Stream Types

JOSEPH W. CUTLER, University of Pennsylvania, USA
CHRISTOPHER WATSON, University of Pennsylvania, USA
EMEKA NKURUMEH, California Institute of Technology, 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 internal *parallel structure* of the stream to support deterministic stream processing on parallel and distributed systems. To these ends, we introduce *stream types*, with operators capturing sequential composition, parallel composition, and iteration, plus a core calculus λ^{ST} of *transformers* over typed streams which naturally supports a number of common streaming idioms, including punctuation, windowing, and parallel partitioning, as first-class constructions. λ^{ST} 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 illustrate the programming style supported by the rich types of λ^{ST}, we present a number of examples written in delta, a prototype high-level language design based on λ^{ST}.

Additional Key Words and Phrases: Type Systems, Stream Processing, Ordered Logic, Bunched Implication

ACM Reference Format:
Joseph W. Cutler, Christopher Watson, Emeka Nkurumeh, Phillip Hilliard, Harrison Goldstein, Caleb Stanford, and Benjamin C. Pierce. 2024. Stream Types. In . ACM, New York, NY, USA, 80 pages.

1 INTRODUCTION

What is the type of a stream? A straightforward answer, dating back to the early days of functional programming [17], 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 simple perspective has been immensely successful: the current programming models exposed by the most popular distributed stream processing eDSLs (e.g., Flink [18, 29], Beam [34], Storm [33], and Heron [30]) typically offer just one type, `Stream t`.

This homogeneous treatment of streams leaves something to be desired. 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 factors like network latency [56, 64]. Another issue with the homogeneous stream abstraction is that temporal patterns like *bracketedness* (every "begin" event has a following "end") or the fact that with exactly k events are expected to arrive on a stream are invisible in its type. Programmers get no help from the type system to ensure such properties when producing a stream, nor can they rely on them when consuming a stream.

Our principal contribution is a novel logical foundation for typed stream processing that can precisely describe streams with both complex sequential patterns and parallel structure. On this foundation, we build a calculus called λ^{ST} that is (a) expressive and type-safe for streams with such

PLDI'24, ,
2018.

complex temporal patterns and (b) deterministic, even when inputs can arrive from multiple sources in parallel. We also present delta, an experimental language design based on λ^{ST}. (A full-blown distributed implementation of delta is left for future work.)

Programs in λ^{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 λ^{ST} calculus is therefore designed to be *incremental*, producing partial outputs from partial inputs on the fly.

A λ^{ST} program is interpreted as a function mapping any prefix of its input(s) to a prefix of its output plus a "resultant" term 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 [42, 51], which syntactically describe partially ordered series/parallel data, and by work by Alur et al. [3] and Mamouras et al. [56], where streams are modeled as partially ordered sets. We discover a suitable proof theory for this two-product formalism in a variant of O'Hearn and Pym's Logic of Bunched Implications (BI) [60]. BI is well known as a foundation for separation logic [63], where its "separating conjunction" allows for local reasoning about separate regions of the heap in imperative programs. In λ^{ST}, we replace spatial separation with *temporal separation*: one product describes pairs of streams separated sequentially in time; the other describes pairs of temporally independent streams whose elements may arrive in interleaved fashion.

Concretely, our contributions are:

(1) We propose *stream types*, a static discipline for distributed stream processing that generalizes the traditional homogeneous view of streams to a richer nested-parallel-and-sequential structure, and define λ^{ST}, a calculus of stream processing transformers inspired by a Curry-Howard-like correspondence with an ordered variant of BI. Terms in λ^{ST} are high level programs in a functional style that conceptually transform whole streams at once.
(2) We equip λ^{ST} with an operational semantics interpreting terms as incremental transformers that accept and produce finite prefixes of streams. Our main result is a powerful *homomorphism theorem* (Theorem 3.2) guaranteeing that the result of a transformer does not depend on how the input stream is divided into prefixes. This theorem implies that the semantics is deterministic: all interleavings of parallel sub-streams yield the same final result.
(3) We present delta, an experimental high-level functional language prototype based on the λ^{ST} calculus that serves as a tool for exploring the potential of richly typed stream programming. We demonstrate by example how delta enables type-safe programming for streams with complex patterns and how it prevents nondeterminism. 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.

Section 2 explores some concrete cases where λ^{ST}'s structured types can prevent common stream processing bugs and enable cleaner programming patterns. Section 3 presents Kernel λ^{ST}, a minimal subset with just the features needed to state and understand the main technical results. Section 4 extends this presentation to Full λ^{ST}. Section 5 develops several further examples. Sections 6 and 7 discuss related and future work. An overview of our prototype implementation of delta can be found in Appendix A; technical details omitted from the main paper in Appendix C; and expanded versions of the examples in Appendix B.

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. Conventional stream processing systems would give this transformation a type like `Stream Int` $\longrightarrow$ `Stream (Start + Int + Stop)`, which expresses only the types of events in the output, not the temporal invariant that the `Start` must come before all the data and the `Stop` after.

These simple types are even more problematic when *consuming* streams. Suppose another transformer wants 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 second transformer must *re-parse* the stream to compute the averages, requiring additional logic for various special cases (e.g., `Stop` before `Start`, empty `Start`/`Stop` pairs) that cannot actually occur in the stream it will see.

In λ^{ST}, we can express the required 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 this invariant; conversely, a downstream transformer can assume that its input will adhere to it.

Enforcing deterministic parallelism. A second limitation of homogeneous streams is that they impose a *total ordering* on their component events. In other words, for each pair of events in the stream, the transformer can tell which came first. This is problematic in a world where stream transformers work over data that is logically only partially ordered—e.g., because it comes from separate sources.[1]

For example, consider a system with two sensors, each producing one reading per second and and sending them via 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 into a single one, 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 come from the same sensor, after which the averages will all be bogus.

The problem with this transformer is that it is not deterministic: its result can depend on external factors like network latency. Bugs of this type can easily occur in practice [56, 64] and can be very difficult to track down, since they may only manifest under rare conditions [49].

Once again, this is a failure of type structure. In λ^{ST}, we can prevent it by giving 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 (indeed, all well-typed λ^{ST} programs are, as we show in Section 3.3); operationally, it waits for events to arrive on both of its input streams before sending them along as a pair.

[1] The same objection applies for stream processing systems that impose total *per key* ordering of a parallelized stream—cf. `KeyedStream` in Flink—since data associated with a given key may also come from multiple sources in parallel.

3 KERNEL λ^{ST}

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

The *concatenation* constructor $\cdot$ describes streams that *vary* over time: if s and t are stream types, then $s \cdot t$ describes a stream on which all the elements of s arrive first, followed by the elements of t. A producer of a stream of type $s \cdot t$ must first produce a stream of type s and then a stream of type t, while a consumer can assume that the incoming data will first consist of data of type s and then of type t. The transition point between the s and t parts is handled automatically by λ^{ST}'s semantics: the underlying data of a stream of type $s \cdot t$ includes a *punctuation marker* [72] indicating the cross-over. One consequence of this is that, unlike Kleene Star for regular languages, 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.

On the other hand, the *parallel* stream type $s \| t$ describes a stream with two parallel substreams of types s and t. 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 parts independently. To enable this, each element in the 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 $\texttt{Int}^\star$.

