# Stream Types

JOSEPH W. CUTLER, University of Pennsylvania, USA
CHRISTOPHER WATSON, University of Pennsylvania, USA
PHILLIP HILLIARD, University of Pennsylvania, USA
HARRISON GOLDSTEIN, University of Pennsylvania, USA
CALEB STANFORD, University of California, Davis, USA
BENJAMIN C. PIERCE, University of Pennsylvania, USA

We propose a rich foundational theory of typed data streams and stream transformers, motivated by two high-level goals: (1) the type of a stream should be able to express complex *sequential patterns* of events over time, and (2) it should describe the *parallel structure* of the stream to enable deterministic stream processing on parallel and distributed systems. To this end, we introduce *stream types*, with operators capturing sequential composition, parallel composition, and iteration, plus a core calculus of *transformers* over typed streams which naturally supports a number of common streaming idioms, including punctuation, windowing, and parallel partitioning, as first-class constructions. The calculus exploits a Curry-Howard-like correspondence with an ordered variant of the logic of Bunched Implication to program with streams compositionally and uses Brzozowski-style derivatives to enable an incremental, event-based operational semantics. To validate our design, we provide a reference interpreter and machine-checked proofs of the main results.

**ACM Reference Format:**
Joseph W. Cutler, Christopher Watson, Phillip Hilliard, Harrison Goldstein, Caleb Stanford, and Benjamin C. Pierce. 2023. Stream Types. 1, 1 (July 2023), 68 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 INTRODUCTION

What is the type of a stream? A straightforward answer, dating back to the early days of functional programming [16], is that a stream is an unbounded sequence of items of a single fixed type, produced by one part of a system (or the external world) and consumed by another. This perspective has been immensely successful, with the programming models exposed by the most popular distributed stream processing eDSLs (e.g., Flink [17, 25], Beam [30], Storm [29], and Heron [26]) typically offering just one type, `Stream t`.

However, this homogeneous treatment of streams leaves something to be desired in the world of modern stream processing. For one thing, streaming data sometimes arrives at a processing node from multiple sources *in parallel*. Using arrival times to impose an "incidental" order on such parallel data can make it difficult to ensure that processing is deterministic because downstream results may then depend on external factors like network latency [49, 55]. Another issue is that temporal patterns like *bracketedness* (every "begin" event has a following "end") or streams with exactly $k$ events are invisible in the stream's type. Programmers cannot trust the type system to ensure these properties when producing a stream, nor can they rely on them when consuming a stream; see Section 2 for examples.

Authors' addresses: Joseph W. Cutler, jwc@seas.upenn.edu, University of Pennsylvania, Philadelphia, Pennsylvania, USA; Christopher Watson, ccwatson@seas.upenn.edu, University of Pennsylvania, Philadelphia, Pennsylvania, USA; Phillip Hilliard, pdh@seas.upenn.edu, University of Pennsylvania, Philadelphia, Pennsylvania, USA; Harrison Goldstein, hgo@seas.upenn.edu, University of Pennsylvania, Philadelphia, Pennsylvania, USA; Caleb Stanford, cdstanford@ucdavis.edu, University of California, Davis, Davis, California, USA; Benjamin C. Pierce, bcpierce@cis.upenn.edu, University of Pennsylvania, Philadelphia, Pennsylvania, USA.

2023. XXXX-XXXX/2023/7-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

We offer a new logical foundation for typed stream processing that can describe streams with *both* complex sequential patterns and parallel structure. On this foundation, we build a calculus called $\lambda^{\text{ST}}$ that is (a) expressive and type-safe for streams with complex temporal patterns and (b) deterministic, even in the presence of parallel inputs.

Programs in $\lambda^{\text{ST}}$ are intuitively batch processors that operate over entire streams at once. But, since streams are in general unbounded, stream transformers can't actually wait for "the entire input stream" to arrive before producing any output. The operational semantics of the $\lambda^{\text{ST}}$ calculus is therefore *incremental*, producing partial outputs from partial inputs on the fly. Concretely, a $\lambda^{\text{ST}}$ program is interpreted as a function mapping any prefix of its input(s) to a prefix of its output plus a "derivative" term that is ready to transform the rest of the inputs to the rest of the output.

Our stream types include two kinds of products, one representing a pair of streams in temporal sequence, the other a pair of streams in parallel. This structure is inspired both by Concurrent Kleene Algebras [35, 45], which syntactically describe partially ordered series/parallel data, and work by Alur et al. [3] and Mamouras et al. [49], where streams are modeled as partially ordered sets. We find a suitable proof theory for this two-product formalism in a variant of O'Hearn and Pym's Logic of Bunched Implications (BI) [52]. BI has famously been used as a foundation for separation logic [54], where the "separating conjunction" allows for local reasoning about separate regions of the heap in imperative programs. In $\lambda^{\text{ST}}$, we replace spatial separation with *temporal separation*: one product describes pairs of streams separated sequentially in time; the other, independent pairs of streams whose elements may arrive in interleaved fashion.

Concretely, our contributions are:

(1) We propose *stream types*, a type discipline for distributed stream processing that generalizes the traditional homogeneous view to a richer nested-parallel-and-sequential structure.
(2) We define a calculus $\lambda^{\text{ST}}$ of stream processing transformers, inspired by a Curry-Howard-like correspondence with an ordered variant of BI. Terms in $\lambda^{\text{ST}}$ are high level programs in a functional style that conceptually transform whole streams.
(3) We equip $\lambda^{\text{ST}}$ with an operational semantics interpreting terms as incremental transformers that accept and produce finite prefixes of streams. Our main technical result is a powerful *homomorphism theorem* (Theorem 3.3) guaranteeing that the result of a transformer does not depend on how the input stream is divided into successive prefixes.
(4) We formalize the type system and semantics in Coq and prove that the incremental semantics is deterministic: all interleavings of parallel substreams yield the same final result.
(5) We demonstrate by example how $\lambda^{\text{ST}}$ enables type-safe programming for streams with complex patterns and prevents nondeterminism and how programming patterns from stream processing practice are elegantly supported by this richer model, including MapReduce-like pipelines, temporal integrity constraints, windowing, punctuation, parallelism control, routing, and side outputs.
(6) We show that the high-level operational semantics of $\lambda^{\text{ST}}$ can be implemented atop conventional sequential streams by compiling stream-typed terms to string transducers, opening a path to a future distributed implementation based on an existing industrial-strength runtime.

The rest of the paper is structured as follows. Section 2 explores some concrete cases where $\lambda^{\text{ST}}$'s structured types can prevent common stream processing bugs and enable cleaner programming patterns. Section 3 presents Kernel $\lambda^{\text{ST}}$, a minimal subset with just the features needed to state and understand our main results. Section 4 extends this presentation to the full $\lambda^{\text{ST}}$. Section 5 develops several examples and argues that $\lambda^{\text{ST}}$ solves the problems presented in Section 2. Section 6 shows how to compile $\lambda^{\text{ST}}$ terms to string transducers. Sections 7 and 8 discuss related and future work. Overviews of the Coq formalization of our main results and our prototype interpreter in Haskell

can be found in Appendix A, technical details omitted from the main paper in Appendix B, and expanded versions of the examples in Appendix C.

## 2 MOTIVATING EXAMPLES

*Types for temporal invariants.* Consider a stream of brightness data coming from a motion sensor, where each event in the stream is a number between 0 and 100. Suppose we want a stream transformer that acts as a threshold filter, sending out a "`Start`" event when the brightness level goes above level 50, forwarding along brightness values until the level dips below the threshold, and sending a final "`Stop`" event. For example:

$$11, 30, 52, 56, 53, 30, 10, 60, 10, \ldots \implies \texttt{Start}, 52, 56, 53, \texttt{Stop}, \texttt{Start}, 60, \texttt{Stop}, \ldots$$

The output of the transformer should satisfy the following *temporal invariant*: each start event must be followed by one or more data events, and then one end event. Unfortunately, usual stream processing systems would give this transformation a type that looks like `Stream Int` $\rightarrow$ `Stream (Start + Int + Stop)`, which expresses only the types of events in the output, not the temporal invariant on them.

These simple types become even more of a problem when *consuming* streams. Suppose a second transformer wanted to consume the output stream of type `Stream (Start + Int + Stop)` and compute the average brightness between each start/end pair. We know a priori that the stream is well bracketed, but the type does not say so. Thus, the transformer must *re-parse* the stream to compute the averages, requiring additional logic for various special cases (e.g., empty start/end pairs) that cannot actually occur in the real stream!

In $\lambda^{\text{ST}}$, we can express this invariant with the type $(\texttt{Start} \cdot \texttt{Int} \cdot \texttt{Int}^\star \cdot \texttt{End})^\star$, specifying that the stream consists of a start message, at least one `Int`, and an end message, repeatedly. A well-typed transformer with this output type is guaranteed to enforce the desired invariant; conversely, a downstream transformer can assume that its input will adhere to the invariant.

*Enforcing deterministic parallelism.* A second limitation of homogeneous streams is that they impose a *total ordering* on their component events.[1] In other words, for each pair of events in the stream, the transformer can tell which came first. Imposing a total ordering like this is problematic in a world where stream transformers work over data that is logically partially ordered—e.g., because it comes from separate sources. For example, consider a pair of sensors, each producing one reading per second and and sending them different network connections to a single transformer that averages them pairwise, producing a composite reading each second.

A natural way to do this is to merge the two streams, group adjacent pairs of elements (i.e., impose a size-two tumbling window), and average the pairs, but this is subtly wrong: a network delay could cause a pair of consecutive elements in the merged stream to be from the same sensor, after which the averages will all be bogus. The problem here is that this transformer is not deterministic: its result can depend on external factors like network latency. Bugs of this type can easily occur in practice [49, 55] and can be very difficult to track down, since they may only manifest under rare conditions [43].

Once again, this is a failure of type structure. In $\lambda^{\text{ST}}$, we would give the merged stream the type $(\texttt{Sensor1} \| \texttt{Sensor2})^\star$, capturing the fact that it is a stream of *parallel pairs* of readings from the two sensors. We can write a strongly typed merge operator that produces this type, given parallel streams of type $\texttt{Sensor1}^\star$ and $\texttt{Sensor2}^\star$. This merge operator is deterministic because all

[1] In practical stream processing systems, this is often a total ordering *per key* for a parallelized stream—cf. `KeyedStream` in Flink—but the same objections apply.

well-typed $\lambda^{\mathrm{ST}}$ programs are (Section 3.4); operationally, it waits for events to arrive on both of its input streams before sending them along as a pair.

## 3 KERNEL $\lambda^{\mathrm{ST}}$

In this section, we define the most important constructors of stream types together with the corresponding features of the term language; these form the "kernel" of the $\lambda^{\mathrm{ST}}$ calculus. The rest of the types and terms of full $\lambda^{\mathrm{ST}}$ will layered on bit by bit in Section 4.

The *concatenation type* constructor describes streams that *vary* over time: if $s$ and $t$ are types of streams, then a stream of type $s \cdot t$ behaves first like a stream of type $s$, and then, if the $s$ part finishes, like a stream of type $t$. Operationally, streams of type $s \cdot t$ can include a *punctuation marker* [60] indicating where they cross over from their $s$ part to their $t$ part. For example, streams of type $s^\star$ are distinguishable from streams of type $s^\star \cdot s^\star$ because a transformer accepting the latter can see when its input crosses from the first $s^\star$ to the second.

The *parallel type* constructor, written $s \| t$, describes streams with two parallel substreams of types $s$ and $t$. Each element in a parallel stream is tagged to indicate which substream it belongs to. This means that streams of type $s \| t$ are isomorphic, but not identical, to streams of type $t \| s$, and similarly $\texttt{Int}^\star \| \texttt{Int}^\star$ is not the same as $\texttt{Int}^\star$. Semantically, the $s$ and $t$ components are produced and consumed independently: a transformer that produces $s \| t$ may send out an entire $s$ first and then a $t$, or an entire $t$ and then the $s$, or any interleaving of the two. Conversely, a transformer that accepts $s \| t$ must handle all these possibilities uniformly by processing the $s$ and $t$ independently.

Parallel types can be combined with concatenation types in interesting ways. For example, a stream of type $(s \| t) \cdot r$ consists of a stream of interleaved items from $s$ and $t$, followed (once all the $s$'s and $t$'s have arrived) by a stream of type $r$. By contrast, a stream of type $(s \cdot t) \| (s' \cdot t')$ has two parallel components, one a stream described by $s$ followed by a stream described by $t$ and the other an $s'$ followed by a $t'$. The fact that the parallel type is on the outside means that the change-over points from $s$ to $t$ and $s'$ to $t'$ are completely independent.

The base type 1 describes a stream containing just one data item, itself a unit value. The other base type is $\varepsilon$, the type of the empty stream containing no data; it is the unit for both the $\cdot$ and $\|$ constructors—i.e., $s \cdot \varepsilon$, $\varepsilon \cdot s$, $\varepsilon \| s$ and $s \| \varepsilon$ are all equivalent to $s$, in the sense that there are $\lambda^{\mathrm{ST}}$ transformers that convert between them. In summary, the Kernel $\lambda^{\mathrm{ST}}$ stream types are given by the grammar on the top left in Figure 1. (Of course, these kernel types can only describe streams of fixed, finite size. In Section 4.3 we will introduce unbounded streams via the Kleene star type $s^\star$.)

What about terms? Our goal is to develop a language of core terms $e$, typed by stream types, where well-typed terms $x : s \vdash e : t$ are interpreted as incremental stream transformers accepting a stream described by $s$ and producing one described by $t$. But one input stream is not enough: in order to be compositional, the type system needs to be able to handle stream transformers with multiple parallel and sequential inputs. To enable this, we draw upon results from proof theory for insight. Both the types $s \cdot t$ and $s \| t$ are *product types*, in the sense that streams of these types should contain the data of a stream of type $s$ and a stream of type $t$—although the temporal structure differs between the two. In situations where a logic (or type theory) includes two products, standard techniques from proof theory tell us that the corresponding typing judgment requires a context with two *context formers*.

The first of these, written with a comma ($\Gamma$, $\Delta$), is standard in intuitionistic logics. The second, written with a semicolon ($\Gamma$; $\Delta$), is the interesting one, behaving intuitively like the parallel type constructor. A context $\Gamma$, $\Delta$ describes inputs to a transformer arriving from two different parts of the environment, one structured according to $\Gamma$ and the other according to $\Delta$. On the other hand, the semicolon corresponds to concatenation: a context $\Gamma$; $\Delta$ means that data will first arrive from

$$s, t, r := 1 \mid \varepsilon \mid s \cdot t \mid s \| t \qquad \Gamma ::= \cdot \mid \Gamma, \Gamma \mid \Gamma; \Gamma \mid x : s$$

$$\frac{\Gamma \vdash e_1 : s \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash (e_1, e_2) : s \| t}\ \textsc{Par-R} \qquad \frac{\Gamma(x : s,\ y : t) \vdash e : r}{\Gamma(z : s \| t) \vdash \mathtt{let}_{\Gamma(-)}\ (x, y)\ =\ z\ \mathtt{in}\ e : r}\ \textsc{Par-L}$$

$$\frac{\Gamma \vdash e_1 : s \qquad \Delta \vdash e_2 : t}{\Gamma; \Delta \vdash (e_1; e_2) : s \cdot t}\ \textsc{Cat-R} \qquad \frac{\Gamma(x : s;\ y : t) \vdash e : r}{\Gamma(z : s \cdot t) \vdash \mathtt{let}_{\Gamma(-)}\ (x; y)\ =\ z\ \mathtt{in}\ e : r}\ \textsc{Cat-L}$$

$$\frac{}{\Gamma \vdash \mathtt{sink} : \varepsilon}\ \textsc{Eps-R} \qquad \frac{}{\Gamma \vdash () : 1}\ \textsc{One-R} \qquad \frac{}{\Gamma(x : s) \vdash (x : s @ \Gamma(-)) : s}\ \textsc{Var}$$

$$\frac{pf : \Gamma \leq \Gamma' \qquad \Gamma' \vdash e : s}{\Gamma \vdash \mathtt{subctx}\,(pf, e) : s}\ \textsc{Subctx}$$

Fig. 1. Kernel $\lambda^{\mathrm{ST}}$ syntax and typing rules

the environment according to $\Gamma$, then according to $\Delta$. Such *bunched* contexts were first introduced in the logic of Bunched Implication [52], the basis of modern separation logic[54]. In BI, the separating context former is commutative, while our semicolon context former is *not*.

Formally, stream contexts are drawn from the grammar at the top right of Figure 1. Our typing judgment is now $\Gamma \vdash e : s$. Intuitively, this means that $e$ is a transformer from a collection of streams structured like $\Gamma$ to a single stream structured like $s$.

The *structural rules* for manipulating these contexts will determine which sequents are provable and hence which programs are type-correct. For example, including an exchange rule for the comma connective—allowing $\Gamma$, $\Delta$ to be rewritten as $\Delta$, $\Gamma$—ensures that $s \| t$ is interprovable with $t \| s$; a proof of this fact will be a transformer that swaps the parallel components of the stream. On the other hand, $s \cdot t$ is manifestly different from $t \cdot s$: in a stream of the first type, $s$ arrives before $t$; in the second, $t$ before $s$. In other words, concatenation is an *ordered* product. The consequence of this is that our type system does not include an exchange rule for semicolon contexts. A contraction rule for semicolon—converting from $\Gamma$ to $\Gamma; \Gamma$—is similarly undesirable, as every use of this structural rule would require the runtime system to save and then "replay" a potentially unbounded incoming stream.

In summary, our type system is substructural. The semicolon context former is ordered (no exchange) and affine (no contraction), while the comma context former is fully structural. Both context formers are associative, with the empty context serving as a unit for both.

### 3.1 Kernel Typing Rules

The typing rules for Kernel $\lambda^{\mathrm{ST}}$ are collected in Figure 1. For technical reasons, the rules are presented in sequent-calculus style, rather than the more familiar natural deduction style. The main difference is that sequent calculi have left and right rules—describing how to eliminate a connective when it's in the context or the result type, respectively—as opposed to natural deduction's introduction and elimination rules. (In the presence of a Cut rule, the two presentations are equivalent. The semantics of Cut itself is a bit complex, though, and we choose the sequent-calculus presentation to isolate this complexity in one place.)

The most straightforward typing rule is the right rule for parallel (Par-R). From a context $\Gamma$, we can produce a stream of type $s \| t$ by producing $s$ and $t$ independently from $\Gamma$, using transformers

$e_1$ and $e_2$. We write the combined transformer as a "parallel pair", $(e_1, e_2)$. Semantically, this transformer operates by copying the inputs arriving on $\Gamma$, passing the copies to $e_1$ and $e_2$, and pairing up the outputs into a parallel stream. Similarly, the Cat-R rule is used to produce a stream of type $s \cdot t$. It uses a similar pairing syntax—if term $e_1$ has type $s$ and $e_2$ has type $t$, then the "sequential pair" $(e_1; e_2)$ has type $s \cdot t$—but the context in the conclusion differs. Since $e_1$ needs to be run before $e_2$, the part of the input stream that $e_1$ depends on ($\Gamma$) must arrive before the part that $e_2$ depends on ($\Delta$). Semantically, this term will operate by accepting data from the $\Gamma$ part of the context and running $e_1$; once the $\Gamma$ part is finished and the $\Delta$ part starts to arrive, it will switch to running $e_2$.

These right rules describe how to *produce* a stream of parallel or concatenation type; the corresponding left rules describe how to *use* a variable of one of these types appearing somewhere in the context. Syntactically, the terms take the form of let-bindings: we can deconstruct variables of type $s \cdot t$ or $s \| t$ as pairs of variables of type $s$ and $t$, connected by either ; or , as appropriate. We use the standard BI notation $\Gamma(-)$ for a context with a hole, and $\Gamma(\Delta)$ when this hole has been filled with the context $\Delta$. In particular, $\Gamma(x : s)$ is a context with a distinguished variable $x$ at some leaf.

The Par-L rule says that if $z$ is a variable of type $s \| t$ somewhere in the context, we can let-bind it as a pair of variables $x$ and $y$ of type $s$ and $t$ and use these in a continuation term $e$ of type $r$. When typing $e$, the variables $x$ and $y$ appear in the same position as the original variable $z$, but separated by a comma—i.e., $x$ and $y$ are assumed to arrive in parallel. Similarly, the parallel rule Cat-L says that, if a variable $z$ of type $s \cdot t$ appears in the context, it can be let-bound to a pair of variables $x$ and $y$ of types $s$ and $t$, which are again used in the continuation $e$. This time, though, $x$ and $y$ are separated by a semicolon—i.e., the substream bound to $x$ will arrive first, followed by the substream bound to $y$.

Eps-R and One-R are the right rules for the two base types, witnessed by the terms `sink` and `()`. Semantically, `sink` does nothing: it accepts inputs on $\Gamma$ and produces no output. On the other hand, `()` emits a unit value as soon as it receives its first input, and then never emits anything else.

The variable rule (Var) says that, if $x : s$ is a variable somewhere in the context, then we can simply return it. Semantically, it works by dropping everything in the context except for the $s$-typed data for $x$, which it forwards along. The corresponding proof term is $(x : s @ \Gamma(-))$, where $x$ is the variable itself, $s$ is its type, and $\Gamma(-)$ is surrounding context. The annotations $s$ and $\Gamma(-)$ are included because they will be needed by the operational semantics. Because the input to a term will itself be tree-structured, the context location $\Gamma(-)$ is required to look up the data corresponding to $x$ in the input tree. And the output type $s$ is required when this variable lookup comes back empty handed because the prefix of the context that has arrived so far stops before it gets to any data corresponding to this variable; in this case, the semantics of the Var rule needs to construct an "empty" value, whose shape depends on the type $s$. (Although these annotations are required at runtime, we expect they will be inferrable in a high-level language based on $\lambda^{\mathrm{ST}}$—see Section 8.)

Finally, the rule SubCtx encodes all of the structural rules as a subtyping relation on the context. For example, the weakening rule for semicolon contexts is written, $\Gamma; \Delta \leq \Gamma$, and the exchange rule is $\Gamma, \Delta \leq \Delta, \Gamma$. Each of these structural rules has semantic content. For example, choosing to use the comma contraction rule $\Gamma \leq \Gamma, \Gamma$ requires duplicating the input at runtime. Because of this, the structural rules are marked in the syntax explicitly as `subctx` $(pf, e)$, where $pf$ is a proof term describing which structural rules structural rules were used. See Appendix B.7 for details.

*Examples.* To show the typing rules in action, here are two small examples of transformers written in Kernel $\lambda^{\mathrm{ST}}$.[2] The first is a simple "parallel-swap" transformer, which accepts a stream $z$ of type

[2]We elide the annotations on variables and let-bindings, as well as terms for structural rules, following the style used for the examples in Section 5. Fully explicit versions can be found in Appendix C, and elaboration from a high level language to the core is discussed in Section 8.

$$\frac{}{\mathtt{oneb} : \mathtt{batch}\,(1)} \qquad \frac{}{\mathtt{epsb} : \mathtt{batch}\,(\varepsilon)}$$

$$\frac{b : \mathtt{batch}\,(s) \qquad b' : \mathtt{batch}\,(t)}{\mathtt{catb}\,(b, b') : \mathtt{batch}\,(s \cdot t)} \qquad \frac{b : \mathtt{batch}\,(s) \qquad b' : \mathtt{batch}\,(t)}{\mathtt{parb}\,(b, b') : \mathtt{batch}\,(s \| t)}$$

$$\frac{b : \mathtt{batch}\,(s)}{b : \mathtt{batch}\,((x : s))} \qquad \frac{b : \mathtt{batch}\,(\Gamma) \qquad b' : \mathtt{batch}\,(\Gamma')}{\mathtt{catb}\,(b, b') : \mathtt{batch}\,(\Gamma; \Gamma')} \qquad \frac{b : \mathtt{batch}\,(\Gamma) \qquad b' : \mathtt{batch}\,(\Gamma')}{\mathtt{parb}\,(b, b') : \mathtt{batch}\,(\Gamma, \Gamma')}$$

Fig. 2. Batches of types and contexts

$s \| t$, and outputs a stream of type $t \| s$, swapping the "positions" of the parallel substreams: $z : s \| t \vdash \mathtt{let}\ (x, y) = z\ \mathtt{in}\ (y, x) : t \| s$. It works by splitting the variable $z : s \| t$ into variables $x : s$ and $y : t$ and yielding a parallel pair with the order reversed. The second example is a "prepend" transformer, which takes a variable $x : s$ and outputs a stream of type $1 \cdot s$, prepending a unit value to the front: $x : s \vdash ((); x) : 1 \cdot s$.

### 3.2 Batches and Batch Processing Semantics

Although our goal is to develop an incremental semantics for stream transformers, it turns out that we need a "batch semantics" as a subroutine, where the full contents of a stream is provided to a transformer and it produces its entire output all at once. A *batch*, then, is a structured value representing the full history of a completed stream.

Stream types dictate the shape of their batches. In particular, a batch of a parallel stream type $s \| t$ is a pair of a batch of $s$ and a batch of $t$. Crucially, this definition encodes no information about any particular interleaving of the batch of $s$ and the batch of $t$. Similarly, a batch of a concatenation $s \cdot t$ is also a batch of $s$ and a batch of $t$. The grammar of batches is

$$b ::= \mathtt{oneb} \mid \mathtt{epsb} \mid \mathtt{catb}\,(b, b') \mid \mathtt{parb}\,(b, b')$$

where $\mathtt{catb}\,(b, b')$ and $\mathtt{parb}\,(b, b')$ construct batches for parallel and concatenation types and oneb and epsb are the unique batches of types 1 and $\varepsilon$. Batches are given types by the judgment $b : \mathtt{batch}\,(s)$, the rules of which are given in Figure 2. This relation is further lifted to contexts: $\Gamma,\Delta$ and $\Gamma;\Delta$ are the batches of $\mathtt{parb}\,(b, b')$ and $\mathtt{catb}\,(b, b')$, respectively, when $b$ and $b'$ are batches of $\Gamma$ and $\Delta$.

The batch semantics is defined by the judgment $b \Rightarrow e \Rightarrow b'$, pronounced "running term $e$ on batch $b$ produces batch $b'$" (see Figure 3). The following theorem establishes the correctness of the semantics: well typed terms $\Gamma \vdash e : s$ take batches of $\Gamma$ to batches of $s$.

Theorem 3.1 (Batch Safety). *If $\Gamma \vdash e : s$ and $b : \mathtt{batch}\,(\Gamma)$, then there is a unique $b'$ such that $b \Rightarrow e \Rightarrow b'$; moreover, $b' : \mathtt{batch}\,(s)$.*

This and all of the theorems in the paper are formalized in Coq (see Appendix A) and discussed in more detail in the Appendix B.

The cases for the two pairing constructs in the batch semantics are straightforward. In B-Par-R, we take the input batch, pass it to both $e_1$ and $e_2$, and then combine the results as a parallel pair of batches. In B-Cat-R, the input must be a concatenation batch because the conclusion of the Cat-R rule has a semicolon context $\Gamma;\Delta$. We pass the first batch to $e_1$ and the second to $e_2$, and we combine the results as a concatenation pair of batches. On the other hand, the batch semantics for let-bindings is trivial, since is it easy to verify that a batch of $\Gamma(z : s \cdot t)$ is exactly the same as a

$$\frac{b \Rightarrow e_1 \Rightarrow b_1 \qquad b \Rightarrow e_2 \Rightarrow b_2}{b \Rightarrow (e_1, e_2) \Rightarrow \mathsf{parb}\,(b_1, b_2)}\ \text{B-Par-R} \qquad \frac{b \Rightarrow e \Rightarrow b'}{b \Rightarrow \mathtt{let}_{\Gamma(-)}\ (z,x)\ =\ y\ \mathtt{in}\ e \Rightarrow b'}\ \text{B-Par-L}$$

$$\frac{b_1 \Rightarrow e_1 \Rightarrow b_1' \qquad b_2 \Rightarrow e_2 \Rightarrow b_2'}{\mathsf{catb}\,(b_1, b_2) \Rightarrow (e_1; e_2) \Rightarrow \mathsf{catb}\,(b_1', b_2')}\ \text{B-Cat-R} \qquad \frac{b \Rightarrow e \Rightarrow b'}{b \Rightarrow \mathtt{let}_{\Gamma(-)}\ (z;x)\ =\ y\ \mathtt{in}\ e \Rightarrow b'}\ \text{B-Cat-L}$$

$$\frac{}{b \Rightarrow \mathtt{sink} \Rightarrow \mathsf{epsb}}\ \text{B-Eps-R} \qquad \frac{}{b \Rightarrow () \Rightarrow \mathsf{oneb}}\ \text{B-One-R} \qquad \frac{b \rightsquigarrow_{\Gamma(-)} b'}{b \Rightarrow (x : s @ \Gamma(-)) \Rightarrow b'}\ \text{B-Var}$$

$$\frac{\mathsf{coeBatch}\,(pf, b) \sim b' \qquad b' \Rightarrow e \Rightarrow b''}{b \Rightarrow \mathsf{subctx}\,(pf, e) \Rightarrow b''}\ \text{B-SubCtx}$$

Fig. 3. Batch semantics rules (input and output batched are red, terms are blue)

batch for $\Gamma(x : s;\ y : t)$, and similarly for parallel types and comma contexts. For this reason, both left rules can simply pass their input batches to the continuation and return the result.

The rules B-Eps-R and B-One-R simply return the unique batches of their respective types.

The variable rule (B-Var) takes the input batch $b$ and uses the context location $\Gamma(-)$ to look up the part of $b$ that corresponds to the variable $x$. This lookup is accomplished with an auxiliary relation called *batch projection*, written $b \rightsquigarrow_{\Gamma(-)} b'$ and defined in Appendix B.5. For example, suppose $b = \mathsf{catb}\,(b_1, \mathsf{parb}\,(b_2, b_3))$ is a batch for the context $(x : s;\ (y : t,\ z : r))$. Then we have $b \rightsquigarrow_{(x:s;\,(-,\,z:r))} b_2$ holds, i.e., $b_2$ is the batch located at the hole where $y$ occurs.

The rule for context subtyping accepts a batch $b$, rearranges it by transporting it along the subtyping relation $pf$, and then runs it through $e$. This rule uses another auxiliary relation called *batch transport* and written $\mathsf{coeBatch}\,(pf, b) \sim b'$. This relation transforms a batch for $\Gamma$ into a (unique) batch for $\Gamma'$ when $pf : \Gamma \leq \Gamma'$. See Appendix B.7 for details.

### 3.3 Prefixes and Incremental Semantics

We're now ready to discuss the *incremental* semantics of Kernel $\lambda^{\mathrm{ST}}$. The natural notion of "value" for an incremental semantics is finite prefixes of streams: that is, a well-typed term $\Gamma \vdash e : s$ should accept any prefix of a stream of type $\Gamma$ and produce a prefix of a stream of type $s$.

(Ultimately, we will be interested in running transformers one event at a time, as discussed in Section 6. Even so, the generalization to arbitrary prefixes here is needed to define the semantics compositionally, since the behavior of a composite transformer in response to a single event may involve generating and processing multiple events internally.)

The possible prefixes of a stream can be calculated from its type. In particular, there are two prefixes of a stream of type 1: the empty prefix, written $\mathsf{onepA}$, and the prefix containing the single element (), written $\mathsf{onepB}$. Similarly, the unique stream of type $\varepsilon$ has a single prefix, the empty prefix, which we write $\mathsf{epsp}$.

What about $s \| t$? A parallel stream of type $s \| t$ is conceptually a pair of independent streams of type $s$ and $t$, so a prefix of a parallel stream should be a pair $\mathsf{parp}(p_1, p_2)$, where $p_1$ is a prefix of a stream of type $s$, and $p_2$ is a prefix of a stream of type $t$. As with batches of streams of parallel type, this definition includes no information about orderings between the two parallel components: the prefix $\mathsf{parp}(p_1, p_2)$ equally represents a situation where $p_1$ arrived first and the $p_2$ did, one where $p_2$ arrived before $p_1$, and one where the data for $p_1$ and $p_2$ arrived in some interleaved order. In a nutshell, this definition is what guarantees deterministic processing. By representing all possible

$$\frac{}{\texttt{epsp} : \texttt{prefix}\,(\varepsilon)} \qquad \frac{}{\texttt{onepA} : \texttt{prefix}\,(1)} \qquad \frac{}{\texttt{onepB} : \texttt{prefix}\,(1)}$$

$$\frac{p : \texttt{prefix}\,(s) \quad p' : \texttt{prefix}\,(t)}{\texttt{parp}(p, p') : \texttt{prefix}\,(s \| t)} \qquad \frac{p : \texttt{prefix}\,(s)}{\texttt{catpA}(p) : \texttt{prefix}\,(s \cdot t)} \qquad \frac{b : \texttt{batch}\,(s) \quad p : \texttt{prefix}\,(t)}{\texttt{catpB}(b, p) : \texttt{prefix}\,(s \cdot t)}$$

Fig. 4. Prefixes for types

interleavings using the same prefix value, we ensure that a transformer that operates on these values cannot possibly depend on ordering information that isn't present in the type.

Finally, let's consider the prefixes of streams of type $s \cdot t$. One case is a prefix that only includes data from $s$ because it is cut off before reaching the point where the $s \cdot t$ stream stops being $s$ and starts being $t$. We write such a prefix as $\texttt{catpA}(p)$, with $p$ a prefix of type $s$. The other case is where the prefix does include the crossover point—i.e., it consists of a complete batch of $s$ plus a prefix of $t$. We write this as $\texttt{catpB}(b, p)$, with $b$ a batch of $s$ and $p$ a prefix of $t$.

We formalize all these possibilities as a judgment $p : \texttt{prefix}\,(s)$ (see Figure 4). We further lift this judgment to contexts, with prefixes of streams of comma contexts behaving like prefixes of streams of parallel type and prefixes of streams of semicolon contexts behaving like prefixes of streams of concatenation type.

Every type $s$ has a distinguished *empty prefix*, written $\texttt{emp}_s$ and defined by straightforward recursion on $s$. There is also a natural operation that takes a batch and turns it into a prefix, "forgetting" the fact that it was a complete batch. We write this operation as $(b)^\circ$. See Appendix B.1 for details.

*Derivatives.* After a well-typed transformer $\Gamma \vdash e : s$ accepts a prefix $p$ of type $\Gamma$ and produces a prefix $p'$ of type $s$, it should transition to a new transformer $e'$ that is prepared to accept the rest of the input stream to produce the rest of the output stream. In other words, $e'$ should have type "the rest of a stream of type $s$, after $p'$" in context "the rest of a stream of type $\Gamma$, after $p$".

The "rest" of a type or context is, intuitively, its *derivative* in the sense of standard Brzozowski derivatives of regular expressions [15]. Formally, we define a 3-place relation, $\delta_p\,(s) \sim s'$, pronounced "the derivative of $s$ with respect to $p$ is $s'$" (see Figure 5). This relation is actually a partial function that is defined when $p : \texttt{prefix}\,(s)$; in this case, we write $\delta_p\,(s)$ for the unique $s'$ such that $\delta_p\,(s) \sim s'$.

