Nondeterministic Asynchronous Dataflow in Isabelle/HOL

Rafael Castro Gonçalves Silva
Department of Computer Science, University of Copenhagen

Laouen Fernet
Department of Computer Science, University of Copenhagen

Dmitriy Traytel
Department of Computer Science, University of Copenhagen

Abstract

We formalize nondeterministic asynchronous dataflow networks in Isabelle/HOL. Dataflow networks are comprised of operators that are capable of communicating with the network, performing silent computations, and making nondeterministic choices. We represent operators using a shallow embedding as codatatypes. Using this representation, we define standard asynchronous dataflow primitives, including sequential and parallel composition and a feedback operator. These primitives adhere to a number of laws from the literature, which we prove by coinduction using weak bisimilarity as our equality. Albeit coinductive and nondeterministic, our model is executable via code extraction to Haskell.

2012 ACM Subject Classification Security and privacy → Logic and verification; Computing methodologies → Distributed algorithms; Software and its engineering → Data flow languages

Keywords and phrases dataflow, verification, coinduction, Isabelle/HOL

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Data stream processing frameworks like Flink [12], Kafka [21], Spark Streaming [33], and Timely Dataflow [26] are widely used in industry to compute with and analyze streams of data at a large scale. With a long-term objective of formally verifying the correctness of programs expressed in such frameworks in mind, we focus on their common algebraic *dataflow* foundation in this paper. The dataflow programming paradigm represents a streaming computation as a graph of *operators*, which are themselves data stream transformers.

We focus on *asynchronous* dataflow in which operators work autonomously without blocking the overall computation following the mantra "if I cannot make progress maybe someone else can." Asynchronous computation is naturally *nondeterministic*: the order in which operators are invoked is not predetermined and might affect the to overall computation's behavior.

Bergstra et al. [3] provide an algebra of asynchronous dataflow operators and building blocks (including sequential and parallel composition and a feedback loop construct), axiomatize the properties of these operators, and give two instances satisfying the axioms. We develop a new instance in Isabelle/HOL, represented as a shallow embedding as a coinductive datatype (short: codatatype). Our operators are essentially nondeterministic input-output machines with silent actions (Section 2). The coinductive representation allows us to keep the state implicit when composing operators. We use corecursion to define composition, feedback, and the various basic asynchronous operators of Bergstra et al. (Section 3). We use countable sets to represent nondeterminism, which simplifies our definition of composition and feedback operators.

We validate that our operators work as intended by proving all but one of Bergstra et al.'s axioms by coinduction (Section 4). Thereby, we use weak bisimilarity as the notion of operator equality. Some of our originally proved properties deviate mildly from Bergstra et

al.'s axioms due to a simplification we made in some operators' definitions and the fact that Isabelle does not have empty types. (Types are used by both us and Bergstra et al. to index the inputs and outputs of an operator. Bergstra et al.'s empty types signify that an operator has no inputs or no outputs). We show how to close this gap and obtain precisely Bergstra et al.'s axioms by working with the subtype of well-behaved operators (Section 5).

With the help of Isabelle's code generator, we extract executable Haskell code from our formalization (Section 6). To this end, we represent countable sets as a quotient of lazy lists. This allows us to run operators, which is useful for testing. A particular challenge is the executability of the union of a countable set of countable sets as the corresponding merge function on lazy lists. We conclude with discussions of related work (Section 7) and particular formalization aspects (Section 8). Our Isabelle formalization is available as supplementary material.

2 Operators as a Codatatype

Isabelle/HOL's codatatypes [6] formalize potentially infinite objects such as infinite streams (*stream* type) and lazy lists (*llist* type). We introduce operators as:

$$\begin{aligned}&\textbf{codatatype}\ (\mathsf{inputs}\colon {'i}, \mathsf{outputs}\colon {'o}, {'d})\ op = \\ &\quad \mathsf{Read}\ {'i}\ ({'d} \Rightarrow ({'i}, {'o}, {'d})\ op) \mid \mathsf{Write}\ (({'i}, {'o}, {'d})\ op)\ {'o}\ {'d} \\ &\quad \mid \mathsf{Silent}\ ({'i}, {'o}, {'d})\ op \mid \mathsf{Choice}\ (({'i}, {'o}, {'d})\ op)\ cset\end{aligned}$$

This **codatatype** command introduces a new type constructor, *op* (written postfix), with type parameters $'i$ indexing the input ports, $'o$ indexing the output ports, and $'d$ representing the type of data items (or events) an operator processes. An operator will perform one of four actions corresponding to the four codatatype constructors: (1) read an event from a given input port and continue based on the read event, (2) write an event to a given output port and continue as specified, (3) execute an internal (silent) action and continue as specified, and (4) nondeterministically choose a continuation from a countable set (type *cset*) of options. Operators are possibly infinite trees with nodes labeled by one of these four actions with Read branching over the read event $'d$ and Choice branching over the countable set of choices. For an operator to be finite, the operator must eventually reach the only leaf $\oslash \equiv \mathsf{Choice}\ \{\}_c$ along all branches. Note that we use the subscript $_c$ to distinguish sets and operations on them from their countable set counterparts: *set* and *cset* are separate types in Isabelle.

The **codatatype** command also introduces the functions $\mathsf{inputs} :: ({'i}, {'o}, {'d})\ op \Rightarrow {'i}\ set$ and $\mathsf{outputs} :: ({'i}, {'o}, {'d})\ op \Rightarrow {'o}\ set$ that collect the sets of used ports as well as a map function on the ports $\mathsf{map\_op} :: ({'i_1} \Rightarrow {'i_2}) \Rightarrow ({'o_1} \Rightarrow {'o_2}) \Rightarrow ({'i_1}, {'o_1}, {'d})\ op \Rightarrow ({'i_2}, {'o_2}, {'d})\ op$. Note that an operator does not necessarily have to use all available ports. For example, an operator of type $(nat, nat, nat)\ op$ does not necessarily have an infinite number of input and output ports; rather, it uses some subset of naturally-numbered ports, given by inputs and outputs.

Isabelle requires all functions to be total. For recursive functions on datatypes, totality means termination. In contrast, corecursive functions, which produce elements of a codatatype, are guaranteed to be total if they are productive—that is, they must always eventually produce a codatatype constructor. The **corec** command facilitates the definition of corecursive functions, in which a codatatype constructor guards all corecursive calls and all call contexts are comprised of friendly functions [5], thereby maintaining productivity and totality.

For example, we define two uncommunicative operators using **corec** in Figure 1. The operator $\odot$ corecurses forever with the Silent constructor. The operator $\otimes$ corecurses forever with the (singleton) Choice constructor as shown in the lemma spin_op_code. The actual **corec** definition is slightly more convoluted. The codatatype nests the corecursive occurrence of *op*,

corec silent_op ($\odot$) **where**
  $\odot$ = Silent $\odot$

corec spin_op ($\otimes$) **where**
  $\otimes$ = Choice ($(\lambda\_.\ \otimes)$`$_c$ {()}$_c$)

lemma spin_op_code:
  $\otimes$ = Choice {$\otimes$}$_c$

Figure 1 Two uncommunicative operators

datatype $('i, 'o, 'd)$ IO = Inp $'i$ $'d$ | Out $'o$ $'d$ | Tau

inductive step **where**
  step (Inp p x) (Read p f) (f x) | step (Out q x) (Write op q x) op | step Tau (Silent op) op
| $op \in_c ops \implies$ step io op $op' \implies$ step io (Choice ops) op'

definition sim where
  sim R op_1 op_2 = $(\forall io\ op_1'.$ step io op_1 $op_1' \longrightarrow (\exists op_2'.$ step io op_2 $op_2' \wedge R$ op_1' $op_2'))$

coinductive bisim (**infix** $\sim$ 40) **where**
  sim ($\sim$) op_1 $op_2 \implies$ sim ($\sim$) op_2 $op_1 \implies op_1 \sim op_2$

Figure 2 Labeled transition system and strong bisimilarity

in the countable set type *cset*, which means that any corecursive call guarded by the Choice constructor must be applied using the map function on *cset*, i.e., the image function `$_c$ (written infix). To this end, we pull the corecursive call out of the singleton set. We also use similar transformations in some subsequent operators. To improve readability, we only present the user-friendly derived equations, such as spin_op_code, rather than the original **corec** command.

2.1 Operators Equivalences

The initial examples $\oslash$, $\odot$, and $\otimes$ represent operators that appear to be identical to an external observer examining their (lack of) communication. However, these operators are syntactically different, which prevents us from proving their equality using *op*'s coinduction principle. Syntactic equality proves too rigid to support more nuanced semantic reasoning.

This limitation leads us to Milner's classic approach of using *bisimilarity* [25], which is an equivalence relation based on labeled transition systems (LTS). In an LTS, transitions are associated with labels that represent the performed actions. For our operators, these actions can be a read, a write, or a silent action (τ). Two operators are bisimilar if their corresponding transition systems can mutually simulate each other's transitions. In this work, we formalize both *strong* bisimilarity, which requires matching transitions one-by-one in simulations, and *weak* bisimilarity, which allows one to abstract away finite sequences of silent actions in simulations.

In Figure 2, we define the datatype IO for labels and our LTS as the inductive predicate step, where the first three introduction rules associate the Read, Write, and Silent operator constructors with the appropriate labels. The Choice constructor does not have a label on its own; it depends on the existence of a label from a step made by an operator in its set of choices.