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 interleaved 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 ε, 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 λ^{ST} transformers that convert between them.

In summary, the Kernel λ^{ST} stream types are given by the grammar on the top left in Figure 1. (So far, these types can only describe streams of fixed, finite size. In Section 4.2 we will enrich the kernel type system with unbounded streams via the Kleene star type $s^\star$.)

What about terms? Recall that 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 stream transformers accepting a stream described by s and producing one described by t. A term e runs by accepting some inputs as described by s, producing some outputs as described by t and then stepping to a new term e' with an updated type, ready to accept the rest of the input and produce the rest of the output. This process happens reactively: output is only produced when an input arrives. The formal semantics of λ^{ST} are described in Section 3.2.

To represent stream transformers with multiple parallel and sequential inputs, 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 a stream of either of these types contains both the data of a stream of type s and a stream of type t—although the temporal structure differs between the two. A standard observation from proof

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

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

$$\frac{\Gamma \vdash e_1 : s \qquad \Delta \vdash e_2 : t}{\Gamma\,;\Delta \vdash (e_1; e_2) : s \cdot t}\ \text{T-Cat-R} \qquad \frac{\Gamma(x : s\,; y : t) \vdash e : r}{\Gamma(z : s \cdot t) \vdash \texttt{let}_t\ (x; y)\ =\ z\ \texttt{in}\ e : r}\ \text{T-Cat-L}$$

$$\frac{}{\Gamma \vdash \texttt{sink} : \varepsilon}\ \text{T-Eps-R} \qquad \frac{}{\Gamma \vdash () : 1}\ \text{T-One-R} \qquad \frac{}{\Gamma(x : s) \vdash x : s}\ \text{T-Var}$$

$$\frac{\Gamma \leq \Gamma' \qquad \Gamma' \vdash e : s}{\Gamma \vdash e : s}\ \text{T-SubCtx}$$

Fig. 1. Kernel λ^{ST} syntax and typing rules

theory is that, in situations where a logic or type theory includes two products, the corresponding typing judgment requires a context with two *context formers*.[2]

The first context former, written with a comma (Γ, Δ), describes inputs to a transformer arriving in parallel, one component structured according to Γ and the other according to Δ. The second context former, written with a semicolon ($\Gamma\,;\Delta$) describes inputs that will first arrive from the environment according to Γ, then according to Δ.

These interpretations are enforced by restricting the ways that these contexts can be manipulated using *structural rules*. Comma contexts can be manipulated in all the ways standard contexts can: they can be reordered—from Γ, Δ to Δ, Γ — duplicated, and dropped. Semicolon contexts, on the other hand, are ordered and affine: a context $\Gamma\,;\Delta$ cannot be freely rewritten to a context $\Delta\,;\Gamma$, and we cannot duplicate a context Γ into one like $\Gamma\,;\Gamma$. These restrictions enforce the meaning of $\Gamma\,;\Delta$ as data arriving according to Γ and then Δ: to exchange them would be to allow a consumer to assume that the data is sent in the opposite order, and to duplicate is to assume that the data input will be replayed.

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 each. The full list of structural rules can be found in Appendix C.6. Formally, stream contexts are drawn from the grammar at the top right of Figure 1.

3.1 Kernel Typing Rules

The typing rules for Kernel λ^{ST} are collected in Figure 1. The typing judgment, written $\Gamma \vdash e : s$, says that e is a stream transformer from a collection of streams structured like Γ, to a single stream structured like s.

The most straightforward typing rule is the right rule for parallel (T-Par-R). It says that, from a context Γ, we can produce a stream of type $s \| t$ by producing s and t independently from Γ, using transformers e_1 and e_2. We write the combined transformer as a "parallel pair" (e_1, e_2). Semantically,

[2]Such *bunched* contexts were first introduced in the logic of Bunched Implication [60], the basis of modern separation logic [63]. Our bunched contexts differ from those of BI by the choice of structural rules: our substructural type former is affine ordered, while the BI one is linear.

it operates by copying the inputs arriving on Γ, passing the copies to e_1 and e_2, and pairing up the outputs into a parallel stream. Similarly, the T-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 (Γ) must arrive before the part that e_2 depends on (Δ). Semantically, this term will operate by accepting data from the Γ part of the context and running e_1; once the Γ part is used up and the Δ 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 that deconstruct variables of type $s \cdot t$ (or $s \| t$) as pairs of variables of type s and t, connected by ; (or ,). 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 Δ. In particular, $\Gamma(x : s)$ is a context with a distinguished variable x.

The T-Par-L rule says that if z is a variable of type $s \| t$ somewhere in the context, we can replace its binding with with a pair of bindings for variables x and y of types s and t and use these in a continuation term e of final 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 rule T-Cat-L says that, if a variable z of type $s \cdot t$ appears somewhere in the context, it can be let-bound to a pair of variables x and y of types s and t that 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 and be processed first, followed by the substream bound to y.

T-Eps-R and T-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 Γ and produces no output. On the other hand, `()` emits a unit value as soon as it receives its first input and never emits anything else.

The variable rule (T-Var) says that, if $x : s$ is a variable somewhere in the context, then we can simply send it along the output stream. Semantically, it works by dropping everything in the context except for the s-typed data for x, which it forwards along.

The rule T-SubCtx bundles together all of the structural rules as a subtyping relation on contexts. For example, the weakening rule for semicolon contexts is written, $\Gamma ; \Delta \leq \Gamma$ and the comma exchange rule is $\Gamma , \Delta \leq \Delta , \Gamma$.

Examples and Non-Examples. To show the typing rules in action, here are two small examples of transformers written in Kernel λ^{ST}, as well as three examples of programs which are *rejected* by the type system. The first example is a simple "parallel-swap" transformer, which accepts a stream z of type $s \| t$, and outputs a stream of type $t \| s$, swapping the "positions" of the parallel substreams:

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

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 first and most important *non*-example is the lack of a corresponding "cat-swap" term, which would accept a stream z of type $s \cdot t$, and produce a stream of type $t \cdot s$. This program is undesirable because it is not implementable without a space leak. Implementing it requires the entire stream of type s to be saved in memory to emit it after the stream of type t.[3] The

$$\dfrac{\dfrac{\dfrac{\lightning}{x : s \vdash y : t}\ \text{T-Var} \quad \dfrac{\lightning}{y : t \vdash x : s}\ \text{T-Var}}{x : s ; y : t \vdash (y; x) : t \cdot s}\ \text{T-Cat-R}}{z : s \cdot t \vdash \texttt{let}_s\ (x; y)\ =\ z\ \texttt{in}\ (y; x) : t \cdot s}\ \text{T-Cat-L}$$

[3] A program with this behavior is implementable in λ^{ST}, but requires a special program construct—see Section 4.5—ensuring that leaky programs like this one cannot be written accidentally.