The derivative of the type 1 with respect to the empty prefix onepA is 1 (the "rest" of the stream is the entire stream), and the derivative with respect to the full prefix onepB is $\varepsilon$ (there is no more stream left after the unit element has arrived). For parallel, the derivative is taken component-wise. The most interesting cases are those for the concatenation type. If the prefix has the form $\texttt{catpA}(p)$, the derivative $\delta_{\texttt{catpA}(p)}\,(s \cdot t)$ is $\left(\delta_p\,(s)\right) \cdot t$, i.e., some of the $s$ has gone by but not all, and we still expect $t$ to come after it. On the other hand, if the prefix has the form $\texttt{catpB}(b, p)$, the derivative $\delta_{\texttt{catpB}(b,p)}\,(s \cdot t)$ is just $\delta_p\,(t)$, i.e., the $s$ component is complete, and the rest of the stream is just the part of $t$ after $p$.

*Incremental Semantics.* The incremental semantics is given by a judgment $p \Rightarrow e \downarrow e' \Rightarrow p'$, pronounced "running the core term $e$ on the input prefix $p$ yields the output prefix $p'$ and steps to $e'$." The following correctness theorem establishes correctness for this semantics: if we run a well-typed core term on a prefix from the context type, it will return a prefix with the result type and step to a term that can accept the rest of the context and produce the rest of the result.

$$\frac{}{\delta_{\mathsf{epsp}}\,(\varepsilon) \sim \varepsilon} \qquad \frac{}{\delta_{\mathsf{onepA}}\,(1) \sim 1} \qquad \frac{}{\delta_{\mathsf{onepB}}\,(1) \sim \varepsilon} \qquad \frac{\delta_p\,(s) \sim s'}{\delta_{\mathsf{catpA}(p)}\,(s \cdot t) \sim s' \cdot t}$$

$$\frac{\delta_p\,(t) \sim t'}{\delta_{\mathsf{catpB}(b,p)}\,(s \cdot t) \sim t'} \qquad \frac{\delta_p\,(s) \sim s' \qquad \delta_{p'}\,(t) \sim t'}{\delta_{\mathsf{parp}(p,p')}\,(s \| t) \sim s' \| t'}$$

Fig. 5. Derivatives

Theorem 3.2 (Correctness of the Incremental Semantics). *If* $\Gamma \vdash e : s$ *and* $p : \mathtt{prefix}\,(\Gamma)$, *then there are unique* $e'$ *and* $p'$ *such that* $p \Rightarrow e \downarrow e' \Rightarrow p'$; *moreover,* $p' : \mathtt{prefix}\,(s)$ *and* $\delta_p\,(\Gamma) \vdash e' : \delta_{p'}\,(s)$.

The rules for the incremental semantics of Kernel $\lambda^{\mathrm{ST}}$ are gathered in Figure 6 and described in the remainder of this subsection; the full set of rules for all of $\lambda^{\mathrm{ST}}$ can be found in Appendix B.10.

*Incremental Semantics of the Right Rules.* The right rules for parallel and concatenation are the simplest to understand, so we address them first. For P-Par-R, we accept a prefix $p$ and use it to run the component terms $e_1$ and $e_2$, independently producing outputs $p_1$ and $p_2$ and stepping to new terms $e_1'$ and $e_2'$. The pair term $(e_1, e_2)$ then steps to $(e_1', e_2')$ and produces the output $\mathtt{parp}(p_1, p_2)$.

There are two right rules for the concatenation pair $(e_1; e_2) : s \cdot t$. Because the conclusion of the corresponding typing rule ensures that the input context has the form $\Gamma;\Delta$, there are two possible shapes of input prefixes: either $\mathtt{catpA}(p)$, with $p$ a prefix of type $\Gamma$, or else $\mathtt{catpB}(b, p)$, with $b$ a batch of type $\Gamma$ and $p$ a prefix of type $p$. If the input is $\mathtt{catpA}(p)$, the rule P-Cat-R-1 applies: we run $e_1$ with $p$, which returns $p'$ as output and $e_1'$ as a new term. The original term then steps to $(e_1'; e_2)$ and returns $\mathtt{catpA}(p')$. (Since the input prefix includes no data for the second half of the context, we do not need to run $e_2$ at all.) On the other hand, if the input is $\mathtt{catpB}(b, p)$, the rule P-Cat-R-2 applies. Here, the data required to run $e_1$ has arrived as a full batch $b$, so we run $e_1$ using the batch semantics to produce an output batch $b'$. We then use the prefix $p$ to run $e_2$, returning $p'$ and stepping to $e_2'$. The entire term then outputs $\mathtt{catpB}(b', p')$ and steps to $e_2'$. Note that the pair is eliminated in the process: we step from $(e_1; e_2)$ to just $e_2'$. To see why, note that, for the correctness theorem to hold, the resulting term must have type $\delta_{\mathtt{catpB}(b',p')}\,(s \cdot t) = \delta_p\,(t)$ in context $\delta_{\mathtt{catpB}(b,p)}\,(\Gamma;\Delta) = \delta_p\,(\Delta)$—but this is the typing for $e_2'$.

*Incremental Semantics of Context Subtyping.* The incremental semantics for context subtyping is conceptually similar to the batch semantics. Given $pf : \Gamma \leq \Gamma'$, we accept a prefix $p$ of type $\Gamma$ in the input, coerce it with a relation $\mathtt{coePfx}\,(pf, p) \sim p'$ to a prefix of type $\Gamma'$, pass it to the premise term $e$, and return the result $p''$. The difference comes in how we decide what term to step to. In the recursive call, the premise term $e$ steps to $e'$, which is well typed in context $\delta_{p'}\,(\Gamma')$. Meanwhile, the overall conclusion must be well typed in context $\delta_p\,(\Gamma)$. These two contexts are related by a subtyping rule—intuitively, the one given by the *derivative* of the proof term $pf$ with respect to $p$. To calculate this derivative, we use yet another relation $\delta_p\,(pf) \sim pf'$ (defined in Figure B.66 in Appendix B), which takes a prefix $p$ of type $\Gamma$ together with a proof term $pf : \Gamma \leq \Gamma'$ and produces a proof term $pf' : \delta_p\,(\Gamma) \leq \delta_{p'}\,(\Gamma')$. In the rule P-SubCtx-1, we use this to update the proof term and step to the term $\mathtt{subctx}\,(pf', e')$. However, it is sometimes the case that $\delta_p\,(\Gamma) = \delta_{p'}\,(\Gamma')$ on the nose, in which case the structural rule is no longer needed. We write this as $\delta_p\,(pf) \sim \bot$, and use it in P-SubCtx-2, where we simply step to the term $e'$.

$$\frac{p@\Gamma(-) \rightsquigarrow \bot@\Gamma'(-)}{p \Rightarrow (x : s@\Gamma(-)) \downarrow (x : s@\Gamma'(-)) \Rightarrow \mathsf{emp}_s}\ \text{P-Var-1}$$

$$\frac{p@\Gamma(-) \rightsquigarrow p'@\Gamma'(-)}{p \Rightarrow (x : s@\Gamma(-)) \downarrow (x : \delta_p(s)@\Gamma'(-)) \Rightarrow p'}\ \text{P-Var-2} \qquad \frac{p@\Gamma(-) \rightsquigarrow b}{p \Rightarrow (x : s@\Gamma(-)) \downarrow \mathsf{sink}_b \Rightarrow (b)^\circ}\ \text{P-Var-}$$

$$\frac{p \Rightarrow e_1 \downarrow e_1' \Rightarrow p_1 \qquad p \Rightarrow e_2 \downarrow e_2' \Rightarrow p_2}{p \Rightarrow (e_1, e_2) \downarrow (e_1', e_2') \Rightarrow \mathsf{parp}(p_1, p_2)}\ \text{P-Par-R}$$

$$\frac{(p@\Gamma(-) \rightsquigarrow \bot@\Gamma'(-)) \vee (p@\Gamma(-) \rightsquigarrow p'@\Gamma'(-)) \qquad p \Rightarrow e \downarrow e' \Rightarrow p'}{p \Rightarrow \mathsf{let}_{\Gamma(-)}\ (x,y) = z\ \mathsf{in}\ e \downarrow \mathsf{let}_{\Gamma'(-)}\ (x,y) = z\ \mathsf{in}\ e' \Rightarrow p'}\ \text{P-Par-L-1}$$

$$\frac{p@\Gamma(-) \rightsquigarrow \_@p \Rightarrow e \downarrow e' \Rightarrow p'}{p \Rightarrow \mathsf{let}_{\Gamma(-)}\ (x,y) = z\ \mathsf{in}\ e \downarrow e' \Rightarrow p'}\ \text{P-Par-L-2}$$

$$\frac{p \Rightarrow e_1 \downarrow e_1' \Rightarrow p'}{\mathsf{catpA}(p) \Rightarrow (e_1; e_2) \downarrow (e_1'; e_2) \Rightarrow \mathsf{catpA}(p')}\ \text{P-Cat-R-1}$$

$$\frac{b \Rightarrow e_1 \Rightarrow b' \qquad p \Rightarrow e_2 \downarrow e_2' \Rightarrow p'}{\mathsf{catpB}(b,p) \Rightarrow (e_1; e_2) \downarrow e_2' \Rightarrow \mathsf{catpB}(b', p')}\ \text{P-Cat-R-2}$$

$$\frac{(p@\Gamma(-) \rightsquigarrow \bot@\Gamma'(-)) \vee (p@\Gamma(-) \rightsquigarrow \mathsf{catpA}(\_)@\Gamma'(-)) \qquad p \Rightarrow e \downarrow e' \Rightarrow p'}{p \Rightarrow \mathsf{let}_{\Gamma(-)}\ (z;x) = y\ \mathsf{in}\ e \downarrow \mathsf{let}_{\Gamma'(-)}\ (z;x) = y\ \mathsf{in}\ e' \Rightarrow p'}\ \text{P-Cat-L-1}$$

$$\frac{(p@\Gamma(-) \rightsquigarrow \mathsf{catpB}(\_,\_)@\_) \vee (p@\Gamma(-) \rightsquigarrow b) \qquad p \Rightarrow e \downarrow e' \Rightarrow p'}{p \Rightarrow \mathsf{let}_{\Gamma(-)}\ (z;x) = y\ \mathsf{in}\ e \downarrow e' \Rightarrow p'}\ \text{P-Cat-L-2}$$

$$\frac{}{p \Rightarrow \mathsf{sink} \downarrow \mathsf{sink} \Rightarrow \mathsf{epsp}}\ \text{P-Eps-R} \qquad \frac{}{p \Rightarrow () \downarrow \mathsf{sink} \Rightarrow \mathsf{onepB}}\ \text{P-One-R}$$

$$\frac{\mathsf{coePfx}(pf, p) \sim p' \qquad p' \Rightarrow e \downarrow e' \Rightarrow p'' \qquad \delta_p(pf) \sim pf'}{p \Rightarrow \mathsf{subctx}(pf, e) \downarrow \mathsf{subctx}(pf', e') \Rightarrow p''}\ \text{P-SubCtx-1}$$

$$\frac{\mathsf{coePfx}(pf, p) \sim p' \qquad p' \Rightarrow e \downarrow e' \Rightarrow p'' \qquad \delta_p(pf) \sim \bot}{p \Rightarrow \mathsf{subctx}(pf, e) \downarrow e' \Rightarrow p''}\ \text{P-SubCtx-2}$$

Fig. 6. Incremental semantics of Kernel $\lambda^{\mathsf{ST}}$

*Incremental Semantics of Variables and Prefix Projection.* The rules for a variable term $(x : s@\Gamma(-))$ are a bit more involved, but the basic idea is straightforward: given a prefix $p$ of type $\Gamma(x : s)$, we use the location $\Gamma(-)$ of $x$ within the context to traverse $p$ and locate the sub-prefix corresponding to $x$. A slight complication arises from the fact that traversing $p$ to look for the data corresponding to $x$ might lead to three different results. First, the input prefix might include *none* of the data for the variable $x$, as in the prefix $p = \mathsf{catpA}(p')$ for the context $\Gamma_0 = (y : t; (x : s; z : r))$, using the one-hole context $\Gamma(-) = (y : t; (-; z : r))$. Second, the input prefix might include a *prefix* of the data for the variable $x$, as in the prefix $\mathsf{catpB}(b, \mathsf{catpA}(p))$ for the same $\Gamma_0$. In this case, $p$ is the result of projecting out the data for $x$. And third, the input might include an entire *batch*

corresponding to the variable $x$, as in the prefix $\mathtt{catpB}(b, \mathtt{catpB}(b', p))$ for $\Gamma_0$. Here, the batch $b'$ is the result of projecting out $x$.

To account for these three situations, we define a trio of relations. The four-place relation $p@\Gamma(-) \rightsquigarrow \bot@\Gamma'(-)$ holds in the first situation, when data for $x$ has not arrived yet. Next, the relation $p@\Gamma(-) \rightsquigarrow p'@\Gamma'(-)$ holds in the second situation, where $p'$ is the prefix corresponding to $x$. Last, the relation $p@\Gamma(-) \rightsquigarrow b$ holds when there is a batch $b$ located at $\Gamma(-)$ in $p$. In the first two relations, $\Gamma'(-)$ is the one-hole context corresponding the location of $x$ in the derivative of $\Gamma(x : s)$ with respect to $p$. These take the place of the original $\Gamma(-)$ in the term that $(x : s@\Gamma(-))$ steps to. The rules defining (slightly more general versions of) these relations can be found in Appendix B.5.

Now we need three variable rules, one for each possibility. When the prefix $p$ includes no data for $x$ (so $p@\Gamma(-) \rightsquigarrow \bot@\Gamma'(-)$ holds), P-Var-1 applies. We return $\mathtt{emp}_s$, the empty prefix of type $s$, and step to the term $(x : s@\Gamma'(-))$ with the updated location of $x$ in the remaining context. If $p$ includes a prefix $p'$ for $x$, P-Var-2 applies. We return the prefix $p'$ and step to the term $\left(x : \delta_{p'}\ (s)\ @\Gamma'(-)\right)$ with the updated location of $x$ and its updated type after $p'$ has been sent out. Last, if $p$ includes an entire batch $b$ for $x$, P-Var-3 applies. We return $(b)^\circ$ (the result of turning $b$ into a prefix) and step to $\mathtt{sink}_b$, a term that produces no output ($\mathtt{sink}_b$ is defined (in Appendix B.9) by recursion on $b$ and has type $\delta_{(b)^\circ}\ (s)$, as required by the correctness theorem).

*Incremental Semantics of Left Rules.* The prefix-to-prefix parts of the left rules for concatenation and parallel are straightforward: we accept a prefix $p$ of type $\Gamma(z : s \otimes t)$, where $\otimes$ is one of the two products, and pass it on to the continuation term.

For both connectives, the term that we step to depends on the "size" of the input prefix $p$. This term must be well typed in context $\delta_p\ (\Gamma(z : s \otimes t))$, but, if $p$ is large enough, the derivative context will no longer include the connective being eliminated, in which case we must remove the `let` binding. For example, given $(z : s \| t;\ u : r) \vdash \mathtt{let}\ (x, y)\ =\ z\ \mathtt{in}\ u : r$, if we have $p = \mathtt{catpB}(b, p')$, then $\delta_p\ ((z{:}s \| t)\ ;\ (u{:}r)) = \left(u : \delta_{p'}\ (r)\right)$. The stream has advanced past the segment which contained the connective being eliminated, so the `let`-binding itself no longer makes sense. In the case of the left rule for parallel, this happens when the prefix $p$ includes a batch of $z : s \| t$, as shown in the example above. In the case of the left rule for concatenation, it happens when $p$ includes either a batch of $z : s \cdot t$ or a prefix $\mathtt{catpB}(b, p)$ that includes a batch $b$ of $s$; in this case the derivative $\delta_p\ (\Gamma(z : s \cdot t))$ looks like $\Gamma'(z : \delta_p\ (t))$ for some $\Gamma'(-)$; the connective has again been deleted by the derivative, and the `let`-binding is no longer required.

### 3.4 The Homomorphism Property and Determinism

Our incremental semantics is designed to run a stream transformer on an "input chunk" of any size. In practical terms, whenever one or more input events are ready, a transformer can take a step that consumes these events and produces the appropriate output. Later, when more of the input arrives, it can take another step to produce more output, and so on. But this flexibility has a cost, as it raises the question of *coherence*—whether we are guaranteed to arrive at the same final output depending on how we carve up a transformer's input into a series of prefixes. Fortunately, this is indeed guaranteed.

Coherence follows from our main technical result, a *homomorphism theorem* that says running a term $e$ on a prefix $p_1$ and then running the resulting term $e'$ on a prefix $p_1'$ of the remaining stream produces the same end result as running $e$ on the combined prefix.

Theorem 3.3 (Homomorphism Theorem). *Suppose*

(1) $\Gamma \vdash e : s$
(2) $p_1 : \mathtt{prefix}\,(\Gamma)$

(3) $p_1' : \mathtt{prefix}\,(\delta_{p_1}\,(\Gamma))$
(4) $p_2 : \mathtt{prefix}\,(s)$
(5) $p_2' : \mathtt{prefix}\,(\delta_{p_2}\,(s))$
(6) $p_1 \Rightarrow e \downarrow e' \Rightarrow p_2$
(7) $p_1' \Rightarrow e' \downarrow e'' \Rightarrow p_2'$.
*Then* $p_1 \cdot_\Gamma p_1' \Rightarrow e \downarrow e'' \Rightarrow p_2 \cdot_s p_2'$.

The operation $p \cdot_s p'$ here is type-indexed *prefix concatenation*, which takes a prefix $p$ of type $s$ and a prefix $p'$ of type $\delta_p\,(s)$ and produces the prefix of type $s$ that is first $p$, and then $p'$. Formally, we define this as a 4-place partial inductive relation $p \cdot_s p' \sim p''$, which is defined when $p$ and $p'$ have types $s$ and $\delta_p\,(s)$, respectively. The operation $p \cdot_\Gamma p'$ does the same for prefixes of contexts. See Appendix B.4 for details.

The homomorphism theorem not only justifies the prefixes-at-a-time perspective; it also implies deterministic processing of parallel streams. Intuitively, determinism states that the results of a stream transformer do not depend on the particular order in which parallel data arrives. We formalize this through the following scenario. Suppose $\Gamma, \Gamma' \vdash e : s$ is a term with two parallel contexts serving as its input, and suppose that $p_1$ is a prefix of type $\Gamma$ and $p_2$ a prefix of type $\Gamma'$. There are two different ways of running $e$ on this data. One is to first run $e$ on $p_1$ (passing the empty prefix for the $\Gamma'$ component) and then run the resulting term on $p_2$ (with an empty prefix for $\Gamma$). The other does the opposite, first running $e$ on $p_2$ and then running the resulting term on $p_1$. Determinism says that these strategies produce equal results. It is proved by observing that the homomorphism theorem guarantees that each of these options is equivalent to running $e$ on $\mathtt{parp}(p_1, p_2)$.

Theorem 3.4 (Determinism). *Suppose*
(1) $\Gamma, \Gamma' \vdash e : s$
(2) $\mathtt{parp}(p_1, p_2) : \mathtt{prefix}\,(\Gamma, \Gamma')$
(3) $\mathtt{parp}(p_1, \mathtt{emp}_{\Gamma'}) \Rightarrow e \downarrow e_1 \Rightarrow p_1'$ *and* $\mathtt{parp}(\mathtt{emp}_\Gamma, p_2) \Rightarrow e_1 \downarrow e_2 \Rightarrow p_2'$
(4) $\mathtt{parp}(\mathtt{emp}_\Gamma, p_2) \Rightarrow e \downarrow e_1' \Rightarrow p_1''$ *and* $\mathtt{parp}(p_1, \mathtt{emp}_{\Gamma'}) \Rightarrow e_1' \downarrow e_2' \Rightarrow p_2''$.
*Then* $e_2 = e_2'$ *and* $p_1' \cdot_s p_2' = p_1'' \cdot_s p_2''$.

Two crucial observations make the proof work. First, prefixes are morally equivalence classes of sequences of stream elements, up to the possible reorderings defined by their type [57]. When presented syntactically, these equivalence classes serve as *normal forms* for all the possible interleavings of the same data. The homomorphism theorem then guarantees that these normal forms are processed compositionally, and so are independent of the actual temporal ordering of parallel data—it suffices to compute on aggregate normal forms.

## 4 FULL $\lambda^{\mathbf{ST}}$

We now introduce—more briefly—the remaining types and terms of $\lambda^{\mathrm{ST}}$ that are not part of Kernel $\lambda^{\mathrm{ST}}$. Along the way, we explore both type-theoretic and logical consequences of the concepts in $\lambda^{\mathrm{ST}}$, as well as features that are more practically motivated.

### 4.1 Cut

A logically inclined reader might have wondered about the absence of a *cut rule* in Kernel $\lambda^{\mathrm{ST}}$. In sequent calculus-style presentations of logics, a cut rule supports proofs with "lemmas": we can prove a proposition and then use it as a premise to help prove another.[3] Transported along the

[3]Traditionally in sequent calculi, the cut rule is introduced only to be immediately shown to be admissible. We have reason to believe that the cut rule in $\lambda^{\mathrm{ST}}$ will indeed be admissible—Frumin [32] has proven that BI plus an arbitrary set of structural

proofs-as-programs principle, cut corresponds to *sequencing*: running one transformer and then passing its output as an argument to another. Given a transformer term $e_1$ whose output type is used in the input of another term $e_2$, we can cut them together to form a single term $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ that operates as the sequential composition of $e_1$ followed by $e_2$.

One way to think about the term $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ is that it "internalizes" the sequential composition of stream transformers, in the following sense. In an eventual high-level stream-processing language based on $\lambda^{\text{ST}}$, a programmer might write one transformer to describe the behavior of each node in a distributed stream processing system and wire these together into a dataflow graph, connecting the output of one transformer to the inputs of another by way of links that might traverse the network. This is an *external* form of composition, occurring outside the bounds of the calculus. The term $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ *internalizes* this composition by forming a single core term that behaves like the external sequential composition of $e_1$ and $e_2$. This step from external to internal composition of transformers corresponds to *operator fusion*, a key optimization technique in the stream processing literature.

The typing rule for cut is shown to the right. If $e_1$ has type $s$ in context $\Delta$, and $e_2$ has type $t$ in a context $\Gamma(x : s)$ with a variable of type $s$, we can form the cut term $\texttt{let}_{\Gamma(-)}\ x = e_1\ \texttt{in}\ e_2$, which has type $t$ in context $\Gamma(\Delta)$. (The one-hole context $\Gamma(-)$ is included as an annotation on the term for the same reason that it is included in the terms for concatenation and parallel let-bindings: it is needed at runtime.)

$$\frac{\Delta \vdash e_1 : s \qquad \Gamma(x : s) \vdash e_2 : t}{\Gamma(\Delta) \vdash \texttt{let}_{\Gamma(-)}\ x = e_1\ \texttt{in}\ e_2 : t}\ \textsc{Cut}$$

Running a cut incrementally intuitively takes the following steps: (1) accept an input prefix $p$ and project out the component $p_\Delta$ corresponding to the subcontext $\Delta$, (2) run $p_\Delta$ through $e_1$ to get a prefix $p_s$, (3) bind $p_s$ to $x$ by substituting it back into $p$ where $p_\Delta$ was to get a prefix of type $\Gamma(x : s)$, and (4) run that through $e_2$, producing the final output. (Full details can be found in Appendix B.10.) An important point to note is that this semantics is non-blocking: even if no data for $\Delta$ has arrived, we still run $e_2$, potentially producing output.

### 4.2 Sums

The first type constructor that appears in $\lambda^{\text{ST}}$ but not in Kernel $\lambda^{\text{ST}}$ is the sum type, written $s + t$. Sums in $\lambda^{\text{ST}}$ are *tagged* unions: a stream of type $s + t$ is either a stream of type $s$ or a stream of type $t$, and a consumer can distinguish between the two. Streams of type $s$ are not the same as streams of type $s + s$, and streams of type $s + t$ are equivalent to but not the same as streams of type $t + s$. Operationally, a producer of a sum stream sends a tag bit prior to sending the rest of the stream, to say which side it is choosing. Conversely, a consumer of $s + t$ first reads the bit to learn if it should expect a stream of type $s$ or one of type $t$ and then gets a stream of that type.

Batches of $s + t$ are either a tagged batch of $s$, written $\texttt{sumbA}\,(b)$, or a tagged batch of $t$, written $\texttt{sumbB}\,(b)$. A prefix of $s + t$ can be a prefix of one of $s$ or one of $t$, written $\texttt{sumpA}(p)$ or $\texttt{sumpB}(p)$. A third possible prefix is $\texttt{sumpEmp}$, the empty prefix of type $s + t$, which does not even say which way the rest of the stream is going to go. The derivatives with respect to these prefixes are defined by (a) the empty prefix taking nothing off the type ($\delta_{\texttt{sumpEmp}}\ (s + t) = s + t$) and (b) the two injections reducing to taking the derivative of the corresponding branch of the sum ($\delta_{\texttt{sumpA}(p)}\ (s + t) = \delta_p\ (s)$ and $\delta_{\texttt{sumpB}(p)}\ (s + t) = \delta_p\ (t)$).

The typing rules for sums are the normal injections on the right (Sum-R-1 and a symmetric rule Sum-R-2) and a case analysis principle on the left (Sum-L). The right rules operate by prepending

rules admits cut—but we have not proven it ourselves for the specific set of rules and constructs in $\lambda^{\text{ST}}$. Because the point of cut elimination is to enable effective proof search, whereas we are most interested in the calculus from a programming perspective, we will not dwell on this point here.

their respective tag and then running the embedded term. The left rule performs case analysis: if the incoming stream is a left, then it is processed with $e_1$; if a right, then $e_2$.

$$\frac{\Gamma \vdash e : s}{\Gamma \vdash \mathtt{inl}\,(e) : s + t}\ \textsc{Sum-R-1} \qquad \frac{\Gamma(x : s) \vdash e_1 : r \qquad \Gamma(y : t) \vdash e_2 : r}{\Gamma(z : s + t) \vdash \mathtt{case}_{\Gamma(-),s,t,r}(z, x.e_1, y.e_2) : r}\ \textsc{Sum-L}$$

The batch semantics for the Plus-L and Plus-R rules are straightforward, but the incremental semantics is a bit more subtle. When a prefix arrives at a sum case term, the incremental semantics must dispatch on the tag that says if the stream is a left or a right. But that prefix might not include a tag (if it is `sumpEmp`) or it might not include any value for the variable at all (as in the rule P-Var-1 discussed earlier); in these cases, we have no way of determining which branch to run. The solution to this conundrum is to run neither! Instead, we hold on to the prefix, adding incoming data to the buffer until the tag arrives. Once we get a prefix that includes the tag, we continue by running the corresponding branch. Note that this buffering is necessarily a blocking operation.[4] Even if data arrives on part of the input that is in parallel with the variable being scrutinized, the case analysis must block until the tag arrives. This requires a slightly generalized typing rule that includes a *buffer* $p : \mathtt{prefix}\,(\Gamma(z : s + t))$ of the context type in the term. As prefixes arrive, we append to this buffer until we get the tag. When the buffer is empty ($p = \mathtt{emp}_{\Gamma(z:s+t)}$, e.g., before the transformer starts running), the generalized rule simplifies to the Sum-L rule in shown above. Full details on the generalized rule and the incremental semantics for case analysis can be found in Appendix B.10.

### 4.3 Star

Full $\lambda^{\mathrm{ST}}$ also includes a type constructor for unbounded streams,[5] written $s^\star$; it describes a stream that contains zero or more sub-streams of type $s$, in sequence. This formulation is inspired by the Kleene star from the theory of regular languages: zero or more concatenations of a stream type $s$ with itself. In regular languages, $r^\star$ is equal to $\varepsilon + r \cdot r^\star$. Interpreted in the language of stream types, this equation says that streams of type $s^\star$ are either empty ($\varepsilon$) or a stream of type $s$, followed by another stream of type $s^\star$—i.e., $s^\star$ can be understood as the least fixpoint of the stream type operator $x \mapsto \varepsilon + s \cdot x$. It is a list, whose elements are separated in time. The definitions of batches, prefixes, and typing rules for star all follow from this perspective.

For example, since a batch of $s + t$ is either a batch of $s$ or a batch of $t$, a batch of $\varepsilon + s \cdot s^\star$ is either a batch of $\varepsilon$ (the unit value), or a batch of $s \cdot s^\star$, which is a pair of a batch of $s$ and another batch of $s^\star$. This gives us the possible batches of type $s^\star$: the empty batch `[]` and the "cons batch" $b :: b'$.

Similarly, we cook our definition of the prefixes of type $s^\star$ so that $\mathtt{prefix}(s^\star) = \mathtt{prefix}(\varepsilon + s \cdot s^\star)$. The empty prefix of type $s^\star$, written `stpEmp`, is effectively the empty prefix of the sum that makes up $s^\star$. The second form of prefix—the "done" prefix of type $s^\star$—is written `stpDone`. It corresponds to the left injection of the sum, and receiving it means that the stream has ended. Note that, despite containing no data, this prefix is not empty: it conveys the information that the stream is complete. The final two cases correspond to the right injection of the sum, i.e., a prefix of type $s \cdot s^\star$. This is either $\mathtt{stpA}(p)$, with $p$ a prefix of $s$, or $\mathtt{stpB}(b, p)$, with $b$ a batch of $s$ and $p$ another prefix of $s^\star$.

For derivatives, the empty prefix leaves the type as is ($\delta_{\mathtt{stpEmp}}\,(s^\star) = s^\star$). Because no data will arrive after the done prefix, the derivative of $s^\star$ with respect to `stpDone` is $\varepsilon$, the type of the empty stream. In the case for $\mathtt{stpA}(p)$, after some of an $s$ has been received, the remainder of $s^\star$ looks like the remainder of the first $s$ followed by some more $s^\star$, so the derivative is defined as $\delta_{\mathtt{stpA}(p)}\,(s^\star) = (\delta_p\,(s)) \cdot s^\star$. Finally, $\delta_{\mathtt{stpB}(b,p)}\,(s^\star) = \delta_p\,(s^\star)$.

[4] Depending on the rest of the context, it could also require unbounded memory! In Section 7, we discuss how the type system might allow us to statically detect such space leaks.

[5] We do not need to distinguish between unbounded finite streams and "truly infinite" ones, because our incremental operational semantics cannot distinguish between a stream that goes on forever and a finite but as yet incomplete stream.

The typing rules for star are again motivated by the analogy with lists. There are right rules for `nil` and `cons`, and a case analysis principle for the left rule. The "nil" rule Star-R-1 corresponds to the left injection of the sum $s^\star = \varepsilon + s \cdot s^\star$. From any context, we can produce $s^\star$ by simply ending the stream. The "cons" rule Star-R-2 is the right injection. From a context $\Gamma;\Delta$, we can produce an $s^\star$ by producing one $s$ from $\Gamma$ and the remaining $s^\star$ from $\Delta$. Operationally, this should run the same way as the Cat-R rule: by first running $e_1$ using the $\Gamma$ part of the context to produce the first $s$, and then switching over to running $e_2$ to produce the tail $s^\star$ once we start to receive $\Delta$-data. The Star-L rule is a case analysis principle for star streams: either a stream is empty, or it includes one $s$ followed by an $s^\star$. The fact that the head $s$ will come first and the tail $s^\star$ later tells us that the variables $x : s$ and $xs : s^\star$ should be separated by a semicolon.

$$\Gamma \vdash \mathtt{nil} : s^\star$$

$$\frac{\Gamma \vdash e_1 : s \qquad \Delta \vdash e_2 : s^\star}{\Gamma;\Delta \vdash e_1 :: e_2 : s^\star}\ \textsc{Star-R-2}$$

$$\frac{\Gamma(\cdot) \vdash e_1 : r \qquad \Gamma(x : s; xs : s^\star) \vdash e_2 : r}{\Gamma(z : s^\star) \vdash \mathtt{case}_{\Gamma(-),s,r}\ (p; z, e_1, x.xs.e_2) : r}\ \textsc{Star-L}$$

The incremental and batch semantics for the right rules are straightforward: the rules for Star-R-1 are like those for Eps-R, while the rules for Star-R-2 are like those for Cat-R. But like Plus-L rule, the Star-L rule needs to wait until it knows if the stream will be empty or include at least one $s$ before it can pick a continuation to run. For this reason, Star-L must also be generalized to include a prefix buffer, and its semantics are also blocking in the same way as Plus-L. For full details on the batch and incremental semantics for star, see Appendix B.10.

### 4.4 Recursion

At this point, although the $\lambda^{\mathrm{ST}}$ type system now includes the type $s^\star$, we cannot write very interesting transformers over $s^\star$ streams: we need a way to define transformers recursively. Because $\lambda^{\mathrm{ST}}$ has no function types, we accomplish this by adding explicit term-level recursion and recursive call operators.

The term $\mathtt{fix}(e)$ denotes a recursive transformer with body $e$. Recursive calls are made with a constant term `rec`, which refers back to $e$. This back-reference works in the same way that uses of the variable $x$ in the body of a traditional fix point $\mathtt{fix}(x.e)$ refer to the term $\mathtt{fix}(x.e)$ itself. This function-free approach is approach is inspired by the concept of *cyclic proofs* [14, 20, 24] from proof theory, where derivations may refer back to themselves. Alternatively, one can think of this construction as defining our terms and proof trees as infinite *coinductive* trees; then the term-level `fix` operator defines terms as *co*fixpoints. Section 5 shows how these recursion operators can be used to define recursive transformers that process star streams like `map` and `filter`.

To type these new terms, we extend the Kernel $\lambda^{\mathrm{ST}}$ typing judgment to include a *recursion signature* $\Sigma ::= \mathtt{NoRec} \mid \Gamma \to s$, written on the turnstile: $\Gamma \vdash_\Sigma e : s$. The point of this addition is to ensure that recursive calls are well typed: if a recursive transformer $\mathtt{fix}(e)$ is typed in context $\Gamma$ with type $s$, then all instances of `rec` in $e$ must also occur in a position where a program from $\Gamma$ to $s$ is expected. A recursion signature of the form `NoRec` says that the term being typed is not in the body of a recursive function, while a recursion signature of the form $\Gamma \to s$ means that we are under a `fix`, where the recursive transformer being defined takes $\Gamma$ to $s$. The Rec rule shown on the right says that, if we are in the body of a recursive program from $\Gamma$ to $s$, we can make a recursive call. The Fix rule defines a recursive transformer from $\Gamma$ to $s$ by typing the body $e$ with a recursion signature $\Gamma \to s$. (Note that Fix clobbers any existing recursion signature, so using the Rec rule inside a multiply-nested recursive program refers to the innermost one. This means that $\lambda^{\mathrm{ST}}$ does not support mutually recursive definitions. (We have yet to run up against

$$\frac{}{\Gamma \vdash_{\Gamma \to s} \mathtt{rec} : s}\ \textsc{Rec}$$

$$\frac{\Gamma \vdash_{\Gamma \to s} e : s}{\Gamma \vdash_\Sigma \mathtt{fix}(e) : s}\ \textsc{Fix}$$

this restriction in practice, but generalizing to allow nested recursions is straightforward, though notationally heavy.)