Strong simulation follows from the definition sim (Figure 2) that takes a relation R and two operators op_1 and op_2, which must meet the following condition: whenever op_1 steps to op_1' with a label io, there must exist a corresponding step from op_2 to some op_2' with the same label io, such that R relates op_1' and op_2'. Relation R is a *strong simulation* if $\forall op_1\ op_2.\ R\ op_1\ op_2 \longrightarrow$ sim $R\ op_1\ op_2$. Moreover, R is a *strong bisimulation* when both R and its converse R^{-1} are strong simulations. Strong bisimilarity is the largest strong bisimulation.

In Isabelle/HOL, a coinductive predicate is defined as the greatest fixed point of a corresponding predicate transformer on the predicate lattice following the Knaster—Tarski's theorem [27]. Coinductive predicates are convenient for defining strong bisimilarity bisim (denoted with $\sim$, Figure 2) because they provide a useful coinduction principle [1]. We reformulate and mildly strengthen this coinduction principle for $\sim$, to improve readability and us-

lemma ex1_op_code:
  ex1_op = Choice {Write ex1_op (1 :: *nat*) (42 :: *nat*), ⊘}$_c$
lemma ex2_op_code:
  ex2_op = Choice {Write ex2_op (1 :: *nat*) (42 :: *nat*), ex2_op}$_c$
lemma ex3_op_code:
  ex3_op = Choice {Write ex3_op (1 :: *nat*) (42 :: *nat*), Silent ex3_op}$_c$

lemma step_ex1_op_elim:
  assumes step *io* ex1_op *op*′ **obtains** *op*′ = ex1_op **and** *io* = Out 1 42
lemma step_ex1_op_intro:
  assumes *io* = Out 1 42 **and** *op*′ = ex1_op **shows** step *io* ex1_op *op*′

lemma step_ex2_op_elim:
  assumes step *io* ex2_op *op*′ **obtains** *op*′ = ex2_op **and** *io* = Out 1 42
lemma step_ex2_op_intro:
  assumes *io* = Out 1 42 **and** *op*′ = ex2_op **shows** step *io* ex2_op *op*′

lemma step_ex3_op_elim:
  assumes step *io* ex3_op *op*′
  obtains *op*′ = ex3_op **and** *io* = Out 1 42 | *op*′ = ex3_op **and** *io* = Tau
lemma step_ex3_op_intro1:
  assumes *io* = Out 1 42 **and** *op*′ = ex3_op **shows** step *io* ex3_op *op*′
lemma step_ex3_op_intro2:
  assumes *io* = Tau **and** *op*′ = ex3_op **shows** step *io* ex3_op *op*′

Figure 3 Examples of operators, and their step elimination and introduction rules

ability, by incorporating basic upto-techniques for strong bisimilarity (yielding bisimilarity up to reflexivity, up to symmetry, and up to bisimilarity) via the auxiliary inductive predicate $\mathcal{S}$:

inductive bisim_upto ($\mathcal{S}$) **for** R **where**
  $R\ x\ y \Longrightarrow \mathcal{S}\ R\ x\ y \mid x \sim y \Longrightarrow \mathcal{S}\ R\ x\ y \mid x = y \Longrightarrow \mathcal{S}\ R\ x\ y \mid \mathcal{S}\ R\ y\ x \Longrightarrow \mathcal{S}\ R\ x\ y$
lemma bisim_coinduct: $R\ op_1\ op_2 \Longrightarrow$
  $(\bigwedge op_1\ op_2\ io\ op_1'.\ R\ op_1\ op_2 \Longrightarrow \mathsf{step}\ io\ op_1\ op_1' \Longrightarrow \exists op_2'.\ \mathsf{step}\ io\ op_2\ op_2' \wedge$
  $\mathcal{S}\ R\ op_1'\ op_2') \Longrightarrow$
  $(\bigwedge op_1\ op_2\ io\ op_2'.\ R\ op_1\ op_2 \Longrightarrow \mathsf{step}\ io\ op_2\ op_2' \Longrightarrow \exists op_1'.\ \mathsf{step}\ io\ op_1\ op_1' \wedge$
  $\mathcal{S}\ R\ op_1'\ op_2') \Longrightarrow$
  $op_1 \sim op_2$

A coinduction proof that a coinductive predicate holds for given arguments thus involves exhibiting a witness relation R that relates these arguments. Isabelle's `coinduction` proof method [6] generates a canonical instantiation for R when applied to a specific goal.

This principle is first used to demonstrate that $\sim$ indeed constitutes an equivalence relation, i.e., it is reflexive, symmetric, and transitive. The next proof by coinduction shows that $\oslash \sim \otimes$, which is a consequence that neither operator can make a step. Three more intricate examples are given in Figure 3. These operators output the natural number 42 at port 1 until they may eventually stop doing so. They illustrate our general technique for proving strong (and weak) bisimilarity: we derive step elimination and introduction rules for specific operators, which allow us to prove the cases resulting from the application of the `coinduction` proof method, e.g., to prove ex1_op $\sim$ ex2_op. Note that neither ex1_op nor ex2_op are strongly bisimilar to ex3_op, because ex3_op can perform Tau actions while the other two cannot. This leads us towards the notion of weak bisimilarity, which can abstract silent steps away.

fun estep **where** estep Tau = (step Tau)$^{==}$ | estep io = step io
definition wstep io = (step Tau)** OO (estep io) OO (step Tau)**
definition wsim where
  wsim R op_1 op_2 = ($\forall io\ op_1'$. step io op_1 op_1' $\longrightarrow$ ($\exists op_2'$. wstep io op_2 op_2' $\wedge$ R op_1' op_2'))
coinductive wbisim (**infix** $\approx$ 40) **where**
  wsim ($\approx$) op_1 op_2 $\Longrightarrow$ wsim ($\approx$) op_2 op_1 $\Longrightarrow$ $op_1 \approx op_2$
inductive wbisim_upto ($\mathcal{W}$) **for** R **where**
  $R\ x\ y \Longrightarrow \mathcal{W}\ R\ x\ y \mid x \approx y \Longrightarrow \mathcal{W}\ R\ x\ y \mid x = y \Longrightarrow \mathcal{W}\ R\ x\ y \mid \mathcal{W}\ R\ y\ x \Longrightarrow \mathcal{W}\ R\ x\ y$
lemma wbisim_coinduct: R op_1 op_2 $\Longrightarrow$
  ($\bigwedge op_1\ op_2\ io\ op_1'$. R op_1 op_2 $\Longrightarrow$ step io op_1 op_1' $\Longrightarrow$ $\exists op_2'$. wstep io op_2 op_2' $\wedge$ $\mathcal{W}$ R op_1' op_2') $\Longrightarrow$
  ($\bigwedge op_1\ op_2\ io\ op_2'$. R op_1 op_2 $\Longrightarrow$ step io op_2 op_2' $\Longrightarrow$ $\exists op_1'$. wstep io op_1 op_1' $\wedge$ $\mathcal{W}$ R op_1' op_2') $\Longrightarrow$
  $op_1 \approx op_2$

Figure 4 Weak bisimilarity and corresponding coinduction principle

To define weak bisimilarity, we first define what constitutes a *weak step*, denoted by wstep in Figure 4. We use Isabelle's notation: infix OO is the relation composition, the superscript ** denotes the reflexive-transitive closure, and the superscript $^{==}$ the reflexive closure. A weak step consists of a finite, possibly empty sequence of Tau steps, followed by a single *extended step* (estep) and another sequence of Tau steps. The extended step introduces reflexivity for Tau actions, which enables operators that lack Tau steps to simulate operators that include Tau steps.

The definition of wsim is similar to sim, but it uses a weak step in the implication's conclusion. Specifically, for op_1 to be simulated by op_2, every step taken by op_1 with label io must be matched using a corresponding weak step by op_2 with the same label io. The *weak bisimulation* coinductive predicate wbisim (denoted with $\approx$) is analogous to $\sim$.

We show that $\approx$ is an equivalence relation by coinduction using wbisim_coinduct following Milner [25, p. 109]. We also show that strong bisimilarity implies weak bisimilarity: $op_1 \sim op_2 \implies op_1 \approx op_2$. Lastly, we have ex2_op $\approx$ ex3_op as an example of equivalence of operators in which one Tau step of ex3_op is simulated by a "reflexive step" of ex2_op.

2.2 Trace Equivalence

The concept of an external observer analyzing communication channels is formalized through the notion of trace. More specifically, we adopt Jonsson's view [19]: a trace is a potentially non-terminating, linearly ordered sequence of communication events produced by an operator. Hence, the meaning of *trace equivalence* is the equality of set of traces of two operators.