$$\frac{}{\texttt{epsEmp} : \texttt{prefix}\,(\varepsilon)} \qquad \frac{}{\texttt{oneEmp} : \texttt{prefix}\,(1)} \qquad \frac{}{\texttt{oneFull} : \texttt{prefix}\,(1)}$$

$$\frac{p : \texttt{prefix}\,(s) \quad p' : \texttt{prefix}\,(t)}{\texttt{parPair}(p, p') : \texttt{prefix}\,(s \| t)} \qquad \frac{p : \texttt{prefix}\,(s)}{\texttt{catFst}(p) : \texttt{prefix}\,(s \cdot t)} \qquad \frac{p' : \texttt{prefix}\,(t) \quad p : \texttt{prefix}\,(s) \quad p\ \texttt{maximal}}{\texttt{catBoth}(p, p') : \texttt{prefix}\,(s \cdot t)}$$

Fig. 2. Prefixes for Types

natural term for this program would be $\texttt{let}_s\ (x; y)\ =\ z\ \texttt{in}\ (y; x)$, but this does not typecheck, as attempting to build a derivation gets stuck. Applying the syntax-directed rules gets us to a point where we must show that y has type t in a context with only x, and that x has type s in a context with only y. This is because the T-Cat-R splits the context, but the variables are listed in the opposite order than we'd need. The lack of a structural rule to let us permute the x and y in the context means that there is nothing to do here, and so a typechecker will reject this program.

The second example is a "broadcast" transformer, which takes a variable $x : s$ and outputs a stream of type $s \| s$, duplicating the variable of type s, and sending it out to two parallel outputs: $x : s \vdash (x, x) : s \| s$.

Another non-example is the "replay" transformer, which would take a variable $x : s$ and produce a stream $s \cdot s$ which repeats the input stream twice. This is the concat type equivalent of the broadcast transformer, but it is undesirable for the same reason as the cat-swap program: it would require saving the entire incoming stream of type s in order to replay it. This time, the failure of the natural term $(x; x)$ to typecheck is down to a lack of contraction rule for semicolon contexts: we are not permitted to turn a context $x : s$ into a context $x : s\,; x : s$.

The last non-example is a "tie-breaking" transformer, which would take a stream $z : \texttt{Int} \| \texttt{Int}$ of two ints in parallel and produce a stream of type Int, forwarding along the Int that arrived first. This program (like others that require inspecting the interleaving of data in a stream of type $s \| t$) is not expressible. In Section 3.3, we'll prove that a well-typed program cannot implement this behavior.

3.2 Prefixes and Semantics

Next we define the semantics of Kernel λ^{ST}. The natural notion of "values" in this semantics is finite prefixes of streams: the meaning of a well-typed term $\Gamma \vdash e : s$ is a function that accepts an environment mapping variables in Γ to prefixes of streams and produces a prefix of a stream of type s.

Because the streams that λ^{ST} programs operate over are more structured than traditional homogeneous streams—including cross-over punctuation in streams of type $s \cdot t$ and disambiguating tags in streams of type $s \| t$—the prefixes are also more structured. A prefix in λ^{ST} is not a simple sequence of data items, but a structured value whose possible shapes are determined by its type.

For example, there are two prefixes of a stream of type 1: the empty prefix, written oneEmp, and the prefix containing the single element (), written oneFull. Similarly, the unique stream of type ε has a single prefix, the empty prefix, which we write epsEmp.

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 $\texttt{parPair}(p_1, p_2)$, where p_1 is a prefix of a stream of type s, and p_2 is a prefix of a stream of type t. Crucially, this definition encodes no information about any interleaving of p_1 and p_2: the prefix $\texttt{parPair}(p_1, p_2)$ equally represents a situation where all of p_1 arrived first and then all of p_2, one where p_2 arrived before p_1, and many others where the elements of p_1 and p_2 arrived in some interleaved order. In a nutshell, this

$$\frac{}{\eta \;:\; \mathsf{env}\,(\cdot)} \qquad \frac{\eta(x) \mapsto p \qquad p \;:\; \mathtt{prefix}\,(s)}{\eta \;:\; \mathsf{env}\,(x : s)} \qquad \frac{\eta \;:\; \mathsf{env}\,(\Gamma) \qquad \eta \;:\; \mathsf{env}\,(\Delta)}{\eta \;:\; \mathsf{env}\,(\Gamma\,,\Delta)}$$

$$\frac{\eta \;:\; \mathsf{env}\,(\Gamma) \qquad \eta \;:\; \mathsf{env}\,(\Delta) \qquad (\eta\;\mathtt{maximalOn}\;\Gamma) \vee (\eta\;\mathtt{emptyOn}\;\Delta)}{\eta \;:\; \mathsf{env}\,(\Gamma\,;\Delta)}$$

Fig. 3. Environments for Contexts

definition is what guarantees deterministic processing. By representing all possible 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 cuts off before reaching the point where the $s \cdot t$ stream stops carrying elements of s and starts on t. We write such a prefix as $\mathtt{catFst}(p)$, 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 "completed" prefix of s plus a prefix of t. We write this as $\mathtt{catBoth}(p, p')$, with p a prefix of s and p' a prefix of t. The requirement that p be completed is formalized by the judgment $p\ \mathtt{maximal}$, which ensures that the prefix p describes an entire completed stream (see Appendix C.3). We formalize all these possibilities as a judgment $p \;:\; \mathtt{prefix}\,(s)$, shown in Figure 2.

Every type s has a distinguished *empty prefix*, written $\mathtt{emp}_s$ and defined by straightforward recursion on s (see Appendix C.1). We lift the idea of prefixes from types to contexts, defining an *environment* η for a context Γ to be a mapping from the variables $x : s$ in Γ to prefixes of the corresponding types s. We write this with a judgment $\eta \;:\; \mathsf{env}\,(\Gamma)$ (Figure 3). Along with ensuring that η has well-typed bindings for all variables, the judgment ensures that the prefixes respect the order structure of the context. In particular, an environment η for a semicolon context $\Gamma\,;\Delta$ must assign prefixes *in order*: the prefixes for Γ, the earlier part of the context, must all be maximal before the prefixes Δ can begin. In other words, either η assigns maximal prefixes to every variable in Γ—which we write $\eta\;\mathtt{maximalOn}\;\Gamma$—or η assigns empty prefixes to every variable in Δ—which we write $\eta\;\mathtt{emptyOn}\;\Delta$.

One might worry that these structured stream prefixes might be incompatible with a future distributed implementation atop an existing stream processing substrate. Fortunately, they are not: by viewing a λ^{ST} stream as a series of single-event prefixes, each consisting of a data item plus some extra tag bits, we recover the traditional homogeneous view. Moreover, this wire representation incurs only a constant overhead: the maximum size of the tag bits on a stream element of type s is bounded by the syntactic depth of s (See Appendix C.11).

Semantics. We describe how well-typed λ^{ST} terms execute with an operational semantics. Given a well-typed term $\Gamma \vdash e : s$ and an input environment $\eta \;:\; \mathsf{env}\,(\Gamma)$, the semantics describes how to run e with η to produce an output prefix p. It also describes how to produce a "resultant" term e', whose purpose is to continue the computation once further data on the input stream data arrives. Formally the semantics is given by a judgment $\eta \Rightarrow e \downarrow e' \Rightarrow p$, which we pronounce "running the core term e on the input environment η yields the output prefix p and steps to e'." The rules for this judgment are gathered in Figure 4 and described below; the full set of rules for all of λ^{ST} can be found in Appendix C.9.