In both the incremental and batch semantics, we only run transformers that are well typed with a closed (NoRec) recursion signature. To run $\texttt{fix}(e)$ in either semantics, we simply unfold the recursion one step, substituting $\texttt{fix}(e)$ itself for instances of $\texttt{use}$ in the body $e$. Naturally, this can lead to non-termination—for example, there are no finite derivations of $p \Rightarrow \texttt{fix(rec)} \downarrow e \Rightarrow p'$ for any $p, e, p'$, since $\texttt{fix(rec)}$ unfolds to itself.[6] The bound the depth of evaluation, we *step index* both semantic judgments by adding a fuel parameter that decreases when we unfold a use of Fix. The incremental semantic judgment then looks like $p \Rightarrow e \downarrow^n e' \Rightarrow p'$ and means that when we run $e$ on $p$, it steps to $e'$ producing $p'$ and unfolding at most $n$ uses of $\texttt{fix}$ along the way. The incremental semantics rule for Fix is presented in Figure 7; all the other rules in the incremental and batch semantics for $\lambda^{\text{ST}}$ need to be updated to propagate the step index compositionally. The only place the step index decreases is in P-Fix, and all base cases run in any amount of gas to ensure that the step index is monotone.

$$\frac{p \Rightarrow e\,[\texttt{fix}(e)/\texttt{use}] \downarrow^n e' \Rightarrow p'}{p \Rightarrow \texttt{fix}(e) \downarrow^{n+1} e' \Rightarrow p'}\ \text{P-Fix}$$

Fig. 7. Fix Incremental Semantics

The inclusion of a step index now means that there are well-typed terms about which the $\lambda^{\text{ST}}$ semantics say nothing at all. In particular, an "infinite generator" term $\cdot \vdash_{\text{NoRec}} \texttt{rec}(() :: \texttt{use}) : 1^\star$, which runs forever and should produce an infinite sequence of unit values, has no meaning in $\lambda^{\text{ST}}$. Semanticists may find this behavior odd, but it mimics the incremental semantics of present-day stream processing systems, which wait for a step of computation to terminate before sending out any of its results.

The inclusion of recursion also requires an update to the correctness theorem: if a well typed term takes a step on a well-typed input *using some amount of gas*, then the output and resulting term are also well typed.

Theorem 4.1 (Incremental Semantics Correctness). *If* $\cdot \mid \Gamma \vdash_\emptyset e : s$, *and* $p : \texttt{prefix}\,(\Gamma)$, *and* $p \Rightarrow e \downarrow^n e' \Rightarrow p'$, *then* $p' : \texttt{prefix}\,(s)$ *and* $\cdot \mid \delta_p\,(\Gamma) \vdash_\emptyset e' : \delta_{p'}\,(s)$

A similarly updated theorem statement for the homomorphism theorem—and many other theorems and lemmas used in the proofs of the correctness and homomorphism theorems—can be found in Appendix B.10.

### 4.5 Stateful Transformers

In the $\lambda^{\text{ST}}$ typing judgment $\Gamma \vdash e : s$, the variables in $\Gamma$ range over *future values* that have yet to arrive at the transformer $e$. The ordered nature of semicolon contexts means that variables further to the right in $\Gamma$ correspond to data that will arrive further in the future. This imposes a strong restriction on programming: if earlier values in the stream are to be used at all, they must be used *before* later values, and once a value in the stream has "gone past" there is no way to refer to it again.

This limitation can be seen as arising from an asymmetry in the design of Kernel $\lambda^{\text{ST}}$. By using variables from the $\Gamma$ context, a term $e$ can refer to values that will arrive in the future; but it has no way of referring to values that have arrived in the *past*.

From a programming perspective, referring to variables from the past requires *state*. Many important streaming functions (e.g., map and filter) are stateless, but many others (e.g., "running

[6] Viewed logically, this means that $\lambda^{\text{ST}}$ is unsound as a proof system: $\texttt{rec(use)}$ is a proof of any sequent. Cyclic proof systems usually ensure soundness by imposing a guardedness condition [14] which requires certain rules be applied before a back-edge can be inserted in the derivation tree. Because we are not primarily concerned with $\lambda^{\text{ST}}$ as a logic at the moment, we leave a guardedness condition to future work.

sums") do require state. Moreover, state isn't *just* about referring to values that have arrived previously: many stateful transformers must maintain accumulators and other scratch values that were never actually received on an input stream.

What types do streams from the past have? We saw in Section 3 that values of completed streams are batches. And the batches of type $s \| t$ and of type $s \cdot t$ are essentially the same, since the partial order structure of the data no longer matters. For example, once a complete stream of type $(\texttt{Int}^\star \| \texttt{Int}^\star) \cdot \texttt{Int}^\star$ has been received, we can regard the data that has been received as a value of the conventional type $(\texttt{list(Int)} \times \texttt{list(Int)}) \times \texttt{list(Int)}$ from the simply typed lambda-calculus (STLC). This suggests that, while parts of streams that will arrive in the future have stream types, parts of streams that have arrived in the past should have standard STLC types.

Accordingly, we define an operation called flatten, written $\langle s \rangle$, that transforms stream types into STLC types. The important cases are $\langle s \cdot t \rangle = \langle s \| t \rangle = \langle s \rangle \times \langle t \rangle$, and $\langle s^\star \rangle = \texttt{list}\,(\langle s \rangle)$.

We then extend the typing judgment of $\lambda^{\text{ST}}$ to include a second context, $\Omega$, called the *historical context*, writing $\Omega \mid \Gamma \vdash_\Sigma e : s$. The historical context is fully structural: $\Omega ::= \cdot \mid \Omega, x : A$, where the types $A$ are drawn from some set of STLC types including at least products, sums, a unit, and a list type. Operationally, the historical context behaves like a standard context in a functional programming language: at the top level, terms to be run with the batch or incremental semantics must be typed in an empty historical context; at runtime, historical variables get their values by way of substitution.

Rather than giving a set of ad-hoc rules for manipulating values from the historical context, we parameterize the $\lambda^{\text{ST}}$ calculus over an arbitrary language with terms $M$, typing judgment $\Omega \vdash M : A$, and terminating big-step semantics $M \downarrow v$. We call any such fixed choice of language the *history language*. Programs from the history language can be embedded in $\lambda^{\text{ST}}$ programs using the HistPgm rule, which says that a historical program $M : \langle s \rangle$ that accesses the historical context $\Omega$ can be used in place of a $\lambda^{\text{ST}}$ term of type $s$. Operationally, as soon as any prefix of the input arrives, we simply run the embedded historical program to completion and yield the result as its stream output (after converting it into a batch of type $s$).

$$\frac{\Omega \vdash M : \langle s \rangle}{\Omega \mid \Gamma \vdash_\Sigma \langle M : s \rangle : s}\ \textsc{HistPgm}$$

How does information get added to the historical context? Intuitively, a variable in $\Gamma$ (a stream that will arrive in the future) can be moved to $\Omega$, where streams that have arrived in the past are saved, by waiting long enough for the future to become the past! Formally, we define an operation called "wait," which allows us to specify part of the incoming context and *block* this subcomputation until that part of the input stream has arrived in full. Once it has, we can bind it to the variables in the historical context and continue by running $e$.

The Wait rule encodes the typing content of this behavior. It allows us to specify a subcontext $\Delta$ of the input, and then flatten and move it to the historical context so that the continuation $e$ can refer to its variables as historical variables. Semantically, this works by *waiting* for $\Delta$: buffering in prefixes of the context until a whole batch of $\Delta$ has arrived. Once we have a batch, we substitute it into $e$ and continue running the resulting term.[7] This buffering is implemented the same way as in the left rules for plus and star, by generalizing the typing rule to include an explicit prefix buffer. The generalized typing rule for Wait as well as the incremental and batch semantics for both constructs can be found in Appendix B.8 and Appendix B.10.

$$\frac{\Omega, \langle \Delta \rangle \mid \Gamma(\cdot) \vdash_\Sigma e : s}{\Omega \mid \Gamma(\Delta) \vdash_\Sigma \texttt{wait}_{\Gamma(-),\Delta,s}\,(e) : s}\ \textsc{Wait}$$

[7] The semantics of the Wait rule is reminiscent of the "blocking reads" of Kahn Process Networks, where every read from a parallel stream blocks all other reads to ensure determinism. Here, we choose a subcontext and block the rest of the program until it is complete and in memory. For a subcontext $\Delta$ that is a single variable, this is essentially the same.

Most of the remaining typing rules in $\lambda^{\text{ST}}$ change only by adding an $\Omega$ to the typing judgment everywhere; the historical context just goes along for the ride. The only meaningful change is in the Fix and Rec rules. First, recursion signatures change to include a historical context: $\Sigma ::= \texttt{NoRec} \mid (\Omega \mid \Gamma \to s)$. Then, when defining a recursive transformer, the programmer must set up the $\Omega$ context with any accumulators or auxiliary data that the program needs and provide initial values for each. Then, when making recursive calls with Rec, the programmer decides how those values those values will be updated before continuing with the next iteration of the program.

The syntax for recursive calls thus becomes $\texttt{rec} \mathbin{@} [M_1, \ldots, M_n]$, where the $M_i$ are historical programs computing updated values of the accumulators (i.e., all of the variables in the historical context in the recursion signature). Similarly, the recursive definition syntax gets extended to $\texttt{fix}\,(x_1 : A_1, \ldots, x_n : A_n) \mathbin{@} [M_1, \ldots, M_n]\,.e$, where the $x_i : A_i$ define the shape of the historical context in the body of $e$ and the $M_i$ are its initial values. See Appendix B.8 for details.

## 5 EXAMPLES

In this section, we show how $\lambda^{\text{ST}}$ addresses the problems of type-safe programming with temporal patterns and deterministic processing of parallel data from Section 2 and how some other characteristic streaming idioms are expressed in $\lambda^{\text{ST}}$.

The process of encoding any of these common streaming idioms in $\lambda^{\text{ST}}$ begins by determining the input and output types. Because these idioms are from the homogeneously typed world, we must work to "extract the stream types" from the situation. Then, we build a term of that type. In many cases, we will see that the high-level style of $\lambda^{\text{ST}}$ means that the streaming programs we derive strongly resemble well-known standard functional programming idioms. But while the programs look like high-level functions computing over entire streams at once, the incremental semantics allows them to be run event by event. We've tested all of the incremental semantics on all of the examples in this section with a prototype interpreter, written in Haskell: see Appendix A and in the supplementary materials for details about our implementation.

Many of the idioms we'll discuss are conceptually higher-order functions, like map, filter, and fold. But $\lambda^{\text{ST}}$ is a first-order language: the calculus has no function types. We treat higher-order functions as macros: transformers like map, filter, and fold are all parameterized by a term which represents the function argument. In Section 8, we'll discuss a high level language design which could support this.

We present all examples in this section with a syntax which is a sugared version of our core terms. First, type and context annotations on variables, left rules, cut and wait are omitted, and structural rules are written silently (without their explicit terms $\texttt{subctx}\,(pf, e)$). We also use a syntax `wait x then e end` for the $\texttt{wait}_\Delta(e)$ operator when $\Delta$ is a single variable `x` and `e` the continuation. Embedded historical programs are written surrounded with angle brackets: $\langle \texttt{M} \rangle$. The syntax for defining recursive programs is `fix with (y1 = M1, ... ,yn = Mn). e`. This defines a recursive transformer with accumulator variables `y1,...,yn` which take initial values `M1,...Mn`. For recursive calls, we use an argument-passing syntax `rec(xs)` to make clear which subcontext a recursive call is being performed on, and write square brackets `rec(x1,...xM)[acc1,...,accN]` to update the values of any accumulators in the historical context. Last, we sometimes use a natural deduction-style elimination form (like `let (x,y) = e1 in e2`) instead of a left rule: this can be de-sugared into a cut followed by a use of the corresponding left rule.

We present the full core terms for all these examples in Appendix C. In Section 8, we'll discuss potential methods for elaborating a higher-level syntax like this one down into $\lambda^{\text{ST}}$.

*Map.* Given a transformer from `s` to `t`, we can lift it to a transformer from $\texttt{s}^\star$ to $\texttt{t}^\star$. Note that this type is more general than the standard `map` function on homogeneous streams, which has type

(a → b) → (Stream a → Stream b): our types $s$ and $t$ can be arbitrary stream types: they need not be singletons.

The map transformer is parameterized by a transformer e from $s$ to $t$. The code for map is essentially identical to the traditional functional code.

```
map (e : s → t) (xs : s*) : t*=
  fix. case xs of
       | nil => nil
       | y :: ys => e(y) :: rec(ys)
```

*Filter.* Similarly, given a "predicate" transformer e from $s$ to $1 + 1$ (i.e., Booleans), we can filter an incoming stream of type $s^\star$ to include the elements of type $s$ (and which may consist of many events) for which e outputs inl().

The code is identical to the standard functional filter, yet it runs incrementally: as soon as e sees that y passes the filter, the function will start to forward it along. See Appendix C.3 for details.

*Singletons, Head, Tail.* In the homogeneous model, stream types are always conceptually unbounded. But in practical situations, some streams are only expected to contain a single element—a constraint that cannot be expressed with homogeneous streams. Using stream types, we can write stream transformers which are statically known to only produce a single output. For example, the "head" function is trivially expressible in the same manner as head on lists. (Exercise: try writing the term for tail on star streams. This requires a use of wait and an accumulator argument like in fold, as shown below.)

```
head (xs : s*) : ε + s =
  fix. case xs of
       | nil => inl(sink)
       | y :: ys => inr(y)
```

*Fold.* In $\lambda^{\mathrm{ST}}$, we can can express a fold which returns only the final accumulator, as opposed to traditional streaming folds which are *running* folds, outputting a stream of all intermediate results (we can also do the running fold; see Appendix C.4). Fold maintains an accumulator of type $\langle t \rangle$ in the historical context, which gets updated by a streaming step function e : ⟨t⟩ | s → t that takes a stream argument of type $s$ and a historical accumulator argument of type $\langle t \rangle$, and produces a $t$. Then, the whole fold takes a stream $xs$ of type $s^\star$ and an initial accumulator init : $\langle t \rangle$ and produces a stream of accumulator values $t^\star$.

```
fold (e : ⟨t⟩ | s → t) (xs : s*) : t =
    fix with (y = m0).
      case xs of
      | nil => y
      | x' :: xs' =>
          let y' = e(x')[y] in
          wait y' then rec(xs')[y'] end
```

The code for fold cases on the incoming stream. If the stream is empty, we return the accumulator y. If there is a head, it uses the term $e$ to compute the next accumulator y' from the head x' and the previous accumulator value. Then, we pull y' into the historical context with a wait and recurse to handle the tail of the input stream with the updated accumulator y'.

*Brightness Levels.* The "input protocol" from the brightness-levels example in Section 2 can be encoded as the type $(\mathtt{Int} \cdot \mathtt{Int}^\star)^\star$: a stream of nonempty sequences of Ints, which represent the nonempty runs of light levels greater than some threshold. Writing a transformer that assumes the input invariant is easy. For example, to average the brightness-levels in each run, we can simply map an average operation—which takes $\mathtt{Int} \cdot \mathtt{Int}^\star$ to Int—over the incoming stream to produce a stream $\mathtt{Int}^\star$ of averages: · | xs : (int · int*)* ⊢ map(avg)(xs) : int*.

By contrast, with a homogeneous stream type like $(\mathtt{Start} + \mathtt{Int} + \mathtt{End})^\star$, this operation would need to be written in a low-level and stateful manner, remembering the current run of Ints until an End event arrives, averaging, and handling the divide-by-zero error which could in principle (although not in practice) occur if no Ints arrived between a Start and an End. In $\lambda^{\mathrm{ST}}$ the program is instead high-level and functional, describing what happens to each run of Ints above the threshold.

The actual per-run average operation can be defined by simply waiting on the whole run to arrive, then computing the average with an embedded historical program, as shown to the right.

```
wait x,xs then
  ⟨(x + sum xs) / (length xs + 1)⟩
end
```

The final thresholding operation—which takes $\texttt{Int}^\star$ and produces the runs of elements above the threshold $(\texttt{Int} \cdot \texttt{Int}^\star)^\star$—is somewhat complex because it requires statefully parsing the input stream, depending on if we're currently on a run of elements above the threshold or not. A certain amount of stateful logic for constructing the thresholded stream is unavoidable: similar code would be required in a language based on homogeneous streams. But in $\lambda^{\text{ST}}$, type safety guarantees that (1) the transformer *does in fact* output a stream which adheres to the protocol, and (2) the downstream transformer does not have to replicate this parsing logic. The code can be found in Appendix C.5.

*Side Outputs & Error Handling.* A common streaming idiom is the use of "side outputs" for reporting errors. Operations include extra output streams where error messages are sent as they arise at runtime. In most frameworks, side outputs are a second-class mechanism: the error streams cannot be transformed or used in a manner other than dumping them to a log somewhere. $\lambda^{\text{ST}}$ provides a first-class account of side outputs, by encoding them as a parallel output type. A function $s \to t$ that may produce errors of type $e$ can have type $s \to t \| e^\star$. Alternately, errors can be handled inline in the traditional typed functional manner, using a sum type $s + e$.

*Partitioning.* Partitioning is a crucial streaming idiom where a homogeneous stream of data is split into two or more parallel streams, which are then routed to different downstream nodes in the dataflow graph, thus exposing parallelism and increasing potential throughput. Appendix C.6 shows how two different partitioning strategies can be implemented in $\lambda^{\text{ST}}$: a *round-robin partitioner*, which fairly distributes an incoming stream of type $s^\star$ into a parallel pair of streams $s^\star \| s^\star$ by by sending the first element to the left branch, the second to the right, and so on, and a *hash-based partitioner*, which routes stream elements based on a hash of their data modulo the number of downstream processing nodes.

*Windowing and Punctuation. Windowing* is a core concept in stream processing systems, where common aggregation operations like moving averages or sums are defined over "windows"—groupings of consecutive events, gathered together into a set. In $\lambda^{\text{ST}}$, these transformers are just `map`s over a stream whose elements are windows. Given a per-window aggregation transformer `f` that takes an individual window $s^\star$ to a result type $t$ plus a "windowing strategy" `win` which takes a stream $r^\star$ and turns it into a stream of windows $s^{\star\star}$, we can write the windowed operation as `xs : r⋆ |- map(f)(win(xs)) : t⋆`. In Appendix C.7, we define both sliding and tumbling size-based window operators, as well as punctuation-based windowing, where windows are delimited by punctuation marks inserted into the stream.

## 6 COMPILING TO HOMOGENEOUS STREAMS

Our new logical foundation for stream processing represents a significant departure from the traditional homogeneous view. The values that $\lambda^{\text{ST}}$ programs operate over are highly structured syntactic objects, not the simple sequences that modern stream processing systems are designed to handle. Given this, one might wonder whether we have thrown the baby out with the streamwater (as it were): have we given up the distributed fault tolerance and message delivery guarantees that mature implementations have worked so hard to achieve? Fortunately, we have not. We prove in this section that stream types can be encoded using traditional homogeneous streams. This result implies that we can build a distributed stream processing application by writing a $\lambda^{\text{ST}}$ transformer to run at each node, equipping each with adapters that convert between the structured prefixes expected by the $\lambda^{\text{ST}}$ incremental semantics and the homogeneous streams of tagged events used by an industrial-strength stream processing substrate.

The key idea is that a structured $\lambda^{\mathrm{ST}}$ stream with an arbitrarily complex type can be represented as a homogeneous, totally ordered sequence over a fixed alphabet of *events*. Sequences of events can be translated into $\lambda^{\mathrm{ST}}$ prefixes and vice versa. Then, given a $\lambda^{\mathrm{ST}}$ term $e$ and a sequence of events $xs$, we can turn $xs$ into a prefix $p$, run $e$ with $p$, and turn the output back into an event sequence.

Translating from prefixes to event sequences is a kind of serialization: we go from the structured prefix representation to a wire format in the form of a sequence of tagged events. Formally, this translation is nondeterministic: a parallel pair prefix $\mathtt{parp}(p_1, p_2)$ can be serialized to any tagged interleaving of the serializations of $p_1$ and $p_2$. The other direction, going from event sequences to prefixes, consists of deserializing the unstructured event sequence representation into a structured one. Crucially, deserializing any event sequence that is a serializations of a given prefix will yield that prefix back.

The grammar of events is as follows:

$$x ::= \mathtt{oneev} \mid \mathtt{parevA}\,(x) \mid \mathtt{parevB}\,(x) \mid +_{\mathsf{puncA}} \mid +_{\mathsf{puncB}} \mid \cdot_{\mathsf{punc}} \mid \mathtt{catevA}\,(x)$$

Like prefixes, events have types. Intuitively, an event of type $s$ is an element that could arrive as the *first* element in a stream of type $s$. We write the typing relation as $x : \mathtt{event}\,(s)$ (the rules are given in Appendix B.11 and explained intuitively below). Then, the derivative $\delta_x\,(s) \sim s'$ means that $s'$ is the type of streams that follow event $x$ in a stream of type $s$. We lift the definition of events to sequences by saying that a sequence of events $xs$ has type $s$ if the first event $xs_0$ has type $s$, the second event $xs_1$ has type $\delta_{xs_0}\,(s)$, and so on.

The unique event of type 1 is $\mathtt{oneev}$, and there are no events of type $\varepsilon$. The events $\mathtt{parevA}\,(x)$ and $\mathtt{parevB}\,(x)$ have type $s \| t$ when $x$ is an event of type $s$ and $t$, respectively—i.e., an event of a parallel pair type is a tagged event of one of the two types. There are two events of type $s + t$, corresponding to the first bit that says the stream will be of type $s$ ($+_{\mathsf{puncA}}$) or of type $t$ ($+_{\mathsf{puncB}}$). These events are *punctuation* [60]: they carry no data and serve only to inform a transformer of the choice of left or right. The derivatives with respect to these events reduce the stream to the corresponding branch: $\delta_{+_{\mathsf{puncA}}}\,(s + t) = s$, and vice versa.

Finally, an event of type $s \cdot t$ can be one of two possibilities. The first is an event $\mathtt{catevA}\,(x)$, where $x$ is an event of type $s$. The second is $\cdot_{\mathsf{punc}}$, which is a punctuation mark which says that we're done with the $s$ part of $s \cdot t$, and are ready to start with the $t$ part. The derivative $\delta_{\cdot_{\mathsf{punc}}}\,(s \cdot t) = t$ moves the stream past the $s$ part.

Moreover, the set of events that will ever be needed to represent a stream of a given type is *finite*: for a fixed type $s$, the size of the possible events that could arrive on a stream of type $s$ is bounded.

Theorem 6.1 (Bounded Event Size). *For all $s$, there is some $N$ such that for any $xs : \mathtt{events}\,(s)$ and any $x \in xs$, we have that $|x| \leq N$, where $|\cdot|$ denotes the size of the AST.*

*Serialization and Deserialization.* Serializing a prefix $p$ of type $s$ into a sequence of events is written with a relation $p \dagger^s xs$, where $p$ and $s$ are inputs and $xs$ is an output. For a given $p$ and $s$, there may be many $xs$ for which this holds; in particular, all possible interleavings of any parallel data in $p$ give valid serializations. To deserialize, we use a relation going the other way, written $xs \hookrightarrow^s p$. This one is deterministic but not injective: each event sequence uniquely determines a prefix, but the same prefix may result from many sequences of events. Serialization and deserialization round-trip in the expected way: serializing a prefix to an event sequence and then deserializing it yields the same prefix that we started with.

*Homogeneous Stream Transformers.* Now we can finally describe how we compile a $\lambda^{\mathrm{ST}}$ term to a homogeneous stream transformer—that is, an element of the greatest set $H$ satisfying the equation

$H \cong \mathtt{Seq}(X) \to 1 + 2^{\mathtt{Seq}(X)} \times H$, where $X$ is the set of all events and $\mathtt{Seq}(A)$ is the set of finite sequences of elements of $A$. An element $h$ of $H$ is a nondeterministic *event handler*. Given an input sequence of events, $h$ may either halt (returning the left injection of the sum), or nondeterministically produce an output sequence of events and step to a new state $h' \in H$.

We compile terms to handlers as follows, defining a map $h : \mathtt{Ctx} \times \mathtt{Tm} \times \mathtt{Ty} \times \mathbb{N} \to H$ which takes a 4-tuple of a context, term, type, and amount of gas (for recursive calls) to a handler. Intuitively, $h(\Gamma, e, s, n)$ takes an event sequence $xs$, deserializes it to obtain a prefix $p$, and runs $e$ on $p$ (using the incremental prefix semantics) to obtain $e'$ and $p'$. It then returns nondeterministically one of the potential serializations of $p'$ and steps to the handler defined by $e'$, with the proper derivative type. If the handler happens to run out of gas because of too many nested recursive calls, it simply halts and produces no output.

*Definition 6.2 (Term to Handler Compilation).* If

(1) $xs \hookrightarrow^{\Gamma} p$
(2) $p \Rightarrow e \downarrow^{n} e' \Rightarrow p'$
(3) $Y = \{ys \mid p' \dagger^{s} ys\}$
(4) $h' = h(\delta_p(\Gamma), e', \delta_{p'}(s), n)$,

then $h(\Gamma, e, s, n)(xs) = \mathtt{inr}(Y, h')$; otherwise $h(\Gamma, e, s, n)(xs) = \mathtt{inl}()$.

## 7 RELATED WORK

Streams as a programming abstraction can be traced back decades, to early work in the programming languages [16, 42, 58, 59] and database [1, 2, 5–7, 19, 48] communities. While streams are commonly viewed as homogeneous sequences, more interesting treatments of streams and stream-like data have been proposed. Streams in the database literature are sometimes viewed as time-varying relations, while the PL community has produced formalisms like process calculi and functional reactive programming. To our knowledge, ours is first type system for streams capturing both (1) heterogeneous patterns of events over time and (2) combinations of parallel and sequential data.

*Sequential, homogeneous streams and dataflow programs.* Traditionally, streams have been viewed as coinductive sequences [16]: a stream of `A` has a single (co)constructor, $\mathtt{cocons} : \mathtt{Stream} A \to (A \times \mathtt{Stream}\ A)$ and acts as a lazily evaluated infinite list. This is the setting of traditional *dataflow programming* [58]. A major challenge in reasoning about dataflow over sequential streams is the nondeterminism arising from operators whose output may depend on the order in which events arrive on multiple input streams. Kahn's seminal process networks [42] (including their restriction to synchronous networks [12, 47, 59]) avoid this problem by allowing only *blocking reads* of messages passed between processes on FIFO queues. In contrast, the semantics of $\lambda^{\mathrm{ST}}$ leverages the type structure to guarantee deterministic parallel processing *without* blocking in many cases. For example, in the context of a `Cut` rule, if the type system can detect statically that a transformer is using two parallel streams safely, it can read from them simultaneously.

*Partitioned streams.* Building on streams as homogeneous sequences, modern stream processing systems such as Flink [17, 25], Spark Streaming [28, 63], Samza [27, 51], and Storm [29] support dynamic partitioning: a stream type defines one stream with many parallel substreams (where the number of substreams and assignment of data to substreams is determined at runtime). The type $\mathtt{Stream}\ \mathtt{t}$ in these systems is implicitly a parallel composition of homogeneous streams: $\mathtt{t}^{\star} \| \cdots \| \mathtt{t}^{\star}$. Unlike in $\lambda^{\mathrm{ST}}$, these parallel substreams cannot have more general types. Dynamic parallelism is difficult to support in the context of universal calculi for streaming; for example, Brooklet [56] and the DON Calculus [21] support data parallelism only as an optimization pass in limited cases. This is because stream partitioning does not in general preserve the semantics of

the source program and can introduce undesirable nondeterminism [34, 49, 55]. While $\lambda^{\mathrm{ST}}$ does not support dynamic partitioning, we hope to address it in future work; see Section 8.

*Streams as time-varying relations.* In the database literature, streams are often viewed as relations (sets of tuples) that vary over time. Stream management systems in the early 2000s pioneered this paradigm, including Aurora [2] and Borealis [1], TelegraphCQ [19] and CACQ [48], and STREAM [5]. A time-varying relation can be viewed as either a function from timestamps to finite relations or an infinite set of timestamped values; this correspondence was elegantly exploited by early streaming query languages such as CQL [6, 7] and remains popular today [11, 39]. Time-varying relations can be expressed in $\lambda^{\mathrm{ST}}$ using Kleene star and concatenation: a relation of tuples of type `T` timestamped by `Time` can be expressed as $(\mathtt{T}^\star \cdot \mathtt{Time})^\star$. We can also express the common pattern where parallel streams are synchronized by a single timestamp (again, modulo dynamic partitioning) with types like $((\mathtt{T}^\star \| \mathtt{T}^\star) \cdot \mathtt{Time})^\star$. Each `Time` event is a punctuation mark containing the timestamp of the prior set of tuples [41, 61]. Traditional systems include separate APIs for operations that modify punctuation (e.g., a *delay* function that increments timestamps); whereas in our system they are ordinary stream operators and punctuation markers are ordinary events.

*Streams as pomsets.* A sweet spot between the homogeneous sequential and relational viewpoints is identified by prior work treating streams as *pomsets* (partially ordered multisets) [3, 43–45, 49], inspired by prior work in concurrency theory [22, 50]. In a pomset, data items may be completely ordered (a sequence), completely unordered (a bag), or somewhere in between. While existing works have proposed pomset-based types for streams[3, 49], their types do not support concatenation and do not come with type *systems*—programs must be shown to be well typed semantically, rather than statically.

*Functional reactive programming (FRP)* [23] treats programs as incremental, reactive state machines written using functional combinators. The fundamental abstraction is a "signal": a time-varying value `Sig(A) = Time -> A`. Work on type systems for FRP has used modal and substructural types [8, 9, 18, 46] to guarantee properties like causality, productivity, and space leak freedom. While our type system is not *designed* to address these issues, it does incidentally have bearing on them. For one, the existence of our incremental semantics demonstrates that our type system enforces causality: since outputs that have been incrementally emitted cannot be retracted or changed, the type system must ensure that past outputs cannot depend on future inputs. Similarly, potential space leaks can be detected statically by checking if only bounded-sized types are buffered using `wait` or the buffering built into the left rules for sums and star. Our current calculus does not guarantee productivity (new inputs eventually produce new outputs), but in Section 8 we discuss how to remedy this by imposing guardedness conditions on recursive calls.

Work by Jeffrey [40] permits the type of a signal to vary over time, using dependent types inspired by Linear Temporal Logic [53]. The system includes an *until* type that behaves like our concatenation type: a signal of type $A\,U\,B$ is a signal of type $A$, followed by a signal of type $B$. However, unlike parallel streams in our setting, time updates in steps, discretely; i.e., there's a synchronous ordering between elements of one signal and elements on another. Concurrently with our work, Bahr and Møgelberg [10] proposes a modal type system to weaken the synchronicity assumption; however, it still treats signals as homogeneous.

*Session types and process calculi.* Another large body of work with a vision similar to ours is session types for process calculi [36], where types describe complex sequential protocols between communicating processes as they evolve through time. A main difference from our work is that the session type of a process describes the *protocol* for its communications with other processes—i.e., the sequence of sends and receives on different channels—while the stream type of a $\lambda^{\mathrm{ST}}$ program

describes only the data that it communicates. Indeed, a stream transformer might display many patterns of communication with downstream transformers: it can run in "batch mode"—sending exactly one output after accepting all available input—or in a sequence smaller steps, sending along partial outputs as it receives partial inputs. Also, a single channel in a process calculus cannot carry parallel substreams of events that are internally ordered but not ordered relative to each other. Recently, Frumin et al. [33] proposed a session-types interpretation of BI that the bunched structure very differently from $\lambda^{\mathrm{ST}}$. In particular, processes of type $A * B$ and $A \wedge B$ both behave semantically like a process of type $A$ in parallel with a process of type $B$; in $\lambda^{\mathrm{ST}}$, $s \cdot t$ and $s \| t$ describe very different streams.

*Concurrent Kleene Algebras and regular expression types.* Finally, stream types are also inspired by Concurrent Kleene Algebras (CKAs) [35] and related syntaxes for pomset languages [45], though we are apparently the first to use these formalisms as *types* in a programming language, rather than as a tool for reasoning about concurrency. In particular, traditional applications of Kleene algebra such as NetKAT [4] and Concurrent NetKAT [62], use KA to model *programs*, whereas in $\lambda^{\mathrm{ST}}$ we use the KA structure to describe the data that programs exchange, while the programs themselves are written in a separate language. We have also been inspired by languages for programming with XML data [13, 31, 37, etc.] using types based on regular expressions.

## 8 CONCLUSIONS AND FUTURE WORK

We have proposed a new static type system for stream programming, strongly connected to a novel variant of BI logic and intended to capture both complex temporal patterns and deterministic parallel processing.

In Section 5, we presented a speculative higher-level syntax for $\lambda^{\mathrm{ST}}$ terms. Two significant challenges in building a real, statically typed, higher-level language on this foundation are (a) inferring type and context annotations on variables, cut, and left rules, and (b) inferring usages of structural rules. The first, we believe, is relatively easy: given a context and a variable, it is straightforward to find the one-hole context in which the variable resides, and thus the variable's type. A promising beginning for the second is a recent "labeled" formulation of the BI sequent calculus [38].

In Section 6, we showed how to run $\lambda^{\mathrm{ST}}$ terms as sequential automata, demonstrating the feasibility in principle of compiling $\lambda^{\mathrm{ST}}$ down to programs for an existing stream processing system like Apache Storm and thus inheriting its desirable fault-tolerance and delivery guarantees. We plan to build such a compiler and use it as a platform for experimenting with type-enabled optimizations and resource usage analysis.

Finally, we plan to extend the theory of stream types by (1) investigating a denotational semantics in pomsets, (2) adding support for *bags* (unbounded parallelism, the parallel analog of Kleene star) to enable dynamic partitioning, and (3) adding a *guardedness* condition on recursive calls to ensure termination and hence productivity.

## REFERENCES


[1] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Uğur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley Zdonik. 2005. The Design of the Borealis Stream Processing Engine. In *Second Biennial Conference on Innovative Data Systems Research (CIDR)*.