In Figure 5, we define visible actions using the datatype *VIO*, along with a function that transforms *VIO* into *IO*. We then introduce the concept of an operator being *finished*, meaning that the operator can no longer produce any visible actions. This is formalized using the coinductive predicate wfinished (the prefix w means we are in a weak setting that does not observe Tau steps). Next, we define the traces of an operator using the coinductive predicate wtraced, which consists of two cases: the base case, which associates the empty lazy list (LNil) with the operator being finished, and the coinductive case, which prepends (using LCons) the visible action from a weak step to a trace of the operator after this weak step. Lastly, trace equivalence $\equiv_t$ is the equality of the set of traces. As expected, we show that weak bisimilarity implies trace equivalence $op_1 \approx op_2 \implies op_1 \equiv_t op_2$.

datatype $('i, 'o, 'd)$ *VIO* = VInp $'i$ $'d$ | VOut $'o$ $'d$

fun io_of_vio **where** io_of_vio (VInp p x) = Inp p x | io_of_vio (VOut p x) = Out p x

coinductive wfinished **where**
  $(\forall op.\ op \in ops \longrightarrow$ wfinished $op)$ $\Longrightarrow$ wfinished (Choice ops)
| wfinished op $\Longrightarrow$ wfinished (Silent op)

coinductive wtraced **where**
  wfinished op $\Longrightarrow$ wtraced op LNil
| wstep (io_of_vio vio) op op' $\Longrightarrow$ wtraced op' lxs $\Longrightarrow$ wtraced op (LCons vio lxs)

definition wtraces $op = \{lxs.$ wtraced op $lxs\}$

abbreviation wtrace_equiv (**infix** $\equiv_t$ 40) **where**
  $op_1 \equiv_t op_2 \equiv$ wtraces op_1 = wtraces op_2

Figure 5 Traces and trace equivalence

2.3 Numeral Types for Ports

A typical choice for input/output port types is to use Isabelle/HOL's existing numeral types *0*, *1*, *2*, *3*... which model sets of the respective finite cardinality. For example, an operator with two inputs and three outputs can be represented with the type $(2, 3, 'd)$ *op*.

Because all Isabelle/HOL types must be nonempty, the numeral type *0* is peculiar: it is isomorphic to the type of natural numbers. Its "cardinality" is still 0, because in Isabelle's standard library the cardinality of infinite sets is defined to be 0. This raises a challenge when working with operators that should have no inputs or outputs, such as operator of type $(0, 'o, 'd)$ *op* or $('i, 0, 'd)$ *op*. Although these operators should not be capable of reading or writing data, they may use the infinitely many ports available in *0*.

To address this limitation, we introduce a type class called *defaults*, that defines a set of elements named defaults. (A similar type class singling out a particular default value already existed in Isabelle, but working with a set results in better closure properties for sum types.) We instantiate the *defaults* type class for *0* by setting its defaults to UNIV (the universal set containing all elements of the specified type). For other numeral types *1*, *2*, ..., their *defaults* instance defines defaults = {}. We also make the sum type + an instance of *defaults* by taking the union of the defaults from both sides. Hence, the type $0 + 0$, just as *0*, has no non-default elements. The set of all non-default elements is $\mathfrak{U}$ = UNIV − defaults. We introduce another type class *all_defaults* where $\mathfrak{U} = \{\}$, as a subclass of *defaults*. Naturally, we also provide instances of *all_defaults* for *0* and +. Finally, when defining operators, we assume that both input and output ports belong to the *defaults* and *countable* type classes and follow the convention to only use ports in $\mathfrak{U}_c$ (the countable set counterpart of $\mathfrak{U}$).

3 Asynchronous Dataflow Operators

This section explains the set of operators of the network algebra for asynchronous dataflow. Following Bergstra et al. [3], they are divided into two categories: 1. The *basic network algebra*, which include three operators for structuring the network: sequential composition, parallel composition, and feedback. Additionally, there are two constants: identity and transposition; 2. The additional asynchronous dataflow constants: split, sink, merge, dummy source, asynchronous copy, and asynchronous equality test. Except for the sink, all operators are stateful, maintaining internal buffers to store communication data. These buffers function as intermediate first-in-first-out communication channels where data can reside indefinitely.

```
lemma id_op_code: id_op buf = Choice
  (((λp. Read p (λx. id_op (BENQ p x buf)))`_c 𝔘_c) ∪_c
   ((λp. Write (id_op (BTL p buf)) p (BHD p buf))`_c {p ∈_c 𝔘_c | buf p ≠ []}))
abbreviation id_empty_op (ℐ) where ℐ ≡ id_op (λ_. [])
lemma transp_op_code: transp_op buf = Choice
  (((λp. Read p (λx. transp_op (BENQ p x buf)))`_c 𝔘_c) ∪_c
   ((λp. Write (transp_op (BTL p buf)) (case_sum Inr Inl p) (BHD p buf))
    `_c {p ∈_c 𝔘_c | buf p ≠ []}))
abbreviation transp_empty_op (𝒳) where 𝒳 ≡ transp_op (λ_. [])
```

Figure 6 The identity and transposition operators

3.1 Buffer Infrastructure

Most of our definitions will include a buffer function $buf :: 'a \Rightarrow 'd\ list$ (where $'a$ is the type of the relevant ports). This buffer function buf represents the operator's current state, e.g., what data has already been read (and on which ports) but not yet written. In the rest of the paper, we will make use of the following convenience functions on buffers: BHD gives the first element, on the given port, in the buffer; BTL dequeues this first element; BENQ enqueues an element to the buffer; BULK_BENQ (or infix $\gg$) concatenates the data contained in two buffers. Buffers act as queues: we add elements to the list's end and remove from its start.

```
type_alias buf = list
definition BHD :: 'a ⇒ ('a ⇒ 'd buf) ⇒ 'd where BHD p buf = hd (buf p)
definition BTL :: 'a ⇒ ('a ⇒ 'd buf) ⇒ ('a ⇒ 'd buf) where
  BTL p buf = buf(p := tl (buf p))
definition BENQ :: 'a ⇒ 'd ⇒ ('a ⇒ 'd buf) ⇒ ('a ⇒ 'd buf) where
  BENQ p x buf = buf(p := buf p @ [x])
definition BULK_BENQ (infixr ≫ 65) where buf_1 ≫ buf_2 = (λp. buf_2 p @ buf_1 p)
```

3.2 Network Algebra Operators

We present building blocks for defining arbitrary asynchronous dataflow operators. We recall the convention established in the previous section: only ports in $\mathfrak{U}$ can be used. The first two primitives are the *identity* and *transposition* operators, defined respectively as id_op and transp_op. Their code equations are given in Figure 6.

The operator id_op is a stream delayer: it can read data from any of its (non-default) input ports and enqueue it in its corresponding internal buffer; the operator can also dequeue data from any non-empty buffer and write data to the corresponding output port. The operator transp_op has type $('m + 'n, 'n + 'm, 'd)\ op$, i.e., it operates on a sum type for the input ports and transposes the ports in the output. For instance, the data read on some input port Inl p, where $p :: 'm$, is enqueued on the internal buffer of the transposition operator and may eventually be written on output port Inr p. We abbreviate the initial states of these operators (in which internal buffers are empty) by $\mathcal{I}$ and $\mathcal{X}$ respectively.

To combine operators, we define a general *composition* operator in Figure 7. To compose two operators, we must specify the *wiring* between them, as well as the buffer associated with this wiring. Given operators $op_1 :: ('i_1, 'o_1, 'd)\ op$ and $op_2 :: ('i_2, 'o_2, 'd)\ op$, the wire is a partial function $'o_1 \rightharpoonup 'i_2$ defining how op_1's output ports of are connected to op_2's input ports.

The definition of comp_op uses the choices function and sound_reads abbreviation. The function choices descends through Choice constructors until it reaches a non-Choice

```
lemma comp_op_code: comp_op wire buf op1 op2 = Choice
  (((λop. case op of
     Read p f  ⇒ Read (Inl p) (λx. comp_op wire buf (f x) op2)
   | Write op p x ⇒ (case wire p of
       None ⇒ Write (comp_op wire buf op op2) (Inl p) x
     | Some q ⇒ Silent (comp_op wire (BENQ q x buf) op op2))
   | Silent op ⇒ Silent (comp_op wire buf op op2))
    `c choices op1) ∪c
   ((λop. case op of
     Read p f ⇒ if p ∈ ran wire
       then Silent (comp_op wire (BTL p buf) op1 (f (BHD p buf)))
       else Read (Inr p) (λx. comp_op wire buf op1 (f x))
   | Write op p x ⇒ Write (comp_op wire buf op1 op) (Inr p) x
   | Silent op ⇒ Silent (comp_op wire buf op1 op))
    `c sound_reads wire buf (choices op2)))
definition pcomp_op :: ('i1,'o1,'d) op ⇒ ('i2,'o2,'d) op ⇒
    ('i1 + 'i2,'o1 + 'o2,'d) op (infixl ∥ 64) where
  op1 ∥ op2 = comp_op (λ_. None) (λ_. []) op1 op2
definition scomp_op :: ('i1,'o1,'d) op ⇒ ('o1,'o2,'d) op ⇒
    ('i1,'o2,'d) op (infixl • 65) where
  op1 • op2 = map_op projl projr (comp_op Some (λ_. []) op1 op2)
```

Figure 7 The general, parallel, and sequential composition operators

constructor. It thus computes all steps an operator can make (e.g., choices $\otimes$ = {} and Silent op' $\in_c$ choices op $\longleftrightarrow$ step Tau op op'). It returns a countable set of operators: Read, Write, and Silent are returned as singletons and for Choice ops we obtain the union of choices applied to each operator in ops; formally choices (Choice ops) = $\bigcup_c$(choices `$_c$ ops). The function sound_reads $wire$ buf ops filters the Read operators in ops to only keep those whose port either is not connected by the wire (i.e., they communicate with the external world) or the buffer on the port connected by the wire is not empty. This makes reading from the empty buffer an impossible action. The standard ran function returns the range of a partial function.

The comp_op operator proceeds by stepping one of the operators independently. In particular, either of the operators may be neglected and never take steps. This design is intentional, as it reflects the behavior of independent nodes in a decentralized network, where no global orchestrator ensures a fair or synchronized execution schedule. Moreover, the left operator is restricted to reading inputs from the external environment, while its outputs can be directed either to the wire's buffers or to the external world, depending on the wiring. Conversely, the second operator can read either from the buffer or from the external environment, again based on the wiring, but its outputs are limited to the external world. Interactions with the wire's buffers are recorded by comp_op as a silent action.