The following theorem establishes the soundness of the Kernel λ^{ST} semantics, formalizing the intuitive description given above: If we run a well-typed core term e on an environment η of the context type, it will return a prefix p with the result type s, and step to a term e' which is well typed

$$\frac{\eta(x) \mapsto p}{\eta \Rightarrow x \downarrow x \Rightarrow p}\ \text{S-Var}$$

$$\frac{\eta \Rightarrow e_1 \downarrow e_1' \Rightarrow p_1 \qquad \eta \Rightarrow e_2 \downarrow e_2' \Rightarrow p_2}{\eta \Rightarrow (e_1, e_2) \downarrow (e_1', e_2') \Rightarrow \mathtt{parPair}(p_1, p_2)}\ \text{S-Par-R}$$

$$\frac{\eta(z) \mapsto \mathtt{parPair}(p_1, p_2) \qquad \eta[x \mapsto p_1, y \mapsto p_2] \Rightarrow e \downarrow e' \Rightarrow p'}{\eta \Rightarrow \mathtt{let}\ (x,y)\ =\ z\ \mathtt{in}\ e \downarrow \mathtt{let}\ (x,y)\ =\ z\ \mathtt{in}\ e' \Rightarrow p'}\ \text{S-Par-L}$$

$$\frac{\eta \Rightarrow e_1 \downarrow e_1' \Rightarrow p \qquad \neg\,(p\ \mathtt{maximal})}{\eta \Rightarrow (e_1; e_2) \downarrow (e_1'; e_2) \Rightarrow \mathtt{catFst}(p)}\ \text{S-Cat-R-1}$$

$$\frac{\eta \Rightarrow e_1 \downarrow e_1' \Rightarrow p \qquad p\ \mathtt{maximal} \qquad \eta \Rightarrow e_2 \downarrow e_2' \Rightarrow p'}{\eta \Rightarrow (e_1; e_2) \downarrow e_2' \Rightarrow \mathtt{catBoth}(p, p')}\ \text{S-Cat-R-2}$$

$$\frac{\eta(z) \mapsto \mathtt{catFst}(p) \qquad \eta[x \mapsto p, y \mapsto \mathtt{emp}_t] \Rightarrow e \downarrow e' \Rightarrow p'}{\eta \Rightarrow \mathtt{let}_t\ (x;y)\ =\ z\ \mathtt{in}\ e \downarrow \mathtt{let}_t\ (x;y)\ =\ z\ \mathtt{in}\ e' \Rightarrow p'}\ \text{S-Cat-L-1}$$

$$\frac{\eta(z) \mapsto \mathtt{catBoth}(p, p') \qquad \eta[x \mapsto p, y \mapsto p'] \Rightarrow e \downarrow e' \Rightarrow p''}{\eta \Rightarrow \mathtt{let}_t\ (x;y)\ =\ z\ \mathtt{in}\ e \downarrow \mathtt{let}\ x = \mathtt{sink}_p\ \mathtt{in}\ e'[z/y] \Rightarrow p''}\ \text{S-Cat-L-2}$$

$$\frac{}{\eta \Rightarrow \mathtt{sink} \downarrow \mathtt{sink} \Rightarrow \mathtt{epsEmp}}\ \text{S-Eps-R}$$

$$\frac{}{\eta \Rightarrow () \downarrow \mathtt{sink} \Rightarrow \mathtt{oneFull}}\ \text{S-One-R}$$

Fig. 4. Incremental semantics of Kernel λ^{ST}

in context "the rest of" Γ after η and has type "the rest of" s after p. The "rest" of a type/context after a prefix/environment is, intuitively, its *derivative* with respect to the prefix/environment, in the sense of standard Brzozowski derivatives of regular expressions [16] — we make this formal in Section 3.2.1. Most critically, the types of the variables in e and e' are different: if x has type s in e, then x has type $\delta_{\eta(x)} s$ in e', having already consumed $\eta(x)$.

Theorem 3.1 (Soundness of the Kernel λ^{ST} Semantics). *Suppose:* $\Gamma \vdash e : s$ *and* $\eta\ :\ \mathsf{env}\,(\Gamma)$. *Then, there are* p *and* e' *such that* $\eta \Rightarrow e \downarrow e' \Rightarrow p$, *with* $p\ :\ \mathtt{prefix}\,(s)$ *and* $\delta_\eta\,(\Gamma) \vdash e' : \delta_p\,(s)$

Appendix C.9 presents the proof of the soundness theorem for Full λ^{ST} (see Section 4).

In light of this theorem, our operational semantics can be thought of as defining a reactive state machine. Well typed terms $\Gamma \vdash e : s$ are the states, while the semantics judgment defines the transition function: when new inputs η arrive, we step the semantics to produce an output prefix p, and step to a new state $\delta_\eta\,(\Gamma) \vdash e' : \delta_p\,(s)$. This form of semantics — a state machine with terms themselves as states, typed by derivatives — predates our work, having been pioneered by the Esterel programming language [14].

Semantics of the Right Rules. The right rules for parallel and concatenation are the simplest to understand. For S-Par-R, we accept an environment η 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 $\texttt{parPair}(p_1, p_2)$.

There are two rules, S-Cat-R-1 and S-Cat-R-2, for running the concatenation pair $(e_1; e_2) : s \cdot t$. In either case, we begin by running e_1 with the environment η, producing a prefix p and term e_1'. If p is not maximal, we stop there: future inputs will allow the first component to produce the rest of s, so it is not yet time to start running e_2 to produce t. This case is handled by S-Cat-R-1, where the resulting term is $(e_1'; e_2)$ and the output prefix is $\texttt{catFst}(p)$.

On the other hand, if p is maximal, then we run e_2, which steps to e_2' and produces a prefix p' using rule S-Cat-R-2, where the entire term then outputs $\texttt{catBoth}(p, p')$, 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'. This is because we are done producing the s part of the $s \cdot t$, and so a subsequent step of evaluation only has to run e_2' to produce the rest of the t.

Semantics of Variables. The variable semantics S-Var is a simple variable lookup. We look up the prefix bound to the variable x in the environment, return it, and then step to x itself.

Semantics of Left Rules. The semantics of the left rules for concatenation and parallel are similar, both accepting an environment η with a binding for $z : s \otimes t$, where $\otimes$ is one of the two products, binding variables x and y of type s and t to the two components of the product, and using the updated environment to run the continuation term.

In the case of the semantics of the left rule for parallel (S-Par-L), looking up z of type $s \| t$ will always yield a prefix $\texttt{parPair}(p_1, p_2)$. The rule binds p_1 to x and p_2 to y and runs the continuation term, stepping to e' and producing the output prefix p. Then the whole term steps to $\texttt{let}\ (x, y) = z\ \texttt{in}\ e'$ and produces p.

The left rule for concatenation has two cases, depending on what kind of prefix comes back from the lookup for z. If the lookup yields is $\texttt{catFst}(p)$, the rule S-Cat-L-1 applies. Since no data for y has arrived, we bind y to $\texttt{emp}_t$, the empty prefix of type t, and run the continuation [4]. If the result comes back as $\texttt{catBoth}(p, p')$, then the rule S-Cat-L-2 applies, so we run the continuation with p and p' bound to x and y.