[2] Daniel J Abadi, Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. 2003. Aurora: A New Model and Architecture for Data Stream Management. *The VLDB Journal* 12, 2 (2003). https://doi.org/10.1007/s00778-003-0095-z

[3] Rajeev Alur, Phillip Hilliard, Zachary G Ives, Konstantinos Kallas, Konstantinos Mamouras, Filip Niksic, Caleb Stanford, Val Tannen, and Anton Xue. 2021. Synchronization Schemas. *Invited contribution, Principles of Database Systems.*

[4] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic foundations for networks. *Acm sigplan notices* 49, 1 (2014), 113–126.

[5] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. 2004. *STREAM: The Stanford Data Stream Management System.* Technical Report 2004-20. Stanford InfoLab.

[6] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2003. CQL: A language for continuous queries over streams and relations. In *International Workshop on Database Programming Languages*. Springer.

[7] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal* 15, 2 (2006). https://doi.org/10.1007/s00778-004-0147-z

[8] Patrick Bahr, Christian Uldal Graulund, and Rasmus Ejlers Møgelberg. 2019. Simply RaTT: A Fitch-Style Modal Calculus for Reactive Programming without Space Leaks. *Proc. ACM Program. Lang.* 3, ICFP, Article 109 (jul 2019), 27 pages. https://doi.org/10.1145/3341713

[9] Patrick Bahr, Christian Uldal Graulund, and Rasmus Ejlers Møgelberg. 2021. Diamonds Are Not Forever: Liveness in Reactive Programming with Guarded Recursion. *Proc. ACM Program. Lang.* 5, POPL, Article 2 (jan 2021), 28 pages. https://doi.org/10.1145/3434283

[10] Patrick Bahr and Rasmus Ejlers Møgelberg. 2023. Asynchronous Modal FRP. arXiv:2303.03170 [cs.PL]

[11] Edmon Begoli, Tyler Akidau, Fabian Hueske, Julian Hyde, Kathryn Knight, and Kenneth Knowles. 2019. One SQL to Rule Them All-an Efficient and Syntactically Idiomatic Approach to Management of Streams and Tables. In *International Conference on Management of Data (SIGMOD)*.

[12] Albert Benveniste, Paul Caspi, Stephen A Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert De Simone. 2003. The synchronous languages 12 years later. *Proc. IEEE* 91, 1 (2003).

[13] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. 2003. CDuce: An XML-Centric General-Purpose Language. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 51–63.

[14] James Brotherston. 2005. Cyclic Proofs for First-Order Logic with Inductive Definitions. In *Automated Reasoning with Analytic Tableaux and Related Methods (Lecture Notes in Computer Science)*, Bernhard Beckert (Ed.). Springer, Berlin, Heidelberg, 78–92. https://doi.org/10.1007/11554554_8

[15] Janusz A Brzozowski. 1964. Derivatives of regular expressions. *J. ACM* 11, 4 (1964).

[16] William H Burge. 1975. Stream processing functions. *IBM Journal of Research and Development* 19, 1 (1975).

[17] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 36, 4 (2015).

[18] Andrew Cave, Francisco Ferreira, Prakash Panangaden, and Brigitte Pientka. 2014. Fair Reactive Programming. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 361–372. https://doi.org/10.1145/2535838.2535881

[19] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J Franklin, Joseph M Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R Madden, Fred Reiss, and Mehul A Shah. 2003. TelegraphCQ: continuous dataflow processing. In *ACM SIGMOD International Conference on Management of Data (SIGMOD*. 668–668.

[20] Farzaneh Derakhshan. 2021. *Session-Typed Recursive Processes and Circular Proofs.* Ph. D. Dissertation. Caregie Mellon University. https://www.andrew.cmu.edu/user/fderakhs/publications/Dissertation_Farzaneh.pdf

[21] Philip Dexter, Yu David Liu, and Kenneth Chiu. 2022. The essence of online data processing. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 899–928.

[22] Volker Diekert and Grzegorz Rozenberg. 1995. *The Book of Traces.* World Scientific. https://doi.org/10.1142/2563

[23] Conal Elliott and Paul Hudak. 1997. Functional reactive animation. In *Second ACM SIGPLAN International Conference on Functional Programming (ICFP)*.

[24] Jérôme Fortier and Luigi Santocanale. 2013. Cuts for circular proofs: semantics and cut-elimination. In *Computer Science Logic 2013 (CSL 2013) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 23)*, Simona Ronchi Della Rocca (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 248–262. https://doi.org/10.4230/LIPIcs.CSL.2013.248 ISSN: 1868-8969.

[25] Apache Software Foundation. 2019. Apache Flink. https://flink.apache.org/. (Accessed July 2022.).

[26] Apache Software Foundation. 2019. Apache Heron (originally Twitter Heron). https://heron.incubator.apache.org/. (Accessed July 2022.).

[27] Apache Software Foundation. 2019. Apache Samza. https://samza.apache.org/. (Accessed July 2022.).

[28] Apache Software Foundation. 2019. Apache Spark Streaming. https://spark.apache.org/streaming/. (Accessed July 2022.).

[29] Apache Software Foundation. 2019. Apache Storm. https://storm.apache.org/. (Accessed July 2022.).

[30] Apache Software Foundation. 2021. Apache Beam. https://beam.apache.org/. (Accessed July 2022.).

[31] Alain Frisch, Giuseppe Castagna, and Veronique Benzaken. 2002. Semantic Subtyping. In *Logic in Computer Science (LICS)*.

[32] Dan Frumin. 2022. Semantic Cut Elimination for the Logic of Bunched Implications, Formalized in Coq. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Philadelphia, PA, USA) *(CPP 2022)*. Association for Computing Machinery, New York, NY, USA, 291–306. https://doi.org/10.1145/3497775.3503690

[33] Dan Frumin, Emanuele D'Osualdo, Bas van den Heuvel, and Jorge A. Pérez. 2022. A Bunch of Sessions: A Propositions-as-Sessions Interpretation of Bunched Implications in Channel-Based Concurrency. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 155 (oct 2022), 29 pages. https://doi.org/10.1145/3563318

[34] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. 2014. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)* 46, 4 (2014).

[35] CAR (Tony) Hoare, Bernhard Möller, Georg Struth, and Ian Wehrman. 2009. Concurrent Kleene Algebra. In *CONCUR 2009-Concurrency Theory: 20th International Conference, CONCUR 2009, Bologna, Italy, September 1-4, 2009. Proceedings 20*. Springer, 399–414.

[36] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty asynchronous session types. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 273–284.

[37] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. 2005. Regular Expression Types for XML. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 27, 1 (Jan. 2005), 46–90. Preliminary version in ICFP 2000.

[38] Zhe Hou, Alwen Tiu, and Rajeev Gore. 2015. A Labelled Sequent Calculus for BBI: Proof Theory and Proof Search. arXiv:1302.4783 [cs.LO]

[39] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Uğur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stan Zdonik. 2008. Towards a streaming SQL standard. *Proceedings of the VLDB Endowment* 1, 2 (2008).

[40] Alan Jeffrey. 2012. LTL Types FRP: Linear-Time Temporal Logic Propositions as Types, Proofs as Functional Reactive Programs. In *Proceedings of the Sixth Workshop on Programming Languages Meets Program Verification* (Philadelphia, Pennsylvania, USA) *(PLPV '12)*. Association for Computing Machinery, New York, NY, USA, 49–60. https://doi.org/10.1145/2103776.2103783

[41] Theodore Johnson, Shanmugavelayutham Muthukrishnan, Vladislav Shkapenyuk, and Oliver Spatscheck. 2005. A heartbeat mechanism and its application in Gigascope. In *31st International Conference on Very Large Data Bases (VLDB)*. VLDB Endowment.

[42] Gilles Kahn. 1974. The semantics of a simple language for parallel programming. *Information Processing* 74 (1974).

[43] Konstantinos Kallas, Filip Niksic, Caleb Stanford, and Rajeev Alur. 2020. DiffStream: differential output testing for stream processing programs. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020).

[44] Konstantinos Kallas, Filip Niksic, Caleb Stanford, and Rajeev Alur. 2022. Stream Processing With Dependency-Guided Synchronization. In *Principles and Practice of Parallel Programming (PPoPP)*.

[45] Tobias Kappé, Paul Brunet, Bas Luttik, Alexandra Silva, and Fabio Zanasi. 2019. On series-parallel pomset languages: Rationality, context-freeness and automata. *Journal of Logical and Algebraic Methods in Programming* 103 (2019), 130–153. https://doi.org/10.1016/j.jlamp.2018.12.001

[46] Neelakantan R. Krishnaswami. 2013. Higher-Order Functional Reactive Programming without Spacetime Leaks. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 221–232. https://doi.org/10.1145/2500365.2500588

[47] Edward A Lee and David G Messerschmitt. 1987. Synchronous data flow. *Proc. IEEE* 75, 9 (1987).

[48] Samuel Madden, Mehul Shah, Joseph M Hellerstein, and Vijayshankar Raman. 2002. Continuously Adaptive Continuous Queries over Streams. In *ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 12 pages. https://doi.org/10.1145/564691.564698

[49] Konstantinos Mamouras, Caleb Stanford, Rajeev Alur, Zachary G Ives, and Val Tannen. 2019. Data-trace types for distributed stream processing systems. In *40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[50] Antoni Mazurkiewicz. 1986. Trace theory. In *Advanced course on Petri nets*. Springer.

[51] Shadi A Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H Campbell. 2017. Samza: Stateful Scalable Stream Processing at LinkedIn. *Proceedings of the VLDB Endowment* 10, 12 (2017).

[52] Peter W O'Hearn and David J Pym. 1999. The logic of bunched implications. *Bulletin of Symbolic Logic* 5, 2 (1999), 215–244.

[53] Amir Pnueli. 1977. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. ieee, 46–57.

[54] John C Reynolds. 2002. Separation logic: A logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 55–74.

[55] Scott Schneider, Martin Hirzel, Buğra Gedik, and Kun-Lung Wu. 2013. Safe data parallelism for general streaming. *IEEE Trans. Comput.* 64, 2 (2013).

[56] Robert Soulé, Martin Hirzel, Robert Grimm, Buğra Gedik, Henrique Andrade, Vibhore Kumar, and Kun-Lung Wu. 2010. A universal calculus for stream processing languages. In *European Symposium on Programming (ESOP)*. Springer.

[57] Caleb Stanford. 2022. *Safe Programming over Distributed Streams.* Ph. D. Dissertation. University of Pennsylvania.

[58] Robert Stephens. 1997. A survey of stream processing. *Acta Informatica* 34, 7 (1997).

[59] William Thies, Michal Karczmarek, and Saman Amarasinghe. 2002. StreamIt: A language for streaming applications. In *International Conference on Compiler Construction*. Springer.

[60] Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. 2003. Exploiting Punctuation Semantics in Continuous Data Streams. *IEEE Trans. on Knowl. and Data Eng.* 15, 3 (mar 2003), 555–568. https://doi.org/10.1109/TKDE.2003.1198390

[61] Peter A Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. 2003. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering* 15, 3 (2003).

[62] Jana Wagemaker, Nate Foster, Tobias Kappé, Dexter Kozen, Jurriaan Rot, and Alexandra Silva. 2022. Concurrent NetKAT: Modeling and analyzing stateful, concurrent networks. In *European Symposium on Programming*. Springer International Publishing Cham, 575–602.

[63] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. 2013. Discretized Streams: Fault-tolerant Streaming Computation at Scale. In *24th Symposium on Operating Systems Principles (SOSP)*. ACM. https://doi.org/10.1145/2517349.2522737

## A MECHANIZATION AND REFERENCE INTERPRETER

The supplemental material contains a Coq formalization of our main results and a reference interpreter written in Haskell.

The Coq formalization includes the following:

Table 1. Overview of Coq Development

| Name | Description | Location |
|---|---|---|
| Basics | Definitions of types, contexts, batches, and prefixes | `verif/basics.v` |
| CtxSub | Context subtyping and its semantics | `verif/ctxsubty.v` |
| PProj | Prefix and batch projection definitions and theorems | `verif/pproj.v` |
| PrefixConcat | Prefix and batch concatenation definitions and theorems | `verif/prefixconc` |
| Events | Events, serialization and deserialization | `verif/events.v` |
| Language | Syntax of terms, the typing judgment, batch and incremental semantics, correctness and the homomorphism property | `verif/language.v` |

The Haskell interpreter assumes well-typed terms, failing if anything is ill-typed; we have not translated the type system definition to Haskell. We do not have proofs that the interpreter (a function) and the Coq definition of the semantics (an inductive relation) agree, but we have tested a majority of the important theorems from Coq via QuickCheck. We also use QuickCheck to check the homomorphism theorem for fixed terms. The important directories include:

Table 2. Overview of Haskell Interpreter

| Name | Description | Location |
|---|---|---|
| Types | Types and contexts | `src/Types.hs` |
| Interpreter | Code for the interpreter | `src/Interpreter.hs` |
| Examples | Core terms for examples | `src/Examples.hs` |

The Coq development has been tested with Coq version 8.17.0 using CoqHammer Tactics version 1.3.2. The Haskell development has been tested with GHC version 9.2.7 and Stack version 2.9.3.

## B TECHNICALITIES

This appendix collects technical definitions that did not fit in the main body of the paper.

### B.1 Basics

Stream types are defined by the following grammar. The base types included are the unit type 1 which types streams that contain exactly one unit element, the type of the empty stream $\varepsilon$, and the type of streams consisting of a single integer, $\mathtt{Int}$. Larger types include the concatenation type $s \cdot t$, the sum type $s + t$, the parallel stream type $s \| t$, and the star type $s^\star$.

$$s, t, r ::= \quad 1 \mid \varepsilon \mid \mathtt{Int} \mid s \cdot t \mid s + t \mid s \| t \mid s^\star$$

Contexts in the stream types calculus system have a bunched structure. The context former $\Gamma,\ \Delta$ corresponds to the parallel type, while the context former $\Gamma;\ \Delta$ corresponds to the concatenation type. The two context formers share a unit, written as "$\cdot$".

$$\Gamma ::= \quad \cdot \mid \Gamma,\ \Gamma \mid \Gamma;\ \Gamma \mid x : s$$

A stream type is *nullable* if it includes no data. Nullable types are parallel combinations of $\varepsilon$s.

*Definition B.1 (Nullable).* We define a judgment $s\ \mathtt{nullable}$ as follows:

$$\frac{}{\varepsilon\ \mathtt{nullable}} \qquad \frac{s\ \mathtt{nullable} \qquad t\ \mathtt{nullable}}{s \| t\ \mathtt{nullable}}$$

We extend to contexts in the natural way:

$$\frac{}{\cdot\ \mathtt{nullable}} \qquad \frac{\Gamma\ \mathtt{nullable} \qquad \Gamma'\ \mathtt{nullable}}{\Gamma,\ \Gamma'\ \mathtt{nullable}}$$

Batches are defined as in the paper, with a typing relation $b : \mathtt{batch}(s)$ and a context typing relation $g : \mathtt{batch}(\Gamma)$.

*Definition B.2 (Batch).* The grammar of batches is given by:

$$b ::= \mathtt{oneb} \mid \mathtt{batch}_{\mathtt{Int}}(n) \mid \mathtt{epsb} \mid \mathtt{catb}(b, b') \mid \mathtt{parb}(b, b') \mid \mathtt{sumbA}(b) \mid \mathtt{sumbB}(b) \mid [] \mid b :: b'$$

We define the batch has type relation as:

$$\frac{}{\mathtt{oneb} : \mathtt{batch}(1)} \qquad \frac{}{\mathtt{epsb} : \mathtt{batch}(\varepsilon)} \qquad \frac{b : \mathtt{batch}(s) \qquad b' : \mathtt{batch}(t)}{\mathtt{catb}(b, b') : \mathtt{batch}(s \cdot t)}$$

$$\frac{b : \mathtt{batch}(s) \qquad b' : \mathtt{batch}(t)}{\mathtt{parb}(b, b') : \mathtt{batch}(s \| t)} \qquad \frac{b : \mathtt{batch}(s)}{\mathtt{sumbA}(b) : \mathtt{batch}(s + t)} \qquad \frac{b : \mathtt{batch}(t)}{\mathtt{sumbB}(b) : \mathtt{batch}(s + t)}$$

$$\frac{}{[] : \mathtt{batch}(s^\star)} \qquad \frac{b : \mathtt{batch}(s) \qquad b' : \mathtt{batch}(s^\star)}{b :: b' : \mathtt{batch}(s^\star)}$$

And we lift this to contexts by:

$$\frac{}{\mathtt{epsb} : \mathtt{batch}(\cdot)} \qquad \frac{b : \mathtt{batch}(\Gamma) \qquad b' : \mathtt{batch}(\Delta)}{\mathtt{catb}(b, b') : \mathtt{batch}(\Gamma;\ \Delta)} \qquad \frac{b : \mathtt{batch}(\Gamma) \qquad b' : \mathtt{batch}(\Delta)}{\mathtt{parb}(b, b') : \mathtt{batch}(\Gamma,\ \Delta)}$$

Prefixes are also like in the main paper, a typing relation $p : \mathtt{prefix}(s)$ and a context typing relation $p : \mathtt{prefix}(\Gamma)$.

*Definition B.3 (Prefix).* The grammar of prefixes is given by:

$$\begin{aligned} p ::= {} & \mathsf{onepA} \mid \mathit{onepB} \mid \mathsf{epsp} \mid \mathsf{parp}(p, p') \\ & \mid \mathsf{catpA}(p) \mid \mathsf{catpB}(b, p) \\ & \mid \mathsf{sumpEmp} \mid \mathsf{sumpA}(p) \mid \mathsf{sumpB}(p) \\ & \mid \mathsf{stpEmp} \mid \mathsf{stpDone} \\ & \mid \mathsf{stpA}(p) \mid \mathsf{stpB}(b, p) \end{aligned}$$

$$\frac{}{\mathsf{epsp} : \mathsf{prefix}(\varepsilon)} \qquad \frac{}{\mathsf{onepA} : \mathsf{prefix}(1)} \qquad \frac{}{\mathsf{onepB} : \mathsf{prefix}(1)}$$

$$\frac{p : \mathsf{prefix}(\Gamma) \qquad p' : \mathsf{prefix}(\Gamma')}{\mathsf{parp}(p, p') : \mathsf{prefix}(\Gamma, \Gamma')} \qquad \frac{p : \mathsf{prefix}(s)}{\mathsf{catpA}(p) : \mathsf{prefix}(s \cdot t)}$$

$$\frac{b : \mathsf{batch}(s) \qquad p : \mathsf{prefix}(t)}{\mathsf{catpB}(b, p) : \mathsf{prefix}(s \cdot t)} \qquad \frac{}{\mathsf{sumpEmp} : \mathsf{prefix}(s + t)} \qquad \frac{p : \mathsf{prefix}(s)}{\mathsf{sumpA}(p) : \mathsf{prefix}(s + t)}$$

$$\frac{p : \mathsf{prefix}(t)}{\mathsf{sumpB}(p) : \mathsf{prefix}(s + t)} \qquad \frac{}{\mathsf{stpEmp} : \mathsf{prefix}(s^\star)} \qquad \frac{}{\mathsf{stpDone} : \mathsf{prefix}(s^\star)}$$

$$\frac{p : \mathsf{prefix}(s)}{\mathsf{stpA}(p) : \mathsf{prefix}(s^\star)} \qquad \frac{b : \mathsf{batch}(s) \qquad p : \mathsf{prefix}(s^\star)}{\mathsf{stpB}(b, p) : \mathsf{prefix}(s^\star)}$$

We lift this to contexts by:

$$\frac{p : \mathsf{prefix}(\Gamma) \qquad p' : \mathsf{prefix}(\Gamma')}{\mathsf{parp}(p, p') : \mathsf{prefix}(\Gamma, \Gamma')} \qquad \frac{p : \mathsf{prefix}(\Gamma)}{\mathsf{catpA}(p) : \mathsf{prefix}(\Gamma; \Gamma')}$$

$$\frac{b : \mathsf{batch}(\Gamma) \qquad p : \mathsf{prefix}(\Gamma')}{\mathsf{catpB}(b, p) : \mathsf{prefix}(\Gamma; \Gamma')}$$

For each type $s$, we define the "empty" prefix $\mathsf{emp}_s$ inductively on the structure of $s$. In principle, one could define a single empty prefix shared by all types, but this would over-represent in many cases: the pointed empty prefix $\mathsf{emp}$ would not be the same as $\mathsf{parp}(\mathsf{emp}, \mathsf{emp})$, for example. While type-indexing the definition requires us to carry around more types at runtime (to compute the empty prefix of the right type), we view this as a preferable choice.

*Definition B.4 (Empty Prefix).* The empty prefix is defined as follows:

$(\varepsilon)$ $\mathsf{emp}_\varepsilon = \mathsf{epsp}$
$(1)$ $\mathsf{emp}_1 = \mathsf{onepA}$
$(s \| t)$ $\mathsf{emp}_{s\|t} = \mathsf{parp}(\mathsf{emp}_s, \mathsf{emp}_t)$
$(s + t)$ $\mathsf{emp}_{s+t} = \mathsf{sumpEmp}$
$(s \cdot t)$ $\mathsf{emp}_{s\cdot t} = \mathsf{catpA}(\mathsf{emp}_s)$
$(s^\star)$ $\mathsf{emp}_{s^\star} = \mathsf{stpEmp}$

We lift this to contexts in the natural way, with $\mathsf{emp}_\cdot = \mathsf{epsp}$, and $\mathsf{emp}_{\Gamma;\Delta} = \mathsf{catpA}(\mathsf{emp}_\Gamma)$, and $\mathsf{emp}_{\Gamma,\Delta} = \mathsf{parp}(\mathsf{emp}_\Gamma, \mathsf{emp}_\Delta)$.

Given a batch, we can forget about its batch structure and demote it to a prefix.

*Definition B.5 (Batch to Prefix Lift).*

$$
\begin{aligned}
(\mathsf{oneb})^\circ &= \mathsf{onepB} \\
(\mathsf{epsb})^\circ &= \mathsf{epsp} \\
(\mathsf{catb}\,(b, b'))^\circ &= \mathsf{catpB}(b, (b')^\circ) \\
(\mathsf{parb}\,(b, b'))^\circ &= \mathsf{parp}((b)^\circ, (b')^\circ) \\
(\mathsf{sumbA}\,(b))^\circ &= \mathsf{sumpA}((b)^\circ) \\
(\mathsf{sumbB}\,(b))^\circ &= \mathsf{sumpB}((b)^\circ) \\
([])^\circ &= \mathsf{stpDone} \\
(b :: b')^\circ &= \mathsf{stpB}(b, (b')^\circ)
\end{aligned}
$$

This function takes well-typed batches to well-typed prefixes.

Theorem B.6 (Batch To Prefix Lift Well-Typed). *If* $b : \mathtt{batch}\,(s)$, *then* $(b)^\circ : \mathtt{prefix}\,(s)$

Proof. `basics.v:prefixOf_correct` □

If a batch converted to a prefix is well-typed, then the batch was also well-typed.

Theorem B.7 (Batch To Prefix Lift Well-Typed Converse). *If* $(b)^\circ : \mathtt{prefix}\,(s)$, *then* $b : \mathtt{batch}\,(s)$.

Proof. `basics.v:prefixOf_inv` □

Theorem B.8 (Batch To Prefix Lift Well-Typed (Context)). *If* $b : \mathtt{batch}\,(\Gamma)$, *then* $(b)^\circ : \mathtt{prefix}\,(\Gamma)$

Proof. `basics.v:prefixOf_correct_ctx` □

## B.2 Derivatives

*Definition B.9 (Derivatives).* We define a 3-place relation $\delta_p\,(s) \sim s'$ between a prefix and two types.

$$
\frac{}{\delta_{\mathsf{epsp}}\,(\varepsilon) \sim \varepsilon} \qquad \frac{}{\delta_{\mathsf{onepA}}\,(1) \sim 1} \qquad \frac{}{\delta_{\mathsf{onepB}}\,(1) \sim \varepsilon} \qquad \frac{\delta_p\,(s) \sim s'}{\delta_{\mathsf{catpA}(p)}\,(s \cdot t) \sim s' \cdot t}
$$

$$
\frac{\delta_p\,(t) \sim t'}{\delta_{\mathsf{catpB}(b,p)}\,(s \cdot t) \sim t'} \qquad \frac{\delta_p\,(s) \sim s' \qquad \delta_{p'}\,(t) \sim t'}{\delta_{\mathsf{parp}(p,p')}\,(s \| t) \sim s' \| t'} \qquad \frac{}{\delta_{\mathsf{sumpEmp}}\,(s + t) \sim s + t}
$$

$$
\frac{\delta_p\,(s) \sim s'}{\delta_{\mathsf{sumpA}(p)}\,(s + t) \sim s'} \qquad \frac{\delta_p\,(t) \sim t'}{\delta_{\mathsf{sumpB}(p)}\,(s + t) \sim t'} \qquad \frac{}{\delta_{\mathsf{stpEmp}}\,(s^\star) \sim s^\star} \qquad \frac{}{\delta_{\mathsf{stpDone}}\,(s^\star) \sim \varepsilon}
$$

$$
\frac{\delta_p\,(s) \sim s'}{\delta_{\mathsf{stpA}(p)}\,(s^\star) \sim s' \cdot s^\star} \qquad \frac{\delta_p\,(s^\star) \sim s'}{\delta_{\mathsf{stpB}(\_,p)}\,(s^\star) \sim s'}
$$

We lift this to contexts in the natural way:

$$
\frac{\delta_p\,(\Gamma) \sim \Gamma'}{\delta_{\mathsf{catpA}(p)}\,(\Gamma;\, \Delta) \sim \Gamma';\, \Delta} \qquad \frac{\delta_p\,(\Delta) \sim \Delta'}{\delta_{\mathsf{catpB}(b,p)}\,(\Gamma;\, \Delta) \sim \Delta'} \qquad \frac{\delta_p\,(\Gamma) \sim \Gamma' \qquad \delta_{p'}\,(\Delta) \sim \Delta'}{\delta_{\mathsf{parp}(p,p')}\,(\Gamma,\, \Delta) \sim \Gamma',\, \Delta'}
$$

Derivatives are functions defined when the prefix input is well-typed.

Theorem B.10 (Derivative Function). *For any* $p$ *and* $s$, *there is at most one* $s'$ *such that* $\delta_p\,(s) \sim s'$. *If* $p : \mathtt{prefix}\,(s)$, *then such an* $s'$ *exists.*

Proof. Existence by `basics.v:derivrel_fun`, uniqueness by `basics.v:derivrel_det`. □

When it's guaranteed to exist, we write this $s'$ simply as $\delta_p\ (s)$. The empty prefix is the identity for the derivative operator.

Theorem B.11 (Empty Prefix Derivative). $\delta_{\mathsf{emp}_s}\ (s) = s$.

Proof. `basics.v:derivrel_emp` □

Theorem B.12 (Derivative Function (Contexts)). *For any $p$ and $\Gamma$, there is at most one $\Gamma'$ such that $\delta_p\ (\Gamma) \sim \Gamma'$. If $p$ : `prefix` $(\Gamma)$, then such an $\Gamma'$ exists.*

Proof. Existence by `basics.v:derivrelCtx_fun`, uniqueness by `basics.v:derivrelCtx_det`. □

Theorem B.13 (Empty Prefix Derivative (Contexts)). $\delta_{\mathsf{emp}_s}\ (\Gamma) \sim \Gamma$.

Proof. `basics.v:derivrelCtx_emp` □

If we take a batch, convert it to a prefix, and then take a derivative by that prefix, the resulting type is nullable. Intuitively, this is because "the rest" of a stream after a batch should be empty.

Theorem B.14 (Nullable Batch Derivative). *If $b$ : `batch` $(s)$ then $\delta_{(b)^\circ}\ (s)$ `nullable`*

Proof. `basics.v:batch_lift_derivrel` □

Theorem B.15 (Nullable Batch Derivative). *If $b$ : `batch` $(\Gamma)$ then $\delta_{(b)^\circ}\ (\Gamma)$ `nullable`*

Proof. `basics.v:batch_lift_derivrelCtx` □

### B.3 Done Batches

For a nullable type $s$, there is a unique "empty batch", which we refer to as the done batch.

*Definition B.16 (Done Batch).*

$$\frac{}{\epsilon\ \mathsf{done}\ \mathsf{epsb}} \qquad \frac{s\ \mathsf{done}\ b \qquad t\ \mathsf{done}\ b'}{s \| t\ \mathsf{done}\ \mathsf{parb}\ (b, b')}$$

Theorem B.17 (Done Batch Correctness). *For any $s$, there is at most one $b$ such that $s$ `done` $b$. Moreover, such a $b$ exists exactly when $s$ `nullable`.*

Proof. Uniqueness by `basics.v:doneBatch_det`, existence by `basics.v:doneBatchTheorem` □

### B.4 Concatenation

*Prefix-Batch Concatenation.* Given a prefix $p$ of $s$, and a batch $b$ of $\delta_p\ (s)$, we often want to consider the batch of $s$ given by $p$, followed by $b$. The following functional relation gives a way of computing this. We also lift this definition to contexts.

*Definition B.18 (Prefix-Batch Concatenation).* We define a relation $p \cdot_s b \sim b'$.

$$\frac{}{\mathsf{epsp} \cdot_\varepsilon \mathsf{epsb} \sim \mathsf{epsb}} \qquad \frac{}{\mathsf{onepA} \cdot_1 \mathsf{oneb} \sim \mathsf{oneb}} \qquad \frac{}{\mathsf{onepB} \cdot_1 \mathsf{epsb} \sim \mathsf{oneb}}$$

$$\frac{p_1 \cdot_s p_1' \sim b_1' \qquad p_2 \cdot_t p_2' \sim b_2'}{\mathsf{parp}(p_1, p_2) \cdot_{s\|t} \mathsf{parb}\,(b_1, b_2) \sim \mathsf{parb}\,\left(b_1', b_2'\right)} \qquad \frac{}{\mathsf{sumpEmp} \cdot_{s+t} b \sim b}$$

$$\frac{p \cdot_s b \sim b'}{\mathsf{sumpA}(p) \cdot_{s+t} b \sim \mathsf{sumbA}\,(b')} \qquad \frac{p \cdot_t b \sim b'}{\mathsf{sumpB}(p) \cdot_{s+t} b \sim \mathsf{sumbB}\,(b')}$$

$$\frac{p \cdot_s b_1 \sim b_1'}{\mathsf{catpA}(p) \cdot_{s\cdot t} \mathsf{catb}\,(b_1, b_2) \sim \mathsf{catb}\,\left(b_1', b_2\right)} \qquad \frac{p \cdot_t b' \sim b''}{\mathsf{catpB}(b, p) \cdot_{s\cdot t} b' \sim \mathsf{catb}\,(b, b'')}$$

$$\frac{}{\mathsf{stpEmp} \cdot_{s^\star} b \sim b} \qquad \frac{}{\mathsf{stpDone} \cdot_{s^\star} \mathsf{epsb} \sim []} \qquad \frac{p \cdot_s b_1 \sim b_1'}{\mathsf{stpA}(p) \cdot_{s^\star} \mathsf{catb}\,(b_1, b_2) \sim b_1' :: b_2}$$

$$\frac{p \cdot_{s^\star} b' \sim b''}{\mathsf{stpB}(b, p) \cdot_{s^\star} b' \sim b :: b''}$$

We lift this to contexts in the natural way

$$\frac{p_1 \cdot_\Gamma p_1' \sim b_1' \qquad p_2 \cdot_\Delta p_2' \sim b_2'}{\mathsf{parp}(p_1, p_2) \cdot_{\Gamma,\Delta} \mathsf{parb}\,(b_1, b_2) \sim \mathsf{parb}\,\left(b_1', b_2'\right)} \qquad \frac{p \cdot_\Gamma b_1 \sim b_1'}{\mathsf{catpA}(p) \cdot_{\Gamma;\Delta} \mathsf{catb}\,(b_1, b_2) \sim \mathsf{catb}\,\left(b_1', b_2\right)}$$

$$\frac{p \cdot_\Delta b' \sim b''}{\mathsf{catpB}(b, p) \cdot_{\Gamma;\Delta} b' \sim \mathsf{catb}\,(b, b'')}$$

Theorem B.19 establishes the functional behavior of this relation. When the preconditions are satisfied ($p : \mathtt{prefix}\,(s)$ and $b : \mathtt{batch}\left(\delta_p\,(s)\right)$), we write $p \cdot_s b$ for the unique $b'$ that it guarantees.

Theorem B.19 (Batch Prefix Concatenation Function). *For all $s, p, b$, there is at most one $b'$ such that $p \cdot_s b \sim b'$. If $p : \mathtt{prefix}\,(s)$ and $b : \mathtt{batch}\left(\delta_p\,(s)\right)$, then such a $b'$ exists, and it satisfies $b' : \mathtt{batch}\,(s)$.*

Proof. Existence by `prefixconcat.v:prefixBatchConcatRel_fun`,
uniqueness by `prefixconcat.v:prefixBatchConcatRel_det`,
type-correctness by `prefixconcat.v:prefixBatchConcatRel_correct`. □

If we take a batch $b$ and demote it to a prefix with $(b)^\circ$, then concatenating any batch $b'$ to it just gives $b$ back. Intuitively, this is because if $b : \mathtt{batch}\,(s)$, then the type $\delta_{(b)^\circ}\,(s)$ is nullable, and so any batch $b'$ of this type contains no data.

Theorem B.20 (Batch Prefix Concatenation of Prefix Lift). *If $b : \mathtt{batch}\,(s)$ and $\delta_{(b)^\circ}\,(s) \sim s'$ and $b' : \mathtt{batch}\,(s')$, then $(b)^\circ \cdot_s b' = b$.*

Proof. `prefixconcat.v:prefixBatchConcatRel_prefixOf` □

Conversely, concatenating a batch $b$ onto the empty prefix $\mathsf{emp}_s$ gives the whole batch $b$. This theorem relies on Theorem B.13, which ensures that $\delta_{\mathsf{emp}_b}\,(s) = s$.

Theorem B.21 (Batch Prefix Concatenation of Empty). *If $b : \mathtt{batch}\,(s)$, then $\mathsf{emp}_s \cdot_s b = b$.*

Proof. `prefixconcat.v:prefixBatchConcatRel_emp` □

All of the three previous theorems all have identical versions for batches and prefixes of contexts.