We consider two special cases for the wiring function. When $wire$ = (λ_. None), the two operators are not interacting with each other and we thus obtain *parallel* composition (infix ∥): the sum types of input and output ports reflect this. When $wire$ = Some, it means that the output ports of op_1 are connected with the input ports of op_2 so that all the data that op_2 may read must have been written by op_1. Thus, we obtain *sequential* composition (infix •). Here, map_op projl projr is applied to remove the sum types, since in this case we know that the sequential composition's input ports are the input ports of the first operator, and its output ports are the outputs ports of the second operator. Moreover, we prove

```
corec loop_op :: ('o ⇁ 'i) ⇒ ('i ⇒ 'd buf) ⇒ ('i,'o,'d) op ⇒ ('i,'o,'d) op where
  loop_op wire buf op = Choice ((λop. case op of
    Read p f ⇒ if p ∈ ran wire
      then Silent (loop_op wire (BTL p buf) (f (BHD p buf)))
      else Read p (λx. loop_op wire buf (f x))
  | Write op' p x ⇒ (case wire p of
      None ⇒ Write (loop_op wire buf op') p x
    | Some q ⇒ Silent (loop_op wire (BENQ q x buf) op'))
  | Silent op' ⇒ Silent (loop_op wire buf op'))
    `c sound_reads wire buf (choices op))
definition feedback_op (_ ↑ [66] 65) where
  op ↑ = map_op projl projl (loop_op
    (case_sum (λ_. None) (λp. if p ∈ defaults then None else Some (Inr p)))
    (case_sum undefined (λ_. [])) op)
```

Figure 8 The loop and feedback operators

inputs (comp_op *wire buf* op_1 op_2) ⊆ Inl ` inputs $op_1$ ∪ Inr ` (inputs op_2 − ran *wire*) and outputs (comp_op *wire buf* op_1 op_2) ⊆ Inl ` (outputs $op_1$ − dom *wire*) ∪ Inr ` outputs op_2.

Similarly to comp_op we define a general loop operator loop_op in Figure 8, which utilizes the choices function and is also parameterized by a wiring. The wiring determines when data is going to/coming from the outside environment or the loop's own buffer (which is both written to and read from). We give a standard wiring configuration using the sum type, where the left-side ports communicate with the external environment, and the right-side ports handle the looping-back mechanism. We call loop_op with this wiring instance the *feedback* operator (_ ↑). Notably, ports in the defaults set are excluded from looping-back, enabling the definition of feedback operators without any looping-back ports (e.g., when the sum's right-side is the *0* type). We show that loop_op has similar inputs/outputs lemmas as the ones for comp_op.

The composition and loop operators combine other operators. Thus it is useful to have congruence rules for them. Below, we show comp_op's and loop_op's congruence rules for weak bisimilarity. Similar rules also hold for strong bisimilarity as well as for the derived operators.

lemma wbisim_comp_op_cong: $op_1 \approx op_1' \implies op_2 \approx op_2' \implies$
  comp_op *wire buf* op_1 op_2 ≈ comp_op *wire buf* op_1' op_2'

lemma wbisim_loop_op_cong: $op \approx op' \implies$
  loop_op *wire buf op* ≈ loop_op *wire buf op'*

Now, we describe the remaining operators shown in Figure 9. The dummy source operator (i) is defined as the sequential composition of ⊘ with the $\mathcal{I}$. Since ⊘ does not produce any output, the identity operator, in turn, also generates no output. In contrast, the sink operator (!) is capable of reading (and ignoring) data from all its usable input ports.

To informally describe the final operators, we consider a fixed usable port p of type $'m :: \{countable, defaults\}$. The split operator ($\Lambda$) reads data from p and nondeterministically buffers it either on the left (Inl p) or on the right (Inr p). Eventually, it may send that input data out through the corresponding port (Inl p or Inr p). The merge operator ($\mathcal{V}$) acts as the dual of Λ by nondeterministically reading data from either the left (Inl p) or the right (Inr p) port and buffering it. $\mathcal{V}$ may eventually send this input data out through port p. The asynchronous copy operator ($\mathcal{C}$) behaves similarly to (Λ), but it buffers the data on both sides (Inl p) and (Inr p). Finally, the equality test operator ($\mathcal{Q}$) resembles ($\mathcal{V}$). However, it operates with optional data and only outputs the data when the heads of both buffers are equal; otherwise, it outputs None.

```
abbreviation dummy_source_op (i) where i ≡ ⊘ • ℐ
corec sink_op :: ('i :: {countable, defaults}, 'o, 'd) op (!) where
  ! = Choice ((λp. Read p (λ_. !))`_c 𝔘_c)
lemma split_op_code: split_op buf = Choice
  (((λp. Read p (λx. split_op (BENQ (Inl p) x buf)))`_c 𝔘_c) ∪_c
   ((λp. Read p (λx. split_op (BENQ (Inr p) x buf)))`_c 𝔘_c) ∪_c
   ((λp. Write (split_op (BTL p buf)) p (BHD p buf))`_c {p ∈_c 𝔘_c | buf p ≠ []}))
abbreviation split_empty_op (Λ) where Λ ≡ split_op (λ_. [])
lemma merge_op_code: merge_op buf = Choice
  (((λp. Read (Inl p) (λx. merge_op (BENQ (Inl p) x buf)))`_c 𝔘_c) ∪_c
   ((λp. Read (Inr p) (λx. merge_op (BENQ (Inr p) x buf)))`_c 𝔘_c) ∪_c
   ((λp. Write (merge_op (BTL (Inl p) buf)) p (BHD (Inl p) buf))
     `_c {p ∈_c 𝔘_c | buf (Inl p) ≠ []}) ∪_c
   ((λp. Write (merge_op (BTL (Inr p) buf)) p (BHD (Inr p) buf))
     `_c {p ∈_c 𝔘_c | buf (Inr p) ≠ []}))
abbreviation merge_empty_op (𝒱) where 𝒱 ≡ merge_op (λ_. [])
lemma acopy_op_code: acopy_op buf = Choice
  (((λp. Read p (λx. acopy_op (BENQ (Inr p) x (BENQ (Inl p) x buf))))`_c 𝔘_c) ∪_c
   ((λp. Write (acopy_op (BTL (Inl p) buf)) (Inl p) (BHD (Inl p) buf))
     `_c {p ∈_c 𝔘_c | buf (Inl p) ≠ []}) ∪_c
   ((λp. Write (acopy_op (BTL (Inr p) buf)) (Inr p) (BHD (Inr p) buf))
     `_c {p ∈_c 𝔘_c | buf (Inr p) ≠ []}))
abbreviation acopy_empty_op (𝒞) where 𝒞 ≡ acopy_op (λ_. [])
lemma aeq_op_code: aeq_op buf = Choice
  (((λp. Read (Inl p) (λx. aeq_op (BENQ (Inl p) x buf)))`_c 𝔘_c) ∪_c
   ((λp. Read (Inr p) (λx. aeq_op (BENQ (Inr p) x buf)))`_c 𝔘_c) ∪_c
   ((λp. (if BHD (Inl p) buf = BHD (Inr p) buf
     then Write (aeq_op (BTL (Inr p) (BTL (Inl p) buf))) p (BHD (Inl p) buf)
     else Write (aeq_op (BTL (Inr p) (BTL (Inl p) buf)) p None)))
     `_c {p ∈_c 𝔘_c | buf (Inl p) ≠ [] ∧ buf (Inr p) ≠ []}))
abbreviation aeq_empty_op (𝒬) where 𝒬 ≡ aeq_op (λ_. [])
```

Figure 9 Additional operators

4 Asynchronous Dataflow Properties

We show that the axioms from Tables 1, 2, and 3 presented in Bergstra et al. [3] with the exception of axiom A1 in their Table 3 are valid in our shallow embedding. We discuss insights from proving the axioms and explain formulation differences. To make referencing transparent, we will use the axiom numbering of Bergstra et al.'s, but we merge their Tables 2 and 3 (which use the same axiom numbers) into a single table.

First, we establish a notation for sequentially composing an operator with identity: $op \vdash$ denotes $op \bullet \mathcal{I}$ and $\dashv op$ is $\mathcal{I} \bullet op$. This pre-/post-composition is added in our formalization to certain lemmas to ensure that the operator exhibits the required asynchronous behavior. This allows for a more precise account on where these additions are needed. For instance, the process algebra model from Bergstra et al. [3] surrounds most operators by identities on both sides. Changes to axiom statements of this nature are highlighted in gray. In Section 5, we demonstrate how these differences can be eliminated through the introduction of a new type.

