texttt{ (} C \texttt{ => } e^+ \texttt{)}^+ \\ \mid\ & f \texttt{ (} e^+ \texttt{)} \mid \texttt{(} \texttt{ restart } f \texttt{ every } e \texttt{)} \texttt{ (} e^+ \texttt{)} \end{aligned}$$

$$eq ::= x^+ = e^+ ; \qquad d ::= x^{ck}_{ty} \qquad ck ::= \bullet \mid ck \texttt{ on } C \texttt{ (} x \texttt{)}$$

$$n ::= \texttt{node } f \texttt{ (} d^+ \texttt{) returns (} d^+ \texttt{) var } d^* \texttt{ let } eq^+ \texttt{ tel}$$

$$g ::= \texttt{(type } ty = \texttt{(| C)}^+\texttt{)}^*\ n^+$$

The expressions comprise primitive constants c, enumeration constants C, variables x, unary operator applications $\diamond$, binary operator applications $\oplus$, the synchronous and case operators presented earlier, node instances, and resettable node instances. The primitive constants and most of the unary and binary operators are defined directly by CompCert, with adaptations to test equality and inequality of enumeration values, to provide a unary cast operator, and to accommodate for bool being an enumeration type in Vélus and an integer type in Clight. The `when` and `merge` operators compare variable values with enumeration constants; the standard syntactic sugar for bool is used in the example. The `merge` and `case` operators match exhaustively on enumerated types. Some expressions allow lists of subexpressions and the corresponding semantic operators are lifted directly to lists of streams.

Equations pair lists of variables and expressions. A node declaration wraps a list of equations with declarations of input, output, and local variables. The declarations include a type and a clock type. Integer and floating-point types are inherited from CompCert. Enumeration types are declared with their constructors. A clock type is either $\bullet$, representing the base clock of a node, or ck `on` $C(x)$, representing a clock ck sampled when the variable x equals the enumeration constant C. Finally, a program g (for 'global') is a list of declarations of enumeration types and nodes.

B. Relational Semantics

The intuitive meaning of the examples was given by writing streams of values next to variables, expressions, and equations, leaving gaps to align values with cycles of the whole program. The relational semantics formalizes this idea [8], [7].

The coinductive type family $\mathsf{Stream}\ A$ has one constructor, *cons*, written $\cdot : A \to \mathsf{Stream}\ A \to \mathsf{Stream}\ A$. The relational semantics involves infinite sequences of *synchronous values*,

$$\mathsf{svalue} := v \mid \mathsf{abs},$$

where v is an enumeration value, or an integer or floating-point value from CompCert, and absence is represented explicitly.

The semantics of a node requires a history H that maps each variable to a stream satisfying the constraints given by its equations. For `actdef`, we have $H(k) = \mathsf{F} \cdot \mathsf{F} \cdot \mathsf{T} \cdot \mathsf{F} \cdots$, $H(x) = \mathsf{abs} \cdot \mathsf{abs} \cdot 4 \cdot \mathsf{abs} \cdots$ and $H(y) = 0 \cdot 0 \cdot 4 \cdot 0 \cdots$.

The constraints on streams and histories in Vélus are encoded by a set of mutually inductive predicates that follow the syntactic structure of programs. For a program G, a node

name f and a list xs of input streams, ys is a valid output of the node only if $G \vdash f(xs) \Downarrow ys$, defined as,

$$\frac{G.f = n \qquad H(n.\mathsf{in}) = xs \qquad H(n.\mathsf{out}) = ys \qquad \forall eq \in n.\mathsf{eqs},\ G, H, (\mathsf{base\text{-}of}\ xs) \vdash eq}{G \vdash f(xs) \Downarrow ys}$$

where $G.f$ maps the function name to its syntax. There must exist a history H that associates input variables to xs and output variables to ys, and that satisfies the node equations.

The rule for equations is parameterized by a global base clock bs, which is a stream of booleans that reflects when values are available relative to synchronous cycles of the whole program. Within a node, the base clock is instantiated with $\mathsf{base\text{-}of}\ xs$, true only when at least one of the input streams in xs is present. An equation $x = e$ is satisfied if the dataflow semantics of the expression e matches the value of x in H.

$$\frac{G, H, bs \vdash e \Downarrow H(x)}{G, H, bs \vdash x = e}$$

For readability, we present equations having only a single variable and expression. The generalization in Rocq to the form $x^+ = e^+$ is tedious but conceptually simple.

Other rules associate each expression with a list of streams defined by coinductive semantic operators, as shown below for constants. The symbol $\equiv$ represents stream equality.

$$\frac{s \equiv \mathsf{const}\ bs\ [\![\mathsf{c}]\!]}{G, H, bs \vdash \mathsf{c} \Downarrow [s]} \qquad \begin{array}{l} \mathsf{const}\ (\mathsf{T} \cdot bs)\ v = v \cdot \mathsf{const}\ bs\ v \\ \mathsf{const}\ (\mathsf{F} \cdot bs)\ v = \mathsf{abs} \cdot \mathsf{const}\ bs\ v \end{array}$$

We present the rules for other synchronous operators in section III-C, together with the functional definitions since this reduces repetition and aids comparison.

Compiler Correctness: The relational model is convenient for reasoning about compiler correctness for two reasons.

First, it allows reasoning about mutually recursive stream equations as conjunctions of predicates on environments associating variable names to streams. Such predicates are readily manipulated in an interactive theorem prover by induction, introduction, and rule inversion. Essentially, we reason about least fixpoints of equations without, until now, proving that they exist. This is not completely satisfactory for a machine-checked end-to-end proof. We should rather minimize and simplify the assumptions in the main theorem so that its interpretation is straightforward and so that its proof considers all relevant details. For us, this means showing that the semantic predicates are coherent and that the syntactic dependency analysis implies the existence of a suitable fixpoint.

Second, by assuming that errors do not occur, the very many rule inversions in the compiler correctness proof only yield interesting cases. With explicit errors, one would have to continually, in each induction, in each compiler pass, preserve an erroneous behavior or show that it cannot occur. The flip side of our assumption is that the correctness proof does not guarantee error preservation. This approach is only reasonable if error freedom can be guaranteed on source programs in one way or another.

C. Constructive Cpo Library

A Kahn network [29] is a set of processes that communicate solely by reading and writing on unbounded queues; once a process chooses to read a queue it must wait for data. A process is modeled as a continuous function between tuples of finite and infinite sequences, a network of processes is modeled as the fixpoint of such functions, and, thanks to closure under arbitrary composition and recursion, a network is itself a process. C. Paulin-Mohring developed a library for reasoning about constructive fixpoints and used it to formalize this model in Rocq [36]. We recall the elements that are reused in our functional definitions.

The library uses dependent records, in the manner of type classes, to define partial orders, monotonic functions, (ω-)complete partial orders, and continuous functions. The definitions formalize standard concepts from denotational semantics, which we recall here.

A *preorder* bundles any type with a reflexive and transitive relation $\preceq$, and extends naturally to an equivalence relation, $x \simeq y$ iff $x \preceq y$ and $y \preceq x$, and thus a *partial order*.

The type of *monotonic functions* between partial orders, $O_1 \to_{\mathsf{m}} O_2$, is for values that pair a function $f : O_1 \to O_2$ with a proof of its monotonicity: $\forall x\, y, x \preceq y$ implies $f(x) \preceq f(y)$. Such values can be considered simply as functions that have additional proof obligations and properties. The type itself is partially ordered: $f \preceq g$ iff $\forall x, f\ x \preceq g\ x$.

A *complete partial order* (cpo) extends a partial order O by distinguishing a bottom element, $\bot : O$, and defining a function to calculate the *least upper bound* (lub) of any monotonic sequence, $\mathsf{lub} : (\mathbb{N} \to_{\mathsf{m}} O) \to O$. A monotonic sequence represents successive approximations of type O, which the lub operation combines into an ideal value.

The *continuous functions* between cpos, $D_1 \to_{\mathsf{c}} D_2$, pair a monotonic function $f : D_1 \to_{\mathsf{m}} D_2$ with a proof that it is continuous: $\forall h : \mathbb{N} \to_{\mathsf{m}} D_1, f(\mathsf{lub}(h)) \preceq \mathsf{lub}(f \circ h)$. The type itself is a cpo: $\bot := \lambda\text{-}.\bot_{D_2}$ and $\mathsf{lub} := \lambda f\, x.\mathsf{lub}_{D_2}(\lambda n.f\ n\ x)$.

The point of all of this is to permit the definition of an operator on continuous functions $\mathsf{FIXP} : (D \to_{\mathsf{c}} D) \to_{\mathsf{c}} D$, such that $\mathsf{FIXP}\ f \simeq f\ (\mathsf{FIXP}\ f)$, and an associated induction principle which specifies that, for every admissible, that is, stable by lub, property P, $P \bot \to (\forall x, P\, x \to P\,(f\, x)) \to P\,(\mathsf{FIXP}\ f)$. This operator is essential for expressing infinite unfoldings and also the process of converging to a solution for mutually recursive definitions.