Theorem B.22 (Batch Prefix Concatenation Function (Contexts)). *For all $\Gamma, p, b$, there is at most one $b'$ such that $p \cdot_\Gamma b \sim b'$. If $p$ : $\mathtt{prefix}\,(\Gamma)$ and $b$ : $\mathtt{batch}\,(\delta_p\,(\Gamma))$, then such a $b'$ exists, and it satisfies $b'$ : $\mathtt{batch}\,(\Gamma)$.*

Proof. Existence by `prefixconcat.v:prefixBatchConcatCtxRel_fun`,
uniqueness by `prefixconcat.v:prefixBatchConcatCtxRel_det`,
type-correctness by `prefixconcat.v:prefixBatchConcatCtxRel_correct`. □

Theorem B.23 (Batch Prefix Concatenation of Prefix Lift (Contexts)). *If $b$ : $\mathtt{batch}\,(\Gamma)$ and $b'$ : $\mathtt{batch}\,(\delta_{(b)^\circ}\,(\Gamma))$, then $(b)^\circ \cdot_\Gamma b' = b$.*

Proof. `prefixconcat.v:prefixBatchConcatCtxRel_prefixOf` □

Theorem B.24 (Batch Prefix Concatenation of Empty (Contexts)). *If $b$ : $\mathtt{batch}\,(s)$, then $\mathtt{emp}_s \cdot_s b = b$.*

Proof. `prefixconcat.v:prefixBatchConcatCtxRel_emp` □

*Prefix-Prefix Concatenation.* More generally, we often want to concatenate a prefix $p$ of $s$ with a prefix $p'$ of $\delta_p\,(s)$. This is defined with another 4-place, type-indexed relation.

*Definition B.25 (Prefix Concatenation).* We define a relation $p \cdot_s p' \sim p''$.

$$\frac{}{\mathtt{epsp} \cdot_\varepsilon \mathtt{epsp} \sim \mathtt{epsp}}$$

$$\frac{p \;:\; \mathtt{prefix}\,(1)}{\mathtt{onepA} \cdot_1 p \sim p} \qquad \frac{}{\mathtt{onepB} \cdot_1 \mathtt{epsp} \sim \mathtt{onepB}}$$

$$\frac{p_1 \cdot_s p_1' \sim p_1'' \qquad p_2 \cdot_t p_2' \sim p_2''}{\mathtt{parp}(p_1, p_2) \cdot_{s\|t} \mathtt{parp}(p_1', p_2') \sim \mathtt{parp}(p_1'', p_2'')}$$

$$\frac{p \cdot_s p' \sim p''}{\mathtt{catpA}(p) \cdot_{s\cdot t} \mathtt{catpA}(p') \sim \mathtt{catpA}(p'')} \qquad \frac{p \cdot_s b \sim b'}{\mathtt{catpA}(p) \cdot_{s\cdot t} \mathtt{catpB}(b, p') \sim \mathtt{catpB}(b', p')}$$

$$\frac{p \cdot_t p' \sim p''}{\mathtt{catpB}(b, p) \cdot_{s\cdot t} p' \sim \mathtt{catpB}(b, p'')}$$

$$\frac{}{\mathtt{sumpEmp} \cdot_{s+t} p \sim p} \qquad \frac{p \cdot_s p' \sim p''}{\mathtt{sumpA}(p) \cdot_{s+t} p' \sim \mathtt{sumpA}(p'')} \qquad \frac{p \cdot_t p' \sim p''}{\mathtt{sumpB}(p) \cdot_{s+t} p' \sim \mathtt{sumpB}(p'')}$$

$$\frac{}{\mathtt{stpEmp} \cdot_{s\star} p \sim p} \qquad \frac{}{\mathtt{stpDone} \cdot_{s\star} \mathtt{epsp} \sim \mathtt{stpDone}} \qquad \frac{p \cdot_s p' \sim p''}{\mathtt{stpA}(p) \cdot_{s\star} \mathtt{catpA}(p') \sim \mathtt{stpA}(p'')}$$

$$\frac{p \cdot_s b \sim b'}{\mathtt{stpA}(p) \cdot_{s\star} \mathtt{catpB}(b, p') \sim \mathtt{stpB}(b', p')} \qquad \frac{p \cdot_{s\star} p' \sim p''}{\mathtt{stpB}(b, p) \cdot_{s\star} p' \sim \mathtt{stpB}(b, p'')}$$

We lift this to contexts.

$$\frac{p_1 \cdot_\Gamma p_1' \sim p_1'' \qquad p_2 \cdot_\Delta p_2' \sim p_2''}{\mathtt{parp}(p_1, p_2) \cdot_{\Gamma,\Delta} \mathtt{parp}(p_1', p_2') \sim \mathtt{parp}(p_1'', p_2'')}$$

$$\frac{p \cdot_\Gamma p' \sim p''}{\mathtt{catpA}(p) \cdot_{\Gamma;\Delta} \mathtt{catpA}(p') \sim \mathtt{catpA}(p'')} \qquad \frac{p \cdot_\Gamma b \sim b'}{\mathtt{catpA}(p) \cdot_{\Gamma;\Delta} \mathtt{catpB}(b, p') \sim \mathtt{catpB}(b', p')}$$

$$\frac{p \cdot_\Delta p' \sim p''}{\mathtt{catpB}(b, p) \cdot_{\Gamma;\Delta} p' \sim \mathtt{catpB}(b, p'')}$$

This relation is a function when the inputs are well-typed. Because of this, when $p : \texttt{prefix}\,(s)$ and $p' : \texttt{prefix}\,(\delta_p\,(s))$, we write $p \cdot_s p'$ for the unique $p''$ that the following theorem guarantees.

Theorem B.26 (Prefix Concatenation Function). *For all $p, p'$ and $s$, there is at most one $p''$ such that $p \cdot_s p' \sim p''$. If $p : \texttt{prefix}\,(s)$ and $p' : \texttt{prefix}\,(\delta_p\,(s))$, then such a $p''$ exists, and satisfies:*

(1) $p'' : \texttt{prefix}\,(s)$
(2) $\delta_{p''}\,(s) = \delta_{p'}\,(\delta_p\,(s))$

Proof. Existence by `prefixconcat.v:prefixConcatRel_fun`,
uniqueness by `prefixconcat.v:prefixConcatRel_det`,
well-typedness and derivative condition by `prefixconcat.v:prefixConcatRel_correct`. □

Concatenating a prefix $p$ to the empty prefix yields $p$ back, and...

Theorem B.27 (Prefix Concatenation Empty 1). *If $p : \texttt{prefix}\,(s)$, then $\texttt{emp}_s \cdot_s p = p$*

Proof. `prefixconcat.v:prefixConcatRel_emp` □

Concatenating the empty prefix to a prefix $p$ also gives $p$.

Theorem B.28 (Prefix Concatenation Empty 2). *If $p : \texttt{prefix}\,(s)$, then $p \cdot_s \texttt{emp}_{\delta_p(s)} = p$*

Proof. `prefixconcat.v:prefixConcatRel_emp'` □

Theorem B.29 (Prefix Concatenation Batch Lift 1). *If $b : \texttt{batch}\,(s)$ and $p : \texttt{prefix}\,(\delta_{(b)^\circ}\,(s))$ then $(b)^\circ \cdot_s p = (b)^\circ$*

Proof. `prefixconcat.v:prefixConcatRel_prefixOf` □

Theorem B.30 (Prefix Concatenation Batch Lift 2). *If $p \cdot_s b \sim b'$, then $p \cdot_s (b)^\circ \sim (b')^\circ$.*

Proof. `prefixconcat.v:prefixConcatRel_prefixOf'` □

Batch-to-prefix concatenation is associative.

Theorem B.31 (Batch Prefix Concatenation Associativity). $p \cdot_s \left(p' \cdot_{\delta_p(s)} b\right) = (p \cdot_s p') \cdot_s b$

Proof. `prefixconcat.v:prefixBatchConcatRel_assoc` □

Prefix concatenation is also associative.

Theorem B.32 (Prefix Concatenation Associativity). $p \cdot_s \left(p' \cdot_{\delta_p(s)} p''\right) = (p \cdot_s p') \cdot_s p''$

Proof. `prefixconcat.v:prefixConcatRel_assoc` □

All of the previous theorems about prefix concatenation for types are also needed for contexts.

Theorem B.33 (Prefix Concatenation Function (Contexts)). *For all $p, p'$ and $\Gamma$, there is at most one $p''$ such that $p \cdot_\Gamma p' \sim p''$. If $p : \texttt{prefix}\,(\Gamma)$ and $p' : \texttt{prefix}\,(\delta_p\,(\Gamma))$, then such a $p''$ exists, and satisfies:*

(1) $p'' : \texttt{prefix}\,(\Gamma)$
(2) $\delta_{p''}\,(\Gamma) = \delta_{p'}\,(\delta_p\,(\Gamma))$

Proof. Existence by `prefixconcat.v:prefixConcatCtxRel_fun`,
uniqueness by `prefixconcat.v:prefixConcatCtxRel_det`,
well-typedness and derivative condition by `prefixconcat.v:prefixConcatCtxRel_correct`. □

Theorem B.34 (Batch Prefix Concatenation Associativity (Contexts)). $p \cdot_\Gamma \left(p' \cdot_{\delta_p(\Gamma)} b\right) = (p \cdot_\Gamma p') \cdot_\Gamma b$

Proof. `prefixconcat.v:prefixBatchConcatCtxRel_assoc` □

Theorem B.35 (Prefix Concatenation Associativity (Contexts)). $p \cdot_\Gamma \left(p' \cdot_{\delta_p(\Gamma)} p''\right) = (p \cdot_\Gamma p') \cdot p''$

Proof. `prefixconcat.v:prefixConcatCtxRel_assoc` □

Theorem B.36 (Prefix Concatenation Batch Lift 1 (Contexts)). *If* $b : \mathtt{batch}\,(\Gamma)$ *and* $p : \mathtt{prefix}\,(\delta_{(b)^\circ}(\Gamma))$, *then* $(b)^\circ \cdot_\Gamma \Gamma = (b)^\circ$

Proof. `prefixconcat.v:prefixConcatCtxRel_prefixOf` □

Theorem B.37 (Prefix Concatenation Batch Lift 2 (Contexts)). *If* $p \cdot_\Gamma b \sim b'$, *then* $p \cdot_\Gamma (b)^\circ \sim (b')^\circ$.

Proof. `prefixconcat.v:prefixConcaCtxtRel_prefixOf'` □

### B.5 Projection Relations

*Batch Projection.* Given a batch or prefix of a context $\Gamma(\Delta)$, we often want to project out the part of the data corresponding to the subtree $\Delta$. In the case of batches, this always exists: a batch of $\Gamma(\Delta)$ contains a full batch of $\Delta$, located by following the hole in $\Gamma(-)$ down to $\Delta$. We also often want to substitute a new batch in where the old one was — this corresponds to a sort of "re-binding" of the subcontext $\Delta'$ to a new one. The batch projection relation $b \rightsquigarrow_{\Gamma(-)} b', f$ computes both of these things: $b'$ is the batch found at the $\Gamma(-)$-position in $b$, and $f$ is a function from batches to batches such that $f(b'')$ is the batch $b$, but with $b''$ where $b'$ used to be.

*Definition B.38 (Batch Projection).* We inductively define the relation $b \rightsquigarrow_{\Gamma(-)} b'$ as follows

$$\frac{}{b \rightsquigarrow_{-} b, \mathtt{id}} \qquad \frac{b_1 \rightsquigarrow_{\Gamma_1(-)} b, f \qquad g(b') = \mathtt{parb}\,(f(b'), b_2)}{\mathtt{parb}\,(b_1, b_2) \rightsquigarrow_{\Gamma_1(-),\Gamma_2} b, g}$$

$$\frac{b_2 \rightsquigarrow_{\Gamma_2(-)} b, f \qquad g(b') = \mathtt{parb}\,(b_1, f(b'))}{\mathtt{parb}\,(b_1, b_2) \rightsquigarrow_{\Gamma_1,\Gamma_2(-)} b, g} \qquad \frac{b_1 \rightsquigarrow_{\Gamma_1(-)} b, f \qquad g(b') = \mathtt{catb}\,(f(b'), b_2)}{\mathtt{catb}\,(b_1, b_2) \rightsquigarrow_{\Gamma_1(-);\Gamma_2} b, g}$$

$$\frac{b_2 \rightsquigarrow_{\Gamma_2(-)} b, f \qquad g(b') = \mathtt{catb}\,(b_1, f(b'))}{\mathtt{catb}\,(b_1, b_2) \rightsquigarrow_{\Gamma_1;\Gamma_2(-)} b, g}$$

Batch projection is deterministic, in the sense that the batch $b$ and the context $\Gamma(-)$ determine $b'$ and $f$.

Theorem B.39 (Batch Projection Determinism). *For all* $b$, *and* $\Gamma(-)$, *there is at most one pair* $(b', f)$ *such that* $b \rightsquigarrow_{\Gamma(-)} b', f$.

Proof. `pproj.v:bproj_det` □

The following establishes the fact that batch projection is a function: if the input is well-typed, there is an output pair which makes the relation hold (exactly one, by the previous theorem).

Theorem B.40 (Batch Projection Function). *For all* $b$, *and* $\Gamma(-)$, *if* $b : \mathtt{batch}\,(\Gamma(\Delta))$, *then there are* $b'$ *and* $f$ *such that* $b \rightsquigarrow_{\Gamma(-)} b', f$.

Proof. `pproj.v:bproj_fun` □

This theorem establishes the correctness condition for batch projection. If the input is well-typed and the relation $b \rightsquigarrow_{\Gamma(-)} b', f$ holds, then $b'$ has the expected type ($\Delta$), and the substitution function takes well-typed batches to well-typed batches.

Theorem B.41 (Batch Projection Correctness). *If $b \rightsquigarrow_{\Gamma(-)} b', f$, then for all $\Delta$, if $b$ : batch $(\Gamma($ we have that $b'$ : batch $(\Delta)$ and $f(b') = b$. Moreover, for any $\Delta'$ and $b''$ such that $b''$ : batch $(\Delta')$, we have $f(b'')$ : batch $(\Gamma(\Delta'))$.*

Proof. `pproj.v:brpoj_correct` □

*Prefix Projection.* With prefixes, the projection story is a bit more complicated. If $p$ : prefix $(\Gamma(\Delta))$, exactly one of three possibilities can occur. First, $p$ can contain no $\Delta$-data, like in the case $\Gamma(-) = \cdot; -$, with $p = \mathtt{catpA}(\mathtt{epsp})$. Next, $p$ may contain some prefix of $\Delta$, like when $\Gamma(-) = -, \Gamma'$, and $p = (p_1, p_2)$. In this case, the prefix of $\Delta$ that $p$ contains is precisely $p_1$. Last, $p$ may contain an entire batch of $\Delta$, as in the case $\Gamma(-) = -; \cdot$, with $p = \mathtt{catpB}(b, \mathtt{epsp})$. In this case, the batch in question is exactly $b$.

Because of this trichotomy, prefix projection actually consists of three mutually disjoint relations, exactly one of which must hold when $p$ is well-typed. They are written as follows:

- $p@\Gamma(-) \rightsquigarrow \bot@\Gamma'(-)$ holds when $p$ contains none of $\Delta$
- $p@\Gamma(-) \rightsquigarrow p'@\Gamma'(-)@f$ holds when $p$ contains a prefix $p'$ of $\Delta$
- $p@\Gamma(-) \rightsquigarrow b@f$ holds when $p$ contains a batch $b$ of $\Delta$.

In some cases where we use prefix projection, we will also need to compute how the context $\Gamma(-)$ context changes when we take the derivative of $\Gamma(\Delta)$ with respect to the $p$. When $p \rightsquigarrow_{\Gamma(-)} \bot$, the derivative $\delta_p\ (\Gamma(\Delta))$ looks like $\Gamma'(\Delta)$: some of the surrounding context has been chipped away, but since $p$ contains none of $\Delta$, it remains untouched. When $p \rightsquigarrow_{\Gamma(-)} p'$, the derivative $\delta_p\ (\Gamma(\Delta))$ looks like $\Gamma'\left(\delta_{p'}\ (\Delta)\right)$. The outer context has changed, and $\Delta$ has been reduced by $p'$. Lastly, when $p \rightsquigarrow_{\Gamma(-)} b$, the derivative $\delta_p\ (\Gamma(\Delta))$ is a hole-free context $\Gamma'$: because $p$ contained a whole batch of $\Delta$, the derivative context has no $\Delta$ left. In the first two cases, the semantics will need these $\Gamma'(-)$s, and so we include computing them in the prefix projection relation.

These relations also include one feature that wasn't presented in the body of the paper: when some of $\Delta$ was found (a prefix or a batch), we also need a function for *substituting* a different prefix or a batch (respectively) back in – this is the $f$ in the $p@\Gamma(-) \rightsquigarrow p'@\Gamma'(-)@f$ and $p@\Gamma(-) \rightsquigarrow b@f$. For example, if $p = \mathtt{parp}(p_1, p_2)$ and $\Gamma(-) = -, \Gamma'$, then $p@\Gamma(-) \rightsquigarrow p_1@\left(-, \delta_{p_2}\ (\Gamma')\right)@f$, and for all $p'$, we have that $f(p') = \mathtt{parp}(p', p_2)$: the prefix $p'$ replaces the projected out prefix.

*Definition B.42 (Prefix Projection).*

For a well-typed $p$, exactly one of these three relations must hold.

Theorem B.43 (Prefix Projection Trichotomy). *For any prefix $p$ and $\Gamma(-)$, at most one of the following three things holds:*

(1) $p@\Gamma(-) \rightsquigarrow \bot@\Gamma'(-)$
(2) $p@\Gamma(-) \rightsquigarrow p'@\Gamma'(-)@f$, *where $f$ is a function from prefixes to prefixes.*
(3) $p@\Gamma(-) \rightsquigarrow b@f$, *where $f$ is a function from batches to prefixes.*

*Moreover, when $p$ : prefix $(\Gamma(\Delta))$, exactly one of three holds.*

Proof. Existence is `pproj.v:pproj_tricot`, and uniqueness is `pproj.v:pproj_det`. □

The following theorem establishes the correctness of empty prefix projection. If there is no $\Delta$ in $p$, then $p$ is actually a prefix of $\Gamma(\Delta')$ for any $\Delta'$ (because it contains only data for the surrounding $\Gamma(-)$), and all the derivatives line up.

$$\textsc{pproj-here}\ \frac{}{p@- \rightsquigarrow p@-@id} \qquad \textsc{pproj-,1-emp}\ \frac{p_1@\Gamma_1(-) \rightsquigarrow \bot@\Gamma_1'(-) \qquad \delta_{p_2}(\Gamma_2) \sim \Gamma_2'}{\mathsf{parp}(p_1,p_2)@\Gamma_1(-),\Gamma_2 \rightsquigarrow \bot@\Gamma_1'(-),\Gamma_2'}$$

$$\textsc{pproj-,1-pfx}\ \frac{p_1@\Gamma_1(-) \rightsquigarrow p'@\Gamma_1'(-)@f \qquad f'(p'') = \mathsf{parp}(f(p''),p_2) \qquad \delta_{p_2}(\Gamma_2) \sim \Gamma_2'}{\mathsf{parp}(p_1,p_2)@\Gamma_1(-),\Gamma_2 \rightsquigarrow p'@\Gamma_1'(-),\Gamma_2'@f'}$$

$$\textsc{pproj-,1-bat}\ \frac{p_1@\Gamma_1(-) \rightsquigarrow b@f \qquad f'(b') = \mathsf{parp}(f(b'),p_2)}{\mathsf{parp}(p_1,p_2)@\Gamma_1(-),\Gamma_2 \rightsquigarrow b@f'} \qquad \textsc{pproj-,2-emp}\ \frac{p_2@\Gamma_2(-) \rightsquigarrow \bot@\Gamma_2'(-) \qquad \delta_{p_1}(\Gamma_1) \sim \Gamma_1'}{\mathsf{parp}(p_1,p_2)@\Gamma_1,\Gamma_2(-) \rightsquigarrow \bot@\Gamma_1',\Gamma_2'(-)}$$

$$\textsc{pproj-,2-pfx}\ \frac{p_2@\Gamma_2(-) \rightsquigarrow p'@\Gamma_2'(-)@f \qquad f'(p'') = \mathsf{parp}(p_1,f(p'')) \qquad \delta_{p_1}(\Gamma_1) \sim \Gamma_1'}{\mathsf{parp}(p_1,p_2)@\Gamma_1,\Gamma_2(-) \rightsquigarrow p'@\Gamma_1',\Gamma_2'(-)@f'}$$

$$\textsc{pproj-,2-bat}\ \frac{p_2@\Gamma_2(-) \rightsquigarrow b@f \qquad f'(p'') = \mathsf{parp}(p_1,f(p''))}{\mathsf{parp}(p_1,p_2)@\Gamma_1(-),\Gamma_2 \rightsquigarrow b@f'} \qquad \textsc{pproj-;1-a-emp}\ \frac{p@\Gamma_1(-) \rightsquigarrow \bot@\Gamma_1'(-)}{\mathsf{catpA}(p)@\Gamma_1(-);\Gamma_2 \rightsquigarrow \bot@\Gamma_1'(-);\Gamma_2}$$

$$\textsc{pproj-;1-a-pfx}\ \frac{p@\Gamma_1(-) \rightsquigarrow p'@\Gamma_1'(-)@f \qquad f'(p'') = \mathsf{catpA}(f(p''))}{\mathsf{catpA}(p)@\Gamma_1(-);\Gamma_2 \rightsquigarrow p'@\Gamma_1'(-);\Gamma_2@f'}$$

$$\textsc{pproj-;1-a-bat}\ \frac{p@\Gamma_1(-) \rightsquigarrow b@f \qquad f'(b') = \mathsf{catpA}(f(b'))}{\mathsf{catpA}(p)@\Gamma_1(-);\Gamma_2 \rightsquigarrow b@f'}$$

$$\textsc{pproj-;1-b}\ \frac{p@\Gamma_1(-) \rightsquigarrow b'@f \qquad f'(b'') = \mathsf{catpB}(b,f(b'')) \qquad \delta_p(\Gamma_2) \sim \Gamma_2'}{\mathsf{catpB}(b,p)@\Gamma_1(-);\Gamma_2 \rightsquigarrow b'@f'}$$

$$\textsc{pproj-;2-a}\ \frac{\delta_p(\Gamma_1) \sim \Gamma_1'}{\mathsf{catpA}(p)@\Gamma_1;\Gamma_2(-) \rightsquigarrow \bot@\Gamma_1';\Gamma_2(-)} \qquad \textsc{pproj-;2-b-emp}\ \frac{p@\Gamma_2(-) \rightsquigarrow \bot@\Gamma_2'(-)}{\mathsf{catpB}(b,p)@\Gamma_1;\Gamma_2(-) \rightsquigarrow \bot@\Gamma_2'(-)}$$

$$\textsc{pproj-;2-b-pfx}\ \frac{p@\Gamma_2(-) \rightsquigarrow p'@\Gamma_2'(-)@f \qquad f'(p'') = \mathsf{catpB}(b,f(p''))}{\mathsf{catpB}(b,p)@\Gamma_1;\Gamma_2(-) \rightsquigarrow p'@\Gamma_2'(-)@f'}$$

$$\textsc{pproj-;2-b-pfx}\ \frac{p@\Gamma_2(-) \rightsquigarrow b'@\Gamma_2'@f \qquad f'(b'') = \mathsf{catpB}(b,f(b''))}{\mathsf{catpB}(b,p)@\Gamma_1;\Gamma_2(-) \rightsquigarrow b'@\Gamma_2'@f'}$$

Fig. 8. Prefix Projection Rules

Theorem B.44 (Prefix Projection Empty Coherence). *If* $p : \mathtt{prefix}(\Gamma(\Delta))$ *and* $p@\Gamma(-) \rightsquigarrow \bot@\Gamma'(-)$ *then for all* $\Delta'$*, we have* $p : \mathtt{prefix}(\Gamma(\Delta'))$ *and* $\delta_p(\Gamma(\Delta')) = \Gamma'(\Delta')$

Proof. `pproj.v:pproj_emp` □

The following theorem says that if we get a prefix $p'$ out of prefix projection, then (1) it actually is a prefix of $\Delta$, (2) substituting it back into $p$ just gives $p$ again, and (3) for any prefix $p''$ of a context $\Delta'$, substituting it into $p$ (i.e. $f(p'')$) gives a prefix of $\Gamma(\Delta')$ such that $\delta_{f(p'')}(\Gamma(\Delta'))$ is of the form $\Gamma'(\delta_{p''}(\Delta'))$, where $\Gamma'$ does not depend on $\Delta'$ or $p''$.

Theorem B.45 (Preifx Projection Prefix Coherence). *If* $p : \texttt{prefix}(\Gamma(\Delta))$ *and* $p@\Gamma(-) \rightsquigarrow p'@\Gamma'(-)@f$, *then*

(1) $p' : \texttt{prefix}(\Delta)$
(2) $f(p') = p$
(3) *For any* $\Delta'$ *and any* $p'' : \texttt{prefix}(\Delta')$, *we have* $f(p'') : \texttt{prefix}(\Gamma(\Delta'))$ *and* $\delta_{f(p'')}(\Gamma(\Delta')) = \Gamma'(\delta_{p''}(\Delta'))$

Proof. `pproj.v:pproj_pfx` □

The following theorem is basically the same story as Theorem B.45. If we look for the $\Delta$ bit of $p : \texttt{prefix}(\Gamma(\Delta))$ and get out a whole batch $b$, then (1) $b$ actually has type $\Delta$, (2) substituting $b$ back into $p$ just yields $p$ again, and (3) for all contexts $\Delta'$ and all batches $b'$ of $\Delta'$, substituting $b'$ into $p$ gives a prefix of $\Gamma(\Delta')$, and all the derivatives of such substitutions are equal to a fixed context (without a hole) $\Gamma'$, which is given by the relation.

Theorem B.46 (Prefix Projection Batch Coherence). *If* $p : \texttt{prefix}(\Gamma(\Delta))$ *and* $p@\Gamma(-) \rightsquigarrow b@f$, *then*

(1) $b : \texttt{batch}(\Delta)$
(2) $f(b) = p$
(3) *For any* $\Delta'$ *and any* $b' : \texttt{prefix}(\Delta')$, *we have* $f(b') : \texttt{prefix}(\Gamma(\Delta'))$

Proof. `pproj.v:pproj_bat` □

*Projection Concatenation Theorems.* This section includes a handful of theorems about how prefix and batch projection relate to prefix concatenation. Given a prefix $p : \texttt{prefix}(\Gamma(\Delta))$ and a prefix $p' : \texttt{prefix}(\delta_p(\Gamma(\Delta)))$, and prefix projections for $p$ and $p'$, how can we characterize the projection for the concatenation of $p$ with $p'$? This section answers all such questions. Note that if the prefix projection for $p$ is empty or a prefix and gives back the new context $\Gamma'(-)$, then $p'$ has type $\Gamma'(\dots)$, and so its projection happens in context $\Gamma'(-)$. All of the following theorems are required to prove subcases of the homomorphism theorem.

If a prefix $p$ contains none of $\Delta$ and is followed by a batch $b$, then the resulting batch $p \cdot b$ has exactly the same batch projection behavior as $b$.

Theorem B.47 (Batch Projection Empty Concatenation). *Suppose the following:*

(1) $p : \texttt{prefix}(\Gamma(\Delta))$
(2) $p@\Gamma(-) \rightsquigarrow \bot@\Gamma'(-)$
(3) $b : \texttt{batch}(\Gamma'(\Delta))$
(4) $b \rightsquigarrow_{\Gamma'(-)} b', f$

*Then, there is an* $f$ *such that:*

(1) $p \cdot_{\Gamma(\Delta)} b \rightsquigarrow_{\Gamma(-)} b', f'$
(2) *For any* $b''$ *and* $\Delta'$ *with* $b'' : \texttt{batch}(\Delta')$, *we have* $f'(b'') = p \cdot_{\Gamma(\Delta')} f(b'')$.

Proof. `pproj.v:bproj_hom_emp` □

If a prefix $p$ contains a prefix of $\Delta$ and is followed by a batch $b$, then the batch projection of $p \cdot b$ is the concatenation of the projections of $p$ and $b$, respectively, and the substitution functions also line up correctly.

Theorem B.48 (Batch Projection Prefix Concatenation). *Suppose the following:*

(1) $p : \texttt{prefix}(\Gamma(\Delta))$
(2) $p@\Gamma(-) \rightsquigarrow p_d@\Gamma'(-)@f$

(3) $b : \mathtt{batch}\left(\Gamma'\left(\delta_{p_d}(\Delta)\right)\right)$
(4) $b \rightsquigarrow_{\Gamma'(-)} b_d, f'$

*Then, there is some $f''$ such that*

(1) $p \cdot_{\Gamma(\Delta)} b \rightsquigarrow_{\Gamma(-)} p_d \cdot_{\Delta} b_s, f''$
(2) *For any $p''$ $b''$ and $\Delta'$ with $p'' : \mathtt{prefix}(\Delta')\ b'' : \mathtt{batch}\left(\delta_{p''}(\Delta')\right)$, we have $f''(b'') = f(p'') \cdot_{\Gamma(\Delta')} f'(b'')$.*

Proof. `pproj.v:bproj_hom_pfx` □

If a prefix $p$ contains a batch of $\Delta$ and is followed by a batch $b$, then the batch projection of $p \cdot b$ is just the original batch of $\Delta$.

Theorem B.49 (Batch Projection Batch Concatenation). *Suppose the following:*

(1) $p : \mathtt{prefix}(\Gamma(\Delta))$
(2) $p@\Gamma(-) \rightsquigarrow b_d@f$
(3) $b : \mathtt{batch}\left(\delta_p(\Gamma(\Delta))\right)$

*Then, there is some $f'$ such that*

(1) $p \cdot_{\Gamma(\Delta)} b \rightsquigarrow_{\Gamma(-)} b_d, f'$
(2) *For any $p'$ $b'$ and $\Delta'$ with $p' : \mathtt{prefix}(\Delta')\ b' : \mathtt{batch}\left(\delta_p(\Gamma(\Delta'))\right)$, we have $f'(b') = f(p') \cdot_{\Gamma(\Delta')} b'$.*

Proof. `pproj.v:bproj_hom_bat` □

If two subsequent prefixes $p$ and $p'$ both have empty projections, then their concatenation does as well.

Theorem B.50 (Prefix Projection Empty Empty Concatenation). *Suppose the following*

(1) $p : \mathtt{prefix}(\Gamma(\Delta))$
(2) $p@\Gamma(-) \rightsquigarrow \bot@\Gamma'(-)$
(3) $p' : \mathtt{prefix}(\Gamma'(\Delta'))$
(4) $p'@\Gamma'(-) \rightsquigarrow \bot@\Gamma''(-)$

*Then, for any $\Delta''$, we have $p \cdot_{\Gamma(\Delta'')} p'@\Gamma(-) \rightsquigarrow \bot@\Gamma''(-)$.*

Proof. `pproj.v:pproj_hom_emp_emp` □

If subsequent prefixes $p$ and $p'$ are such that $p$ has an empty projection and $p'$ has a prefix projection, then the projection of $p \cdot p'$ is that of $p'$.

Theorem B.51 (Prefix Projection Empty Prefix Concatenation). *Suppose the following:*

(1) $p : \mathtt{prefix}(\Gamma(\Delta))$
(2) $p@\Gamma(-) \rightsquigarrow \bot@\Gamma'(-)$
(3) $p' : \mathtt{prefix}(\Gamma'(\Delta))$
(4) $p'@\Gamma'(-) \rightsquigarrow p_d@\Gamma''(-)@f$

*Then, there exists $f'$ such that*

(1) $p \cdot_{\Gamma(\Delta)} p'@\Gamma(-) \rightsquigarrow p_d@\Gamma''(-)@f'$
(2) *For any $\Delta'$ and $p''$ such that $p'' : \mathtt{prefix}(\Delta')$, we have $f'(p'') = p \cdot_{\Gamma(\Delta')} f(p'')$*

Proof. `pproj.v:pproj_hom_emp_pfx` □

If subsequent prefixes $p$ and $p'$ are such that $p$ has an empty projection and $p'$ has a batch projection, then the projection of $p \cdot p'$ is again that of $p'$.

Theorem B.52 (Prefix Projection Empty Batch Concatenation). *Suppose the following:*

(1) $p : \texttt{prefix}(\Gamma(\Delta))$
(2) $p@\Gamma(-) \rightsquigarrow \bot@\Gamma'(-)$
(3) $p' : \texttt{prefix}(\Gamma'(\Delta))$
(4) $p'@\Gamma'(-) \rightsquigarrow b_d@f$

*Then, there exists $f'$ such that*

(1) $p \cdot_{\Gamma(\Delta)} p'@\Gamma(-) \rightsquigarrow b_d@f'$
(2) *For any $\Delta'$ and $b$ such that $b : \texttt{batch}(\Delta')$, we have $f'(b) = p \cdot_{\Gamma(\Delta')} f(b)$*

Proof. `pproj.v:pproj_hom_emp_bat` □

It is impossible for subsequent prefixes $p$ and $p'$ to be such that $p$ has a prefix projection and $p'$ has an empty projection. Once a prefix has reached the hole part of the context, all subsequent prefixes have projection which is at least a prefix, if not a batch.

Theorem B.53 (Prefix Projection Prefix Not Empty Concatenation). *Suppose the following:*

(1) $p : \texttt{prefix}(\Gamma(\Delta))$
(2) $p@\Gamma(-) \rightsquigarrow p_d@\Gamma'(-)@$
(3) $p' : \texttt{prefix}\left(\Gamma'\left(\delta_{p_d}(\Delta)\right)\right)$

*Then, it is never the case that $p'@\Gamma'(-) \rightsquigarrow \bot@\Gamma''(-)$, for any $\Gamma''(-)$*

Proof. `pproj.v:pproj_hom_pfx_not_emp` □

If subsequent prefixes $p$ and $p'$ both have prefix projections, then then the projection of $p \cdot p'$ is the concatenation of those of $p$ and $p'$. The substitution function for $p \cdot p'$ is just the pointwise concatenation of the substitution functions for $p$ and $p'$.