B1: $op_1 \parallel (op_2 \parallel op_3) \approx \mathsf{map\_op}\ \curvearrowright \curvearrowright\ (op_1 \parallel op_2) \parallel op_3$
B2_1: $op \parallel (\mathcal{I} :: (0, 0, 'd)\ op) \approx \mathsf{map\_op\ Inl\ Inl}\ op$
B2_2: $(\mathcal{I} :: (0, 0, 'd)\ op) \parallel op \approx \mathsf{map\_op\ Inr\ Inr}\ op$
B3: $(op_1 \bullet op_2) \bullet op_3 \approx op_1 \bullet (op_2 \bullet op_3)$
B4_1: $op\vdash \bullet \mathcal{I} \approx op\vdash$ B4_2: $\mathcal{I} \bullet \dashv op \approx \dashv op$
B5: $(op_1 \parallel op_2) \bullet (op_3 \parallel op_4) \approx (op_1 \bullet op_3) \parallel (op_2 \bullet op_4)$
B6: $\mathcal{I} \parallel \mathcal{I} \approx \mathcal{I}$ B7: $\mathcal{X} \bullet \mathcal{X} \approx \mathcal{I}$
B8: $(\mathcal{X} :: ('i + 0, 0 + 'i, 'd)\ op) \approx \mathsf{map\_op\ id\ (case\_sum\ Inr\ Inl)}\ \mathcal{I}$
B9: $\mathcal{X} \approx \mathsf{map\_op}\ \curvearrowright \curvearrowright\ (\mathcal{X} \parallel \mathcal{I}) \bullet \mathsf{map\_op\ id}\ \curvearrowleft\ (\mathcal{I} \parallel \mathcal{X})$
B10: $(\dashv op_1 \parallel \dashv op_2) \bullet \mathcal{X} \approx \mathcal{X} \bullet (op_2\vdash \parallel op_1\vdash)$
F1: $\mathcal{I}\uparrow \approx (\mathcal{I} :: (0, 0, 'd)\ op)$ F2: $\mathcal{X}\uparrow \approx \mathcal{I}$
R1: $\mathsf{Inr}\ \grave{-}\ \mathsf{inputs}\ op_1 \cap \mathsf{defaults} = \{\} \Longrightarrow \mathsf{Inr}\ \grave{-}\ \mathsf{outputs}\ op_1 \cap \mathsf{defaults} = \{\} \Longrightarrow$
$op_2 \bullet (op_1\uparrow) \approx ((op_2 \parallel \mathcal{I}) \bullet op_1)\uparrow$
R2: $\mathsf{Inr}\ \grave{-}\ \mathsf{inputs}\ op_1 \cap \mathsf{defaults} = \{\} \Longrightarrow \mathsf{Inr}\ \grave{-}\ \mathsf{outputs}\ op_1 \cap \mathsf{defaults} = \{\} \Longrightarrow$
$(op_1\uparrow) \bullet op_2 \approx (op_1 \bullet (op_2 \parallel \mathcal{I}))\uparrow$
R3: $\mathsf{Inr}\ \grave{-}\ \mathsf{inputs}\ op_2 \cap \mathsf{defaults} = \{\} \Longrightarrow\ \mathsf{Inr}\ \grave{-}\ \mathsf{outputs}\ op_2 \cap \mathsf{defaults} = \{\} \Longrightarrow$
$op_1 \parallel (op_2\uparrow) \approx (\mathsf{map\_op}\ \curvearrowleft \curvearrowleft\ (op_1 \parallel op_2))\uparrow$
R4: $\mathsf{Inr}\ \grave{-}\ \mathsf{inputs}\ op_1 \cap \mathsf{defaults} = \{\} \Longrightarrow \mathsf{Inr}\ \grave{-}\ \mathsf{outputs}\ op_1 \cap \mathsf{defaults} = \{\} \Longrightarrow$
$\mathsf{inputs}\ op_2 \cap \mathsf{defaults} = \{\} \Longrightarrow \mathsf{outputs}\ op_2 \cap \mathsf{defaults} = \{\} \Longrightarrow$
$(\dashv op_1 \bullet (\mathcal{I} \parallel op_2))\uparrow \approx ((\mathcal{I} \parallel op_2) \bullet op_1\vdash)\uparrow$
R5: $\mathsf{Inr}\ \grave{-}\ \mathsf{inputs}\ op = \{\} \Longrightarrow \mathsf{Inr}\ \grave{-}\ \mathsf{outputs}\ op = \{\} \Longrightarrow$
$\mathsf{map\_op\ Inl\ Inl}\ ((op :: ('i + 0, 'o + 0, 'd)\ op)\uparrow) \approx op$
R6: $\mathsf{Inr}\ \grave{-}\ \mathsf{inputs}\ op = \{\} \Longrightarrow \mathsf{Inr}\ \grave{-}\ \mathsf{outputs}\ op = \{\} \Longrightarrow$
$\mathsf{Inr}\ \grave{-}\ \mathsf{Inl}\ \grave{-}\ \mathsf{inputs}\ op = \{\} \Longrightarrow \mathsf{Inr}\ \grave{-}\ \mathsf{Inl}\ \grave{-}\ \mathsf{outputs}\ op = \{\} \Longrightarrow$
$(op\uparrow)\uparrow \approx (\mathsf{map\_op}\ \curvearrowright \curvearrowright\ op)\uparrow$

Table 1 Basic network algebra properties

The first set of axioms is shown in Table 1. First, we identify some notable axioms. Axiom B1 is the associativity of parallel composition and axiom B3 is the associativity of sequential composition. Notice that for B1 we need to map ports with the $\curvearrowright :: ('a + 'b) + 'c \Rightarrow 'a + 'b + 'c$ function so the sum types are properly reassociated. We also define its inverse function $\curvearrowleft$. Bergstra et al. mostly ignore such port mismatches; in a proof assistant more rigor is needed. Both parts of B4 show that an operator sequentially composed with identity can absorb another identity. This follows directly from the $\mathcal{I} \bullet \mathcal{I} \approx \mathcal{I}$ auxiliary lemma. We call such lemmas that remove identities *identity absorption*.

The feedback operator axioms require additional assumptions to guarantee that arbitrary operators do not use default ports when looping-back (on the right). These assumptions use the inverse image function $f\ \grave{-}\ B \equiv \{x.\ f\ x \in B\}$ to capture the right inputs and outputs. Recall that the wiring of $\uparrow$ forces *defaults* ports on the right to not loop back. Without these assumptions, the invariant that the ports on the right are the looping-back ones would be violated. Again, these modifications are highlighted and will be abstracted away in Section 5.

Table 2 is about the asynchronous equality test, merge, copy, split, the dummy source, and sink operators. Axioms A14, A15, A18, and A19 can be seen as a recursive characterization of equality test, merge, copy, and split. The merge and split operators satisfy fewer axioms than equality test and copy. For example, A11 first duplicates inputs and then performs an equality test on them on the subsequent operator, which is bisimilar to the identity operator. For the $\mathcal{V}$ and Λ, the axiom A11 is not valid because nondeterministic split choices do not necessarily align with the nondeterministic merge order. A related reason prevented us from proving A1 for $\mathcal{V}$. We conjecture that $(\mathcal{V} \parallel \mathcal{I}) \bullet \mathcal{V}$ and $\mathsf{map\_op}\ \curvearrowleft\ \mathsf{id}\ ((\mathcal{I} \parallel \mathcal{V}) \bullet \mathcal{V})$ are not weakly bisimilar as they make partial nondeterministic choices in different orders. We do

A1: $(\mathcal{Q} \parallel \mathcal{I}) \bullet \mathcal{Q} \approx \mathsf{map\_op}\ \curvearrowleft\ \mathsf{id}\ ((\mathcal{I} \parallel \mathcal{Q}) \bullet \mathcal{Q})$
A2: $\mathcal{X} \bullet \circledcirc \approx \circledcirc$ for $\circledcirc \in \{\mathcal{Q}, \mathcal{V}\}$ A6: $\circledast \bullet \mathcal{X} \approx \circledast$ for $\circledast \in \{\mathcal{C}, \Lambda\}$
A3$_\mathcal{Q}$: $((\mathsf{i} :: (0, 'a, 'd)\ op) \parallel \mathcal{I}) \bullet \mathcal{Q} \approx (! :: (0 + 'a, 0, 'd)\ op) \bullet \mathsf{i}$
A3$_\mathcal{V}$: $((\mathsf{i} :: (0, 'a, 'd)\ op) \parallel \mathcal{I}) \bullet \mathcal{V} \approx \mathsf{map\_op\ Inr\ id}\ \mathcal{I}$
A4: $\circledcirc \bullet ! \approx ! \parallel !$ for $\circledcirc \in \{\mathcal{Q}, \mathcal{V}\}$ A8: $\mathsf{i} \bullet \circledast \approx \mathsf{i} \parallel \mathsf{i}$ for $\circledast \in \{\mathcal{C}, \Lambda\}$
A5: $\mathcal{C} \bullet (\mathcal{C} \parallel \mathcal{I}) \approx \mathsf{map\_op\ id}\ \curvearrowleft\ (\mathcal{C} \bullet (\mathcal{I} \parallel \mathcal{C}))$
A7: $\mathcal{C} \bullet (! \parallel \mathcal{I}) \approx \mathsf{map\_op\ id\ Inr}\ \mathcal{I}$ A9: $\mathsf{i} \bullet ! \approx \mathcal{I}$
A10: $\mathcal{Q} \bullet \mathcal{C} \approx (\mathcal{C} \parallel \mathcal{C}) \bullet (\mathsf{map\_op}\ \curvearrowright \curvearrowright\ (\mathsf{map\_op}\ \curvearrowleft \curvearrowleft\ (\mathcal{I} \parallel \mathcal{X}) \parallel \mathcal{I})) \bullet (\mathcal{Q}\vdash \parallel \mathcal{Q}\vdash)$
A11: $\mathcal{C} \bullet \mathcal{Q} \approx \mathcal{I}$ A12: $\mathsf{i} \approx (\mathcal{I} :: (0, 0, 'd)\ op)$
A16: $! \approx (\mathcal{I} :: (0, 0, 'd)\ op)$ A13: $\mathsf{i} \approx \mathsf{i} \parallel \mathsf{i}$ A17: $! \approx ! \parallel !$
A14: $\mathsf{map\_op\ id\ Inl}\ (\circledcirc :: (0 + 0, 0, 'd)\ op) \approx \mathcal{I}$ for $\circledcirc \in \{\mathcal{Q}, \mathcal{V}\}$
A15: $\circledcirc \approx \mathsf{map\_op}\ \curvearrowright \curvearrowright\ (\mathsf{map\_op}\ \curvearrowleft \curvearrowleft\ (\mathcal{I} \parallel \mathcal{X}) \parallel \mathcal{I}) \bullet (\circledcirc \parallel \circledcirc)$ for $\circledcirc \in \{\mathcal{Q}, \mathcal{V}\}$
A18: $\mathsf{map\_op\ Inl\ id}\ (\circledast :: (0, 0 + 0, 'd)\ op) \approx \mathcal{I}$ for $\circledast \in \{\mathcal{C}, \Lambda\}$
A19: $\circledast \approx (\circledast \parallel \circledast) \bullet \mathsf{map\_op}\ \curvearrowright \curvearrowright\ (\mathsf{map\_op}\ \curvearrowleft \curvearrowleft\ (\mathcal{I} \parallel \mathcal{X}) \parallel \mathcal{I})$ for $\circledast \in \{\mathcal{C}, \Lambda\}$
F3: $(\mathsf{map\_op\ id\ Inr}\ \circledcirc) \uparrow \approx !$ for $\circledcirc \in \{\mathcal{Q}, \mathcal{V}\}$
F4: $(\mathsf{map\_op\ Inr\ id}\ \circledast) \uparrow \approx \mathsf{i}$ for $\circledast \in \{\mathcal{C}, \Lambda\}$
F5: $((\mathcal{I} \parallel \mathcal{C}) \bullet \mathsf{map\_op}\ \curvearrowright \curvearrowright\ (\mathcal{X} \parallel \mathcal{I}) \bullet (\mathcal{I} \parallel \mathcal{Q})) \uparrow \approx ! \bullet \mathsf{i}$