Both rules output the prefix resulting from running the continuation, but they step to different resulting terms. If $\eta(z) = \texttt{catFst}(p)$, then the resulting term must be another use of Cat-L: the variable z still expects to get some more of the first component of the concatenation, and then the second component. If $\eta(z) = \texttt{catBoth}(p, p')$ on the other hand, the z stream has crossed over to the second part. In this case, we close over the (now not-needed) x variable in e', and connect z to the y input of e' by substituting y for z.

3.2.1 Derivatives. When $p : \texttt{prefix}\,(s)$, we write $\delta_p\,(s)$ for the derivative [16] of s by p: the type of streams that result after a prefix of type p has been "chopped off" the beginning of a stream of type s. Because this operation is partial—$\delta_p\,(s)$ is only defined when $p : \texttt{prefix}\,(s)$—we formally define this as a a 3-place relation, written as $\delta_p\,(s) \sim s'$ and pronounced as "the derivative of s with respect to p is s'" (see Figure 5).

The derivative of the type 1 with respect to the empty prefix oneEmp is 1 (the rest of the stream is the entire stream), and its derivative with respect to the full prefix oneFull is ε (there is no more stream left after the unit element has arrived). For parallel, the derivative is taken component-wise.

[4] This need to compute $\texttt{emp}_t$ at runtime to bind to y is the reason that the term for T-Cat-L, $\texttt{let}_t\ (x; y) = z\ \texttt{in}\ e$, includes a t in the syntax. In Section 4, the case analysis expressions for star types and sum types will have similar annotations for the same reason.

$$\frac{}{\delta_{\texttt{epsEmp}}\,(\varepsilon) \sim \varepsilon} \qquad \frac{}{\delta_{\texttt{oneEmp}}\,(1) \sim 1} \qquad \frac{}{\delta_{\texttt{oneFull}}\,(1) \sim \varepsilon} \qquad \frac{\delta_p\,(s) \sim s'}{\delta_{\texttt{catFst}(p)}\,(s \cdot t) \sim s' \cdot t}$$

$$\frac{\delta_{p'}\,(t) \sim t'}{\delta_{\texttt{catBoth}(p,'p)}\,(s \cdot t) \sim t'} \qquad \frac{\delta_p\,(s) \sim s' \qquad \delta_{p'}\,(t) \sim t'}{\delta_{\texttt{parPair}(p,p')}\,(s \| t) \sim s' \| t'}$$

Fig. 5. Derivatives

The interesting cases are those for the concatenation type. If the prefix has the form $\texttt{catFst}(p)$, the derivative $\delta_{\texttt{catFst}(p)}\,(s \cdot t)$ is $\big(\delta_p\,(s)\big) \cdot t$, i.e., some of the s has gone by but not all, and once it does we still expect t to come after it. On the other hand, if the prefix has the form $\texttt{catBoth}(p, p')$, the derivative $\delta_{\texttt{catBoth}(p,p')}\,(s \cdot t)$ 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'.

This definition gets lifted to contexts and environments pointwise: if $x : s$ is a variable in Γ, the derivative of $\delta_\eta\,(\Gamma)$ has $x : \delta_{\eta(x)}\,(s)$ in the same location.

3.3 The Homomorphism Property and Determinism

The semantics is designed to run a stream transformer on an "input chunk" of any size, from individual input events one at a time all the way up to the entire stream at once. The cost of this flexibility is that it raises the question of *coherence*—i.e., 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 is a corollary of our main technical result, a *homomorphism theorem* that says running a term e on an environment η and then running the resulting term e' on an environment η' of appropriate type produces the same end result as running e on the combined environment.

Theorem 3.2 (Homomorphism Theorem). *Suppose (1)* $\Gamma \vdash e : s$*, (2)* $\eta : \texttt{env}\,(\Gamma)$*, (3)* $\eta' : \texttt{env}\,\big(\delta_\eta\,(\Gamma)\big)$*, (4)* $p : \texttt{prefix}\,(s)$*, (5)* $p' : \texttt{prefix}\,\big(\delta_p\,(s)\big)$*, (6)* $\eta \Rightarrow e \downarrow e' \Rightarrow p$*, and (7)* $\eta' \Rightarrow e' \downarrow e'' \Rightarrow p'$*. Then, if* $\eta \cdot \eta' \Rightarrow e \downarrow e''' \Rightarrow p''$*, we have* $p'' = p \cdot p'$*, and* $e''' = e'$

(The proof goes by induction on the derivation of $\eta \Rightarrow e \downarrow e' \Rightarrow p$, inverting everything in sight. See Appendix C.9 for full details.) The operation $p \cdot p'$ here is *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, this is defined as a 4-place partial inductive relation $p \cdot p' \sim p''$, which is defined when p and p' have types s and $\delta_p\,(s)$, respectively. The operation $\eta \cdot \eta' \sim \eta''$ does the same for environments. See Appendix C.4.

The homomorphism theorem not only justifies running the semantics of prefixes of any size; 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 η is an environment for Γ, Γ'. Write $\eta_1 = \eta|_\Gamma$ and $\eta_2 = \eta|_{\Gamma'}$, for the restrictions of η to the variables in Γ and Γ', respectively. Now, there are two different ways of running e on this data. One is to first run e on $\eta_1 \cup \texttt{emp}_{\Gamma'}$ (which has η_1 bindings for Γ and then the empty prefix for everything in Γ') and then run the resulting term on $\eta_2 \cup \texttt{emp}_\Gamma$ (with an empty prefixes for Γ). The other does the opposite, first running e on $\eta_2 \cup \texttt{emp}_\Gamma$ and then running the resulting term on $\eta_1 \cup \texttt{emp}_{\Gamma'}$. Determinism says that these strategies produce equal results. It is proved in Appendix C.10 by observing that the homomorphism theorem guarantees that each of these options is equivalent to running e on η.

Theorem 3.3 (Determinism). *Suppose (1)* $\Gamma, \Gamma' \vdash e : s$*, (2)* $\eta : \mathtt{prefix}(\Gamma, \Gamma')$*, (3)* $\eta|_\Gamma \cup \mathtt{emp}_{\Gamma'} \Rightarrow e \downarrow e_1 \Rightarrow p_1$ *and* $\eta|_{\Gamma'} \cup \mathtt{emp}_\Gamma \Rightarrow e_1 \downarrow e_2 \Rightarrow p_2$*, (4)* $\eta|_{\Gamma'} \cup \mathtt{emp}_\Gamma \Rightarrow e \downarrow e_1' \Rightarrow p_1'$ *and* $\eta|_\Gamma \cup \mathtt{emp}_{\Gamma'} \Rightarrow e_1' \downarrow e_2' \Rightarrow p_2'$*. Then* $e_2 = e_2'$ *and* $p_1 \cdot p_2 = p_1' \cdot p_2'$*.*

To intuitively see how this theorem follows from homomorphism, note that prefixes are canonical representatives of equivalence classes of sequences of stream elements, up to the possible reorderings defined by their type [67]. 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 the combined normal forms from the two steps.

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

We now sketch the remaining types and terms of λ^{ST} that are not part of Kernel λ^{ST}.