Theorem B.54 (Prefix Projection Prefix Prefix Concatenation). *Suppose the following:*

(1) $p : \texttt{prefix}(\Gamma(\Delta))$
(2) $p@\Gamma(-) \rightsquigarrow p_d@\Gamma'(-)@f$
(3) $p' : \texttt{prefix}\left(\Gamma'\left(\delta_{p_d}(\Delta)\right)\right)$
(4) $p'@\Gamma'(-) \rightsquigarrow p'_d@\Gamma''(-)@f'$

*Then, there exists $f''$ such that:*

(1) $p \cdot_{\Gamma(\Delta)} p'@\Gamma(-) \rightsquigarrow p_d \cdot_\Delta p'_d@\Gamma''(-)@f'$
(2) *For any $p_0$, $p'_0$ and $\Delta_0$, if $p_0 : \texttt{prefix}(\Delta_0)$ and $p'_0 : \texttt{prefix}\left(\delta_{p_0}(\Delta_0)\right)$, then $f''(p_0 \cdot_{\Delta_0} p'_0) = f(p_0) \cdot_{\Gamma(\Delta_0)} f'(p'_0)$.*

Proof. `pproj.v:pproj_hom_pfx_pfx` □

Theorem B.55 (Prefix Projection Prefix Batch Concatenation). *Suppose the following:*

(1) $p : \texttt{prefix}(\Gamma(\Delta))$
(2) $p@\Gamma(-) \rightsquigarrow p_d@\Gamma'(-)@f$
(3) $p' : \texttt{prefix}\left(\Gamma'\left(\delta_{p_d}(\Delta)\right)\right)$
(4) $p'@\Gamma'(-) \rightsquigarrow b_d@f'$

*Then, there exists $f''$ such that:*

(1) $p \cdot_{\Gamma(\Delta)} p'@\Gamma(-) \rightsquigarrow p_d \cdot_\Delta b_d@f'$
(2) *For any $p_0$, $b_0$ and $\Delta_0$, if $p_0 : \texttt{prefix}(\Delta_0)$ and $b_0 : \texttt{batch}\left(\delta_{p_0}(\Delta_0)\right)$, then $f''(p_0 \cdot_{\Delta_0} b_0) = f(p_0) \cdot_{\Gamma(\Delta_0)} f'(b_0)$.*

Proof. `pproj.v:pproj_hom_pfx_bat` □

Theorem B.56 (Prefix Projection Batch Concatenation). *Suppose the following:*

(1) $p$ : $\texttt{prefix}\,(\Gamma(\Delta))$
(2) $p@\Gamma(-) \rightsquigarrow b_d@f$
(3) $p'$ : $\texttt{prefix}\,\big(\delta_p\,(\Gamma(\Delta))\big)$

*Then, there exists $f'$ such that:*

(1) $pprojBatp\ {}_{\cdot\Gamma(\Delta)}\ p'\Gamma(-)b_d\delta_{p'}\ \big(\delta_p\,(\Gamma(\Delta))\big)f'$
(2) *For all $b'$ and $\Delta'$ such that $b'$ : $\texttt{batch}\,(\Delta')$, we have that $f'(b') = f(b')\ {}_{\cdot\Gamma(\Delta')}\ p'$.*

Proof. `pproj.v:pproj_hom_bat` □

### B.6 Historical Contexts

*Definition B.57 (Historical Context).* Contexts $\Omega := \cdot \mid \Omega, x : A$ are fully structural contexts, where the $A$ are STLC types.

A stream type is "flattened" into an STLC type by turning concatenations and parallels into products, and stars into lists.

*Definition B.58 (Type and Context Flatten).* For $s$ a stream type, we define its flattening into an STLC type, denoted $\langle s \rangle$, inductively:

- $\langle 1 \rangle = 1$
- $\langle \varepsilon \rangle = 1$
- $\langle s \cdot t \rangle = \langle s \rangle \times \langle t \rangle$
- $\langle s \| t \rangle = \langle s \rangle \times \langle t \rangle$
- $\langle s + t \rangle = \langle s \rangle + \langle t \rangle$
- $\langle s^\star \rangle = \texttt{list}\,(\langle s \rangle)$

For $\Gamma$ a bunched context, we define its flattening to a standard context, $\langle \Gamma \rangle$ inductively:

- $\langle \cdot \rangle = \cdot$
- $\langle x : s \rangle = x : \langle s \rangle$
- $\langle \Gamma; \Gamma' \rangle = \langle \Gamma \rangle, \langle \Gamma' \rangle$
- $\langle \Gamma, \Gamma' \rangle = \langle \Gamma \rangle, \langle \Gamma' \rangle$

For an STLC value $v : \langle s \rangle$, we write $\texttt{toBatch}_s\,(v)$ for the batch of type $s$ that it corresponds to. Dually, for $b$ : $\texttt{batch}\,(s)$, we write $\langle b \rangle$ for the STLC value of type $\langle s \rangle$ it corresponds to.

*Definition B.59 (Historical Programs and Substitutions).* Fix a language of terms $M$, with type system $\Omega \vdash M : A$. Write its semantics as $M \downarrow v$. We assume that this relation is a decidable partial function, in the sense that $M$ evaluates to at most one $v$, and it is decidable whether or not such a $v$ exists. We write substitutions $\theta : \Omega' \to \Omega$. Substitutions have a contravariant action on terms, written $M[\theta]$: if $\Omega \vdash M : A$, then $\Omega' \vdash M[\theta] : A$.

### B.7 Context Subtyping

The following is a full listing of subtyping rules. This includes:

- The commutative monoid equations for comma contexts
- The non-commutative monoid equations for semicolon contexts.
- Weakenings for comma and semicolon. The semicolon weakenings need to record the context *not* being weakened away in the term for semantic reasons: transporting along these subtyping relations will require emitting empty prefixes, which require the context to construct.

- A rule SubtyNullable(Δ) which says that every context is a subtype of a context Δ if Δ is nullable. This is not intended to be used in terms (or needed in any we have written), but is needed to make the subtyping derivative operation work.
- A rule SubtyWknAll which is a global weakening: every context is a subtype of the empty context.
- A rule SubtyDistr which says that semicolon distributes over comma: $\Gamma;\ (\Delta_1,\ \Delta_2) \leq (\Gamma;\ \Delta_1)\ ,\ (\Gamma;\ \Delta_2)$ by copying the $\Gamma$ to both parallel components, and then forwarding the $\Delta$ components. A program that has the same batch behavior as this is implementable without the structural rule (using wait), but it is less incremental, instead waiting until the entire $\Gamma$ arrives, and then sending $\Gamma$ all at once and forwarding the $\Delta$s.
- A congruence rule, allowing subtyping rules to be applied in a subcontext.

*Definition B.60 (Context Subtyping).*

$$\frac{pf : \Delta \leq \Delta'}{\texttt{SubtyCong}(\Gamma(-), pf) : \Gamma(\Delta) \leq \Gamma(\Delta')} \qquad \frac{}{\texttt{SubtyUnitSemicL} : \Gamma \leq \cdot;\ \Gamma}$$

$$\frac{}{\texttt{SubtyUnitSemicR} : \Gamma \leq \Gamma;\ \cdot} \qquad \frac{}{\texttt{SubtyUnitCommaL} : \Gamma \leq \cdot,\ \Gamma}$$

$$\frac{}{\texttt{SubtyUnitCommaR} : \Gamma \leq \Gamma,\ \cdot} \qquad \frac{}{\texttt{SubtyCommaContr} : \Gamma \leq \Gamma,\ \Gamma}$$

$$\frac{}{\texttt{SubtyCommaComm} : \Gamma_1, \Gamma_2 \leq \Gamma_2, \Gamma_1} \qquad \frac{}{\texttt{SubtyCommaWkn1} : \Gamma,\ \Gamma' \leq \Gamma}$$

$$\frac{}{\texttt{SubtyCommaWkn2} : \Gamma',\ \Gamma \leq \Gamma} \qquad \frac{}{\texttt{SubtySemicWkn1}(\Gamma) : \Gamma;\ \Gamma' \leq \Gamma}$$

$$\frac{}{\texttt{SubtySemicWkn2}(\Gamma) : \Gamma';\ \Gamma \leq \Gamma} \qquad \frac{}{\texttt{SubtyWknAll} : \Gamma \leq \cdot} \qquad \frac{\Gamma\ \texttt{nullable}}{\texttt{SubtyNullable}(\Gamma) : \Gamma' \leq \Gamma}$$

$$\frac{}{\texttt{SubtySemicAssoc1} : (\Gamma_1;\ \Gamma_2)\ ;\ \Gamma_3 \leq \Gamma_1;\ (\Gamma_2;\ \Gamma_3)} \qquad \frac{}{\texttt{SubtySemicAssoc2} : \Gamma_1;\ (\Gamma_2;\ \Gamma_3) \leq (\Gamma_1;\ \Gamma_2)\ ;\ \Gamma_3}$$

$$\frac{}{\texttt{SubtyCommaAssoc1} : (\Gamma_1,\ \Gamma_2)\ ,\ \Gamma_3 \leq \Gamma_1,\ (\Gamma_2,\ \Gamma_3)} \qquad \frac{}{\texttt{SubtyCommaAssoc2} : \Gamma_1,\ (\Gamma_2,\ \Gamma_3) \leq (\Gamma_1,\ \Gamma_2)\ ,\ \Gamma_3}$$

$$\frac{}{\texttt{SubtyDistr} : \Gamma;\ \Delta,\ \Delta' \leq \Gamma;\ \Delta,\ \Gamma;\ \Delta'}$$

Subtyping is deterministic in the sense that a choice of a subtyping rule and a left context $\Gamma$ determine the context $\Delta$ for which $pf : \Gamma \leq \Delta$.

Theorem B.61 (Subtyping Proof and Left Context Determine Right Context). *If* $pf : \Gamma \leq \Delta$ *and* $pf : \Gamma \leq \Delta'$, *then* $\Delta = \Delta'$,.

Proof. ctxsubty.v:tckCtxSubyPf_det □

Subtyping relations can be interpreted as functions which cast batches of the left context to those of the right context. In other words, a subtyping $pf : \Gamma \leq \Delta$ should denote a function $\texttt{batch}(\Gamma) \to \texttt{batch}(\Delta)$. We operationalize this with the following definition: the judgment $\texttt{coeBatch}\,(pf, b) \sim b'$ means that the subtyping proof term turns $b$ (of type $\Gamma$) into $b'$ (of type $\Delta$).

*Definition B.62 (Context Subtyping Batch Transport).*

$$\frac{b \rightsquigarrow_{\Gamma(-)} b', f \qquad \mathsf{coeBatch}\,(pf, b') \sim b''}{\mathsf{coeBatch}\,(\mathsf{SubtyCong}(\Gamma(-), pf), b) \sim f(b'')}$$

$$\frac{}{\mathsf{coeBatch}\,(\mathsf{SubtyUnitSemicL}, \mathsf{catb}\,(\mathsf{epsb}, b)) \sim b} \quad \frac{}{\mathsf{coeBatch}\,(\mathsf{SubtyUnitSemicR}, b) \sim \mathsf{catb}\,(b, \mathsf{epsb}}$$

$$\frac{}{\mathsf{coeBatch}\,(\mathsf{SubtyUnitCommaL}, \mathsf{parb}\,(\mathsf{epsb}, b)) \sim b} \quad \frac{}{\mathsf{coeBatch}\,(\mathsf{SubtyUnitCommaR}, \mathsf{parb}\,(b, \mathsf{epsb})) \sim}$$

$$\frac{}{\mathsf{coeBatch}\,(\mathsf{SubtyCommaContr}, b) \sim \mathsf{parb}\,(b, b)} \quad \frac{}{\mathsf{coeBatch}\,(\mathsf{SubtyCommaComm}, \mathsf{parb}\,(b_1, b_2)) \sim \mathsf{parb}\,(}$$

$$\frac{}{\mathsf{coeBatch}\,(\mathsf{SubtyCommaWkn1}, \mathsf{parb}\,(b, b')) \sim b'} \quad \frac{}{\mathsf{coeBatch}\,(\mathsf{SubtyCommaWkn2}, \mathsf{parb}\,(b, b')) \sim b}$$

$$\frac{}{\mathsf{coeBatch}\,(\mathsf{SubtySemicWkn1}(\_), \mathsf{catb}\,(b, b')) \sim b'} \quad \frac{}{\mathsf{coeBatch}\,(\mathsf{SubtySemicWkn2}(\_), \mathsf{catb}\,(b, b')) \sim b}$$

$$\frac{\dfrac{}{\mathsf{coeBatch}\,(\mathsf{SubtyWknAll}, b) \sim \mathsf{epsb}} \quad \Gamma\ \mathsf{done}\ b'}{\mathsf{coeBatch}\,(\mathsf{SubtyNullable}(\Gamma), b) \sim b'}$$

$$\frac{}{\mathsf{coeBatch}\,(\mathsf{SubtySemicAssoc1}, \mathsf{catb}\,(\mathsf{catb}\,(b_1, b_2), b_3)) \sim \mathsf{catb}\,(b_1, \mathsf{catb}\,(b_2, b_3))}$$

$$\frac{}{\mathsf{coeBatch}\,(\mathsf{SubtySemicAssoc2}, \mathsf{catb}\,(b_1, \mathsf{catb}\,(b_2, b_3))) \sim \mathsf{catb}\,(\mathsf{catb}\,(b_1, b_2), b_3)}$$

$$\frac{}{\mathsf{coeBatch}\,(\mathsf{SubtyCommaAssoc1}, \mathsf{parb}\,(\mathsf{parb}\,(b_1, b_2), b_3)) \sim \mathsf{parb}\,(b_1, \mathsf{parb}\,(b_2, b_3))}$$

$$\frac{}{\mathsf{coeBatch}\,(\mathsf{SubtyCommaAssoc2}, \mathsf{parb}\,(b_1, \mathsf{parb}\,(b_2, b_3))) \sim \mathsf{parb}\,(\mathsf{parb}\,(b_1, b_2), b_3)}$$

$$\frac{}{\mathsf{coeBatch}\,(\mathsf{SubtyDistr}, \mathsf{catb}\,(b, \mathsf{parb}\,(b_1, b_2))) \sim \mathsf{parb}\,(\mathsf{catb}\,(b, b_1), \mathsf{catb}\,(b, b_2))}$$

Naturally, this relation is functional: when the inputs are well-typed, there is a unique well-typed output.

Theorem B.63 (Context Subtyping Batch Transport Function). *For any $pf, b$, there is at most one $b'$ such that* $\mathsf{coeBatch}\,(pf, b) \sim b'$. *If* $b : \mathsf{batch}\,(\Gamma)$ *and* $pf : \Gamma \leq \Gamma'$, *such a $b'$ must exist, and it satisfies* $b' : \mathsf{batch}\,(\Gamma')$.

Proof. Existence by `ctxsuby.v:ctxSubtyPfBatchTransport_fun`,
uniqueness by `ctxsubty.v:ctxSubtyPfBatchTransport_det`,
well-typedness of $b'$ by `ctxsubty.v:ctxSubtyPfBatchTransport_correct`. □

*Definition B.64 (Context Subtyping Prefix Transport).*

$$\frac{p@\Gamma(-) \rightsquigarrow \bot@\Gamma'(-)}{\mathsf{coePfx}\,(\mathsf{SubtyCong}(\Gamma(-), pf), p) \sim p}$$

$$\frac{p@\Gamma(-) \rightsquigarrow p'@\Gamma'(-)@f \qquad \mathsf{coePfx}\,(pf, p') \sim p''}{\mathsf{coePfx}\,(\mathsf{SubtyCong}(\Gamma(-), pf), p) \sim f(p'')}$$

$$\frac{p@\Gamma(-) \rightsquigarrow b@f \qquad \mathsf{coeBatch}\,(pf, b) \sim b'}{\mathsf{coePfx}\,(\mathsf{SubtyCong}(\Gamma(-), pf), p) \sim f(b')}$$

$$\frac{}{\mathsf{coePfx}\,(\mathsf{SubtyUnitSemicL}, p) \sim \mathsf{catpB}(\mathsf{epsb}, p)}$$

$$\frac{}{\mathsf{coePfx}\,(\mathsf{SubtyUnitSemicR}, p) \sim \mathsf{catpA}(p)}$$

$$\frac{}{\mathsf{coePfx}\,(\mathsf{SubtyUnitCommaL}, p) \sim \mathsf{parp}(\mathsf{epsp}, p)}$$

$$\frac{}{\mathsf{coePfx}\,(\mathsf{SubtyUnitCommaR}, p) \sim \mathsf{parp}(p, \mathsf{epsp})}$$

$$\frac{}{\mathsf{coePfx}\,(\mathsf{SubtyCommaContr}, p) \sim \mathsf{parp}(p, p)}$$

$$\frac{}{\mathsf{coePfx}\,(\mathsf{SubtyCommaComm}, \mathsf{parp}(p, p')) \sim \mathsf{parp}(p, p')}$$

$$\frac{}{\mathsf{coePfx}\,(\mathsf{SubtyCommaWkn1}, \mathsf{parp}(p, p')) \sim p'}$$

$$\frac{}{\mathsf{coePfx}\,(\mathsf{SubtyCommaWkn2}, \mathsf{parp}(p, p')) \sim p}$$

$$\frac{}{\mathsf{coePfx}\,(\mathsf{SubtySemicWkn1}(\Gamma), \mathsf{catpA}(p)) \sim \mathsf{emp}_\Gamma}$$

$$\frac{}{\mathsf{coePfx}\,(\mathsf{SubtySemicWkn1}(\_), \mathsf{catpB}(b, p)) \sim p}$$

$$\frac{}{\mathsf{coePfx}\,(\mathsf{SubtySemicWkn2}(\_), \mathsf{catpA}(p)) \sim p}$$

$$\frac{}{\mathsf{coePfx}\,(\mathsf{SubtySemicWkn2}(\_), \mathsf{catpB}(b, p)) \sim (b)^\circ}$$

$$\frac{}{\mathsf{coePfx}\,(\mathsf{SubtyWknAll}, p) \sim \mathsf{epsp}}$$

$$\frac{}{\mathsf{coePfx}\,(\mathsf{SubtyNullable}(\Gamma), p) \sim \mathsf{emp}_\Gamma}$$

$$\frac{}{\mathsf{coePfx}\,(\mathsf{SubtySemicAssoc1}, \mathsf{catpA}(\mathsf{catpA}(p))) \sim \mathsf{catpA}(p)}$$

$$\frac{}{\mathsf{coePfx}\,(\mathsf{SubtySemicAssoc1}, \mathsf{catpA}(\mathsf{catpB}(b, p))) \sim \mathsf{catpB}(b, \mathsf{catpA}(p))}$$

$$\frac{}{\mathsf{coePfx}\,(\mathsf{SubtySemicAssoc1}, \mathsf{catpB}(\mathsf{catb}\,(b, b')\,, p)) \sim \mathsf{catpB}(b, \mathsf{catpB}(b', p))}$$

$$\frac{}{\mathsf{coePfx}\,(\mathsf{SubtySemicAssoc2}, \mathsf{catpA}(p)) \sim \mathsf{catpA}(\mathsf{catpA}(p))}$$

$$\frac{}{\mathsf{coePfx}\,(\mathsf{SubtySemicAssoc2}, \mathsf{catpB}(b, \mathsf{catpA}(p))) \sim \mathsf{catpA}(\mathsf{catpB}(b, p))}$$

$$\frac{}{\mathsf{coePfx}\,(\mathsf{SubtySemicAssoc2}, \mathsf{catpB}(b, \mathsf{catpB}(b', p))) \sim \mathsf{catpB}(\mathsf{catb}\,(b, b')\,, p)}$$

$$\frac{}{\mathsf{coePfx}\,(\mathsf{SubtyCommaAssoc1}, \mathsf{parp}(\mathsf{parp}(p_1, p_2), p_3)) \sim \mathsf{parp}(p_1, \mathsf{parp}(p_2, p_3))}$$

$$\frac{}{\mathsf{coePfx}\,(\mathsf{SubtyCommaAssoc2}, \mathsf{parp}(p_1, \mathsf{parp}(p_2, p_3))) \sim \mathsf{parp}(\mathsf{parp}(p_1, p_2), p_3)}$$

$$\frac{}{\mathsf{coePfx}\,(\mathsf{SubtyDistr}, \mathsf{catpA}(p)) \sim \mathsf{parp}(\mathsf{catpA}(p), \mathsf{catpA}(p))}$$

$$\frac{}{\mathsf{coePfx}\,(\mathsf{SubtyDistr}, \mathsf{catpB}(b, \mathsf{parp}(p_1, p_2))) \sim \mathsf{parp}(\mathsf{catpB}(b, p_1), \mathsf{catpB}(b, p_2))}$$

Theorem B.65 (Context Subtyping Prefix Transport Function). *For any $pf$ and $p$, there is at most one $p'$ such that* $\texttt{coePfx}\,(pf, p) \sim p'$. *If* $p : \texttt{prefix}\,(\Gamma)$ *and* $pf : \Gamma \leq \Gamma'$, *then such a $p'$ exists, and* $p' : \texttt{prefix}\,(\Gamma')$

Proof. Existence by `ctxsubty.v:ctxSubtyPfPrefixTransport_fun`,
uniqueness by `ctxsubty.v:ctxSubtyPfPrefixTransport_det`.
well-typedness of $p'$ by `ctxsubty.v:ctxSubtyPfPrefixTransport_correct` □

The proof terms themselves also need to update with prefixes: if $p : \texttt{prefix}\,(\Gamma)$ and $pf : \Gamma \leq \Gamma'$, then we should be able to find $\delta_p\,(pf)$ so that $\delta_p\,(pf) : \delta_p\,(\Gamma) \leq \delta_{\texttt{coepfx}\,(pf,p)}\Gamma'$. However, it is sometimes the case that $\delta_p\,(\Gamma) = \delta_{\texttt{coepfx}\,(pf,p)}\Gamma'$, in which case the subtyping is no longer needed.

To this end, we define a pair of relations, written $\delta_p\,(pf) \sim \bot$ and $\delta_p\,(pf) \sim pf'$. The first defines the situation where after $p$, the contexts that $pf$ relates agree and the subtyping is no longer needed. The second gives $pf'$, the updated proof term relating the derivative contexts, when necessary. We call these two relations the "none" and "some" subtyping proof term derivative relations because of the use of an option type in the Coq formalism.

*Definition B.66 (Context Subtyping Proof Term Derivatives).*

$$\textsc{cspd-cong-emp}\ \frac{p@\Gamma(-) \rightsquigarrow \bot@\Gamma'(-)}{\delta_p\,(\mathsf{SubtyCong}(\Gamma(-), pf)) \sim \mathsf{SubtyCong}(\Gamma'(-), pf)}$$

$$\textsc{cspd-cong-pfx0}\ \frac{p@\Gamma(-) \rightsquigarrow p'@\Gamma'(-)@f \qquad \delta_{p'}\,(pf) \sim \bot}{\delta_p\,(\mathsf{SubtyCong}(\Gamma(-), pf)) \sim \bot}$$

$$\textsc{cspd-cong-pfx1}\ \frac{p@\Gamma(-) \rightsquigarrow p'@\Gamma'(-)@f \qquad \delta_{p'}\,(pf) \sim pf'}{\delta_p\,(\mathsf{SubtyCong}(\Gamma(-), pf)) \sim \mathsf{SubtyCong}(\Gamma'(-), pf')} \qquad \textsc{cspd-cong-bat}\ \frac{p@\Gamma(-) \rightsquigarrow b@f}{\delta_p\,(\mathsf{SubtyCong}(\Gamma(-), pf)) \sim \bot}$$

$$\textsc{cspd-unitsemicl}\ \frac{}{\delta_p\,(\mathsf{SubtyUnitSemicL}) \sim \bot} \qquad \textsc{cspd-unitsemicr}\ \frac{}{\delta_p\,(\mathsf{SubtyUnitSemicR}) \sim \mathsf{SubtyUnitSemicR}}$$

$$\textsc{cspd-unitcommal}\ \frac{}{\delta_p\,(\mathsf{SubtyUnitCommaL}) \sim \mathsf{SubtyUnitCommaL}} \qquad \textsc{cspd-unitcommar}\ \frac{}{\delta_p\,(\mathsf{SubtyUnitCommaR}) \sim \mathsf{SubtyUnitCommaR}}$$

$$\textsc{cspd-commacontr}\ \frac{}{\delta_p\,(\mathsf{SubtyCommaContr}) \sim \mathsf{SubtyCommaContr}} \qquad \textsc{cspd-commacomm}\ \frac{}{\delta_p\,(\mathsf{SubtyCommaComm}) \sim \mathsf{SubtyCommaComm}}$$

$$\textsc{cspd-commawknA}\ \frac{}{\delta_p\,(\mathsf{SubtyCommaWkn1}) \sim \mathsf{SubtyCommaWkn1}} \qquad \textsc{cspd-commawknB}\ \frac{}{\delta_p\,(\mathsf{SubtyCommaWkn2}) \sim \mathsf{SubtyCommaWkn2}}$$

$$\textsc{cspd-semicwknA-A}\ \frac{}{\delta_{\mathsf{catpA}(p)}\,(\mathsf{SubtySemicWkn1}(\Delta)) \sim \mathsf{SubtySemicWkn1}(\Delta)} \qquad \textsc{cspd-semicwknB-A}\ \frac{}{\delta_{\mathsf{catpB}(b,p)}\,(\mathsf{SubtySemicWkn1}(\Delta)) \sim \bot}$$

$$\textsc{cspd-semicwknB-B}\ \frac{\delta_p\,(\Delta) \sim \Delta'}{\delta_{\mathsf{catpA}(p)}\,(\mathsf{SubtySemicWkn2}(\Delta)) \sim \mathsf{SubtySemicWkn2}(\Delta')}$$

$$\textsc{cspd-wknall}\ \frac{}{\delta_p\,(\mathsf{SubtyWknAll}) \sim \mathsf{SubtyWknAll}} \qquad \textsc{cspd-nullable}\ \frac{\delta_{(b)^\circ}\,(\Delta) \sim \Delta'}{\delta_{\mathsf{catpB}(b,p)}\,(\mathsf{SubtySemicWkn2}(\Delta)) \sim \mathsf{SubtyNullable}(\Delta')}$$

$$\textsc{cspd-semicassocA-A-A}\ \frac{}{\delta_p\,(\mathsf{SubtyNullable}(\Delta)) \sim \mathsf{SubtyNullable}(\Delta)}$$

$$\textsc{cspd-semicassocA-A-B}\ \frac{}{\delta_{\mathsf{catpA}(\mathsf{catpA}(p))}\,(\mathsf{SubtySemicAssoc1}) \sim \mathsf{SubtySemicAssoc1}} \qquad \textsc{cspd-semicassocA-B}\ \frac{}{\delta_{\mathsf{catpB}(b,p)}\,(\mathsf{SubtySemicAssoc1}) \sim \bot}$$

$$\textsc{cspd-semicassocB-A}\ \frac{}{\delta_{\mathsf{catpA}(\mathsf{catpB}(b,p))}\,(\mathsf{SubtySemicAssoc1}) \sim \bot}$$

$$\textsc{cspd-semicassocB-B-A}\ \frac{}{\delta_{\mathsf{catpA}(p)}\,(\mathsf{SubtySemicAssoc2}) \sim \mathsf{SubtySemicAssoc2}} \qquad \textsc{cspd-semicassocB-B-B}\ \frac{}{\delta_{\mathsf{catpB}(b,\mathsf{catpA}(p))}\,(\mathsf{SubtySemicAssoc2}) \sim \bot}$$

$$\textsc{cspd-commaassocA}\ \frac{}{\delta_{\mathsf{catpB}(b,\mathsf{catpB}(b',p))}\,(\mathsf{SubtySemicAssoc2}) \sim \bot}$$

$$\textsc{cspd-commaassocB}\ \frac{}{\delta_p\,(\mathsf{SubtyCommaAssoc1}) \sim \mathsf{SubtyCommaAssoc1}} \qquad \textsc{cspd-distr-A}\ \frac{}{\delta_p\,(\mathsf{SubtyCommaAssoc2}) \sim \mathsf{SubtyCommaAssoc2}}$$

$$\textsc{cspd-distr-B}\ \frac{}{\delta_{\mathsf{catpA}(p)}\,(\mathsf{SubtyDistr}) \sim \mathsf{SubtyDistr}} \qquad \textsc{cspd-distr-B}\ \frac{}{\delta_{\mathsf{catpB}(b,p)}\,(\mathsf{SubtyDistr}) \sim \bot}$$

Theorem B.67 (Context Subtyping Proof Derivative Function). *For any $p$ and $pf$, there is at most one $x$ (either some $pf'$ or $\bot$) such that $\delta_p\,(pf) \sim x$.*

Proof. Existence by `ctxsubty.v:ctxSubtyPfDeriv_fun`,
uniqueness by `ctxsubty.v:ctxSubtyPfDeriv_det`. □

Theorem B.68 (Context Subtyping Proof Derivative Correctness (Some)). *If*

(1) $p$ : $\mathtt{prefix}\,(\Gamma)$
(2) $pf : \Gamma \le \Gamma'$
(3) $\mathtt{coePfx}\,(pf, p) \sim p'$
(4) $\delta_p\,(pf) \sim pf'$

*Then $pf : \delta_p\,(\Gamma) \le \delta_{p'}\,(\Gamma')$.*

Proof. `ctxsubty.v:ctxSubtyPfDeriv_correct_some` □

Theorem B.69 (Context Subtyping Proof Derivative Correctness (None)). *If*

(1) $p$ : $\mathtt{prefix}\,(\Gamma)$
(2) $pf : \Gamma \le \Gamma'$
(3) $\mathtt{coePfx}\,(pf, p) \sim p'$
(4) $\delta_p\,(pf) \sim \bot$

*Then $\delta_p\,(\Gamma) = \delta_{p'}\,(\Gamma')$.*

Proof. `ctxsubty.v:ctxSubtyPfDeriv_correct_some` □

The following four theorems are required as lemmas for the homomorphism theorem. They describe what happens when you coerce two prefixes (or a prefix then a batch) along a subtyping relation one after the other. In short, it's equivalent to coercing the concatenated prefix (or batch). The exact theorem statements depend on the derivative of the proof term $pf$ with respect to the first prefix $p$. If the result is none, the second batch or prefix does not need to be coerced, as it already has the right type on both sides of the subtyping. If the result is some $pf'$,the second batch or prefix gets transported across it.

Theorem B.70 (Prefix and Batch Transport Concatenation (None)). *Suppose the following:*

(1) $pf : \Gamma \le \Delta$
(2) $p$ : $\mathtt{prefix}\,(\Gamma)$
(3) $b$ : $\mathtt{batch}\,(\delta_p\,(\Gamma))$
(4) $\mathtt{coePfx}\,(pf, p) \sim p'$
(5) $\delta_p\,(pf) \sim \bot$

*Then,* $\mathtt{coeBatch}\,(pf, p \cdot_\Gamma b) \sim p' \cdot_\Delta b$

Proof. `ctxsubty.v:ctxSubtyPfBatchTransport_hom_none` □

Theorem B.71 (Prefix and Batch Transport Concatenation (Some)). *Suppose the following:*

(1) $pf : \Gamma \le \Delta$
(2) $p$ : $\mathtt{prefix}\,(\Gamma)$
(3) $b$ : $\mathtt{batch}\,(\delta_p\,(\Gamma))$
(4) $\mathtt{coePfx}\,(pf, p) \sim p'$
(5) $\delta_p\,(pf) \sim pf'$
(6) $\mathtt{coeBatch}\,(pf', b) \sim b'$

*Then,* $\mathtt{coeBatch}\,(pf, p \cdot_\Gamma b) \sim p' \cdot_\Delta b'$

Proof. ctxsubty.v:ctxSubtyPfBatchTransport_hom_some □

Theorem B.72 (Prefix Transport Concatenation (None)). *Suppose:*

- $pf : \Gamma \leq \Delta$
- $p_1 : \texttt{prefix}(\Gamma)$
- $p_2 : \texttt{prefix}\left(\delta_{p_1}(\Gamma)\right)$
- $\texttt{coePfx}(pf, p_1) \sim p_1'$
- $\delta_{p_1}(pf) \sim \bot$

*Then,* $\texttt{coePfx}(pf, p_1 \cdot_\Gamma p_2) \sim p_1' \cdot_\Delta p_2$

Proof. ctxsubty.v:ctxSubyPfPrefixTransport_hom_none □

Theorem B.73 (Prefix Transport Concatenation (Some)). *Suppose:*

- $pf : \Gamma \leq \Delta$
- $p_1 : \texttt{prefix}(\Gamma)$
- $p_2 : \texttt{prefix}\left(\delta_{p_1}(\Gamma)\right)$
- $\texttt{coePfx}(pf, p_1) \sim p_1'$
- $\delta_{p_1}(pf) \sim pf'$
- $\texttt{coePfx}(pf', p_2) \sim p_2'$

*Then,* $\texttt{coePfx}(pf, p_1 \cdot_\Gamma p_2) \sim p_1' \cdot_\Delta p_2'$

Proof. ctxsubty.v:ctxSubyPfPrefixTransport_hom_some □

Theorem B.74 (Subtyping Proof Term Derivative Concatenation (None)). *Suppose:*

(1) $pf : \Gamma \leq \Delta$
(2) $p : \texttt{prefix}(\Gamma)$
(3) $p' : \texttt{prefix}\left(\delta_p(\Gamma)\right)$
(4) $\delta_p(pf) \sim \bot$

*Then,* $\delta_{p \cdot_\Gamma p'}(pf) \sim \bot$

Proof. ctxsubty.v:ctxSubyPfDeriv_hom_none □

Theorem B.75 (Subtyping Proof Term Derivative Concatenation (Some)). *Suppose:*

(1) $pf : \Gamma \leq \Delta$
(2) $p : \texttt{prefix}(\Gamma)$
(3) $p' : \texttt{prefix}\left(\delta_p(\Gamma)\right)$
(4) $\delta_p(pf) \sim pf'$
(5) $\delta_{p'}(pf') \sim x$

*Where $x$ could either be some $pf''$ or $\bot$. Then,* $\delta_{p \cdot_\Gamma p'}(pf) \sim x$

Proof. ctxsubty.v:ctxSubyPfDeriv_hom_some □

### B.8 Type System

*Definition B.76 (Recursion Signature).* A recursion signature $\Sigma$ is either empty (signaling that typechecking is not in the body of a recursive function), or the signature $\Omega \mid \Gamma \longrightarrow s$ of a sequent which defines the recursive function we are currently checking the body of. $\Sigma ::= \emptyset \mid (\Omega \mid \Gamma \longrightarrow s)$