Table 2 Properties of equality test, merge, copy, split, source and sink operators

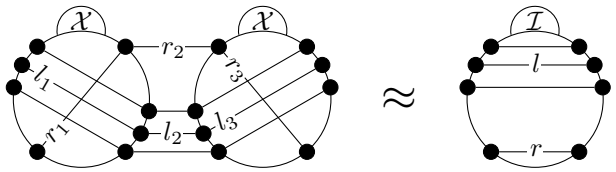

Figure 10 Generalization of axiom B7 where $l = l_1 \gg l_2 \gg l_3$ and $r = r_1 \gg r_2 \gg r_3$

believe that they are trace equivalent, but do not have a formal proof of this statement.

To prove the axioms, we follow the approach outlined in Section 2.1. This involves deriving introduction and elimination rules, such as the ones in Figure 3, for each operator and using these rules to construct the required simulations generated by the coinduction. The coinduction proofs require generalizing the operator's buffers, as using empty buffers from the notations would eventually cause us to be outside of the bisimulation during the coinduction. Consequently, we must consider arbitrary buffers and establish an invariant between them. In most cases, this is straightforward. For example, in the generalization of axiom B7 in Figure 10, the buffers from both sides are related using the $\gg$ function. In this drawing, the left-side ports are of the numeral type *3*, and the right-side ports are of type *1*. Also, we re-use the same symbols as the introduced operator notations to unambiguously identify them.

A common pattern in the proofs involves forward reasoning to move data through the buffers. For instance, consider one of the cases in the B7 proof, where $\mathcal{I}$ on the right-hand side of the weak bisimilarity produces an output from the l buffer. Then the left-hand side must also produce the same output. However, it may be the case that the buffers l_2 and l_3 are empty, and the head of l_1 must traverse the diagram to generate the output. This results in a sequence of Tau steps that move l_1's head to l_2 and then to l_3. The Isar proof language [31] offers fact chaining using **also**, which is well-suited for this kind of transitive reasoning.

The axiom A10 was a challenging one to prove as it required a non-trivial buffer invariant. We consider the diagram in Figure 11, and we assume that $a = a_1 \gg a_2 \gg a_3 \gg a_4 \gg a_5$, and similarly for b, c, and d. Furthermore, we assume $ac = ac_1 \gg ac_2$ and $bd = bd_1 \gg bd_2$. To find the invariant, we analyze the left-to-right simulation first. Assume that the left-hand side does an equality test step with $\mathcal{Q}_0$. This step can only be weakly simulated on the right-

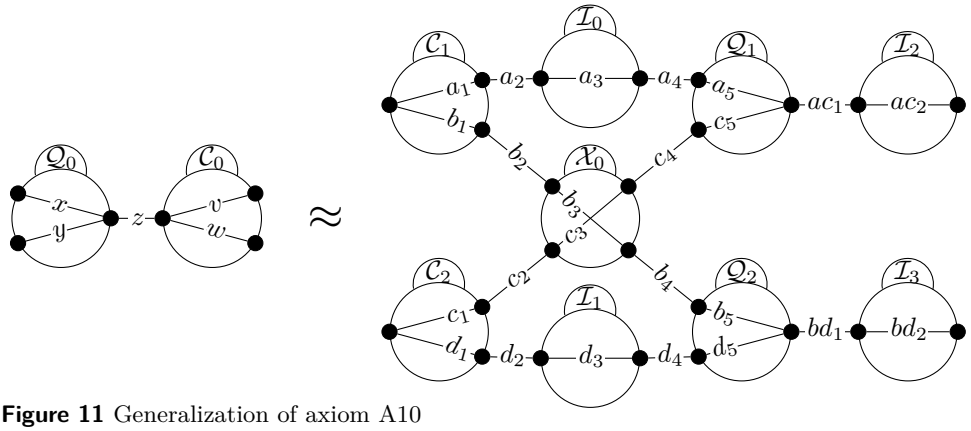

Figure 11 Generalization of axiom A10

hand side if either or both $\mathcal{Q}_1$ and $\mathcal{Q}_2$ can make some sequences of tests to catch up with $\mathcal{Q}_0$. However, there is no way to have a simulation if either $\mathcal{Q}_1$ or $\mathcal{Q}_2$ has tested more than $\mathcal{Q}_0$. The sequence of channels $a\ p$ and $c\ p$ must have the same length as they are consumed together by $\mathcal{Q}_1$, and symmetrically $b\ p$ and $d\ p$. Thus, for every port p, there exist natural numbers n, m, such that $\mathsf{drop}\ n\ (a\ p) = x\ p \wedge \mathsf{drop}\ n\ (c\ p) = y\ p \wedge \mathsf{drop}\ m\ (b\ p) = x\ p \wedge \mathsf{drop}\ m\ (d\ p) = y\ p$. As $\mathcal{Q}_1$ and $\mathcal{Q}_2$ advance independently, we maintain $n = 0 \vee m = 0$ as invariant. The already tested elements in $z \gg v$ and in $z \gg w$ can be ahead of the tested elements on the right-hand side of the equivalence (ac and bd). This means that $(z \gg v)\ p$ is precisely the same as $ac\ p$ followed by the sequence of n elements still to be pairwise tested in $a\ p$ and $c\ p$.

The established invariant must also hold for the right-to-left simulation. We assume an equality test on the right-hand side in $\mathcal{Q}_2$ testing if $\mathsf{BHD}\ p\ b_5$ equals $\mathsf{BHD}\ p\ d_5$. We must avoid testing too little on the left-hand of the equivalence side to keep the invariant. For example, if $b = x$ and $d = y$, this new equality test with b_5 and d_5 will reduce the size of b and d by one; therefore, one equality test on the left-hand side must happen as well.

5 Well-Behaved Operators

We outline the process for eliminating the differences in our lemmas compared to Bergstra et al. [3]: the gray highlights in Table 1 and Table 2. To achieve this, we define a subtype of well-behaved operators using Isabelle's **typedef** command. The new type requires its elements to be weakly bisimilar to some operator surrounded by $\dashv$ and $\vdash$.

typedef $('i :: \{countable, defaults\}, 'o :: \{countable, defaults\}, 'd)\ operator =$
$\{op :: ('i, 'o, 'd) op.\ \exists op' :: ('i, 'o, 'd)\ op.\ op \approx \dashv op' \vdash\}$

Next, we lift every definition to this new type. This includes all operators, weak bisimilarity, and map_op. The lifting is done by the **lift_definition** command [18], which requires a proof that the lifted term respects the subtype's property. When an operator has identity absorption lemmas, it can be lifted without changes; for example, $\mathcal{I}$ can be lifted directly as it satisfies $\mathcal{I} \bullet \mathcal{I} \approx \mathcal{I}$. Similarly, $\bullet$, $\|$, and $\uparrow$ are lifted directly too; they satisfy $\dashv (op_1 \bullet op_2) \vdash \approx (\dashv op_1) \bullet (op_2 \vdash)$, $\dashv (op_1 \parallel op_2) \vdash \approx (\dashv op_1 \vdash) \parallel (\dashv op_2 \vdash)$, and $\dashv (op \uparrow) \vdash \approx (\dashv op \vdash) \uparrow$.