The cpo of streams, central to both Kahn networks and our semantics, is based on the following coinductive type family $\mathsf{Stream}_\epsilon\ A$ with two constructors.

$$\begin{array}{rcl} \mathsf{Stream}_\epsilon\ A := & \cdot & : A \to \mathsf{Stream}_\epsilon\ A \to \mathsf{Stream}_\epsilon\ A \\ \mid & \epsilon & : \mathsf{Stream}_\epsilon\ A \to \mathsf{Stream}_\epsilon\ A \end{array}$$

The first is the same as for standard streams, while the second is a device [11] that permits a constructive and productive lub computation. The partial order and equivalence relations on streams are insensitive to finite sequences of ϵs. The infinite sequence ϵ^ω is the bottom element and the lub is defined by a function that gradually widens its search along a sequence of

$$\mathsf{tl} : \mathsf{Stream}_\epsilon\ A \to_\mathsf{c} \mathsf{Stream}_\epsilon\ A$$
$$\mathsf{tl}\ (a \cdot x) \simeq x$$
$$\mathsf{tl}\ \bot \simeq \bot\ (= \epsilon^\omega)$$

$$\mathsf{app} : \mathsf{Stream}_\epsilon\ A \to_\mathsf{c} \mathsf{Stream}_\epsilon\ A \to_\mathsf{c} \mathsf{Stream}_\epsilon\ A$$
$$\mathsf{app}\ (a \cdot x)\ y \simeq a \cdot y$$
$$\mathsf{app}\ \bot\ y \simeq \bot$$

$$\mathsf{map} : (A \to B) \to \mathsf{Stream}_\epsilon\ A \to_\mathsf{c} \mathsf{Stream}_\epsilon\ B$$
$$\mathsf{map}\ f\ (a \cdot x) \simeq (f\ a) \cdot \mathsf{map}\ f\ x$$
$$\mathsf{map}\ f\ \bot \simeq \bot$$

$$\mathsf{zip} : (A \to B \to C) \to \mathsf{Stream}_\epsilon\ A \to_\mathsf{c} \mathsf{Stream}_\epsilon\ B \to_\mathsf{c} \mathsf{Stream}_\epsilon\ C$$
$$\mathsf{zip}\ f\ (a \cdot x)\ (b \cdot y) \simeq (f\ a\ b) \cdot \mathsf{zip}\ f\ x\ y$$
$$\mathsf{zip}\ f\ \bot\ y \simeq f\ x\ \bot \simeq \bot$$

$$\mathsf{filter} : (A \to \mathsf{bool}) \to \mathsf{Stream}_\epsilon\ A \to_\mathsf{c} \mathsf{Stream}_\epsilon\ A$$
$$\mathsf{filter}\ p\ (a \cdot x) \simeq a \cdot \mathsf{filter}\ p\ x \qquad \text{if } p\ a = \mathsf{T}$$
$$\mathsf{filter}\ p\ (a \cdot x) \simeq \epsilon\ \mathsf{filter}\ p\ x \simeq \mathsf{filter}\ p\ x \qquad \text{if } p\ a = \mathsf{F}$$
$$\mathsf{filter}\ p\ \bot \simeq \bot$$

$$\mathsf{take} : \mathbb{N} \to \mathsf{Stream}_\epsilon\ A \to_\mathsf{c} \mathsf{Stream}_\epsilon\ A$$
$$\mathsf{take}\ 0\ x \simeq \bot$$
$$\mathsf{take}\ (n+1)\ (a \cdot x) \simeq a \cdot (\mathsf{take}\ n\ x)$$
$$\mathsf{take}\ n\ \bot \simeq \bot$$

Fig. 1: Characteristic equations of primitive stream functions

approximated streams until it finds a non-ϵ element, and so on corecursively, otherwise generating an infinite sequence of ϵs. A stream of the form $(\epsilon^*\ v\cdot)^*\epsilon^\omega$ serves to model both a finite stream and a failure to converge to an infinite stream. There is no constructive procedure for determining whether or not a stream is infinite, but one can still reason about this property using the following two predicates.

$$\frac{\mathsf{is\text{-}cons}\ x}{\mathsf{is\text{-}cons}\ (\epsilon\ x)} \qquad \frac{}{\mathsf{is\text{-}cons}\ (a \cdot x)} \qquad \frac{\mathsf{is\text{-}cons}\ x \qquad \mathsf{infinite}\ (\mathsf{tl}\ x)}{\mathsf{infinite}\ x}$$

The $\mathsf{is\text{-}cons}\ x$ predicate indicates that there is at least one non-ϵ element in the stream x, that is, $\exists n\ a\ y, x = \epsilon^n\ a \cdot y$. The predicate $\mathsf{infinite}\ x$ indicates that x contains an infinite number of non-ϵ elements. A single horizontal line represents an inductive definition; double horizontal lines represent a coinductive one.

To use the library, rather than define stream functions as lambda-terms using Rocq's built-in features, one constructs them using tactics from library primitives and their associated proofs of continuity. One then states and proves characteristic equations giving alternate definitions, without proof content, which are more readable and better suited for proof by rewriting modulo partial order and equivalence relations.

Figure 1 presents the characteristic equations for the main library primitives: tl drops the first element of a stream; app concatenates the first value from one stream onto a second stream; map lifts a function on values onto a stream of values; zip is similar but is for functions on two values; filter propagates or removes elements based on a predicate; and take returns at most the first n values. These primitives encode standard functions that we use in our semantic definitions. The tactic-based definitions build on the following primitive, which propagates ϵs.

$$\mathsf{scase} : (A \to \mathsf{Stream}_\epsilon\ A \to_\mathsf{c} \mathsf{Stream}_\epsilon\ B) \to_\mathsf{c} \mathsf{Stream}_\epsilon\ A \to_\mathsf{c} \mathsf{Stream}_\epsilon\ B$$
$$\mathsf{scase}\ f\ (\epsilon\ s) = \epsilon\ (\mathsf{scase}\ f\ s)$$
$$\mathsf{scase}\ f\ (a \cdot s) = f\ a\ s$$

For example, the tl function is defined as $\mathsf{scase}\ (\lambda\ x\ s.\ s)$. The scase skips over ϵs, giving, in particular, $\mathsf{tl}\ \epsilon^\omega \simeq \epsilon^\omega$, and otherwise applies the anonymous function which drops the head of the stream.

A similar insensitivity to ϵ is encoded in the partial order and equivalence relations for $\mathsf{Stream}_\epsilon\ A$, shown below.

$$\frac{x \preceq y}{\epsilon\ x \preceq y} \qquad \frac{y = \epsilon^n\ a \cdot z \qquad x \preceq z}{a \cdot x \preceq y} \qquad \frac{x \preceq y \quad y \preceq x}{x \simeq y}$$

Together this means that the characteristic equation of a stream function only has two cases, one for a finite number of ϵs followed by a value and one for an infinite number of ϵs. It explains why we can largely ignore ϵs in the remainder.

Kahn versus synchronous semantics: Unlike in a synchronous semantics, absence is not modeled explicitly in a Kahn semantics. Loosely speaking, the "synchronous traces" from the example in section I-A become "Kahn traces" by ignoring the spaces used to align the values into columns. For `actdef`, we would have $H(x) = 4 \cdot 3 \cdot 2 \cdots$. Both forms of dataflow semantics are useful. A Kahn semantics, by virtue of its simplicity, is an ideal reference model. A synchronous semantics provides a specification for compilers: absence indicates that a value is neither calculated nor used in a given cycle. Sometimes both semantics are presented together and compared [21, §§3.1 and 3.2]. Here, we only define a synchronous semantics, but informal comparisons with the Kahn model are sometimes made.

III. Functional Semantics with Errors

The functional semantics of our dataflow language is defined using the fixpoint combinator from the cpo library. Section III-A presents the base definitions in which absence, presence, and errors are explicit. Section III-B presents the denotation of nodes as fixpoints of synchronous stream operators, which are themselves defined in section III-C and, for the reset operator, in section III-D. For each stream operator, we present both the original relational definition and our new functional one, and state lemmas that relate the two under the assumption that errors do not occur, since the stream type used in the relational semantics cannot represent finite streams or

explicit errors. The assumption on finite streams is discharged in section III-E. The elimination of most forms of explicit error is described in section III-F.

A. Values

Our functional semantics associates an expression with streams and a node with a function over streams. Unlike the relational model, it is defined for all programs, meaning that wrong behaviors must be represented. We categorize errors into three classes. *Typing errors*, which result from incorrect combinations of values, and which can be excluded by the type system. *Synchronization errors*, which result from incorrect combinations of absence and presence, and which can similarly be excluded by the clock type system. *Run-time errors*, caused by failures of logical or arithmetic operators, and which require special treatment because they cannot always be detected statically. The *synchronous possibly erroneous values* include explicit cases for each error class:

$$\mathsf{sevalue} := v \mid \mathsf{abs} \mid \mathsf{err_{ty}} \mid \mathsf{err_{sync}} \mid \mathsf{err_{rt}}.$$

We write simply err when the distinction between $\mathsf{err_{ty}}$, $\mathsf{err_{sync}}$, or $\mathsf{err_{rt}}$ is unimportant. The type of streams in the functional model is Stream_ϵ sevalue, which we abbreviate to Stream_ϵ.