*Definition B.77 (Type System).*

$$\textsc{Var}\ \frac{}{\Omega \mid \Gamma(x : s) \vdash_\Sigma (x : s@\Gamma(-)) : s} \qquad \textsc{HistPgm}\ \frac{\Omega \vdash M : \langle s \rangle}{\Omega \mid \Gamma \vdash_\Sigma \langle M : s \rangle : s} \qquad \textsc{Par-R}\ \frac{\Omega \mid \Gamma \vdash_\Sigma e_1 : s \qquad \Omega \mid \Gamma \vdash_\Sigma e_2 : t}{\Omega \mid \Gamma \vdash_\Sigma (e_1, e_2) : s \| t}$$

$$\textsc{Par-L}\ \frac{\Omega \mid \Gamma(x : s, y : t) \vdash_\Sigma e : r}{\Omega \mid \Gamma(z : s \| t) \vdash_\Sigma \mathtt{let}_{\Gamma(-)}\ (x, y) = z\ \mathtt{in}\ e : r} \qquad \textsc{Cat-R}\ \frac{\Omega \mid \Gamma \vdash_\Sigma e_1 : s \qquad \Omega \mid \Delta \vdash_\Sigma e_2 : t}{\Omega \mid \Gamma; \Delta \vdash_\Sigma (e_1; e_2) : s \cdot t}$$

$$\textsc{Cat-L}\ \frac{\Omega \mid \Gamma(x : s; y : t) \vdash_\Sigma e : r}{\Omega \mid \Gamma(z : s \cdot t) \vdash_\Sigma \mathtt{let}_{\Gamma(-)}\ (x; y) = z\ \mathtt{in}\ e : r} \qquad \textsc{Eps-R}\ \frac{}{\Omega \mid \Gamma \vdash_\Sigma \mathtt{sink} : \varepsilon} \qquad \textsc{Plus-R-1}\ \frac{\Omega \mid \Gamma \vdash_\Sigma e : s}{\Omega \mid \Gamma \vdash_\Sigma \mathtt{inl}\,(e) : s + t}$$

$$\textsc{Plus-R-2}\ \frac{\Omega \mid \Gamma \vdash_\Sigma e : t}{\Omega \mid \Gamma \vdash_\Sigma \mathtt{inr}\,(e) : s + t}$$

$$\textsc{Plus-L}\ \frac{p : \mathtt{prefix}\,(\Gamma(z : s + t)) \qquad \Omega \mid \Gamma(x : s) \vdash_\Sigma e_1 : r \qquad \Omega \mid \Gamma(y : t) \vdash_\Sigma e_2 : r}{\Omega \mid \delta_p\,(\Gamma(z : s + t)) \vdash_\Sigma \mathtt{case}_{\Gamma(-),s,t,r}\,(p; z, x.e_1, y.e_2) : r}$$

$$\textsc{Star-R-1}\ \frac{}{\Omega \mid \Gamma \vdash_\Sigma \mathtt{nil} : s^\star} \qquad \textsc{Star-R-2}\ \frac{\Omega \mid \Gamma \vdash_\Sigma e_1 : s \qquad \Omega \mid \Delta \vdash_\Sigma e_2 : s^\star}{\Omega \mid \Gamma; \Delta \vdash_\Sigma e_1 :: e_2 : s^\star}$$

$$\textsc{Star-L}\ \frac{p : \mathtt{prefix}\,\big(\Gamma(z : s^\star)\big) \qquad \Omega \mid \Gamma(\cdot) \vdash_\Sigma e_1 : r \qquad \Omega \mid \Gamma(x : s; xs : s^\star) \vdash_\Sigma e_2 : r}{\Omega \mid \delta_p\,\big(\Gamma(z : s^\star)\big) \vdash_\Sigma \mathtt{case}_{\Gamma(-),s,r}\,(p; z, e_1, x.xs.e_2) : r}$$

$$\textsc{Wait}\ \frac{p : \mathtt{prefix}\,(\Gamma(\Delta)) \qquad \Omega, \langle \Delta \rangle \mid \Gamma(\cdot) \vdash_\Sigma e : s}{\Omega \mid \delta_p\,(\Gamma(\Delta)) \vdash_\Sigma \mathtt{wait}_{\Gamma(-),\Delta,s}\,(p; e) : s}$$

$$\textsc{Fix}\ \frac{\Omega' = x_1 : A_1, \ldots, x_n : A_n \qquad \Omega' \mid \Gamma \vdash_{\Omega' \mid \Gamma \to s} e : s \qquad \Omega \vdash M_i : A_i}{\Omega \mid \Gamma \vdash_\Sigma \mathtt{fix}\,(x_1 : A_1, \ldots, x_n : A_n)\ @\ [M_1, \ldots, M_n]\,.e : s}$$

$$\textsc{Rec}\ \frac{\Omega' = x_1 : A_1, \ldots, x_n : A_n \qquad \Omega \vdash M_i : A_i}{\Omega \mid \Gamma \vdash_{(\Omega' \mid \Gamma \to s)} \mathtt{rec}\ @\ [M_1 : A_1, \ldots, M_n : A_n] : s} \qquad \textsc{Cut}\ \frac{\Omega \mid \Delta \vdash_\Sigma e_1 : s \qquad \Omega \mid \Gamma(x : s) \vdash_\Sigma e_2 : r}{\Omega \mid \Gamma(\Delta) \vdash_\Sigma \mathtt{let}_{\Gamma(-)}\ x = e_1\ \mathtt{in}\ e_2 : r}$$

$$\textsc{SubCtx}\ \frac{pf : \Gamma \leq \Gamma'\Omega \mid \Gamma' \vdash_\Sigma e : s}{\Omega \mid \Gamma \vdash_\Sigma \mathtt{subctx}\,(pf, e) : s}$$

*Buffering Rules.* Some typing rules are different from those presented in the body of the paper. In particular, the left rules for star and sums, as well as Wait, include a *buffer* in the term: a prefix of the input context, where we store inputs until we have received enough to run the term. For example, the Wait rule has this buffer $p$, which we gather until it includes a whole batch of $\Delta$.

$$\textsc{Wait}\ \frac{p : \mathtt{prefix}\,(\Gamma(\Delta)) \qquad \Omega, \langle \Delta \rangle \mid \Gamma(\cdot) \vdash_\Sigma e : s}{\Omega \mid \delta_p\,(\Gamma(\Delta)) \vdash_\Sigma \mathtt{wait}_{\Gamma(-),\Delta,s}\,(p; e) : s}$$

The buffer is included in the syntax of the term. Additionally, the context in the conclusion is $\delta_p\,(\Gamma(\Delta))$. If we've buffered $p$ of the input, the term is expecting the rest of the context. Users of the calculus need not worry about this detail: when writing programs and when the program starts

running, the buffer is empty: $p = \mathsf{emp}_{\Gamma(\Delta)}$. By Theorem B.13, $\delta_p\,(\Gamma(\Delta)) = \Gamma(\Delta)$, which returns Wait to the expected rule presented in the body of the paper. The other rules that include buffers are Plus-L and Star-L.

*Generalized Fixpoints.* To account for the inclusion of the historical context, the rules for defining recursive programs by fixpoint and making recursive calls are generalized from the ones in the body of the paper to the rules shown below. The main idea is the same: `fix` registers a term as a recursive function, and `rec` can be used inside the body of the term to refer back to the function being defined. Since the full $\lambda^{\mathrm{ST}}$ calculus includes memory by means of the historical context, we need a mechanism to (a) kick off a recursive function with initial values in bound to historical context, and (b) change values as we make recursive calls.

$$\textsc{Fix}\ \frac{\Omega' = x_1 : A_1, \ldots, x_n : A_n \qquad \Omega' \mid \Gamma \vdash_{\Omega' \mid \Gamma \to s} e : s \qquad \Omega \vdash M_i : A_i}{\Omega \mid \Gamma \vdash_\Sigma \mathtt{fix}\,(x_1 : A_1, \ldots, x_n : A_n)\ @\ [M_1, \ldots, M_n]\ .e : s}$$

$$\textsc{Rec}\ \frac{\Omega' = x_1 : A_1, \ldots, x_n : A_n \qquad \Omega \vdash M_i : A_i}{\Omega \mid \Gamma \vdash_{(\Omega' \mid \Gamma \to s)} \mathtt{rec}\ @\ [M_1 : A_1, \ldots, M_n : A_n] : s}$$

When defining a recursive transformer with Fix, the programmer can choose a context $\Omega'$, different from the current context $\Omega$ for the recursive function to run in. This context includes any accumulators that the transformer will maintain. At definition time, the programmer must provide initial values for each of the accumulator values. Formally, this looks like a list of terms $M_i$, one for each position in the context $\Omega'$, typed in the defining context $\Omega$. The context $\Omega'$ is also included in the updated recursion signature, $\Omega' \mid \Gamma tos$. Then, when making recursive calls, the programmer decides how these values should be updated before continuing with the next iteration. As we will see in Appendix C, accumulator values are often updated with data computed from the stream that has arrived since the last recursive call was made. This new recursion term is written with $\mathtt{rec}\ @\ [M_1, \ldots, M_n]$, where the terms $M_i$ compute define the new accumulator values from the current historical context $\Omega$.

### B.9 Sink Terms

Once we have produced an entire batch $b : \mathtt{batch}\,(s)$, a program $e$ of type $s$ needs to transition to a program emitting nothing: we compute this term from $b$ with $\mathtt{sink}_b$.

*Definition B.78 (Sink Terms).* We define sink terms by recursion on $b$, as follows:

(1) $\mathtt{sink}_{\mathsf{epsb}} = \mathtt{sink}_{\mathsf{oneb}} = \mathtt{sink}_{[]} = \mathtt{sink}_{b_1 :: b_2} = \mathtt{sink}$
(2) $\mathtt{sink}_{\mathsf{parb}(b_1, b_2)} = \left(\mathtt{sink}_{b_1}, \mathtt{sink}_{b_2}\right)$
(3) $\mathtt{sink}_{\mathsf{catb}(b_1, b_2)} = \mathtt{sink}_{b_2}$
(4) $\mathtt{sink}_{\mathsf{sumbA}(b)} = \mathtt{sink}_{\mathsf{sumbB}(b)} = \mathtt{sink}_b$

This program is closed, and has type we expect for a stream transformer that has just emitted an entire batch $b$ of type $s$.

Theorem B.79 (Sink Term Correctness). *When* $b : \mathtt{batch}\,(s)$, *we have* $\cdot \mid \cdot \vdash_\emptyset \mathtt{sink}_b : \delta_{(b)^\circ}\,(s)$.

Proof. `language.v:sinkexpr_tck` □

The relevant concatenation property of sink terms is that they only depend on the the shape of the type $s$ *after* the batch has been emitted, so adding more to the beginning does not change anything.

Theorem B.80 (Sink Term Concatenation). *If* $p : \mathtt{prefix}(s)$ *and* $b : \mathtt{batch}(\delta_p(s))$, *then* $\mathsf{sink}_b = \mathsf{sink}_{p \cdot_s b}$.

Proof. `language.v:sinkepr'_hom` □

Theorem B.81 (Fixpoint Substitution). *For* $e, e_f$ *terms, we define* $e\left[e_f/\mathtt{use}\right]$ *compositionally over the structure of* $e$, *with* $\left(\mathtt{rec} @ \left[\overline{M : A}\right]\right)\left[e_f/\mathtt{use}\right] = \mathtt{fix}\left(\overline{x : A}\right) @ \left[\overline{M}\right].e_f$

*Then, if* $\Omega \mid \Gamma \vdash_{\Sigma \cup \{f:\Omega' \mid \Delta \to t\}} e : s$ *and* $\Omega' \mid \Delta \vdash_{(f:\Omega' \mid \Delta \to t)} e_f : t$, *we have* $\Omega \mid \Gamma \vdash_\Sigma e\left[e_f/\mathtt{use}\right] : s$.

Proof. `language.v:fixSubst_correct` □

## B.10 Semantics

*Batch Semantics.* We define a whole-batch semantics of the language: given an input batch and a program $e$, we way that $b \Rightarrow e \Rightarrow^n b'$ if $e$ produces output $b'$ after doing at most $n$ unfoldings of fixpoints.

*Definition B.82 (Batch Semantics).* We define a 4-place relation $b \Rightarrow e \Rightarrow^n b'$ as follows.

$$\text{B-Var}\ \frac{b \rightsquigarrow_{\Gamma(-)} b', \_}{b \Rightarrow (x : s @ \Gamma(-)) \Rightarrow^n b'} \qquad \text{B-HistPgm}\ \frac{M \downarrow v}{b \Rightarrow \langle M : s \rangle \Rightarrow^n \mathsf{toBatch}_s(v)}$$

$$\text{B-Par-R}\ \frac{b \Rightarrow e_1 \Rightarrow^{n_1} b_1 \qquad b \Rightarrow e_2 \Rightarrow^{n_2} b_2}{b \Rightarrow (e_1, e_2) \Rightarrow^{n_1+n_2} \mathsf{parb}(b_1, b_2)} \qquad \text{B-Par-L}\ \frac{b \Rightarrow e \Rightarrow^n b'}{b \Rightarrow \mathtt{let}_{\Gamma(-)}\ (z, x) = y\ \mathtt{in}\ e \Rightarrow^n b'}$$

$$\text{B-Cat-R}\ \frac{b_1 \Rightarrow e_1 \Rightarrow^{n_1} b_1' \qquad b_2 \Rightarrow e_2 \Rightarrow^{n_2} b_2'}{\mathsf{catb}(b_1, b_2) \Rightarrow (e_1; e_2) \Rightarrow^{n_1+n_2} \mathsf{catb}(b_1', b_2')} \qquad \text{B-Cat-L}\ \frac{b \Rightarrow e \Rightarrow^n b'}{b \Rightarrow \mathtt{let\_}\ (z; x) = y\ \mathtt{in}\ e \Rightarrow^n b'}$$

$$\text{B-Eps-R}\ \frac{}{b \Rightarrow \mathtt{sink} \Rightarrow^n \mathsf{epsb}}$$

$$\text{B-Plus-R-1}\ \frac{b \Rightarrow e \Rightarrow^n b'}{b \Rightarrow \mathtt{inl}(e) \Rightarrow^n \mathsf{sumbA}(b')} \qquad \text{B-Plus-R-2}\ \frac{b \Rightarrow e \Rightarrow^n b'}{b \Rightarrow \mathtt{inr}(e) \Rightarrow^n \mathsf{sumbB}(b')}$$

$$\text{B-Plus-L-1}\ \frac{p \cdot_{\Gamma(z:s+t)} b \sim b' \qquad b' \rightsquigarrow_{\Gamma(-)} \mathsf{sumbA}(b''), f \qquad f(b'') \Rightarrow e_1 \Rightarrow^n b'''}{b \Rightarrow \mathtt{case}_{\Gamma(-),s,t,r}(p; z, x.e_1, y.e_2) \Rightarrow^n b'''}$$

$$\text{B-Plus-L-2}\ \frac{p \cdot_{\Gamma(z:s+t)} b \sim b' \qquad b' \rightsquigarrow_{\Gamma(-)} \mathsf{sumbB}(b''), f \qquad f(b'') \Rightarrow e_2 \Rightarrow^n b'''}{b \Rightarrow \mathtt{case}_{\Gamma(-),s,t,r}(p; z, x.e_1, y.e_2) \Rightarrow^n b'''}$$

$$\text{B-Star-R-1}\ \frac{}{b \Rightarrow \mathtt{nil} \Rightarrow^n []} \qquad \text{B-Star-R-2}\ \frac{b_1 \Rightarrow e_1 \Rightarrow^{n_1} b_1' \qquad b_2 \Rightarrow e_2 \Rightarrow^{n_2} b_2'}{\mathsf{catb}(b_1, b_2) \Rightarrow e_1 :: e_2 \Rightarrow^{n_1+n_2} b_1' :: b_2'}$$

$$\text{B-Star-L-1}\quad \frac{p \cdot_{\Gamma(z:s^\star)} b \sim b' \qquad b' \rightsquigarrow_{\Gamma(z:s^\star)} [], f \qquad f(\mathsf{epsb}) \Rightarrow e_1 \Rightarrow^n b''}{b \Rightarrow \mathsf{case}_{\Gamma(-),s,r}\,(p; z, e_1, x.xs.e_2) \Rightarrow^n b''}$$

$$\text{B-Star-L-2}\quad \frac{p \cdot_{\Gamma(z:s^\star)} b \sim b' \qquad b' \rightsquigarrow_{\Gamma(z:s^\star)} b_1 :: b_2, f \qquad f(\mathsf{catb}\,(b_1, b_2)) \Rightarrow e_2 \Rightarrow^n b''}{b \Rightarrow \mathsf{case}_{\Gamma(-),s,r}\,(p; z, e_1, x.xs.e_2) \Rightarrow^n b''}$$

$$\text{B-Wait}\quad \frac{p \cdot_{\Gamma(\Delta)} b \sim b' \qquad b' \rightsquigarrow_{\Gamma(-)} b_\Delta, f \qquad f(\mathsf{epsb}) \Rightarrow e\,[\langle b_\Delta\rangle/\langle\Delta\rangle] \Rightarrow^n b''}{b \Rightarrow \mathsf{wait}_{\Gamma(-),\Delta,s}\,(p; e) \Rightarrow^n b''}$$

$$\text{B-SubCtx}\quad \frac{\mathsf{coeBatch}\,(pf, b) \sim b' \qquad b' \Rightarrow e \Rightarrow^n b''}{b \Rightarrow \mathsf{subctx}\,(pf, e) \Rightarrow^n b''}$$

$$\text{B-Fix}\quad \frac{\overline{M_i \downarrow v_i} \qquad b \Rightarrow e_f\,[e_f/\mathsf{use}]\,\left[\overline{v_i/x_i}\right] \Rightarrow^n b'}{b \Rightarrow \mathsf{fix}\left(\overline{x_i : A_i}\right) @ \left[\overline{M_i}\right].e_f \Rightarrow^{n+1} b'}$$

$$\text{B-Cut}\quad \frac{b \rightsquigarrow_{\Gamma(-)} b', f \qquad b' \Rightarrow e_1 \Rightarrow^{n_1} b'' \qquad f(b'') \Rightarrow e_2 \Rightarrow^{n_2} b'''}{b \Rightarrow \mathsf{let}_{\Gamma(-)}\, x = e_1\, \mathsf{in}\, e_2 \Rightarrow^{n_1+n_2} b'''}$$

Theorem B.83 (Batch Semantics Gas Monotonicity). *If* $b \Rightarrow e \Rightarrow^n b'$ *and* $n' \geq n$, *then* $b \Rightarrow e \Rightarrow^{n'} b'$.

Proof. `language.v:batch_sem_mono` □

The batch semantics relation is deterministic in the sense that for any term $e$ and any input batch $b$, it will only ever terminate with at most one value. Note that this is independent of the gas provided: any terminating evaluation will terminate with the same value.

Theorem B.84 (Batch Semantics Inputs Determine Outputs). *If* $b \Rightarrow e \Rightarrow^{n_1} b_1$ *and* $b \Rightarrow e \Rightarrow^{n_2} b_2$, *then* $b_1 = b_2$.

Proof. `language.v:batch_sem_det` □

Theorem B.85 (Batch Step Correctness). *Suppose the following:*

(1) $\cdot \mid \Gamma \vdash_\emptyset e : s$
(2) $b \Rightarrow e \Rightarrow^n b'$
(3) $b : \mathsf{batch}\,(\Gamma)$

*Then,* $b' : \mathsf{batch}\,(s)$.

Proof. `language.v:batch_pres` □

*Incremental Semantics.*

*Definition B.86 (Incremental Semantics).* We define a relation $p \Rightarrow e \downarrow^n e' \Rightarrow p'$.

P-Var-1

$$\frac{p@\Gamma(-) \rightsquigarrow \bot@\Gamma'(-)}{p \Rightarrow (x : s@\Gamma(-)) \downarrow^{n} (x : s@\Gamma'(-)) \Rightarrow \mathsf{emp}_s}$$

P-Var-2

$$\frac{p@\Gamma(-) \rightsquigarrow p'@\Gamma'(-)@f \qquad \delta_{p'}(s) \sim s'}{p \Rightarrow (x : s@\Gamma(-)) \downarrow^{n} (x : s'@\Gamma'(-)) \Rightarrow p'}$$

P-Var-3

$$\frac{p@\Gamma(-) \rightsquigarrow b@\Gamma'@f \qquad \delta_{(b)^\circ}(s) \sim s'}{p \Rightarrow (x : s@\Gamma(-)) \downarrow^{n} \mathsf{subctx}(\mathsf{SubtyWknAll}, \mathsf{sink}_b) \Rightarrow (b)^\circ}$$

P-HistPgm

$$\frac{M \downarrow v \qquad b = \mathsf{toBatch}_s(v)}{p \Rightarrow \langle M : s \rangle \downarrow^{n} \mathsf{subctx}(\mathsf{SubtyWknAll}, \mathsf{sink}_b) \Rightarrow (b)^\circ}$$

P-Par-R

$$\frac{p \Rightarrow e_1 \downarrow^{n_1} e_1' \Rightarrow p_1 \qquad p \Rightarrow e_2 \downarrow^{n_2} e_2' \Rightarrow p_2}{p \Rightarrow (e_1, e_2) \downarrow^{n_1+n_2} (e_1', e_2') \Rightarrow \mathsf{parp}(p_1, p_2)}$$

P-Par-L-1

$$\frac{(p@\Gamma(-) \rightsquigarrow \bot@\Gamma'(-)) \vee (p@\Gamma(-) \rightsquigarrow p'@\Gamma'(-)@\_) \qquad p \Rightarrow e \downarrow^{n} e' \Rightarrow p'}{p \Rightarrow \mathsf{let}_{\Gamma(-)}\ (x,y) = z\ \mathsf{in}\ e \downarrow^{n} \mathsf{let}_{\Gamma'(-)}\ (x,y) = z\ \mathsf{in}\ e' \Rightarrow p'}$$

P-Par-L-2

$$\frac{p@\Gamma(-) \rightsquigarrow \_@\_p \Rightarrow e \downarrow^{n} e' \Rightarrow p'}{p \Rightarrow \mathsf{let}_{\Gamma(-)}\ (x,y) = z\ \mathsf{in}\ e \downarrow^{n} e' \Rightarrow p'}$$

P-Cat-R-1

$$\frac{p \Rightarrow e_1 \downarrow^{n} e_1' \Rightarrow p'}{\mathsf{catpA}(p) \Rightarrow (e_1; e_2) \downarrow^{n} (e_1'; e_2) \Rightarrow \mathsf{catpA}(p')}$$

P-Cat-R-2

$$\frac{b \Rightarrow e_1 \Rightarrow^{n_1} b' \qquad p \Rightarrow e_2 \downarrow^{n_2} e_2' \Rightarrow p'}{\mathsf{catpB}(b,p) \Rightarrow (e_1; e_2) \downarrow^{n_1+n_2} e_2' \Rightarrow \mathsf{catpB}(b', p')}$$

P-Cat-L-1

$$\frac{(p@\Gamma(-) \rightsquigarrow \bot@\Gamma'(-)) \vee (p@\Gamma(-) \rightsquigarrow \mathsf{catpA}(\_)@\Gamma'(-)@\_) \qquad p \Rightarrow e \downarrow^{n} e' \Rightarrow p'}{p \Rightarrow \mathsf{let}_{\Gamma(-)}\ (z;x) = y\ \mathsf{in}\ e \downarrow^{n} \mathsf{let}_{\Gamma'(-)}\ (z;x) = y\ \mathsf{in}\ e' \Rightarrow p'}$$

P-Cat-L-2

$$\frac{(p@\Gamma(-) \rightsquigarrow \mathsf{catpB}(\_,\_)@\_@\_) \vee (p@\Gamma(-) \rightsquigarrow b@\_) \qquad p \Rightarrow e \downarrow^{n} e' \Rightarrow p'}{p \Rightarrow \mathsf{let}_{\Gamma(-)}\ (z;x) = y\ \mathsf{in}\ e \downarrow^{n} e' \Rightarrow p'}$$

P-Eps-R

$$\frac{}{p \Rightarrow \mathsf{sink} \downarrow^{n} \mathsf{sink} \Rightarrow \mathsf{epsp}}$$

P-Plus-R-1

$$\frac{p \Rightarrow e \downarrow^{n} e' \Rightarrow p'}{p \Rightarrow \mathsf{inl}(e) \downarrow^{n} e' \Rightarrow \mathsf{sumpA}(p')}$$

P-Plus-R-2

$$\frac{p \Rightarrow e \downarrow^{n} e' \Rightarrow p'}{p \Rightarrow \mathsf{inr}(e) \downarrow^{n} e' \Rightarrow \mathsf{sumpB}(p')}$$

P-Plus-L-1

$$\frac{p' \cdot_{\Gamma(z:s+t)} p \sim p'' \qquad (p''@\Gamma(-) \rightsquigarrow \bot@\_) \vee (p@\Gamma(-) \rightsquigarrow \mathsf{sumpEmp}@\_@\_)}{p \Rightarrow \mathsf{case}_{\Gamma(-),s,t,r}\ (p'; z, x.e_1, y.e_2) \downarrow^n \mathsf{case}_{\Gamma(-),s,t,r}\ (p''; z, x.e_1, y.e_2) \Rightarrow \mathsf{emp}_r}$$

P-Plus-L-2-1

$$\frac{p' \cdot_{\Gamma(z:s+t)} p \sim p'' \qquad p''@\Gamma(-) \rightsquigarrow \mathsf{sumpA}(p_s)@\Gamma'(-)@f \qquad f(p_s) \Rightarrow e_1 \downarrow^n e_1' \Rightarrow p_r}{p \Rightarrow \mathsf{case}_{\Gamma(-),s,t,r}\ (p'; z, x.e_1, y.e_2) \downarrow^n e_1' \Rightarrow p_r}$$

P-Plus-L-2-2

$$\frac{p' \cdot_{\Gamma(z:s+t)} p \sim p'' \qquad p''@\Gamma(-) \rightsquigarrow \mathsf{sumpB}(p_t)@\Gamma'(-)@f \qquad f(p_t) \Rightarrow e_2 \downarrow^n e_2' \Rightarrow p_r}{p \Rightarrow \mathsf{case}_{\Gamma(-),s,t,r}\ (p'; z, x.e_1, y.e_2) \downarrow^n e_2' \Rightarrow p_r}$$

P-Plus-L-3-1

$$\frac{p' \cdot_{\Gamma(z:s+t)} p \sim p'' \qquad p''@\Gamma(-) \rightsquigarrow \mathsf{sumbA}\,(b)\,@f \qquad f(b) \Rightarrow e_1 \downarrow^n e_1' \Rightarrow p_r}{p \Rightarrow \mathsf{case}_{\Gamma(-),s,t,r}\ (p'; z, x.e_1, y.e_2) \downarrow^n e_1' \Rightarrow p_r}$$

P-Plus-L-3-2

$$\frac{p' \cdot_{\Gamma(z:s+t)} p \sim p'' \qquad p''@\Gamma(-) \rightsquigarrow \mathsf{sumbB}\,(b)\,@f \qquad f(b) \Rightarrow e_2 \downarrow^n e_2' \Rightarrow p_r}{p \Rightarrow \mathsf{case}_{\Gamma(-),s,t,r}\ (p'; z, x.e_1, y.e_2) \downarrow^n e_2' \Rightarrow p_r}$$

P-Star-R-1

$$\frac{}{p \Rightarrow \mathsf{nil} \downarrow^n \mathsf{sink} \Rightarrow \mathsf{stpDone}}$$

P-Star-R-2-1

$$\frac{p \Rightarrow e_1 \downarrow^n e_1' \Rightarrow p'}{\mathsf{catpA}(p) \Rightarrow e_1 :: e_2 \downarrow^n (e_1'; e_2) \Rightarrow \mathsf{stpA}(p')}$$

P-Star-R-2-2

$$\frac{b \Rightarrow e_1 \Rightarrow^n b' \qquad p \Rightarrow e_2 \downarrow^n e_2' \Rightarrow p'}{\mathsf{catpB}(b, p) \Rightarrow e_1 :: e_2 \downarrow^n e_2' \Rightarrow \mathsf{stpB}(b', p')}$$

P-Star-L-Emp

$$\frac{p' \cdot_{\Gamma(x:s^\star)} p \sim p'' \qquad (p''@\Gamma(-) \rightsquigarrow \bot@\Gamma'(-)) \vee (p''@\Gamma(-) \rightsquigarrow \mathsf{stpEmp}@\Gamma'(-)@f)}{p \Rightarrow \mathsf{case}_{\Gamma(-),s,t}\ (p'; z, e_1, x.xs.e_2) \downarrow^n \mathsf{case}_{\Gamma(-),s,t}\ (p''; z, e_1, x.xs.e_2) \Rightarrow \mathsf{emp}_t}$$

P-Star-L-Done-1

$$\frac{p' \cdot_{\Gamma(x:s^\star)} p \sim p'' \qquad p''@\Gamma(-) \rightsquigarrow \mathsf{stpDone}@\Gamma'(-)@f \qquad f(\mathsf{epsp}) \Rightarrow e_1 \downarrow^n e_1' \Rightarrow p'''}{p \Rightarrow \mathsf{case}_{\Gamma(-),s,t}\ (p'; z, e_1, x.xs.e_2) \downarrow^n \mathsf{subctx}\left(\mathsf{SubtyCong}(\Gamma'(-), \mathsf{SubtyWknAll}), e_1'\right) \Rightarrow p'''}$$

P-Star-L-Done-2

$$\frac{p' \cdot_{\Gamma(x:s^\star)} p \sim p'' \qquad p''@\Gamma(-) \rightsquigarrow []@f \qquad f(\mathsf{epsb}) \Rightarrow e_1 \downarrow^n e_1' \Rightarrow p'''}{p \Rightarrow \mathsf{case}_{\Gamma(-),s,t}\ (p'; z, e_1, x.xs.e_2) \downarrow^n e_1' \Rightarrow p'''}$$

P-Star-L-Cons-1

$$\frac{p' \cdot_{\Gamma(x:s^\star)} p \sim p'' \qquad p''@\Gamma(-) \rightsquigarrow \mathsf{stpA}(p_s)@\Gamma'(-)@f \qquad f(p_s) \Rightarrow e_2 \downarrow^n e_2' \Rightarrow p'''}{p \Rightarrow \mathsf{case}_{\Gamma(-),s,t}\ (p'; z, e_1, x.xs.e_2) \downarrow^n \mathsf{let}_{\Gamma(-)}\ (x; xs) = z\ \mathsf{in}\ e_2' \Rightarrow p'''}$$

P-Star-L-Cons-2

$$\frac{p' \cdot_{\Gamma(x:s^\star)} p \sim p'' \qquad p''@\Gamma(-) \rightsquigarrow \mathsf{stpB}(b, p_s)@\Gamma'(-)@f \qquad f(\mathsf{catpB}(b, p_s)) \Rightarrow e_2 \downarrow^n e_2' \Rightarrow p'''}{p \Rightarrow \mathsf{case}_{\Gamma(-),s,t}\ (p'; z, e_1, x.xs.e_2) \downarrow^n e_2' \Rightarrow p'''}$$

P-Star-L-Cons-3

$$\frac{p' \cdot_{\Gamma(x:s^\star)} p \sim p'' \qquad p''@\Gamma(-) \rightsquigarrow b_1 :: b_2@f \qquad f(\mathsf{catb}\,(b_1, b_2)) \Rightarrow e_2 \downarrow^n e_2' \Rightarrow p'''}{p \Rightarrow \mathsf{case}_{\Gamma(-),s,t}\ (p'; z, e_1, x.xs.e_2) \downarrow^n e_2' \Rightarrow p'''}$$

$$\text{P-Wait-1}\quad \frac{p' \cdot_{\Gamma(\Delta)} p \sim p'' \qquad (p''@\Gamma(-) \rightsquigarrow \bot@\_) \vee (p''@\Gamma(-) \rightsquigarrow \_@\_@\_)}{p \Rightarrow \mathtt{wait}_{\Gamma(-),\Delta,s}\,(p';e) \downarrow^n \mathtt{wait}_{\Gamma(-),\Delta,s}\,(p'';e) \Rightarrow \mathtt{emp}_s}$$

$$\text{P-Wait-2}\quad \frac{p' \cdot_{\Gamma(\Delta;\Delta')} p \sim p'' \qquad p''@\Gamma(-) \rightsquigarrow b_\Delta@f \qquad f(\mathtt{epsb}) \Rightarrow e\,[\langle b_\Delta\rangle/\langle\Delta\rangle] \downarrow^n e' \Rightarrow p_s}{p \Rightarrow \mathtt{wait}_{\Gamma(-),\Delta,s}\,(p;e) \downarrow^n e' \Rightarrow p_s}$$

$$\text{P-SubCtx-Some}\quad \frac{\mathtt{coePfx}\,(pf,p) \sim p' \qquad p' \Rightarrow e \downarrow^n e' \Rightarrow p''\delta_p\,(pf) \sim pf'}{p \Rightarrow \mathtt{subctx}\,(pf,e) \downarrow^n \mathtt{subctx}\,(pf',e') \Rightarrow p''}$$

$$\text{P-SubCtx-None}\quad \frac{\mathtt{coePfx}\,(pf,p) \sim p' \qquad p' \Rightarrow e \downarrow^n e' \Rightarrow p''\delta_p\,(pf) \sim \bot}{p \Rightarrow \mathtt{subctx}\,(pf,e) \downarrow^n e' \Rightarrow p''}$$

$$\text{P-Cut-Emp}\quad \frac{p@\Gamma(-) \rightsquigarrow \bot@\Gamma'(-) \qquad p \Rightarrow e_2 \downarrow^n e_2' \Rightarrow p'}{p \Rightarrow \mathtt{let}_{\Gamma(-)}\,x = e_1\,\mathtt{in}\,e_2 \downarrow^n \mathtt{let}_{\Gamma'(-)}\,x = e_1\,\mathtt{in}\,e_2' \Rightarrow p'}$$

$$\text{P-Cut-Pfx}\quad \frac{p@\Gamma(-) \rightsquigarrow p'@\Gamma'(-)@f \qquad p' \Rightarrow e_1 \downarrow^{n_1} e_1' \Rightarrow p'' \qquad f(p'') \Rightarrow e_2 \downarrow^{n_2} e_2' \Rightarrow p'''}{p \Rightarrow \mathtt{let}_{\Gamma(-)}\,x = e_1\,\mathtt{in}\,e_2 \downarrow^{n_1+n_2} \mathtt{let}_{\Gamma'(-)}\,x = e_1'\,\mathtt{in}\,e_2' \Rightarrow p'''}$$

$$\text{P-Cut-Bat}\quad \frac{p@\Gamma(-) \rightsquigarrow b@f \qquad b \Rightarrow e_1 \Rightarrow^{n_1} b' \qquad f(b') \Rightarrow e_2 \downarrow^{n_2} e_2' \Rightarrow p'}{p \Rightarrow \mathtt{let}_{\Gamma(-)}\,x = e_1\,\mathtt{in}\,e_2 \downarrow^{n_1+n_2} e_2' \Rightarrow p'}$$

$$\text{P-Fix}\quad \frac{\overline{M_i \downarrow v_i} \qquad p \Rightarrow e_f\,\left[e_f/\mathtt{use}\right]\left[\overline{v_i/x_i}\right] \downarrow^n e' \Rightarrow p'}{p \Rightarrow \mathtt{fix}\left(\overline{x_i : A_i}\right)\ @\ \left[\overline{M_i}\right].e_f \downarrow^{n+1} e' \Rightarrow p'}$$

*Incremental Semantics of Buffering.* The incremental semantics for Plus-L andStar-L and Wait buffer in their inputs until enough of the input has arrived to run the term, where the particular value of "enough" depends on the rule in question.