Other operators such as $\mathcal{C}$ are lifted with no changes as well because they also have identity absorption lemmas for both sides: $\dashv \mathcal{C} \vdash \approx \mathcal{C}$. We lift the definitions of i and ! using the all_defaults type class constraint to ensure that they have no usable ports for their inputs and output, respectively. This guarantees their identity absorption lemmas. In contrast, the operators $\mathcal{Q}$ and $\mathcal{V}$ satisfy only weaker one-sided absorption lemmas: $\dashv \mathcal{Q} \vdash \approx \mathcal{Q} \vdash$, $\dashv \mathcal{V} \vdash \approx \mathcal{V} \vdash$, Hence, to make the subtype constraint hold, we lift $\dashv \mathcal{Q}$ and $\dashv \mathcal{V}$ instead of $\mathcal{Q}$ and $\mathcal{V}$.

The lifting of map_op requires $\dashv (\mathsf{map\_op}\ f\ g\ op) \vdash \approx \mathsf{map\_op}\ f\ g\ (\dashv op \vdash)$ to hold. This is not true in general, but we can impose sufficient constraints on f and g. They must maintain the exclusive use of ports in the set $\mathfrak{U}_c$ (i.e., $\forall x.\ f\ x \in \mathsf{defaults} \longrightarrow x \in \mathsf{defaults}$). The same property must also hold for the inverse of f and g because in the bisimilarity proof their inverses are necessary to construct the respective simulation. In addition, this lemma further requires that f and g and their inverses are injective on $\mathfrak{U}_c$. Fortunately, all usages of map_op in all tables meet these conditions. Lastly, the highlighted assumptions from Table 1 are also eliminated for the lifted operators because they are weakly bisimilar to $\dashv op' \vdash$ for some op', the ports of $\dashv op' \vdash = \mathcal{I} \bullet op' \bullet \mathcal{I}$ are in $\mathfrak{U}_c$, and weakly bisimilar operators have the same ports.

6 Code Generation

Our operators are infinitary in two respects: they may branch over a countably infinite set of choices and as codatatypes, they may be trees of infinite depth. We use code generation to Haskell [17] to execute operators lazily and produce prefixes of their traces up to a given depth.

A particular challenge for code generation is our usage of the abstract type of countable sets, which are originally defined as a subtype of arbitrary sets. The default code generation setup for sets supports finite and cofinite sets, but not the countable sets we are interested in. Instead, we provide a new setup for countable sets that recasts them as quotients [20] of lazy list modulo reordering of elements. For this purpose, we define the abstraction function cset_of_llist :: '*a llist* ⇒ '*a cset* which is used by the code generator as the only pseudo-constructor of countable sets: a standard way to provide code generation for quotients [16]. Then, operations on countable sets can be made executable by providing a code equation that pattern-matches on this pseudo-constructor and performs the according operation on the underlying lazy lists. For example, countable set union becomes lazy list interleaving:

```
corec (friend) linterleave :: 'a llist ⇒ 'a llist ⇒ 'a llist where
  linterleave xs ys = (case (xs, ys) of (LNil, LNil) ⇒ LNil
  | (LCons x xs', LCons y ys') ⇒ LCons x (LCons y (linterleave xs' ys'))
  | (LCons x xs', LNil) ⇒ LCons x xs' | (LNil, LCons y ys') ⇒ LCons y ys')
lemma [code]: cset_of_llist xs ∪c cset_of_llist ys = cset_of_llist (linterleave xs ys)
```

Things become more challenging for functions on countable sets of more complex types, e.g., countable sets themselves. Specifically, the above recipe fails for the function $\bigcup_c$ :: ('*a cset*) *cset* ⇒ '*a cset*. It is not a problem to define a corresponding function on lazy lists:

```
corec lmerge :: ('a llist) llist ⇒ 'a llist where
  lmerge xss = (case ldropWhile lnull xss of LNil ⇒ LNil
  | LCons xs xss ⇒ LCons (lhd xs) (linterleave (lmerge xss) (ltl xs)))
```

This definition crucially relies on Isabelle's corecursion up-to friends [5], in particular that linterleave is a friendly function (which is proved automatically for its definition). However, the following equation uses the map function on lazy lists on the left and is thus invalid code:

```
lemma ⋃c (cset_of_llist (lmap cset_of_llist xss)) = cset_of_llist (lmerge xss)
```

A dual problem occurs for functions on subtypes with compound return types. Kunčar [22, §6.4] presents and automates a solution for subtypes, which inspires our solution for quotients. We introduce a type '*a cset_llist* that is isomorphic to ('*a cset*) *llist* as a quotient of ('*a llist*) *llist* (with the abstraction function abs_cset_llist :: ('*a llist*) *llist* ⇒ '*a cset_llist* serving as the pseudo-constructor for code generation). Since this new type does not

have a compound domain, the standard lifting of lmerge to the function cset_llist_merge :: '*a cset_llist* ⇒ '*a cset* yields a valid code equation. Thus we have reduced the problem to finding an executable function cset_llist_of :: ('*a cset*) *llist* ⇒ '*a cset_llist*. This function can be defined by lifting the identity on lazy lists. And it can be made executable by lifting the lazy list constructors to '*a cset_llist* and using the following code equations:

lemma [*code*]: LNil′ = abs_cset_llist LNil
LCons′ (cset_of_llist *x*) (abs_cset_llist *xs*) = abs_cset_llist (LCons *x xs*)
cset_llist_of LNil = LNil′ cset_llist_of (LCons *x xs*) = LCons′ *x xs*

Overall, we obtain $\bigcup_c$(cset_of_llist *xss*) = cset_llist_merge (cset_llist_of *xss*), which is executable. We remark that we cannot use Lochbihler and Stoop's framework [24] for executing generated Isabelle code lazily in strict programming languages, because quotient types are not supported by that framework. Thus, we resort to Haskell as the code generation target.

7 Related Work

This work provides mechanized proofs of most asynchronous dataflow algebra axioms by Bergstra et al [3]. They present the two original algebra instances: the relation-based stream transformer model and the process algebra model using the Algebra of Communicating Processes (ACP) [2]. Our operator definitions closely follow their ACP counterparts. One difference is that most of their operators are defined by explicitly pre- and post-composing with identity operators (⊣ *op* ⊢ instead of *op*), whereas we work with the core operators and only use the additional identities where they are necessary. For the stream transformer model, Broy and Ştefănescu [9] provide pen-and-paper proofs for a slightly different algebra.

The recent Isabelle/HOL formalization of time-aware stream processing [30] bears some similarities to our work. It defines stream processing operators using a codatatype with a single constructor that combines both reading and writing: a finite list of outputs is produced as a reaction to reading an input. In contrast, we separate both operations (so that the operator can decide what it wants to do next) and additionally support nondeterministic choices and silent steps. Their approach covers time metadata associated with processed events (which is necessary for aggregating computations) but does not support loops. Moreover, their representation of multiple inputs/outputs using sum types requires to linearize all inputs into a single lazy list. We plan to extend our network algebra operators with capabilities to observe time metadata.

Chappe et al. [14] formalize choice trees—a semantic domain for modeling nondeterministic, recursive, and impure programs, based on interaction trees developed by Xia et al [32] in Rocq. (Later, interaction trees have also been formalized in Isabelle [15].) Our operators can be seen as a domain specific variant of choice trees tailored to asynchronous dataflow. In particular, our types make inputs and output ports explicit, which is used in composition and feedback operators. In contrast, choice trees are monadically composed along their only output. Due to Isabelle's simple types, our operators are restricted to use a fixed type parameter for the data items. In a dependently-typed system, the type of data could be indexed by the corresponding port, much like the interaction type depends on the action type in choice/interaction trees.

The streaming language Flo [23] serves as a framework for representing streaming computations originating from different systems like Flink [12] and DBSP [11]. Flo makes two assumptions on stream processing systems: streaming progress and eager execution. The first assumption means that a program produces as much output as it can given the observed inputs; the second ensures that this output is deterministic. One practical challenge of their approach is the need to demonstrate that operators satisfy both assumptions.

Synchronous dataflow languages have received some attention in mechanization, with the Lustre [13] compiler Vélus [7] being a prominent example. Bergstra et al. [4] also develop algebraic foundations for synchronous dataflow, which subtly differ from the asynchronous ones.

Milner's classic chapter on weak bisimilarity [25] provided us with valuable insights for our proofs. The network algebra axioms were well-behaved in a sense that we did not have to use advanced up-to techniques for weak bisimilarity, such as the ones developed by Pous [28, 29].

8 Discussion

We have presented our Isabelle formalization of a semantic domain for asynchronous dataflow along with operators that satisfy all but one of Bergstra et al.'s axioms [3]. Our formalization comprises 28 000 lines of definitions and proofs. A major bulk of this work are the coinduction proofs of the 51 axioms, each spanning between 12 and almost 4 000 lines.

In terms of proof assistant technology, we make use of Isabelle's support for codatatypes, in particular using nested corecursion through the type of countable sets. While we could define all our desired operators, some definitions needed adjustments from their most natural (presented) formulations to be accepted by **corec**. Our formalization may give insight to developers on how to make this advanced command even more convenient to use.

Although not visible in the end product, we made great use of the Sketch and Explore tool when proving the axioms. Typically, we would invoke *auto* and obtain a number of subgoals to work on. (In case of A10 there are more than fifty subgoals.) The Sketch and Explore tool presents these subgoals as an Isar proof outline, so that once finished the *auto* invocation can become a terminal proof method, which is deemed better proof style.

Our formalization provides reusable artifacts beyond dataflow. In particular, we have outlined an approach for emulating empty types through the use of the *defaults* type class. We have also sketched how to generate executable code for functions on quotient types with compound domain types. Although we were mostly interested in countable sets, we presented a general technique dualizing the existing solution for a similar problem with subtypes.