A fourth error class is of a different nature. A program that associates a variable or expression to a finite stream is said to suffer from a *causality error*. Such errors are excluded by reasoning about instantaneous dependencies between equations.

A Stream_ϵ that is error-free and infinite, that is, without errs and not ending in ϵ^ω, can be converted to a Stream svalue for use in the relational model. In Rocq, this conversion is defined by coinduction on the proof of infinitude of the source stream. This complicates our proofs but is conceptually trivial and the conversion is implicit in later definitions.

To associate identifiers to streams in semantic definitions, we use stream environments SEnv. They are defined by the underlying library as an indexed product of cpos, $\Pi_{i:\mathsf{ident}}\ \mathsf{Stream}_\epsilon$, but can be understood simply as functions $\mathsf{ident} \to \mathsf{Stream}_\epsilon$. That is, they associate each variable to a stream, with undefined variables mapped to $\bot$. Node environments, that associate node identifiers to their denotations, are defined similarly: $\mathsf{FEnv} := \Pi_{i:\mathsf{ident}}\ (\mathsf{SEnv} \to_\mathsf{c} \mathsf{SEnv})$. That is, they associate a function name to a continuous function from an environment defining the input variables to an environment defining output and local variables.

B. Fixpoint Semantics of Nodes

As in Kahn networks [29, §3] and their formalization in Rocq [36, §4.2], a node's semantics is defined as a fixpoint over its equations. The fixpoint for a node n is defined relative to an environment $env_G : \mathsf{FEnv}$ of node denotations and an environment $env_I : \mathsf{SEnv}$ mapping input variables to streams. Formally, we have $\mathsf{FIXP}(\mathsf{denot_N}\ n\ env_G\ env_I)$, where

$$\mathsf{denot_N}\ n\ env_G\ env_I : \mathsf{SEnv} \to_\mathsf{c} \mathsf{SEnv} :=$$
$$\lambda\ env.\ \mathsf{fold\_right}\ (\mathsf{denot_{EQ}}\ env_G\ env_I\ env)\ n.\mathsf{eqs}\ \bot.$$

This function takes env, an approximation to an environment from output and local variables to streams, and calculates a new one by iterating over the equations. By folding over an empty environment $\bot$, where $\forall x, \bot(x) = \epsilon^\omega$, rather than from env, and assuming unique variable definitions in $n.\mathsf{eqs}$, the calculation with $\mathsf{denot_{EQ}}$ amounts to

$$\{\, x \mapsto \mathsf{denot_E}\ env_G\ env_I\ env\ e \mid \forall\ x = e \in n.\mathsf{eqs} \,\} : \mathsf{SEnv}.$$

The semantics of an expression e is, in the general case, a tuple of streams, $\mathsf{denot_E}\ e\ env_G\ env_I\ env : (\mathsf{Stream}_\epsilon)^{|e|}$. The tuple type family, here, depends on the number of streams defined by the expression. This dependence is necessary as each tuple type of a given size is a distinct cpo whose bottom element and partial order distribute over the individual streams. That is, $(e_0, \ldots, e_n) \preceq (e'_0, \ldots, e'_n)$ means $\forall_{0 \le i \le n}\ e_i \preceq e'_i$ and, similarly, $\bot = (\epsilon^\omega, \ldots, \epsilon^\omega)$.

The definition of $\mathsf{denot_E}$ follows the inductive structure of expressions. For variables, it consults env_I for inputs and env for locals and outputs. For node instances, it applies the corresponding function from env_G. Otherwise, it recursively calculates the streams for subexpressions and applies one of the synchronous operators defined in the next section.

Finally, the semantics of a program G is the fixpoint of the node denotations, $\mathsf{denot}\ G := \mathsf{FIXP}(\mathsf{denot_G}\ G) : \mathsf{FEnv}$, where

$$\mathsf{denot_G}\ G : \mathsf{FEnv} \to_\mathsf{c} \mathsf{FEnv} :=$$
$$\lambda\ env_G\ f\ env_I.\ \mathsf{FIXP}(\mathsf{denot_N}\ (G.f)\ env_G\ env_I).$$

This definition permits mutually recursive function definitions. We will exploit this feature to define resettable node instances, even if the compiler rejects such programs since they do not always execute in bounded space and time.

C. Synchronous Stream Operators

Every form of expression is associated with a synchronous stream operator that defines its semantics. For each operator, we present both the relational definition used in compiler correctness proofs and our new functional version. We sketch the correctness conditions and proofs that link them.

a) Constants: The coinductive rule for constants was presented as an example in section II-B. It introduces absent values according to a base clock. The functional rule is identical as it never produces an error.

b) Binary operators: The relational and functional rules for binary operators are presented in figure 2. The relational rule is defined as a coinductive relation between two input streams and an output stream. The functional rule is defined using zip, see figure 1, which maps a function over the elements of two input streams; internally it propagates ϵs so that if either input stream ends in ϵ^ω then so does its output. The relational rule requires input elements to be simultaneously absent or present, since no other cases are defined; if they are not, the functional rule produces an $\mathsf{err_{sync}}$. The relational rule also only holds when the underlying CompCert semantics is defined, $v_1 \oplus v_2 = \mathsf{Some}\ v$; if this is not the case, the functional rule produces $\mathsf{err_{rt}}$. It would be possible, with an extra test, to distinguish type errors, $\mathsf{err_{ty}}$, from run-time errors,

$$\frac{\textsc{lift}\ xs\ ys\ rs}{\textsc{lift}\ (\mathsf{abs} \cdot xs)\ (\mathsf{abs} \cdot ys)\ (\mathsf{abs} \cdot rs)}$$

$$\frac{\textsc{lift}\ xs\ ys\ rs \qquad v_1 \oplus v_2 = \mathsf{Some}\ v}{\textsc{lift}\ (v_1 \cdot xs)\ (v_2 \cdot ys)\ (v \cdot rs)}$$

$$\begin{aligned}
&\mathsf{lift} : \mathsf{Stream}_\epsilon \to_\mathsf{c} \mathsf{Stream}_\epsilon \to_\mathsf{c} \mathsf{Stream}_\epsilon \\
&\mathsf{lift} := \mathsf{zip}\ (\lambda\ \mathsf{abs}, \mathsf{abs} \to \mathsf{abs} \\
&\qquad \mid v_1, v_2 \to (\mathsf{match}\ v_1 \oplus v_2\ \mathsf{with} \\
&\qquad\qquad \mid \mathsf{Some}\ v \to v \\
&\qquad\qquad \mid \mathsf{None} \to \mathsf{err}_\mathsf{rt}) \\
&\qquad \mid \mathsf{err}, _ \mid _, \mathsf{err} \to \mathsf{err} \\
&\qquad \mid _, _ \to \mathsf{err}_\mathsf{sync})
\end{aligned}$$

Fig. 2: Binary operator: $\oplus$

$$\frac{\textsc{when}_C\ cs\ xs\ rs}{\textsc{when}_C\ (\mathsf{abs} \cdot cs)\ (\mathsf{abs} \cdot xs)\ (\mathsf{abs} \cdot rs)}$$

$$\frac{\textsc{when}_C\ cs\ xs\ rs}{\textsc{when}_C\ (C \cdot cs)\ (v \cdot xs)\ (v \cdot rs)}$$

$$\frac{\textsc{when}_C\ cs\ xs\ rs \qquad C \neq C'}{\textsc{when}_C\ (C' \cdot cs)\ (v \cdot xs)\ (\mathsf{abs} \cdot rs)}$$

$$\begin{aligned}
&\mathsf{when}_C : \mathsf{Stream}_\epsilon \to_\mathsf{c} \mathsf{Stream}_\epsilon \to_\mathsf{c} \mathsf{Stream}_\epsilon \\
&\mathsf{when}_C := \mathsf{zip}\ (\lambda\ \mathsf{abs}, \mathsf{abs} \to \mathsf{abs} \\
&\qquad \mid C, v \to v \\
&\qquad \mid C', v \to \mathsf{abs} \\
&\qquad \mid \mathsf{err}, _ \mid _, \mathsf{err} \to \mathsf{err} \\
&\qquad \mid _, _ \to \mathsf{err}_\mathsf{sync})
\end{aligned}$$

Fig. 3: Sampling operator: `when`

err_rt, but we did not find this useful. Finally, the functional rule propagates errors from the inputs.

Lemma 1. For all streams xs and ys, if $\mathsf{lift}\ xs\ ys$ is infinite and error-free then $\textsc{lift}\ xs\ ys\ (\mathsf{lift}\ xs\ ys)$.

c) Sampling: The `when` operator, whose rules appear in figure 3, is parameterized by an enumerated constant C and takes two inputs, a condition stream and an argument stream. As before, the input streams must be synchronous, otherwise the relational rule does not hold and the functional rule produces $\mathsf{err}_\mathsf{sync}$. When the condition stream matches the parameter, the head of the argument stream is propagated. When it does not, an abs is produced. The overall effect is to filter the argument stream based on the condition stream, while maintaining the synchronization of all streams.