To illustrate, consider the rules for Wait.

$$\text{P-Wait-1}\quad \frac{p' \cdot_{\Gamma(\Delta)} p \sim p'' \qquad (p''@\Gamma(-) \rightsquigarrow \bot@\_) \vee (p''@\Gamma(-) \rightsquigarrow \_@\_@\_)}{p \Rightarrow \mathtt{wait}_{\Gamma(-),\Delta,s}\,(p';e) \downarrow^n \mathtt{wait}_{\Gamma(-),\Delta,s}\,(p'';e) \Rightarrow \mathtt{emp}_s}$$

$$\text{P-Wait-2}\quad \frac{p' \cdot_{\Gamma(\Delta;\Delta')} p \sim p'' \qquad p''@\Gamma(-) \rightsquigarrow b_\Delta@f \qquad f(\mathtt{epsb}) \Rightarrow e\,[\langle b_\Delta\rangle/\langle\Delta\rangle] \downarrow^n e' \Rightarrow p_s}{p \Rightarrow \mathtt{wait}_{\Gamma(-),\Delta,s}\,(p;e) \downarrow^n e' \Rightarrow p_s}$$

In both cases, we take the incoming prefix $p$, and concatenate it onto the buffer $p'$ at the context type $\Gamma(\Delta)$, to get the combined prefix $p''$. We then dispatch on whether $p''$ is enough input to run the continuation $e$. In this case, "enough" means that $p''$ contains a whole batch $b$ of $\Delta$. If it does (P-Wait-2), we run the continuation, substituting the batch in. If it does not — $p''$ either gives an empty projection or just a prefix — we simply save $p''$ as the new buffer in the resulting `wait` term, and return the empty prefix in P-Wait-1.

The semantics for Plus-L and Star-L are similar: in all cases, we add the incoming prefix to the buffer, and then project from the buffer. If not enough data has arrived, we return the empty prefix and step to the same term but with an updated buffer.

*Incremental Semantics of Cut.* To run a cut term $\texttt{let}_{\Gamma(-)}\ x = e_1\ \texttt{in}\ e_2$ incrementally, we use the prefix projection operation to pull out the part of the input prefix in the $\Gamma(-)$ hole, run it through $e_1$, substitute the result back into the input prefix, and then run $e_2$. This makes critical use of our prefix projection operation (Appendix B.5), and we have three incremental semantics rules corresponding to its three possibilities.

The first rule (P-Cut-1) covers the case where the input prefix $p$ contains no $\Delta$-data, and includes only data relevant to $e_2$. So, we bypass $e_1$ entirely and run $p$ through $e_2$, stepping to $e_2'$ and producing output $p'$. The prefix $p'$ is returned as the final output, and resulting term is the cut of $e_1$ with the updated term $e_2'$. In the second case (P-Cut-2), the input prefix $p$ contains a prefix $p'$ of type $\Delta$. We run this through $e_1$, which steps to $e_1'$ and produces $p''$. We then substitute $p''$ into $p$ with the computed substitution function $f$, binding $p''$ to $x$ in the context, and then run it through $e_2$, which steps to $e_2'$ and produces $p'''$. The entire term then returns $p'''$, and steps to the let term with both $e_1'$ and $e_2'$ replacing $e_1$ and $e_2$. In the third and final case (P-Cut-3), the input prefix contains a batch $b$ of $\Delta$. We then use the batch semantics to run this through $e_1$, producing a batch $b'$. We then substitute $b'$ into $p$ where $b$ was with $f$ and run the resulting prefix through $e_2$, which steps to $e_2'$ and produces $p'$. Because the part of the stream ($\Delta$) used to produce the $x : s$ that the cut served to bind has completed, the cut is no longer needed, and so the whole term transitions to $e_2'$, returning $p'$ as output.

*Incremental Semantics Theorems.*

Theorem B.87 (Incremental Semantics Inputs Determine Outputs). *If* $p \Rightarrow e \downarrow^n e' \Rightarrow p'$ *and* $p \Rightarrow e \downarrow^{n'} e'' \Rightarrow p''$, *then* $e' = e''$, *and* $p' = p''$.

Proof. `language.v:prefix_sem_det` □

Theorem B.88 (Incremental Semantics Monotonicity). *If* $p \Rightarrow e \downarrow^n e' \Rightarrow p'$ *and* $n' \geq n$, *then* $p \Rightarrow e \downarrow^{n'} e' \Rightarrow p'$.

Proof. `language.v:prefix_sem_mono` □

Theorem B.89 (Prefix Correctness). *Suppose*

(1) $\cdot \mid \Gamma \vdash_\emptyset e : s$
(2) $p \Rightarrow e \downarrow^n e' \Rightarrow p'$
(3) $p : \texttt{prefix}\,(\Gamma)$

*Then,* $p' : \texttt{prefix}\,(s)$ *and* $\cdot \mid \delta_p\,(\Gamma) \vdash_\emptyset e' : \delta_{p'}\,(s)$.

Proof. `language.v:prefix_pres` □

The following theorem proves that sink terms live up to their names. Given any input prefix $p$, the program $\texttt{sink}_b$ will output an empty prefix of the appropriate type, and then step to itself.

Theorem B.90 (Sink Term Semantics Characterization). *If* $b : \texttt{batch}\,(s)$, *then for all* $n$ *and* $p$, *we have* $p \Rightarrow \texttt{sink}_b \downarrow^n \texttt{sink}_b \Rightarrow \texttt{emp}_{\delta_{(b)^\circ}(s)}$.

The following theorem is a lemma on the way to the homomorphism theorem. If $e$ is run on a prefix $p_1$, outputting $p_2$ and stepping to $e'$, and then $e'$ is evaluated on an *batch* $b_1$ outputting a batch $b_2$, then running $e$ on the concatenated input $p_1 \cdot b_1$ yields the concatenated output $p_2 \cdot b_2$. The proof proceeds by induction on the derivation of $e$ running on $p_1$.

Theorem B.91 (Prefix/Batch Semantics Concatenation). *Suppose the following:*

(1) $p_1 : \texttt{prefix}\,(\Gamma)$
(2) $b_1 : \texttt{batch}\,(\delta_{p_1}\,(\Gamma))$

(3) $p_2 : \mathtt{prefix}(s)$
(4) $\cdot \mid \Gamma \vdash_\emptyset e : s$
(5) $p_1 \Rightarrow e \downarrow^{n_1} e' \Rightarrow p_2$
(6) $b_1 \Rightarrow e' \Rightarrow^{n_2} b_2$

*Then,* $p_1 \cdot_\Gamma b_1 \Rightarrow e \Rightarrow^{n_1+n_2} p_2 \cdot_s b_2$

Proof. `language.v:prefix_batch_hom` □

Theorem B.92 (Homomorphism Theorem). *Suppose:*

- $\cdot \mid \Gamma \vdash_\emptyset e : s$
- $p_1 : \mathtt{prefix}(\Gamma)$
- $p_2 : \mathtt{prefix}(\delta_{p_1}(\Gamma))$
- $p_1 \Rightarrow e \downarrow^{n_1} e' \Rightarrow p_1'$
- $p_2 \Rightarrow e' \downarrow^{n_2} e'' \Rightarrow p_2'$

*Then,* $p_1 \cdot_\Gamma p_2 \Rightarrow e \downarrow^{n_1+n_2} e'' \Rightarrow p_1' \cdot_s p_2'$

Proof. `language.v:prefix_hom` □

## B.11 Events

*Events* (see Section 6) allow us to represent a data stream as a sequence of totally ordered items, while retaining information needed to infer the rich structure of the batch or prefix representations. to *Events*. The grammar of events is:

$$x ::= \mathsf{oneev} \mid \mathsf{parevA}(x) \mid \mathsf{parevB}(x) \mid +_{\mathsf{puncA}} \mid +_{\mathsf{puncB}} \mid \cdot_{\mathsf{punc}} \mid \mathsf{catevA}(x)$$

*Definition B.93 (Event Typing Relation).* We define a binary relation $x : \mathtt{event}(s)$ when $x \in \Sigma$ and $s$ is a stream type.

$$\frac{}{\mathsf{oneev} : \mathtt{event}(1)} \qquad \frac{x : \mathtt{event}(s)}{\mathsf{parevA}(x) : \mathtt{event}(s \| t)} \qquad \frac{x : \mathtt{event}(t)}{\mathsf{parevB}(x) : \mathtt{event}(s \| t)}$$

$$\frac{}{+_{\mathsf{puncA}} : \mathtt{event}(s + t)} \qquad \frac{}{+_{\mathsf{puncB}} : \mathtt{event}(s + t)} \qquad \frac{s\ \mathsf{nullable}}{\cdot_{\mathsf{punc}} : \mathtt{event}(s \cdot t)}$$

$$\frac{x : \mathtt{event}(s)}{\mathsf{catevA}(x) : \mathtt{event}(s \cdot t)} \qquad \frac{}{+_{\mathsf{puncA}} : \mathtt{event}(s^\star)} \qquad \frac{}{+_{\mathsf{puncB}} : \mathtt{event}(s^\star)}$$

Note that $s + t$ and $s^\star$ share the same punctuation events. Intuitively, this is because $s^\star$ can be unrolled as $\varepsilon + (s \cdot s^\star)$.

*Definition B.94 (Event Derivative Relation).* We define a ternary relation $\delta_x(s) \sim s'$.

$$\frac{}{\delta_{\mathsf{oneev}}(1) \sim \varepsilon} \qquad \frac{\delta_x(s) \sim s'}{\delta_{\mathsf{parevA}(x)}(s \| t) \sim s' \| t} \qquad \frac{\delta_x(t) \sim t'}{\delta_{\mathsf{parevB}(x)}(s \| t) \sim s \| t'} \qquad \frac{}{\delta_{+_{\mathsf{puncA}}}(s + t) \sim s}$$

$$\frac{}{\delta_{+_{\mathsf{puncB}}}(s + t) \sim t} \qquad \frac{s\ \mathsf{nullable}}{\delta_{\cdot_{\mathsf{punc}}}(s \cdot t) \sim t} \qquad \frac{\delta_x(s) \sim s'}{\delta_{\mathsf{catevA}(x)}(s \cdot t) \sim s' \cdot t} \qquad \frac{}{\delta_{+_{\mathsf{puncA}}}(s^\star) \sim \varepsilon}$$

$$\frac{}{\delta_{+_{\mathsf{puncB}}}(s^\star) \sim s \cdot s^\star}$$

Theorem B.95 (Event Derivative Function). *If* $x : \mathtt{event}(s)$, *there is a unique* $s'$ *such that* $\delta_x(s) \sim s'$.

Proof. events.v:derivrelEv_fun □

Because of Theorem B.95, if we know $x : \texttt{event}(s)$, we may write the unique $s'$ such that $\delta_x(s) \sim s'$ as $\delta_x(s)$.

*Definition B.96 (Events Typing and Derivatives Relations).* We lift event typing to lists by derivatives.

$$\frac{}{[] : \texttt{events}(s)} \qquad \frac{x : \texttt{event}(s) \qquad \delta_x(s) \sim s' \qquad xs : \texttt{events}(s')}{x::xs : \texttt{events}(s)}$$

We also lift derivatives to lists of events in the natural way.

$$\frac{}{\delta_{[]}(s) \sim s} \qquad \frac{\delta_x(s) \sim s' \qquad \delta_{xs}(s') \sim s''}{\delta_{x::xs}(s) \sim s''}$$

Theorem B.97 (Events Derivative Function). *If $xs : \texttt{events}(s)$, there is a unique $s'$ such that $\delta_{xn}(s) \sim s'$.*

Proof. events.v:derivrelEv_list_fun □

Because of Theorem B.97, if we know $xs : \texttt{events}(s)$, we may write the unique $s'$ such that $\delta_{xs}(s) \sim s'$ as $\delta_{xs}(s)$.

Theorem B.98 (Empty List of Events). *For all $s$, we have $[] : \texttt{events}(s)$, and $\delta_{[]}(s) \sim s$*

Proof. Immediate. □

Theorem B.99 (Events Concatenation). *If*

(1) $xs : \texttt{events}(s)$
(2) $\delta_{xs}(s) \sim s'$
(3) $ys : \texttt{events}(s')$
(4) $\delta_{ys}(s') \sim s''$

*Then, $xs{+}{+}ys : \texttt{events}(s)$, and $\delta_{xs{+}{+}ys}(s) \sim s''$.*

*In other words, if $xs : \texttt{events}(s)$ and $ys : \texttt{events}(\delta_{xs}(s))$, then $xs{+}{+}ys : \texttt{events}(s)$ and $\delta_{xs{+}{+}ys}(s) = \delta_{ys}(\delta_{xs}(s))$.*

Proof. events.v:Ev_list_concat □

## Events

### Events to Prefix

*Definition B.100 (Event(s) to Prefix).*

$$\frac{}{\texttt{oneev} \hookrightarrow^{1} \texttt{onepB}} \qquad \frac{x \hookrightarrow^{s} p}{\texttt{parevA}(x) \hookrightarrow^{s \| t} \texttt{parp}(p, \texttt{emp}_t)} \qquad \frac{x \hookrightarrow^{t} p}{\texttt{parevB}(x) \hookrightarrow^{s \| t} \texttt{parp}(\texttt{emp}_s, p)}$$

$$\frac{x \hookrightarrow^{s} p}{\texttt{catevA}(x) \hookrightarrow^{s \cdot t} \texttt{catpA}(p)} \qquad \frac{s \ \texttt{nullable} \qquad s \ \texttt{done} \ b}{\cdot_{\texttt{punc}} \hookrightarrow^{s \cdot t} \texttt{catpB}(b, \texttt{emp}_t)} \qquad \frac{}{+_{\texttt{puncA}} \hookrightarrow^{s+t} \texttt{sumpA}(\texttt{emp}_s)}$$

$$\frac{}{+_{\texttt{puncB}} \hookrightarrow^{s+t} \texttt{sumpB}(\texttt{emp}_t)} \qquad \frac{}{+_{\texttt{puncA}} \hookrightarrow^{s^\star} \texttt{stpDone}} \qquad \frac{}{+_{\texttt{puncB}} \hookrightarrow^{s^\star} \texttt{stpA}(\texttt{emp}_s)}$$

$$\frac{}{[] \hookrightarrow^{s} \texttt{emp}_s} \qquad \frac{t \hookrightarrow^{s} p \qquad \delta_p s \sim s' \qquad ts \hookrightarrow^{s'} p' \qquad p' \cdot p \sim p''}{t :: ts \hookrightarrow^{s} p''}$$

Theorem B.101 (Event to Prefix Function). *If $e \in event(s)$ then there is a unique $p \in prefix(s)$ such that $x \hookrightarrow^{s} p$.*

Proof. `events.v:evToPfx_fun` □

Theorem B.102 (Events to Prefix Function). *If* $xs$ : $\texttt{events}\,(s)$, *then there is a unique* $p \in prefix(s)$ *such that* $xs \hookrightarrow^s p$.

Proof. `events.v:evToPfx_list_fun` □

Theorem B.103 (EToP Relation). *If* $x$ : $\texttt{event}\,(s)$ *and* $x \hookrightarrow^s p$ *then* $p \in prefix(s)$.

Proof. `events.v:evToPfx_correct` □

Theorem B.104 (EStoP Relation). *If* $xs$ : $\texttt{events}\,(s)$ *and* $xs \hookrightarrow^s p$ *then* $p \in prefix(s)$.

Proof. `events.v:evToPfx_list_correct` □

### Prefix to Events

*Definition B.105 (PToES).* In this definition, we write occasionally lift event constructors to lists, writing $\widehat{f}(xs)$ for $[f(x) \mid x \in xs]$. Also, we write $xs \| ys$ for the set of *shuffles* of the lists $xs$ and $ys$.

$$\text{PToES-Eps}\ \frac{}{\texttt{epsp}\ \dagger^{\epsilon}\ []} \qquad \text{PToES-One-A}\ \frac{}{\texttt{onepA}\ \dagger^{1}\ []} \qquad \text{PToES-One-B}\ \frac{}{\texttt{onepB}\ \dagger^{1}\ [\texttt{oneev}]}$$

$$\text{PToES-Par}\ \frac{p\ \dagger^{s}\ xs \qquad p'\ \dagger^{t}\ ys \qquad zs \in \widehat{\texttt{parevA}}\,(xs)\ \|\widehat{\texttt{parevB}}\,(ys)}{\texttt{parp}(p,p')\ \dagger^{s\|t}\ zs} \qquad \text{PToES-Cat-A}\ \frac{p\ \dagger^{s}\ xs}{\texttt{catpA}(p)\ \dagger^{s\cdot t}\ \widehat{\texttt{catevA}}\,(xs)}$$

$$\text{PToES-Cat-B}\ \frac{b^{\circ}\ \dagger^{s}\ xs \qquad p\ \dagger^{t}\ ys}{\texttt{catpB}(b,p)\ \dagger^{s\cdot t}\ \widehat{\texttt{catevA}}\,(xs)\ ++\ \cdot_{\texttt{punc}}::ys} \qquad \text{PToES-Sum-Emp}\ \frac{}{\texttt{sumpEmp}\ \dagger^{s+t}\ []} \qquad \text{PToES-Sum-A}\ \frac{p\ \dagger^{s}\ xs}{\texttt{sumpA}(p)\ \dagger^{s+t}\ +_{\texttt{puncA}}::xs}$$

$$\text{PToES-Sum-B}\ \frac{p\ \dagger^{t}\ xs}{\texttt{sumpB}(p)\ \dagger^{s+t}\ +_{\texttt{puncB}}::xs} \qquad \text{PToES-Star-Emp}\ \frac{}{\texttt{stpEmp}\ \dagger^{s^{\star}}\ []} \qquad \text{PToES-Star-Done}\ \frac{}{\texttt{stpDone}\ \dagger^{s^{\star}}\ +_{\texttt{puncA}}}$$

$$\text{PToES-Star-A}\ \frac{p\ \dagger^{s}\ xs}{\texttt{stpA}(p)\ \dagger^{s^{\star}}\ +_{\texttt{puncB}}::\widehat{\texttt{catevA}}\,(xs)} \qquad \text{PToES-Star-B}\ \frac{b^{\circ}\ \dagger^{s}\ xs \qquad p\ \dagger^{s^{\star}}\ ys}{\texttt{stpB}(b,p)\ \dagger^{s^{\star}}\ +_{\texttt{puncB}}::\widehat{\texttt{catevA}}\,(xs)++\cdot_{\texttt{punc}}::ys}$$

Theorem B.106 (PToES Empty). *If* $p\ \dagger^{s}\ xs$, *then* $xs = []$ *iff* $p = \texttt{emp}_s$

Proof. `events.v:PfxToEvs_Empty` and `events.v:pfxToEvs_empty'` □

Theorem B.107 (PToES Left Total). *If* $p \in prefix(s)$ *then there exists (not necessarily unique)* $xs$ *such that* $p\ \dagger^{s}\ xs$

Proof. `events.v:PfxToEvs_left_total` □

Lemma B.108 (PToES Relation Derivative Agreement). *If*

(1) $p$ : $\texttt{prefix}\,(s)$
(2) $p\ \dagger^{s}\ xs$
(3) $\delta_p\,(s) \sim s'$

*then* $xs$ : $\texttt{events}\,(s)$ *and* $\delta_{xs}\,(s) \sim s'$

Proof. `events.v:pfxToEvs_derivrelEv_correct_and_conf` □

The preceding lemma provides the following:

Theorem B.109 (PToES Relation). *If* $p \in prefix(s)$ *and* $p\ \dagger^{s}\ xs$ *then* $xs$ : $\texttt{events}\,(s)$

Proof. `events.v:pfxToEvs_correct` □

### Event Size

Each event carries tag information about where it appears within a structured stream; this is necessary for us to recover the rich prefix structure. Importantly, for a given stream type there is an upper bound on the amount of tag information to be included on any event in any stream of that type.

*Definition B.110 (Event size).* We define the *size* of an event recursively:

$$\begin{aligned} size(\texttt{oneev}) &= 1 \\ size(\texttt{parevA}\,(x)) &= 1 + size(x) \\ size(\texttt{parevB}\,(x)) &= 1 + size(x) \\ size(+_{\texttt{puncA}}) &= 1 \\ size(+_{\texttt{puncB}}) &= 1 \\ size(\cdot_{\texttt{punc}}) &= 1 \\ size(\texttt{catevA}\,(()\,x)) &= 1 + size(x) \end{aligned}$$

We lift this to lists of events in the natural way:

*Definition B.111 (Event List Size).*

$$\begin{aligned} size([\,]) &= 0 \\ size(x::xs) &= size(x) + size(xs) \end{aligned}$$

To construct an *a priori* bound on the size of any event to appear in stream, we recurse on the type of the stream:

*Definition B.112 (Event size bound).*

$$\begin{aligned} evSizeBound(\varepsilon) &= 0 \\ evSizeBound(1) &= 1 \\ evSizeBound(s \| t) &= 1 + \max\,(evSizeBound(s), evSizeBound(t)) \\ evSizeBound(s \cdot t) &= \max\,(1 + evSizeBound(s), evSizeBound(t)) \\ evSizeBound(s + t) &= \max\,(1, evSizeBound(s), evSizeBound(t)) \\ evSizeBound(s^\star) &= \max\,(1, 1 + evSizeBound(s)) \end{aligned}$$

We now re-state and prove Theorem 6.1:

Theorem B.113 (Bounded Event Size). *For all $s$, there is some $N = evSizeBound(()s)$ such that for any $xs$ : $\texttt{events}\,(s)$ and any $x \in xs$, we have that $|x| \leq N$, where $|\cdot|$ denotes the size of the AST.*

Proof. `events.v:sizeBound_correct` □

### Serialization and Deserialization

We turn now to the final result of 6, that we can serialize a prefix $p$ into a list of events $xs$, secure in the knowledge that when we deserialize $xs$ we will obtain the same prefix $p$.

Towards this result, we introduce a series of lemmas that allow us to use the tag information encoded in each event to recover the prefix structure during deserialization. Observe that the shape of each lemma mirrors that of the corresponding serialization (†) constructor.

Lemma B.114 (EsToP Par Recovery). *If*

(1) $zs \in \widehat{\texttt{parevA}}\,(xs) \,\|\widehat{\texttt{parevB}}\,(ys)$

(2) $xs \hookrightarrow^{s} p$
(3) $ys \hookrightarrow^{t} p'$
*then* $zs \hookrightarrow^{s \| t} \mathsf{parp}(p, p')$

Proof. `events.v:evToPfx_list_par_structural` □

Lemma B.115 (EsToP Cat Recovery). *If* $xs \hookrightarrow^{s} (b)^{\circ}$ *and* $ys \hookrightarrow^{t} p$ *then*

$$\widetilde{\mathsf{catevA}}\,(xs) +\!\!+ \cdot_{\mathsf{punc}} :: ys \hookrightarrow^{s \cdot t} \mathsf{catpB}(b, p)$$

Proof. `events.v:evToPfx_list_cat_structural` □

Lemma B.116 (EsToP Star Recovery). *If* $xs \hookrightarrow^{s} (b)^{\circ}$ *and* $ys \hookrightarrow^{s^{\star}} p$ *then*

$$+_{\mathsf{puncB}} :: \widetilde{\mathsf{catevA}}\,(xs) +\!\!+ \cdot_{\mathsf{punc}} :: ys \hookrightarrow^{s^{\star}} \mathsf{stpB}(b, p)$$

Proof. `events.v:evToPfx_list_star_structural` □

Theorem B.117 (Serialization/Deserialization Round Trip). *If* $p \dagger^{s} xs$, *then* $xs \hookrightarrow^{s} p$.

Proof. `events.v:roundTrip_dir1` □

## C EXAMPLES

This appendix collects some additional examples of programming with our stream types.

### C.1 Examples Illustrating Kernel $\lambda^{\mathbf{ST}}$

The parallel swap term from Section 3 is:

$$z : s \| t \vdash \texttt{let\_}\ (x; y) = z\ \texttt{in}\ ((y : t @ (x : s, -)), (x : s @ -, y : t)) : t \| s$$

And the "prepend" term is:

$$x : s \vdash \texttt{subctx}\,(\texttt{SubtyUnitSemicL}, ((); (x : s @ -))) : 1 \cdot s$$

### C.2 Map

Below is the sugared term for map, as presented in Section 5.

```
map (e : s ⟶ t) (xs : s*) : t* =
  fix. case xs of
       | nil => nil
       | y :: ys => e(y) :: rec(ys)
```

The Haskell encoding of the core term can be found in `Events.hs:maptm`.

### C.3 Filter

Here is the code for the `filter` transformer described in Section 5.

```
filter (e : s ⟶ 2) (xs : s*) : s* =
  fix. case xs of
       | nil => nil
       | y :: ys => let b = e(y) in
                    case b of
                    | inl() => y :: rec(ys)
                    | inr() => rec(ys)
```

The core term corresponding to this one can be found at `Examples.hs:filtertm`.

### C.4 Running Fold

We can also define a *running* fold operation on star streams, which outputs its partial results as it goes.

```
runningfold (e : ⟨t⟩ | s ⟶ t) (xs : s*) : t* =
  fix with (acc = init).
    case xs of
    | nil => nil
    | y :: ys =>
        let acc' = e(y)[acc] in
        wait acc' then
          ⟨ acc' ⟩ :: rec(ys)[acc']
        end
```

This core term can be found at `Examples.hs:foldtm`.

### C.5 Thresholding

The `parseLL` transformer parses a homogeneous stream of ints — as it would be sent from an actual IOT device — into the parsed representation that we can use to define the per-run averaging operation from Section 5. This transformer consumes the input stream, grouping together runs of

elements larger than the threshold `t`. The program operates the same as the functional program of type `[Int] -> [NonEmpty Int]` which computes these runs by way of a fold. The function `parseLL` consumes the input stream until a run starts (an element is larger than the threshold `t`), and then starts running an auxiliary function `parseLLAux` which actually builds the runs.

```
parseLL (t : int) (xs : int*) : (int · int*)* =
    fix. case xs of
         | nil => nil
         | y :: ys => wait y then
                      case ⟨ y > t ⟩ of
                      | true => let (zs,zss) = parseLLAux t xs in (y;zs);zss
                      | false => rec(ys)
                    end
```

`parseLLAux` takes in a stream *xs* of ints, and returns (1) the maximal initial prefix of the stream which is all larger than the threshold, followed by (2) the stream of runs after the first. This is strengthened specification is required to define the operation as a fold-left. `parseLLAux` maintains a Boolean `b` which indicates if the stream is currently in the middle of a run.

```
parseLLAux (t : int) (xs : int*) : int* · (int · int*)* =
    fix with (b = true).
        case xs of
        | nil => (nil ; nil)
        | y :: ys => wait y then
                       case ⟨ y > t ⟩ of
                       | true => let (zs,zss) = rec(ys)[true] in
                                 case b of
                                 | true => (y:zs;zss)
                                 | false => (nil;(y;zs)::zss)
                       | false => rec(ys)[false]
                     end
```

The program begins by casing on the input, and buffering in the head. If the head is less than the threshold, we simply continue. If the head is above the threshold, then we make a recursive call to compute the runs of the rest. If we are already on a run (`b` is true), we cons `y` onto the current head `zs`, and return `(y:zs;zss)`. If we were not on a run, then the initial prefix is empty, and we return `(nil;(y;zs):zss)`. The core terms for this program can be found at `Examples.hs:parseLL` and `Examples.hs:parseLLAux`.

### C.6 Partitioning

A crucial streaming idiom is partitioning, where a homogeneous stream of data is split into two or more parallel streams, which are then routed to different downstream nodes in the dataflow graph. The purpose of partitioning is to expose parallelism: the different downstream operators can be run separately, potentially on different physical machines. Depending on the situation, a programmer may choose to use different partitioning strategies. In $\lambda^{\mathrm{ST}}$, some common partitioning strategies are implementable as transformers.

*Round Robin Partitioning.* A round-robin partitioner fairly distributes an incoming stream of type $s^\star$ into a parallel pair of streams $s^\star \| s^\star$. It does this by sending the first element to the left branch, the second to the right, the third to the left, and so on. In $\lambda^{\mathrm{ST}}$, we write this by maintaining a Boolean accumulator, and negating after each item. If the Boolean is true, we send the element left, if it's false, we send it right.

```
roundRobin (xs : s⋆) : s⋆ ‖ s⋆ =
fix with (b = false).
  case xs of
  | nil => (nil, nil)
  | y :: ys => let (zs,ws) = rec(ys)[not b] in
               if b then (y :: zs,ws) else (zs, y :: ws)
```

The core term for this program can be found at `Examples.hs:roundRobin`

*Hash-Based Partitioning.* A hash-based partitioner routes stream elements based on a hash of their data, mod the number of routes $k$. We fix $k = 2$ for simplicity, and abstract this hashing function as a transformer `hm2` which accepts a stream of type $s$ and produces a stream of type $s+s$. Conceptually, we assume that this program computes a hash mod 2, and returns the input in the left or right position accordingly. Then we can write the hash-based partitioner like this:

```
sumRouter (xs : s⋆) : s⋆ ‖ s⋆ =
    fix. case xs of
         | nil => (nil, nil)
         | y :: ys => let (zs,ws) = rec(ys) in
                      case hm2(y) of
                      | inl(z) => (z :: zs, ws)
                      | inr(w) => (zs, w :: ws)
```

The core term for this program can be found at `Examples.hs:sumRouter`

### C.7 Windowing and Punctuation

Many kinds of windows have been considered in the literature. The most common windows are event-based — windows defined by the number of elements they'll contain — and time-based — windows which contain all the events from a fixed length of time. Windows can also be tumbling — the next window starts after the previous ends — or sliding — every event could begin a new window.

In $\lambda^{\mathrm{ST}}$, windowed operators are just `map`s over a stream whose elements are windows. Given a per-window stream transformer `f` which takes windows $s^\star$ to a result type $t$, and a "windowing strategy" `win` which takes a stream $r^\star$ and turns it into a stream of windows $(s^\star)^\star$, we can write a windowed operation of type $r^\star \to t^\star$ as follows: `xs : r⋆ |- map(f)(win(xs)) : t⋆`.

For example, if we wanted to compute a size-3 sliding sum of a stream of `Int`s, we would use a windower `win` which takes `Int⋆` to $(\texttt{Int}^\star)^\star$ where the inner streams are the windows, and `f` from `Int⋆` to `Int` is the sum operation.

Every per-window function commonly used in stream processing practice operates on entire windows at once, which is accomplished in $\lambda^{\mathrm{ST}}$ by `wait`-ing on the whole window, and then aggregating it with an embedded historical program. For this reason, we focus primarily on the window construction aspect.

*Fixed-Size Tumbling Windows.* The $k$-size tumbling windower creates windows of size $k$, where each new window starts immediately after the last window ended. For instance when $k = 2$, a stream $1, 2, 4, 7, 3, 8, \ldots$ turns into a stream $\langle 1, 2\rangle, \langle 4, 7\rangle, \langle 3, 8\rangle, \ldots$. The code for a fixed-size tumbling window is exactly the functional code for computing $k$-strides of a list using a fold. We maintain a counter of the current window size, and compute a concat-pair of the first (partial) window, and the windows of the remaining stream. The actual size-$k$ tumbling windower is computed by post-composing this transformer with a cons operation.

```
tumblingWindow (k : int) (xs : s⋆) : s⋆ · s⋆⋆ =
```

```
fix with (n = 0).
  case xs of
  | nil => (nil;nil)
  | y :: ys => if n + 1 >= k then let (cur;rest) rec(ys)[0] in (y :: nil; cur :: rest)
               else let (cur;rest) = rec(ys)[n+1] in  (y :: cur; rest)
```

The core term for this program can be found at `Examples.hs:tumblingWindow`

$k$-size window transformers can actually have the even stronger output type $(s^k)^\star$, where $s^k$ is the $k$-fold concatenation of $s$. If the window function being used requires that the windows all have exactly size $k$ (like taking pairwise differences for $k = 2$), this type can be used instead. The following program implements size-2 windows with this stronger type by casing two-deep into the stream at a time, and pairing up elements into concatenation pairs.

```
wind2 (xs : s⋆) : (s · s)⋆
    fix. case xs of
         | nil => nil
         | y :: ys => case ys of
                      | nil => nil
                      | z :: zs => (y;z) :: rec(zs)
```

The core term for this program can be found at `Examples.hs:sumRouter`

*Fixed-Size Sliding Windows.* A $k$-sized sliding windower produces a new window for each new element, including both the new element and the $k-1$ previous ones. The code for this windower keeps the current window under construction in memory. When each new stream element arrives, we emit the current window. For the first $k$ elements, we only add to the window. After $k$, we start evicting from the window.

```
slidingWindower (xs : s⋆) : s⋆⋆ =
    fix with (acc = []).
      case xs of
      | nil => acc :: nil
      | y::ys => wait y then
          let next = ⟨if |acc| < k then y :: acc else y :: (init acc)⟩ in
          ⟨next⟩ :: rec(ys)[next]
      end
```

The core term for this program can be found at `Examples.hs:slidingWindower`

*Punctuation-Based Windows.* Time-based windows are commonly implemented by way of *punctuation*: unit elements inserted into a stream to authoritatively mark that a period of time has ended. This is required because in the presence of network delays, it's impossible to know if a time period is over (and so a window can be emitted) or if there are more elements in the period to arrive. A punctuated stream has type $(1 + s)^\star$, where the punctuation events mark the end of each time period.

The following code computes a windowed stream $(s^\star)^\star$ from a punctuated stream $(1 + s)^\star$ by emitting windows which are the (potentially empty) runs of $s$s between punctuation marks. Like fixed-size tumbling windows, we generalize the return type to compute with a fold, and then the actual operator is defined by post-composing with a cons.

```
puncWindow (xs : (1+s)⋆) : s⋆ · s⋆⋆ =
    fix.
      case xs of
      | nil => (nil ; nil)
```

```
    | y :: ys => let (cur;rest) = rec(ys) in
                 case y of
                  | inl() => (nil ; cur :: rest)
                  | inr(z) => (z :: cur ; rest)
```

The core term for this program can be found at `Examples.hs:puncWindow`

*Merging Streams and Synchronizing Punctuation.* Parallel streams of star type can be *synchronized*, pairing off one element from one stream with one element of another. Given a stream of type $s^\star \| t^\star$, we can produce a stream of type $(s \| t)^\star$. This type's similarity to the standard functional program `zip` is more than just surface level: the program below has essentially the same code.

```
sync (xs : s*, ys : t*) :  (s ‖ t)* =
    fix.
      case xs of
      | nil => nil
      | x' :: xs' =>
        case ys of
        | nil => nil
        | y' :: ys' =>
          wait x',y'then
            ⟨(x',y')⟩ :: rec(xs,ys)
          end
```

The core term for this program can be found at `Examples.hs:sync`

Semantically, this program `waits` until a full element from each of the parallel input streams has arrived, sends them both out, and then continues with zipping the two tails. This is necessarily blocking: the output type guarantees that exactly one $s$ and $t$ will be produced before the next pair begins, and so we must wait for both to arrive before sending the other out. The upshot is that because this program is well typed in $\lambda^{\mathrm{ST}}$, it is necessarily deterministic. This gives us the deterministic merge operation that was needed to prevent the bug when averaging data from a pair of sensors in Section 2.

Moreover, for parallel streams of windows, synchronization enables databases-style *streaming joins* . Given parallel streams $(s^\star)^\star$ and $(t^\star)^\star$, we can synchronize to get $(s^\star \| t^\star)^\star$, and then apply a join operation to each parallel pair of windows.