4.1 Sums

Sum types in λ^{ST}, written $s + t$, 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 tell which. Streams of type s are not the same as streams of type $s + s$, and streams of type $s + t$ are isomorphic to, but not identical to, streams of type $t + s$. Operationally, a producer of a sum stream sends a tag bit before sending the rest of the stream, to tell downstream consumers which side to expect. Conversely, a consumer of $s + t$ first reads the bit to learn which it is getting next.

A prefix of $s + t$ can be a prefix of one of s or one of t, written $\mathtt{sumInl}(p)$ or $\mathtt{sumInr}(p)$, or it can be $\mathtt{sumEmp}$, the empty prefix of type $s + t$, which does not even include the initial tag bit. The derivatives with respect to these prefixes are defined as follows: (a) the empty prefix takes nothing off the type ($\delta_{\mathtt{sumEmp}}\ (s + t) = s + t$) and (b) the two injections reduce to taking the derivative of the corresponding branch of the sum ($\delta_{\mathtt{sumInl}(p)}\ (s + t) = \delta_p\ (s)$ and $\delta_{\mathtt{sumInr}(p)}\ (s + t) = \delta_p\ (t)$).

The typing rules for sums are the normal injections

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

on the right (T-Sum-R-1 and a symmetric rule T-Sum-R-2) and a case analysis rule on the left (T-Sum-L-Surf). The right rules operate by prepending their respective tags and then running the embedded terms. The left rule does case analysis: if the incoming stream z comes from the left of the sum, it is processed with e_1; if from the right, e_2. To run a sum case term, the semantics must dispatch on the tag that says if the stream z being destructed is a left or a right. But the prefix z might not include a tag, if only data from the surrounding context has arrived. In this case, z will map to $\mathtt{sumEmp}$, and we have no way of determining which branch to run. The solution is to run neither! Instead, we hold on to the environment, saving *all* incoming data to the program until the tag arrives. Once we get a prefix that includes the tag, we continue by running the corresponding branch with the accumulated inputs. Note that this buffering is necessarily a blocking operation. [5]

All this requires a slightly generalized typing rule (T-Sum-L) that includes a *buffer environment* $\eta : \mathtt{env}\ (\Gamma(z : s + t))$ of the context type in the term. This buffer holds all of the input data we've seen so far. As prefixes arrive, we append to this buffer until we get the tag.

$$\frac{\eta : \mathtt{env}\ (\Gamma(z : s + t)) \qquad \Gamma(x : s) \vdash e_1 : r \qquad \Gamma(y : t) \vdash e_2 : r}{\delta_\eta\ (\Gamma(z : s + t)) \vdash \mathtt{case}_r\ (\eta; z, x.e_1, y.e_2) : r}\ \textsc{T-Sum-L}$$

[5] Depending on the rest of the context, it could also require unbounded memory! Fortunately, we can easily detect this, and flag it as a warning to the user: running a case on $z : s + t$ in a context $\Gamma(z : s + t)$ could require buffering all variables to the left of z or in parallel with z in the context. Unbounded memory is required if and only if any of those variables have star type.

Accordingly, the context in this rule is $\delta_\eta\ (\Gamma(z : s + t))$: the term is typed in the context consisting of everything after the part of the stream that has so far been buffered.

Fortunately, the only typing rule that a λ^{ST} programmer needs to concern themselves with is T-Sum-L-Surf. While writing the program, and before it runs, the buffer is empty ($\eta = \mathtt{emp}_{\Gamma(z:s+t)}$). In this case, the $\delta_\eta\ (\Gamma(z : s + t)) = \Gamma(z : s + t)$, and so the generalized rule T-Sum-L simplifies to the "surface" rule, T-Sum-L-Surf. Full details can be found in Appendix C.9.

4.2 Star

Full λ^{ST} also includes a type constructor for unbounded streams, written $s^\star$ because it is inspired by the Kleene star from the theory of regular languages. (We do not need to distinguish between unbounded finite streams and "truly infinite" ones, because our operational semantics is based on prefixes: we're always only operating on "the first part" of the input stream, and it doesn't matter whether the part we haven't seen yet is finite or infinite.) The type $s^\star$ describes a stream that consists of zero or more sub-streams of type s, in sequence.

In ordinary regular languages, $r^\star$ is equal to $\varepsilon + r \cdot r^\star$. In the language of stream types, this equation says that a stream of type $s^\star$ is either empty (ε) 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 $x \mapsto \varepsilon + s \cdot x$. The definitions of prefixes and typing rules for star all follow from this perspective.

In particular, $\mathtt{prefix}(s^\star) = \mathtt{prefix}(\varepsilon + s \cdot s^\star)$. The empty prefix of type $s^\star$, written `starEmp`, 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 `starDone`. It corresponds to the left injection of the sum, and receiving it means that the stream has ended. Note that, despite containing no s 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 $\mathtt{starFirst}(p)$, with p a prefix of s, or $\mathtt{starRest}(p, p')$, with p a maximal prefix of type s and p' another prefix of $s^\star$.

For derivatives, the empty prefix leaves the type as-is ($\delta_{\mathtt{starEmp}}\ (s^\star) = s^\star$). Because no data will arrive after the done prefix, the derivative of $s^\star$ with respect to `starDone` is ε. In the case for $\mathtt{starFirst}(p)$, 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 $\delta_{\mathtt{starFirst}(p)}\ (s^\star) = (\delta_p\ (s)) \cdot s^\star$. Finally, $\delta_{\mathtt{starRest}(p,p')}\ (s^\star) = \delta_{p'}\ (s^\star)$.

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 T-Star-R-1 corresponds to the left injection into 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 T-Star-R-2 is the right injection: from a context $\Gamma;\Delta$, we can produce an $s^\star$ by producing one s from Γ and the remaining $s^\star$ from Δ. Operationally, this should run the same way as the T-Cat-R rule: by first running e_1, and if an entire s is produced, continuing by running e_2 to produce some prefix of the tail. The T-Star-L rule is a case analysis principle for streams of star type: either such a stream is empty, or else it comprises 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 in the context. Like T-Sum-L, this rule includes a buffer, collecting input environments until the prefix bound to z is enough to make the decision for which branch of the case to run.

$$\frac{}{\Gamma \vdash \mathtt{nil} : s^\star}\ \text{T-Star-R-1} \qquad \frac{\Gamma \vdash e_1 : s \qquad \Delta \vdash e_2 : s^\star}{\Gamma;\Delta \vdash e_1 :: e_2 : s^\star}\ \text{T-Star-R-2}$$

The semantics of the right rules are straightforward: the rules for T-Star-R-1 are like those for T-Eps-R, while the rules for T-Star-R-2 are like those for T-Cat-R. The semantics of T-Star-L is just like T-Sum-L, buffering input prefixes until either (a) we get $z \mapsto \mathtt{starDone}$, at which point we

run e_1, or (b) we get $z \mapsto \texttt{starFirst}(p)$ or $z \mapsto \texttt{starRest}(p, p')$, in which case we run e_2. For full details, see Appendix C.9.