Lemma 2. For all streams cs and xs, if $\mathsf{when}_C\ cs\ xs$ is infinite and error-free then $\textsc{when}_C\ cs\ xs\ (\mathsf{when}_C\ cs\ xs)$.

In a well-behaved program, the synchronous semantics of `when`, whether relational or functional, is an infinite stream even if the condition stream does not match the parameter infinitely often. This is due to the explicit absence values. In the Kahn semantics [36, §4.2.5], the stream may only contain a finite number of values but the ability to produce an ϵ ensures that the productive requirement for coiterative functions is met.

d) Delay: The `fby` operator takes two inputs, an initial stream and a delayed stream. The Kahn definition, app from figure 1, is simple: the head of the initial stream is prefixed to the delayed stream. In a synchronous semantics, on the other hand, absent values must remain in place, leading to the relational rules at the top of figure 4. The FBY relation propagates abs until values appear on the initial and delayed streams, whence the initial value is propagated and the delayed value is "memorized" in the $\textsc{fby}_1$ relation. The $\textsc{fby}_1$ also propagates abs. When both input values are present, the memorized value, v, is propagated, the initial value, x, is ignored, since only its presence counts, and the delayed value, y, is memorized. This definition is standard [21, figure 2].

The functional definition is more complicated because of its role in fixpoint calculations. Consider, for example, the counter `x = 0 fby (x + 1)`. A definition that read from both inputs before producing an output would have the fixpoint $x = \bot$. Instead, the semantic operator must read from the initial stream, produce an output, then read from the delayed stream. We express this idea in our definition, at bottom in figure 4, as four mutually recursive functions: fby and fby_1, which read from an initial stream and produce a value, and fby_A and fby'_1, which read from a delayed stream. When fby reads abs on the initial stream, it produces abs, and invokes fby_A to check for abs on the delayed stream and otherwise produces a synchronization error. When fby reads v on the initial stream, it propagates it and invokes fby'_1 to check for and memorize a value on the delayed stream. The fby_1 function does the same with a memorized value v. Speculatively producing outputs ensures the productivity of well-formed recursive equations. For errors, these functions do not produce an err and continue as usual, as in the definitions of lift and when. Rather, the output stream becomes $\mathsf{map}\ (\lambda x.\ \mathsf{err})\ xs$, which replaces each element of xs, whether it be abs, v, or err, with err. This ensures that the overall definition of fby is length preserving, which will be important in section III-E.

Lemma 3. For all streams xs and ys, if $\mathsf{fby}\ xs\ ys$ is infinite and error-free then $\textsc{fby}\ xs\ ys\ (\mathsf{fby}\ xs\ ys)$.

D. Resettable Node Instances

The ability to reset dataflow programs is, compared to the classic stream operators, more recent and less studied. As we have seen in the introductory examples, a resettable function instance like `(restart countdown every r)(n)` allows to reinitialize a function instance, here `countdown(n)`, whenever a stream, here defined by `r`, is true.

The relational rule for resettable node instances [7] is modular in that it incorporates the unmodified predicate for node instances, $G \vdash f(xs) \Downarrow ys$, presented in section II-B.

$$\frac{\text{FBY}\ xs\ ys\ rs}{\text{FBY}\ (\mathsf{abs}\cdot xs)\ (\mathsf{abs}\cdot ys)\ (\mathsf{abs}\cdot rs)}$$

$$\frac{\text{FBY}_1\ v_y\ xs\ ys\ rs}{\text{FBY}\ (v_x\cdot xs)\ (v_y\cdot ys)\ (v_x\cdot rs)}$$

$$\frac{\text{FBY}_1\ v\ xs\ ys\ rs}{\text{FBY}_1\ v\ (\mathsf{abs}\cdot xs)\ (\mathsf{abs}\cdot ys)\ (\mathsf{abs}\cdot rs)}$$

$$\frac{\text{FBY}_1\ v_y\ xs\ ys\ rs}{\text{FBY}_1\ v\ (v_x\cdot xs)\ (v_y\cdot ys)\ (v\cdot rs)}$$

$$\begin{aligned}
\mathsf{fby}\ (\mathsf{abs}\cdot xs)\ ys &\simeq \mathsf{abs}\cdot \mathsf{fby}_\mathsf{A}\ xs\ ys\\
\mathsf{fby}\ (v\cdot xs)\ ys &\simeq v\cdot \mathsf{fby}'_1\ \mathsf{None}\ xs\ ys\\
\mathsf{fby}\ (\mathsf{err}\cdot xs)\ ys &\simeq \mathsf{err}\cdot \mathsf{map}\ (\lambda x.\ \mathsf{err})\ xs\\
\\
\mathsf{fby}_\mathsf{A}\ xs\ (\mathsf{abs}\cdot ys) &\simeq \mathsf{fby}\ xs\ ys\\
\mathsf{fby}_\mathsf{A}\ xs\ (\mathsf{err}\cdot ys) &\simeq \mathsf{map}\ (\lambda x.\ \mathsf{err})\ xs\\
\mathsf{fby}_\mathsf{A}\ xs\ (v\cdot ys) &\simeq \mathsf{map}\ (\lambda x.\ \mathsf{err}_\mathsf{sync})\ xs\\
\\
\mathsf{fby}'_1\ \mathsf{None}\ xs\ (v\cdot ys) &\simeq \mathsf{fby}_1\ v\ xs\ ys\\
\mathsf{fby}'_1\ (\mathsf{Some}\ v)\ xs\ (\mathsf{abs}\cdot ys) &\simeq \mathsf{fby}_1\ v\ xs\ ys\\
\mathsf{fby}'_1\ _\ xs\ (\mathsf{err}\cdot ys) &\simeq \mathsf{map}\ (\lambda x.\ \mathsf{err})\ xs\\
\mathsf{fby}'_1\ _\ xs\ (_\cdot ys) &\simeq \mathsf{map}\ (\lambda x.\ \mathsf{err}_\mathsf{sync})\ xs\\
\\
\mathsf{fby}_1\ v\ (\mathsf{abs}\cdot xs)\ ys &\simeq \mathsf{abs}\cdot \mathsf{fby}'_1\ (\mathsf{Some}\ v)\ xs\ ys\\
\mathsf{fby}_1\ v\ (v_x\cdot xs)\ ys &\simeq v\cdot \mathsf{fby}'_1\ \mathsf{None}\ xs\ ys\\
\mathsf{fby}_1\ v\ (\mathsf{err}\cdot xs)\ ys &\simeq \mathsf{err}\cdot \mathsf{map}\ (\lambda x.\ \mathsf{err})\ xs
\end{aligned}$$

Fig. 4: Delay operator: `fby`

The behavior of a resettable node instance is modeled as a sequence of disjoint instances of f, a new one coming into effect whenever the reset stream is true. Each individual instance is isolated using a function $\mathsf{mask}^k\ rs\ xs$ which filters an input stream xs, propagating its values starting from the kth true on rs until just before the next true, and otherwise replacing them with abs. Consider, for example, its operation on the following arbitrary values for rs and xs.

rs	F	F	T	abs	F	T	F	F	F
xs	4	4	4	4	4	5	5	abs	5
$\mathsf{mask}^0\ rs\ xs$	4	4	abs	abs	abs	abs	abs	abs	abs
$\mathsf{mask}^1\ rs\ xs$	abs	abs	4	4	4	abs	abs	abs	abs
$\mathsf{mask}^2\ rs\ xs$	abs	abs	abs	abs	abs	5	5	abs	5
$\vdots$									

The $\mathsf{mask}^0\ rs\ xs$ propagates values from xs up until the first time that rs is true, then remains abs. The $\mathsf{mask}^1\ rs\ xs$ produces abs until rs becomes true, propagates values from xs up until rs is true again, and then remains abs. And so on.

The mask operator is combined with the rule for node instances by universally quantifying over k and filtering both inputs and outputs.

$$\forall k, G \vdash f(\mathsf{mask}^k\ rs\ xs) \Downarrow \mathsf{mask}^k\ rs\ ys.$$

The input and output streams, xs and ys, are constrained by an infinite conjunction of instances of f, each of which applies within a disjoint interval starting from the kth true value of rs. In terms of the example, a distinct instance of `countdown` is applied to each masked interval and the results are recombined as constraints on a single stream ys.