An anecdote from our formalization journey is illustrative. After having completed all axiom proofs, we have noticed that we actually proved the Λ version of A5, which according to Bergstra et al. is not expected to hold in all models. This made us suspicious and we were inspecting the axioms extra carefully. The proof assistant did not help here: Isabelle had happily accepted our proofs. It turned out that we made a mistake in statements: instead of reassociating sum types we were rotating them, which has trivialized several axioms. We rushed to fix the mistake and proved all axioms but A1 for $\mathcal{Q}$ and $\mathcal{V}$. After some (useful) struggle with Isabelle, we realized that our definition of $\mathcal{Q}$ deviated from Bergstra et al.'s: instead of replacing non-equal pairs by None we were dropping such pairs entirely. Fixing $\mathcal{Q}$ meant that we had to revisit all our proofs involving it, including A10. Here, Isabelle turned out to be tremendously helpful: the proof structure did not change and local fixes (in places Isabelle pointed to) were sufficient to bring us back on track. We finally also managed to prove A1 for $\mathcal{Q}$ (but not for $\mathcal{V}$).

We are working on proving A1 for $\mathcal{V}$ using trace equivalence. As future work, we will formalize the history model (i.e., give operators a semantics in terms of input/output relations on lazy lists) and demonstrate that it suffers from the Brock–Andersen anomaly [8, 19]. We have already formalized the trace semantics, which constitutes Jonsson's workaround for this issue [19]. However, a characterization of traces for composition and feedback in terms of their arguments' traces is missing. As a long term objective, we want to make time a first-class citizen in our dataflows, so that we can formalize programs expressed in data stream processing frameworks such as Timely Dataflow [26]. Our work provides a solid foundation for asynchronous dataflow into which we will incorporate promising initial progress in that direction [10, 30].

References

1. Jesper Bengtson. *Formalising process calculi*. PhD thesis, Uppsala University, 2010.
2. Jan A. Bergstra and Jan Willem Klop. Process algebra for synchronous communication. *Inf. Control.*, 60(1-3):109–137, 1984. doi:10.1016/S0019-9958(84)80025-X.
3. Jan A. Bergstra, Cornelis A. Middelburg, and Gheorghe Stefanescu. Network algebra for asynchronous dataflow. *Int. J. Comput. Math.*, 65(1-2):57–88, 1997. doi:10.1080/00207169708804599.
4. Jan A. Bergstra, Cornelis A. Middelburg, and Gheorghe Stefanescu. Network algebra for synchronous dataflow. *CoRR*, abs/1303.0382, 2013. URL: http://arxiv.org/abs/1303.0382.
5. Jasmin Christian Blanchette, Aymeric Bouzy, Andreas Lochbihler, Andrei Popescu, and Dmitriy Traytel. Friends with benefits - implementing corecursion in foundational proof assistants. In Hongseok Yang, editor, *ESOP 2017*, volume 10201 of *LNCS*, pages 111–140. Springer, 2017. doi:10.1007/978-3-662-54434-1_5.
6. Jasmin Christian Blanchette, Johannes Hölzl, Andreas Lochbihler, Lorenz Panny, Andrei Popescu, and Dmitriy Traytel. Truly modular (co)datatypes for Isabelle/HOL. In Gerwin Klein and Ruben Gamboa, editors, *ITP 2014*, volume 8558 of *LNCS*, pages 93–110. Springer, 2014. doi:10.1007/978-3-319-08970-6_7.
7. Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. A formally verified compiler for Lustre. In Albert Cohen and Martin T. Vechev, editors, *PLDI 2017*, pages 586–601. ACM, 2017. doi:10.1145/3062341.3062358.
8. J Dean Brock and William B Ackerman. Scenarios: A model of non-determinate computation. In *Formalization of Programming Concepts: International Colloquium Peniscola, Spain, April 19–25, 1981 Proceedings*, pages 252–259. Springer, 1981.
9. Manfred Broy and Gheorghe Stefanescu. The algebra of stream processing functions. *Theor. Comput. Sci.*, 258(1-2):99–129, 2001. doi:10.1016/S0304-3975(99)00322-9.
10. Matthias Brun, Sára Decova, Andrea Lattuada, and Dmitriy Traytel. Verified progress tracking for Timely Dataflow. In Liron Cohen and Cezary Kaliszyk, editors, *ITP 2021*, volume 193 of *LIPIcs*, pages 10:1–10:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.ITP.2021.10.
11. Mihai Budiu, Tej Chajed, Frank McSherry, Leonid Ryzhyk, and Val Tannen. DBSP: automatic incremental view maintenance for rich query languages. *Proc. VLDB Endow.*, 16(7):1601–1614, 2023. doi:10.14778/3587136.3587137.
12. Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015. URL: http://sites.computer.org/debull/A15dec/p28.pdf.
13. Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A declarative language for programming synchronous systems. In *POPL 1987*, pages 178–188. ACM Press, 1987. doi:10.1145/41625.41641.
14. Nicolas Chappe, Paul He, Ludovic Henrio, Yannick Zakowski, and Steve Zdancewic. Choice trees: Representing nondeterministic, recursive, and impure programs in Coq. *Proc. ACM Program. Lang.*, 7(POPL):1770–1800, 2023. doi:10.1145/3571254.
15. Simon Foster, Chung-Kil Hur, and Jim Woodcock. Formally verified simulations of state-rich processes using interaction trees in Isabelle/HOL. In Serge Haddad and Daniele Varacca, editors, *CONCUR 2021*, volume 203 of *LIPIcs*, pages 20:1–20:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.CONCUR.2021.20.
16. Florian Haftmann, Alexander Krauss, Ondrej Kuncar, and Tobias Nipkow. Data refinement in Isabelle/HOL. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *ITP 2013*, volume 7998, pages 100–115. Springer, 2013. doi:10.1007/978-3-642-39634-2_10.
17. Florian Haftmann and Tobias Nipkow. Code generation via higher-order rewrite systems. In Matthias Blume, Naoki Kobayashi, and Germán Vidal, editors, *FLOPS 2010*, volume 6009 of *LNCS*, pages 103–117. Springer, 2010. doi:10.1007/978-3-642-12251-4_9.

18 Brian Huffman and Ondrej Kuncar. Lifting and transfer: A modular design for quotients in Isabelle/HOL. In Georges Gonthier and Michael Norrish, editors, *CPP 2013*, volume 8307 of *LNCS*, pages 131–146. Springer, 2013. doi:10.1007/978-3-319-03545-1_9.

19 Bengt Jonsson. A fully abstract trace model for dataflow and asynchronous networks. *Distributed Computing*, 7:197–212, 1994.

20 Cezary Kaliszyk and Christian Urban. Quotients revisited for Isabelle/HOL. In William C. Chu, W. Eric Wong, Mathew J. Palakal, and Chih-Cheng Hung, editors, *SAC 2011*, pages 1639–1644. ACM, 2011. doi:10.1145/1982185.1982529.

21 Martin Kleppmann and Jay Kreps. Kafka, Samza and the Unix philosophy of distributed data. *IEEE Data Eng. Bull.*, 38(4):4–14, 2015. URL: http://sites.computer.org/debull/A15dec/p4.pdf.

22 Ondřej Kunčar. *Types, abstraction and parametric polymorphism in higher-order logic*. PhD thesis, Fakultät für Informatik, Technische Universität München, 2016.

23 Shadaj Laddad, Alvin Cheung, Joseph M. Hellerstein, and Mae Milano. Flo: A semantic foundation for progressive stream processing. *Proc. ACM Program. Lang.*, 9(POPL):241–270, 2025. doi:10.1145/3704845.

24 Andreas Lochbihler and Pascal Stoop. Lazy algebraic types in Isabelle/HOL, 2018.

25 Robin Milner. *Communication and concurrency*. PHI Series in computer science. Prentice Hall, 1989.

26 Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In Michael Kaminsky and Mike Dahlin, editors, *SOSP 2013*, pages 439–455. ACM, 2013. doi:10.1145/2517349.2522738.

27 Lawrence C. Paulson. *A Fixedpoint Approach to (Co)Inductive and (Co)Datatype Definitions*, page 187–211. MIT Press, Cambridge, MA, USA, 2000.

28 Damien Pous. Weak bisimulation up to elaboration. In Christel Baier and Holger Hermanns, editors, *CONCUR 2006*, volume 4137 of *LNCS*, pages 390–405. Springer, 2006. doi:10.1007/11817949_26.

29 Damien Pous. New up-to techniques for weak bisimulation. *Theor. Comput. Sci.*, 380(1-2):164–180, 2007. doi:10.1016/J.TCS.2007.02.060.

30 Rafael Castro Gonçalves Silva and Dmitriy Traytel. Time-aware stream processing in Isabelle/HOL. In Tobias Nipkow, Lawrence C. Paulson, and Makarius Wenzel, editors, *Isabelle workshop 2024*, pages 1–19, 2024. URL: https://files.sketis.net/Isabelle_Workshop_2024/Isabelle_2024_paper_8.pdf (accessed: March 2, 2024).

31 Makarius Wenzel et al. The Isabelle/Isar reference manual, 2004.

32 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. ACM Program. Lang.*, 4(POPL):51:1–51:32, 2020. doi:10.1145/3371119.

33 Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: fault-tolerant streaming computation at scale. In Michael Kaminsky and Mike Dahlin, editors, *SOSP 2013*, pages 423–438. ACM, 2013. doi:10.1145/2517349.2522737.