4.3 Let-Binding

Full λ^{ST} also allows for more general let-binding. Given a transformer e whose output is used in the input of another term e', we can compose them to form a single term $\texttt{let}\, x = e\, \texttt{in}\, e'$ that operates as the sequential composition of e followed by e'. The rules for this construct are in Figure 6. Note that this sequencing is not the same kind of sequencing as in a concat-pair $(e; e')$. The latter produces data that follows the sequential pattern $s \cdot t$, while the former is sequential composition of code. When a let binding is run, both terms are evaluated, and the output of the first is passed to the input of the second. An important point to note is that this semantics is non-blocking: even if e produces the empty prefix, we still run e', potentially producing output.

$$\frac{\Gamma(\cdot) \vdash e_1 : r \qquad \Gamma(x : s; xs : s^\star) \vdash e_2 : r \qquad \eta \;:\; \mathsf{env}\left(\Gamma(z : s^\star)\right)}{\delta_\eta\left(\Gamma(z : s^\star)\right) \vdash \mathsf{case}_{s,r}\,(p; z, e_1, x.xs.e_2) : r}\ \text{T-Star-L}$$

The semantic rule S-Let for let-binding (in Figure 6) is a straightforward encoding of this behavior. Given the input environment η, we run the term e, bind the resulting prefix p to x, and run the continuation e', returning its output. The resultant term is another let-binding between the resultant terms of e and e'.

$$\frac{\Delta \vdash e : s \qquad \Gamma(x : s) \vdash e' : t \qquad e\ \texttt{inert}}{\Gamma(\Delta) \vdash \texttt{let}\, x = e\, \texttt{in}\, e' : t}\ \text{T-Let}$$

$$\frac{\eta \Rightarrow e_1 \downarrow e_1' \Rightarrow p \qquad \eta[x \mapsto p] \Rightarrow e_2 \downarrow e_2' \Rightarrow p'}{\eta \Rightarrow \texttt{let}\, x = e_1\, \texttt{in}\, e_2 \downarrow \texttt{let}\, x = e_1'\, \texttt{in}\, e_2' \Rightarrow p'}\ \text{S-Let}$$

Fig. 6. Rules for Let-Bindings

The typing rule T-Let says that if e has type s in context Δ and e' has type t in a context $\Gamma(x : s)$ with a variable of type s, we can form the let-binding term $\texttt{let}\, x = e\, \texttt{in}\, e'$, which has type t in context $\Gamma(\Delta)$. The soundness of the semantics rule S-Let depends on a subtle requirement: e must not produce nonempty output until e' is ready to accept it. This is enforced by the third premise of the T-Let rule, which states that e must be *inert*: it only produces nonempty output when given nonempty input. This restriction rules out let-bindings such as $\texttt{let}\, x = ()\, \texttt{in}\, e'$, since the semantics of `()` always produces nonempty output (namely `oneFull`), even when given an environment mapping every variable to an empty prefix[6]. In actuality, inertness is not a purely syntactic condition on terms, but depends also on typing information. To this end, inertness is tracked like an effect through the type system: see Appendix C.7.1 for details.

4.4 Recursion

To write interesting transformers over $s^\star$ streams, we provide a way to define transformers recursively. Adding a traditional general recursion operator $\texttt{fix}(x.e)$ does not work in our context, as arrow types are required to define functions this way. We instead add explicit term-level recursion and recursive call operators. The program $\texttt{fix}\left(e_{\texttt{args}}\right).(e)$ defines a recursive transformer with body e and initial arguments $e_{\texttt{args}}$. Recursive calls are made inside the body e with a term $\texttt{rec}\left(e_{\texttt{args}}\right)$, which calls the function being defined with arguments $e_{\texttt{args}}$. This back-reference works in the same way that uses of the variable x in the body of a traditional fix point $\texttt{fix}(x.e)$ refer to the term $\texttt{fix}(x.e)$ itself. This function-free approach is approach is inspired by the concept of *cyclic proofs* [15, 23, 28] from proof theory, where derivations may refer back to themselves. Alternatively,

[6]Because such let-bindings are essentially trivial, we expect that they can be eliminated — see Section 7 for more discussion.

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.

Full details of the typing rules and semantics of fixpoints can be found in Appendices C.7 and C.9. In brief, to typecheck a fixpoint term, we simply type its body e, assuming that all instances of the `rec` in e have the same type as the fixpoint itself. Then, to run a fixpoint term $\texttt{fix}\ \left(e_{\texttt{args}}\right).\ (e)$, the rule unfolds the recursion one step by substituting the body e for instances of `rec` in itself, then runs the resulting term, binding all of the arguments to their variables.

Naturally, this can lead to non-termination, as $\texttt{fix(rec)}$ unfolds to itself.[7] To bound the depth of evaluation, we *step index* both semantic judgments by adding a fuel parameter that decreases when we unfold a `fix`. The semantic judgment then looks like $\eta \Rightarrow e \downarrow^n e' \Rightarrow p$: when we run e on η, it steps to e' producing p and unfolding at most n uses of `fix` along the way.

4.5 Stateful Transformers

In the λ^{ST} typing judgment $\Gamma \vdash e : s$, the variables in Γ 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 Γ correspond to data that will arrive further in the future. This imposes a strong restriction on programming: if earlier values in the stream are used at all, they must be used *before* later values; once a value in the stream has "gone by," there is no way to refer to it again. By using variables from the Γ 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*. This limitation is by design: from a programming perspective, referring to variables from the past requires *memory*, which is a resource to be carefully managed in streaming contexts. Of course, while some important streaming functions (e.g., map and filter) can get by without state, but many others (e.g., "running sums") require it. In this section, we add support for stateful stream transformers.

To maintain state from the past, we extend the typing judgment of λ^{ST} to include a second context, Ω, called the *historical context*, which gives types to variables bound to values stored in memory. We write $\Omega \mid \Gamma \vdash e : s$ to mean "e has type s in context Γ and historical context Ω".

What types do variables in the historical context have? Once a complete stream of type $\left(\texttt{Int}^\star \| \texttt{Int}^\star\right) \cdot \texttt{Int}^\star$ has been received and is stored in memory, we may as well regard the data as a value of the standard type $(\texttt{list(Int)} \times \texttt{list(Int)}) \times \texttt{list(Int)}$ from the simply typed lambda-calculus (STLC). In other words, parts of streams that *will* arrive in the future have stream types, parts of streams that *have* arrived in the past can be given standard STLC types. The "flattening" operation $\langle s \rangle$ transforms stream types into STLC types. The interesting cases of its definition are $\langle s \cdot t \rangle = \langle s \| t \rangle = \langle s \rangle \times \langle t \rangle$ and $\langle s^\star \rangle = \texttt{list}\ (\langle s \rangle)$.

The historical context is a fully structural: $\Omega ::= \cdot \mid \Omega, x : A$, where the types A are drawn from some set of conventional lambda-calculus 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 must be typed in an empty historical context; at runtime, historical variables get their values by substitution.