The mask function is defined coinductively as follows.

$$\begin{aligned}
&\mathsf{mask}^k\ rs\ xs := \mathsf{mask}^k_0\ rs\ xs\\
&\mathsf{mask}^k_{k'}\ (\mathsf{F}\cdot rs)\ (x\cdot xs) \simeq\\
&\qquad (\mathsf{if}\ k' = k\ \mathsf{then}\ x\ \mathsf{else}\ \mathsf{abs})\cdot \mathsf{mask}^k_{k'}\ rs\ xs\\
&\mathsf{mask}^k_{k'}\ (\mathsf{abs}\cdot rs)\ (x\cdot xs) \simeq\\
&\qquad (\mathsf{if}\ k' = k\ \mathsf{then}\ x\ \mathsf{else}\ \mathsf{abs})\cdot \mathsf{mask}^k_{k'}\ rs\ xs\\
&\mathsf{mask}^k_{k'}\ (\mathsf{T}\cdot rs)\ (x\cdot xs) \simeq\\
&\qquad (\mathsf{if}\ k'+1 = k\ \mathsf{then}\ x\ \mathsf{else}\ \mathsf{abs})\cdot \mathsf{mask}^k_{k'+1}\ rs\ xs
\end{aligned}$$

It filters the stream xs, propagating its values starting from the kth true on rs and stopping at the next true, and otherwise replacing them with abs.

The relational rule works well in the proofs of compiler correctness by natural deduction, but its infinite set of intersecting constraints is quite far from the ideal of programming real-time systems in a functional programming language. It turns out that a functional form already exists, imagined first with recursive block diagrams [13] and later defined using primitive dataflow operators and recursive node instances [27]:

$$\begin{aligned}
&\mathsf{reset}_f\ rs\ xs := \mathsf{let}\ cs = \mathsf{true\text{-}until}\ rs\ \mathsf{in}\\
&\qquad \mathsf{merge}\ cs\ (f(\mathsf{when}_\mathsf{T}\ cs\ xs))\\
&\qquad\qquad (\mathsf{reset}_f\ (\mathsf{when}_\mathsf{F}\ cs\ rs)\ (\mathsf{when}_\mathsf{F}\ cs\ xs)).
\end{aligned}$$

The local stream cs is true until the first true value on rs at which point it immediately becomes false forever.[2] The `merge` thus returns values from an initial instance of f until cs becomes false, at which point it returns values from the next recursive instance. Such recursive, high-order programs can be expressed in languages like Haskell and Lucid synchrone [38], but to produce code that executes in bounded time and space, a compiler would have to recognize the "once false always false" invariant on cs and the resulting tail recursion through the `merge`, and apply these facts to optimize the allocation of memory for the function instance f. This is difficult, brittle, and best avoided. Instead, in languages for embedded software, recursive node instantiations are forbidden and specific higher-order patterns are provided by language features. This definition, however, is readily expressed using the fixpoint combinator and applied to define the semantics of resettable node instances. We found, though, that the standard

[2]The equation with true-until can be programmed with `cs = (true fby false) || (not rs && (true fby cs))`.

semantics of `merge` [21, figure 2] must be adjusted to obtain the expected fixpoint. Our version, merge′, just like for fby and fby_1', speculatively produces abs in the recursive branch to avoid becoming stuck at $\bot$.

$$\begin{aligned}
&\mathsf{merge}'\ (\mathsf{abs}\cdot s_c)\ s_1\ s_2\\
&\qquad \simeq \mathsf{abs}\cdot(\mathsf{merge}'\ s_c\ (\mathsf{expecta}\ s_1)\ (\mathsf{expecta}\ s_2))\\
&\mathsf{merge}'\ (\mathsf{T}\cdot s_c)\ s_1\ s_2\\
&\qquad \simeq \mathsf{app}\ s_1\ (\mathsf{merge}'\ s_c\ (\mathsf{tl}\ s_1)\ (\mathsf{expecta}\ s_2))\\
&\mathsf{merge}'\ (\mathsf{F}\cdot s_c)\ s_1\ s_2\\
&\qquad \simeq \mathsf{app}\ s_2\ (\mathsf{merge}'\ s_c\ (\mathsf{expecta}\ s_1)\ (\mathsf{tl}\ s_2))\\
&\mathsf{merge}'\ (\mathsf{err}\cdot s_c)\ s_1\ s_2\\
&\qquad \simeq \mathsf{map}\ (\lambda x.\mathsf{err})\ (\mathsf{err}\cdot s_c)
\end{aligned}$$

$$\begin{aligned}
\mathsf{expecta}\ (\mathsf{abs}\cdot s) &\simeq s\\
\mathsf{expecta}\ (v\cdot s) &\simeq \mathsf{map}\ (\lambda x.\mathsf{err}_{\mathsf{sync}})\ s\\
\mathsf{expecta}\ (\mathsf{err}\cdot s) &\simeq \mathsf{map}\ (\lambda x.\mathsf{err})\ s
\end{aligned}$$

The operator first checks the condition stream to select a value to propagate and thus unblock the computation in the recursive application of reset_f. Note that if the condition stream becomes abs^ω then the merge′ becomes abs^ω without propagating any errs from either branch. This does not cause any problems, however, since we assume, and ultimately show or require, the absence of errors in all sub-streams. Another drawback is that incorrect synchronizations between $T\cdot s_c$ and $\mathsf{abs}\cdot s_1$ or between $F\cdot s_c$ and $\mathsf{abs}\cdot s_2$ are ignored. These cannot occur, however, in the definition of reset_f.

Since the reset_f above only uses the primitives `fby`, `merge`, and `when`, it is also a valid Kahn network. In the Kahn interpretation, values on the input streams rs and xs are consumed at the same rate. The synchronous mask definition, however, treats absence explicitly: an abs on rs can be paired with a present value on xs and vice versa, as shown by the successive values of xs and rs in the previous example. The Kahn and synchronous interpretations thus only correspond when rs and xs have the same clock. This constraint is enforced by clock types in Lucid synchrone [38], but not in Scade 6 or Vélus, which, for instance, accept

```
(restart countdown every r)(n when c),
```

even though `r` is not sampled on `c`. Such expressions, where the input signal may be absent and the reset signal present, are produced by the translation of state machines into compositions of simpler operators. They are given a synchronous semantics by the relational rule, but cannot be given a Kahn semantics without somehow expressing the timing relation between input and reset streams.

There is an alternative reset [25, §2.6.4.2] that does not restrict rs and xs, and is therefore sensitive to absence.

$$\begin{aligned}
\mathsf{sreset}_f\ rs\ xs &:= \mathsf{sreset}'_f\ rs\ xs\ (f\ xs)\\
\mathsf{sreset}'_f\ (\mathsf{T}\cdot rs)\ xs\ ys &\simeq \mathsf{sreset}'_f\ (\mathsf{F}\cdot rs)\ xs\ (f\ xs)\\
\mathsf{sreset}'_f\ (\mathsf{F}\cdot rs)\ (x\cdot xs)\ (y\cdot ys) &\simeq y\cdot(\mathsf{sreset}'_f\ rs\ xs\ ys)\\
\mathsf{sreset}'_f\ (\mathsf{abs}\cdot rs)\ (x\cdot xs)\ (y\cdot ys) &\simeq y\cdot(\mathsf{sreset}'_f\ rs\ xs\ ys)
\end{aligned}$$

It applies f to xs initially and whenever rs is true, passing the result in an auxiliary stream ys. The values of ys are propagated one-by-one as false or absent values are read on rs and absent or present values are read on xs. We prove that this functional definition coincides with the previous one when rs and xs have the same clock. We also prove that it is correct with respect to the relational rule.

Theorem 2. If the abs elements of rs and xs are aligned, and if f preserves the alignment of abs elements from its inputs through to its outputs, then $\mathsf{sreset}_f\ rs\ xs \simeq \mathsf{reset}_f\ rs\ xs$.

Theorem 3. The sreset operator satisfies the resettable node relation: $\forall k, G \vdash f(\mathsf{mask}^k\ rs\ xs) \Downarrow \mathsf{mask}^k\ rs\ (\mathsf{sreset}\ f\ rs\ xs)$.

We are not aware of any previous proofs relating these three characterisations of resettable node instances. Our proofs are based on three fundamental properties. We subsequently prove, using the fixpoint induction principle from the cpo library, that our functional semantics satisfies these properties.

a) Invariance to initial absence: nodes stutter on initial absent values, $\forall xs, f(\mathsf{abs}\cdot xs) \simeq \mathsf{abs}\cdot f(xs)$, meaning that not only are these values simply propagated, but also that the stream function is invariant; in operational terms, any internal state is unchanged. The property follows recursively from the fact that the base clock, calculated at node instances, regulates the production of constants (const from section II-B). A node's internal components are not activated until inputs are received.

b) Prefix commutativity: nodes commute with stream prefixes, $\forall n\ xs, f(\mathsf{take}\ n\ xs) \simeq \mathsf{take}\ n\ f(xs)$. This property is a characterization of synchronous stream functions: they need at most n inputs to produce n outputs. In fact, this property implies a weaker one which suffices for theorem 3: $\mathsf{take}\ n\ xs \simeq \mathsf{take}\ n\ ys$ implies $\mathsf{take}\ n\ f(xs) \simeq \mathsf{take}\ n\ f(ys)$.

c) Absence forever: if inputs become absent forever, so do outputs, $\forall n\ xs$, if $\mathsf{tl}^n\ xs \preceq \mathsf{abs}^\omega$ then $\mathsf{tl}^n\ (f(xs)) \preceq \mathsf{abs}^\omega$. Stream prefix inequality is used because full equivalence additionally implies productivity, which is not necessary here. This property is a consequence of the correctness of the clock type system. Unlike invariance to initial absence, it does not require function invariance.

Together, the three properties cover the before, during, and after phases of a mask^k instance.

E. Causality Errors

A causality error occurs when a variable depends instantaneously on itself. Non-causal equations may have no solutions, `x = x + 1`, too many, `x = x`, or require solving constraints, `x = (1 + x) / 2`. In these cases, the fixpoint of the functional semantics is $x = \bot$. To reject problematic programs, compilers analyze the dependency graphs of nodes [14, §4.1]. A node is *causal* if its variables can be ordered $x_1, \ldots, x_n$ such that x_i does not depend instantaneously on $x_{j\geq i}$. Variables at right of a `fby` do not count as instantaneous dependencies: for example, in `z = x fby y`, `z` depends on `x`, but not on `y`. The dependency relation is formalized in Vélus, together with an associated induction principle [9, §2.3].

In the semantics of causal programs, all streams are infinite, that is, none of them end in $\bot$. To prove this, we first show that

all operators are length preserving. We write $|xs| \geq i$ to mean that a stream xs has at least i absent, present, or error values. Formally, $|xs| \geq i := \mathsf{is\text{-}cons}\,(\mathsf{tl}^{i-1}\ xs)$. The only interesting case is for the fby.

Lemma 4. Given streams xs and ys such that $|xs| \geq i+1$ and $|ys| \geq i$, then $|\mathsf{fby}\ xs\ ys| \geq i+1$.

This property in turn implies that iterating a node while successively lengthening the input streams produces a fixpoint whose local and output streams are just as long.

Lemma 5. Given a causal node f, if for all inputs x, $|env_I(x)| \geq i+1$, and for all local and output variables y, $|env(y)| \geq i$, then $|(\mathsf{denot_N}\ f\ env_G\ env_I\ env)(y)| \geq i+1$.

This property is not admissible, so, unlike the other theorems which are shown by fixpoint induction, this one is shown by induction on the syntactic dependency relation.

Now, since $\forall n, |xs| \geq n$ implies xs infinite, the dependency analysis implies that there are no causality errors.

Theorem 4. If a node f in a program G is syntactically causal and if the input streams in the environment env_I are infinite, then the output streams of $\mathsf{denot}\ G\ f\ env_I$ are infinite.

F. Typing and Runtime Errors

Compile-time analyses in the Vélus compiler reject ill-typed programs and otherwise produce a typing context that associates every variable in a node to a type and clock type, $\Gamma(x) = (ty_x, ck_x)$, and associated well-typing predicates. We use the typing context to define well-formed, a predicate on a functional environment env that is true only if for every variable x in Γ: (i) the stream $env(x)$ never contains err, (ii) present values in $env(x)$ have type ty_x, written $env(x) : ty_x$, and (iii) $env(x)$ is *aligned* with the denotation of ck_x. The first two conditions are readily expressed by quantifying over stream elements. The third condition means that whenever $env(x)$ is present, the denotation of ck_x is true, and whenever $env(x)$ is absent, it is false. We formalize this alignment condition as

$$\mathsf{clock\text{-}of}\ env(x) \preceq \mathsf{denot_C}\ ck_x\ env,$$

where clock-of maps present values to true and others to false, and a clock type's denotation is defined inductively:

$$\begin{aligned} \mathsf{denot_C}\ \bullet\ env &:= \mathsf{base\text{-}of}_{f.\mathsf{in}}\ env \\ \mathsf{denot_C}\ (ck\ \mathtt{on}\ C(x))\ env &:= \\ \mathsf{zip}\ (\lambda c\, v.\ c \wedge v = C)\ (\mathsf{denot_C}\ ck\ env)\ env(x)&. \end{aligned}$$

The alignment condition is a prefix inequality because, at any iteration of the fixpoint calculation, the clock stream may have one element more than the variable stream. Such inequalities permit proof by fixpoint induction and thereafter imply equivalence when the left-hand side is infinite.

Assuming a well-formed env, we prove that the denotation of a well-typed expression with no run-time errors, (i) is never err, (ii) has well-typed present values, and (iii) is aligned with its clock. The proof is by induction on the syntax and requires that every semantic operator preserve the invariant.

Freedom from run-time errors is necessary for clock alignment. We formalize it in a predicate no-rte that depends on an expression's semantics in an environment. The only interesting cases are for unary and binary operators, the others descend recursively or hold trivially. The rule for unary operators, below, requires showing that an operator which receives well-typed values never produces a run-time error.

$$\frac{\begin{array}{c}\mathsf{no\text{-}rte}\ env_G\ env_I\ env\ e \\ (\mathsf{denot_E}\ e\ env_G\ env_I\ env : \mathsf{type\text{-}of}\ e \\ \qquad \rightarrow \mathsf{denot_E}\ (\diamond\ e)\ env_G\ env_I\ env \neq \cdots \mathsf{err_{rt}} \cdots)\end{array}}{\mathsf{no\text{-}rte}\ env_G\ env_I\ env\ (\diamond\ e)}$$

Finally, if there are no such run-time errors, then the proof on denotations of well-typed expressions extends to nodes.

Lemma 6. For a well-typed node n, environments env_G and $\mathsf{well\text{-}formed}\ (env_I \uplus env)$, given $\mathsf{no\text{-}rte}\ env_G\ env_I\ env\ n$, then $\mathsf{well\text{-}formed}\ (env_I \uplus \mathsf{denot_N}\ n\ env_G\ env_I\ env)$.

This result extends by fixpoint induction to program denotations. Freedom from run-time errors for a node named f on inputs xs, written $\mathsf{no\text{-}run\text{-}time\text{-}errors}\ G\ f\ xs$, is defined as $\mathsf{no\text{-}rte}\ (\mathsf{denot}\ G)\ env_I\ (\mathsf{denot}\ G\ f\ env_I)\ (G.f)$, where env_I is an environment produced from xs.

IV. Compiler Correctness

Together, the definitions and properties described in section III permit to strengthen the compiler correctness result. Before stating the new theorem, we first note that the individual correctness results from sections III-C and III-D extend to the fixpoints of section III-B, showing that the functional semantics agrees with the relational one.

Theorem 6. For every node f and environment env_I, if executing the denotation of f on inputs env_I results in an infinite and error-free environment, then

$$G \vdash f(\lceil env_I \rceil_{f.\mathsf{in}}) \Downarrow \lceil \mathsf{denot}\ G\ f\ env_I \rceil_{f.\mathsf{out}},$$

where $\lceil \cdot \rceil_l$ represents the conversion from an environment of infinite Stream_ϵs to one on Streams, and its projection onto a list for the variables in l.

This result enables us to prove the new correctness theorem shown at right of the original theorem in figure 5. The first two conditions are unchanged, that is, the compiler must still produce a result and input values must still be well typed. But, since the functional semantics assigns a meaning to all programs, even erroneous ones, the requirement that a program satisfy the semantic relation is replaced by a precise requirement on error freedom. The conclusion guarantees the existence of output streams ys that satisfy both the dataflow semantics of the source program and the trace semantics of the generated assembly program. This stronger theorem provides a practicable proof obligation on source programs, no-run-time-errors, that guarantees correct compilation.

In general, discharging the proof obligation requires reasoning about inputs and dynamic behavior. Often, however, it can be achieved by static analysis. We propose a simple proof of concept, check-ops(G), that suffices to guarantee the dynamic

Theorem 1 (Compiler correctness, original).

if $\mathsf{compile}\ G\ f = \mathsf{OK}\ asm$
and $\mathsf{well\text{-}typed\text{-}inputs}\ G\ f\ xs$
and $G \vdash f(xs) \Downarrow ys$
then $asm \Downarrow \langle \mathsf{Load}(xs(i)) \cdot \mathsf{Store}(ys(i)) \rangle_{i=0}^{\infty}$

Theorem 5 (Compiler correctness, new).

if $\mathsf{compile}\ G\ f = \mathsf{OK}\ asm$
and $\mathsf{well\text{-}typed\text{-}inputs}\ G\ f\ xs$
and $\mathsf{no\text{-}run\text{-}time\text{-}errors}\ G\ f\ xs$
then $\exists ys,\ G \vdash f(xs) \Downarrow ys \wedge asm \Downarrow \langle \mathsf{Load}(xs(i)) \cdot \mathsf{Store}(ys(i)) \rangle_{i=0}^{\infty}$

Fig. 5: At left, the original Vélus correctness theorem [6], [9], at right, our new theorem

condition $\mathsf{no\text{-}run\text{-}time\text{-}errors}\ G\ f\ xs$. Our version of Vélus prints a warning if $\mathsf{check\text{-}ops}(G) = \mathsf{F}$. If compilation succeeds with no warnings, then the generated assembly program is guaranteed to be correct for well-typed inputs. Otherwise, the programmer is responsible for ensuring, and eventually proving, that run-time errors never occur.

Theorem 5 states that the generated assembly program can produce the desired trace. We also show, moreover, thanks to CompCert's determinism theorems, that this is the only trace possible in an environment where loads return successive values from input streams. That is, a corollary strengthens the conclusion to $\exists! ys,\ \cdots \wedge (asm \parallel \mathsf{env}_{xs}) \Downarrow \langle \cdots \rangle_{i=0}^{\infty}$.

A. A Simple Static Check

The $\mathsf{no\text{-}run\text{-}time\text{-}errors}$ predicate is an obligation that, in general, requires reasoning about a program from preconditions and invariants, or otherwise analyzing the program source in one way or another. Since the compiler correctness theorem does not apply when errors occur, any analysis should be applied to the dataflow source program rather than to the generated code. In some cases, a simple check suffices to rule out run-time errors by requiring (i) that a program never casts floats to integers, which fails on extreme values, (ii) that for integer divisions or modulos, the second argument is a constant that is neither zero nor minus one, and (iii) that for bitwise shift operations, the second argument is a constant that is strictly less than the word size of the target architecture. The second condition is restrictive but still useful for embedded control software that only uses integer division for binary search in lookup tables. We implemented a Rocq function $\mathsf{check\text{-}ops}$ that returns true if the above conditions are met and prove that this implies $\mathsf{no\text{-}run\text{-}time\text{-}errors}$.

Theorem 7. For a node f in a well-typed program G, if $\mathsf{check\text{-}ops}(G) = \mathsf{T}$ then $\forall xs,\ \mathsf{no\text{-}run\text{-}time\text{-}errors}\ G\ f\ xs$.

This result combined with theorem 5, the new correctness theorem, guarantees that if $\mathsf{check\text{-}ops}$ returns true, no run-time errors occur and that the source dataflow semantics is preserved by the semantics of the generated assembly.

Theorem 8 (Compiler correctness for checked programs).

if $\mathsf{compile}\ G\ f = \mathsf{OK}\ asm$
and $\mathsf{well\text{-}typed\text{-}inputs}\ G\ f\ xs$
and $\mathsf{check\text{-}ops}(G) = \mathsf{T}$
then $\exists ys,\ G \vdash f(xs) \Downarrow ys$
$\wedge\ asm \Downarrow \langle \mathsf{Load}(xs(i)) \cdot \mathsf{Store}(ys(i)) \rangle_{i=0}^{\infty}$

V. Related Work and Discussion

We aim for a verified compiler with a semantics that is as simple and direct as possible within the constraints of an interactive theorem prover. Dataflow synchronous languages are pure functional languages whose basic values are potentially infinite sequences. It is thus natural to associate a program with a function over streams of type

$$\mathsf{Stream}\ \alpha_1 \times \cdots \times \mathsf{Stream}\ \alpha_n \to \mathsf{Stream}\ \beta_1 \times \cdots \times \mathsf{Stream}\ \beta_m.$$

Existence and determinism are intrinsic properties of such functions, whereas they must be separately proved for relational definitions, that is, for definitions by predicate. The earlier sections essentially present the choices and compromises that we made to realize the ideal of a functional semantics in our context. The related work can also be understood from this perspective.

General-purpose functional languages, like Haskell or a subset of OCaml with Lazy streams, are not used to program safety-critical embedded control software because of the need to statically determine the memory use and execution time of generated code. They can, however, be used to define the semantics of dataflow synchronous languages [16], [18]. Transferring such definitions into an interactive theorem prover requires respecting the constraints of the underlying formal logic: inductive definitions must terminate, coinductive definitions must produce, and domain-theoretic definitions require proof of continuity. All theorem provers address these constraints in one way or another, and many solutions have been developed, see [40, §7] for some historical references.

HOLCF [34] is an extension to the Isabelle interactive theorem prover [35] that combines the HOL and LCF logical systems. It includes a formalization of cpos and continuous functions, and provides mechanisms for defining and reasoning about (partial) recursive functions and infinite values. These features are applied to formalize I/O automata [33], [32], and notably to model execution traces as potentially infinite sequences. Unlike the constructive sequences underlying our development [36], the sequence type formalized in HOLCF allows distinguishing finite total sequences, finite partial sequences, and infinite sequences [33, §6.2]:

$$\mathbf{domain}\ (\alpha)\mathsf{seq} = \mathsf{nil} \mid (\mathsf{HD} : \alpha)\#(\mathbf{lazy}(\mathsf{TL} : (\alpha)\mathsf{seq})).$$

The explicit nil constructor that marks the end of a finite sequence is distinct from the $\bot$ element implicit in a HOLCF **domain** declaration. This makes it possible to model both

automata that terminate and those that diverge by performing infinitely many internal transitions. Since our definitions in section III never refer explicitly to ϵ, we believe that they would transfer directly to the classical context of HOLCF. Certain proof details may even be simpler.

A defining feature of Rocq is that it is both an interactive theorem prover and a programming language. It is natural in this setting to focus on constructive definitions and, as explained in section II-C, the cpo library [36] does so. In principle, building on this library means that our semantics is executable. In practice, however, calculating the first few elements of a simple definition can take hours, even after extraction and compilation. In terms of specification and proof, working with the library involves a lot of technical details, but ultimately enables the development presented in section III. While coinductive proofs can be difficult in Rocq due to restrictions on the underlying proof terms, *coinduction loading* [24] sufficed to overcome these problems in our case.

The ϵ from the Stream_ϵ type [36] is also used in interaction trees [42, Figure 1] (`Tau`), which additionally model termination (`Ret`) and interactions (`Vis`). In HOL4, ϵ is not required for co-recursive definitions and divergence may be modelled directly (`Div`) [30, §3.3]. In any case, interaction trees are less natural for a functional dataflow language, where computations occur both instantaneously (evaluation of expressions) and infinitely often (parallel composition of node equations).

Another approach to handling partial or non-terminating functions in interactive theorem provers is to restrict their domains. Theorems and reduction rules then only apply to a function when a condition on its inputs is satisfied. For stream functions, one such requirement is productivity [1][39, §3]. The syntactic guardedness condition of Lustre, that otherwise cyclic dependencies must be broken by inserting `fby`s, precludes many productivity problems, but not those arising from the sampling operator. Consider, for example, the equation `x = 42 when false`. It is productive in a synchronous semantics, giving a stream of absent values, but not in a Kahn semantics, where it defines an empty stream, since `when` is equivalent to filter.

S. Boulmé and G. Hamon specify dataflow synchronous operators and programs directly in Rocq [4], [5], exploiting its dependent type system to encode the clock types of Lustre and Lucid synchrone [12], [15], [21]. They define a type family for dependent streams $\mathsf{Str}\ A\ c$, indexed by A, the type of values, and c, a *clock* that reflects when values are available. The type of c is clock, which is a synonym for $\mathsf{Stream\ bool}$. The constructor for dependent streams has the type

$$\forall c : \mathsf{clock} \to \mathsf{lvalue}\ A\ (\mathsf{hd}\ c) \to \mathsf{Str}\ A\ (\mathsf{tl}\ c) \to \mathsf{Str}\ A\ c.$$

The first argument, usually inferred, specifies the clock on which the remaining types depend. The second argument is the head of the stream. It has type $\mathsf{lvalue}\ A\ \mathsf{bool}$, which has constructors for present values, $v : \mathsf{lvalue}\ A\ \mathsf{T}$, absent values, $\mathsf{abs} : \mathsf{lvalue}\ A\ \mathsf{F}$, and indeterminate values, $\mathsf{fail} : \mathsf{lvalue}\ A\ \mathsf{T}$. Indeterminate values are used to define a clocked stream $\bot(c)$ from which least fixpoints are built. They are successively replaced by present values, as mutually recursive definitions are evaluated, and disappear completely from streams in well-formed programs. Reasoning about a program by unfolding its fixpoint requires proof that none of its streams depends instantaneously on itself.

Expressing the typing and clock-typing rules as dependent types allows semantic operators to be defined as functions using just the cases of the corresponding relational definitions. Unwanted cases are ruled out by construction. Partial arithmetic and logical operators are not considered in the above development and their treatment would surely require more sophisticated dependent types. With enough wizardry, including, for example, the use of heterogeneous lists [17, §9], it might not be impossible to follow this approach in a verified compiler. The main obstacle would seem to be the interlocking constraints between the types representing the syntax of a program being compiled and those representing its semantics. The compiler would have to build these, essentially validating the typing rules in the process and thereby losing one advantage of the shallow embedding. Proving properties would require manipulating dependent terms with Rocq tactics. We took a different approach.

In the coiterative semantics of [16], length-preserving (synchronous) stream functions are characterized by an initial state of type S and a step function of type

$$S \times (\alpha_1 \times \cdots \times \alpha_n) \to S \times (\beta_1 \times \cdots \times \beta_m).$$

The composition of individual operators gives a state and step function for a complete program. Iterating this step function from the initial state with successive values from input streams gives successive values of output streams. By making the state explicit, it becomes possible to determine whether or not program execution time and memory use are bounded. This semantics is thus well suited for reasoning about compilation, and for defining interpreters and executable semantics [18]. Programs can be distinguished by their internal structure, which must be explicitly abstracted to obtain the input/output equivalence intrinsic to a stream semantics. Non-length-preserving functions like `when` and `merge` are treated by adding an explicit absent value. Obtaining a Kahn semantics requires to explicitly model buffering.

While our development is more complex than strictly necessary for a synchronous semantics, it lays the groundwork for a simple correspondence with a Kahn semantics by mapping abs to ϵ [28, Chapter 3]. Reasoning in a Kahn semantics is preferable because there is no need to account for absence [10], [23] and operator definitions are simpler, notably for `fby`. A formal correspondence would permit to transfer verified properties to the synchronous model, which specifies the behavior of generated code by making precise when values are calculated and used, and ultimately to the generated assembly.

Acknowledgment

Supported by ANR JCJC FidelR 19-CE25-0014-01 “Fidelity in Reactive Systems Design and Compilation”. We thank the anonymous reviewers for their helpful suggestions.

References

[1] Yves Bertot and Ekaterina Komendantskaya. Inductive and coinductive components of corecursive functions in Coq. In *9th Workshop on Coalgebraic Methods in Computer Science (CMCS)*, volume 203, issue 5 of *ENTCS*, pages 25–47, April 2008. Elsevier.

[2] Dariusz Biernacki, Jean-Louis Colaço, Gregoire Hamon, and Marc Pouzet. Clock-directed modular code generation for synchronous data-flow languages. In *Proc. 9th ACM SIGPLAN Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 121–130, June 2008. ACM Press.

[3] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *Proc. 14th Int. Symp. Formal Methods (FM)*, volume 4085 of *LNCS*, pages 460–475, August 2006. Springer.

[4] Sylvain Boulmé and Grégoire Hamon. Certifying synchrony for free. In *Proc. 8th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 2250 of *LNCS*, pages 495–506, December 2001. Springer.

[5] Sylvain Boulmé and Grégoire Hamon. A clocked denotational semantics for Lucid-Synchrone in Coq. Technical report, LIP6, November 2001.

[6] Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. A formally verified compiler for Lustre. In *Proc. 2017 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 586–601, June 2017. ACM Press.

[7] Timothy Bourke, Lélio Brun, and Marc Pouzet. Mechanized semantics and verified compilation for a dataflow synchronous language with reset. *Proc. of the ACM on Programming Languages*, 4(POPL):1–29, January 2020.

[8] Timothy Bourke, Pierre-Évariste Dagand, Marc Pouzet, and Lionel Rieg. Vérification de la génération modulaire du code impératif pour Lustre. In *28ièmes Journées Francophones des Langages Applicatifs (JFLA)*, pages 165–179, January 2017.

[9] Timothy Bourke, Paul Jeanmaire, Basile Pesin, and Marc Pouzet. Verified Lustre normalization with node subsampling. *ACM Trans. Embedded Computing Systems*, 20(5s):Article 98, October 2021.

[10] Cécile Canovas-Dumas and Paul Caspi. A PVS proof obligation generator for Lustre programs. In *Proc. 7th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2000)*, volume 1955 of *LNCS*, pages 179–188, November 2000. Springer.

[11] Venanzio Capretta. General recursion via inductive types. *Logical Methods in Computer Science (LMCS)*, 1(2), July 2005.

[12] Paul Caspi. Clocks in dataflow languages. *Theoretical Computer Science*, 94(1):125–140, March 1992.

[13] Paul Caspi. Towards recursive block diagrams. *Annual Review in Automatic Programming*, 18:81–85, 1994.

[14] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John A. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *Proc. 14th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL)*, pages 178–188, January 1987. ACM Press.

[15] Paul Caspi and Marc Pouzet. Synchronous Kahn networks. In *Proc. 1st ACM SIGPLAN Int. Conf. on Functional Programming (ICFP)*, pages 226–238, May 1996. ACM Press.

[16] Paul Caspi and Marc Pouzet. A co-iterative characterization of synchronous stream functions. In *1st Workshop on Coalgebraic Methods in Computer Science (CMCS)*, volume 11 of *ENTCS*, pages 1–21, March 1998. Elsevier.

[17] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013.

[18] Jean-Louis Colaço, Michael Mendler, Baptiste Pauget, and Marc Pouzet. A constructive state-based semantics and interpreter for a synchronous data-flow language with state machines. *ACM Trans. Embedded Computing Systems*, 22(5s):152:1–152:26, September 2023. Presented at 23rd Int. Conf. on Embedded Software (EMSOFT).

[19] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. A conservative extension of synchronous data-flow with state machines. In *Proc. 5th ACM Int. Conf. on Embedded Software (EMSOFT)*, pages 173–182, September 2005. ACM Press.

[20] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. Scade 6: A formal language for embedded critical software development. In *Proc. 11th Int. Symp. Theoretical Aspects of Software Engineering (TASE)*, pages 4–15, September 2017. IEEE Computer Society.

[21] Jean-Louis Colaço and Marc Pouzet. Clocks as first class abstract types. In *Proc. 3rd Int. Conf. on Embedded Software (EMSOFT)*, volume 2855 of *LNCS*, pages 134–155, October 2003. Springer.

[22] Coq Development Team. *The Coq proof assistant reference manual*. Inria, 2022. v. 8.16.1.

[23] Cécile Dumas Canovas. *Méthodes déductives pour la preuve de programmes Lustre*. PhD thesis, Université Joseph Fourier, Grenoble, France, November 2000.

[24] Jörg Endrullis, Dimitri Hendriks, and Martin Bodin. Circular coinduction in Coq using bisimulation-up-to techniques. In *Proc. 4th Int. Conf. on Interactive Theorem Proving (ITP)*, volume 7998 of *LNCS*, pages 354–369, July 2013. Springer.

[25] Léonard Gérard. *Programmer le parallélisme avec des futures en Heptagon un langage synchrone flot de données et étude des réseaux de Kahn en vue d'une compilation synchrone*. PhD thesis, Université Paris-Sud, Paris, France, September 2013.

[26] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous dataflow programming language LUSTRE. *Proc. IEEE*, 79(9):1305–1320, September 1991.

[27] Gégoire Hamon and Marc Pouzet. Modular resetting of synchronous data-flow programs. In *Proc. 2nd ACM SIGPLAN Int. Conf. on Principles and Practice of Declarative Programming (PPDP)*, pages 289–300, September 2000. ACM.

[28] Paul Jeanmaire. *Une sémantique dénotationnelle pour un compilateur synchrone vérifié*. PhD thesis, PSL University, December 2024.

[29] Gilles Kahn. The semantics of a simple language for parallel programming. In *Proc. Int. Federation for Information Processing (IFIP) Congress 1974*, pages 471–475, August 1974. North-Holland.

[30] Hrutvik Kanabar, Samuel Vivien, Oskar Abrahamsson, Magnus O. Myreen, Michael Norrish, Johannes Åman Pohjola, and Riccardo Zanetti. PureCake: A verified compiler for a lazy functional language. *Proc. of the ACM on Programming Languages*, 7:Article 145, 2023.

[31] Xavier Leroy. Formal verification of a realistic compiler. *Comms. ACM*, 52(7):107–115, 2009.

[32] Olaf Müller. I/O Automata and beyond: Temporal logic and abstraction in Isabelle. In *Proc. 11th Int. Conf. on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 1479 of *LNCS*, pages 331–348, September/October 1998. Springer.

[33] Olaf Müller. *A Verification Environment for I/O Automata Based on Formalized Meta-Theory*. PhD thesis, T.U. München, Germany, September 1998.

[34] Olaf Müller, Tobias Nipkow, David von Oheimb, and Oscar Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9(2):191–223, March 1999.

[35] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[36] Christine Paulin-Mohring. A constructive denotational semantics for Kahn networks in Coq. In *From Semantics to Computer Science: Essays in Honour of Gilles Kahn*, pages 383–413. CUP, 2009.

[37] Basile Pesin. *Verified Compilation of a Synchronous Dataflow Language with State Machines*. PhD thesis, PSL Research University, October 2023.

[38] Marc Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, April 2006.

[39] Vlad Rusu and David Nowak. Defining corecursive functions in Coq using approximations. In *36th European Conf. on Object-Oriented Programming (ECOOP)*, volume 222, pages 12:1–12:24, 2022.

[40] Konrad Slind. Function definition in Higher-Order Logic. In *Proc. 9th Int. Conf. on Theorem Proving in Higher Order Logics (TPHOLs)*, volume 1125 of *LNCS*, pages 381–397, August 1996. Springer.

[41] William W. Wadge and Edward A. Ashcroft. *LUCID, the dataflow programming language*. Academic Press Professional, Inc., 1985.

[42] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction Trees: Representing recursive and impure programs in Coq. *Proc. of the ACM on Programming Languages*, 4(POPL):article 51, January 2019.