Rather than giving a specific set of ad-hoc rules for manipulating values from the historical context, we parameterize the λ^{ST} calculus over an arbitrary language with terms M, typing judgment $\Omega \vdash M : A$, and 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 λ^{ST} programs using the T-HistPgm rule,

$$\frac{\Omega \vdash M : \langle s \rangle}{\Omega \mid \Gamma \vdash \langle M : s \rangle : s}\ \text{T-HistPgm}$$

[7] Cyclic proof systems usually ensure soundness by imposing a guardedness condition [15] which requires certain rules be applied before a back-edge can be inserted in the derivation tree. Because we are not primarily concerned with λ^{ST} as a logic at the moment, we leave a guardedness condition to future work.

which says that a historical program M of type $\Omega \vdash M : \langle s \rangle$ with access the historical context can be used in place of a λ^{ST} term of type s. Operationally, as soon as any prefix of the input arrives, we run the historical program to completion and yield the result as its stream output (after converting it into a value of type s).

How does information get added to the historical context? Intuitively, a variable in Γ (a stream that will arrive in the future) can be moved to Ω, where streams that have arrived in the past are saved, by waiting for the future to become the past! Formally, we define an operation called "wait," which allows the programmer 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 T-Wait-Surf rule encodes the typing content of this behavior. It allows us to specify a variable x of the input, flatten its type, and then move it to the historical context, so that the continuation e can refer to it in historical terms. Semantically, this works by buffering in environments until a maximal prefix for x has arrived. Once we have a full prefix for x, we substitute it into e and continue running the resulting term.[8] This buffering is implemented the same way as in the left rules for plus and star, by generalizing the typing rule T-Wait-Surf to a rule T-Wait which includes an explicit prefix buffer. As with plus and star, the generalized rule simplifies to the surface rule when the buffer is empty. The generalized rule and the semantics of both the wait and historical program constructs can be found in Appendix C.7 and Appendix C.9. The remaining typing rules in λ^{ST} change only by adding an Ω to the typing judgment everywhere.

$$\frac{\Omega, x : \langle s \rangle \mid \Gamma(\cdot) \vdash_\Sigma e : s}{\Omega \mid \Gamma(x : s) \vdash_\Sigma \texttt{wait}_s(x)\,(e) : s}\ \text{T-Wait-Surf}$$

Updated Soundness Theorems. Adding recursion and the historical context requires us to update to the soundness theorem from that of Kernel λ^{ST} to Full λ^{ST}. If a well typed term has (a) closed historical context, and (b) no unbound recursive calls, takes a step on a well-typed input *using some amount of gas*, then the output and resulting term are also well typed. (The proof is by a large but routine induction, first on the derivation of $\eta \Rightarrow e \downarrow e' \Rightarrow p$, and then on the derivation of $\cdot \mid \Gamma \vdash_\emptyset e : s$. See Appendix C.9 for cases.)

Theorem 4.1 (Soundness of the λ^{ST} Semantics). *If* $\cdot \mid \Gamma \vdash_\emptyset e : s$, *and* $\eta : \texttt{env}(\Gamma)$, *and* $\eta \Rightarrow e \downarrow^n e' \Rightarrow p$, *then* $p : \texttt{prefix}(s)$ *and* $\cdot \mid \delta_\eta(\Gamma) \vdash_\emptyset e' : \delta_p(s)$

A similarly updated statement of the homomorphism theorem can be found in Appendix C.9.

5 DELTA

We next show how λ^{ST} addresses the problems that we identified in Section 2 of (a) type-safe programming with temporal patterns and (b) deterministic processing of parallel data. We also show how some other characteristic streaming idioms can be expressed elegantly in λ^{ST}.

The examples in this section are written in delta[9], an experimental language design based on λ^{ST}.delta proposes a high-level functional syntax that, after typechecking, elaborates to λ^{ST} terms. It supports some features not included in the λ^{ST} calculus that we expect will be required in full-blown language designs based on λ^{ST}.

[8] The semantics of the T-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 variable and block the rest of the program until it is complete and in memory.

[9] delta is available at http://www.github.com/alpha-convert/delta

delta Features

While the proof terms of λ^{ST} allow elimination forms (such as `let (x,y) = z in e`) to only be applied to variables (an artifact of the sequent calculus formalism), delta's syntax is a standard one where elimination forms can be applied to arbitrary expressions. delta also includes more types than λ^{ST}, adding base types `Int` and `Bool`.

Functions and Macros. Top-level functions in delta are simply open terms: a function definition `fun f(x : Int*) : Int* = e` elaborates and typechecks to a core term e which satisfies the typing judgment $x : \texttt{Int}^\star \vdash \texttt{e} : \texttt{Int}^\star$. Higher order functions in delta are implemented as *macros*. A function written as `fun g<f : Int -> Int>(x : Int*) : Int* = e` is a macro which takes another function `f : Int* -> Int*` as a parameter. Calls to `g` in other functions then look like `g<f'>`, where `f'` is either (a) another function defined at top level, or (b) a call to yet another macro. If the macro `g` is recursive, its recursive calls do not receive a macro argument — all recursive usages of a macro get passed the initial macro parameter `f`. This discipline ensures that the macro usage does not depend on runtime data, and so higher-order functions can be fully resolved to λ^{ST} terms statically.

Neither of these features — standard top-level functions and higher-order macros — require the use of first-class function types, which λ^{ST} does not currently support. Defining true higher-order functions would allow for streams of *functions*, such as $(s \to t)^\star$. We hope to investigate these in future work; see Section 7.

Functions in delta can also be (prenex-) polymorphic [58]. Polymorphic functions definitions are annotated with an list of their type arguments, like `fun f[s,t](x : s*) : t* = e`. When such a function is called, the type arguments must be passed explicitly like `f[Int,Bool]`.

Historical Arguments and Generalized Wait. Functions in delta can also take arguments for their historical contexts: a function `fun f{acc : Int}(xs : Bool*) : Int* = e` takes an in-memory `Int` argument, and elaborates to a core term that satisfies the typing judgment $\texttt{acc} : \texttt{Int} \mid \texttt{xs} : \texttt{Bool}^\star \vdash e : \texttt{Int}^\star$, where `acc` is in the historical context. When `f` is called, the `acc` argument must be passed a historical program. For example, if `u : Int` is in the current historical context (and `ys : Bool*` in the regular one), `f{u + 1}(ys)` is an acceptable call to `f`.

The `wait` construct is also slightly more general in delta. Instead of just `wait`ing on variables, programmers may `wait` on the result of some expression, and then save its result into memory: this is accomplished with `wait e as x do e' end`.

delta Implementation. The implementation begins by elaborating a high-level surface syntax down to an "elaborated syntax", which eliminates shadowing, resolves function calls, and transforms the syntax into the sequent calculus representation by introducing intermediate variables for subexpressions.

The elaborated syntax is then typechecked, producing *templates* of λ^{ST} terms. These templates serve two roles. First, they are monomorphizers. Since λ^{ST} is a monomorphic calculus, typechecking produces a map from closed types (to plug in for type variables) to raw terms. Second, the templates implement macro expansion.

The typechecker uses a (we believe novel) algorithm for checking our variant of ordered & bunched terms. While we have tested the typechecker with many terms, we have not proved that the algorithm is sound and complete with respect to the declarative type system presented in Appendix C. The interpreter, on the other hand, is very straightforward: it is a direct translation of the λ^{ST} semantics into code.

More details about the project structure of the delta prototype can be found in Appendix A.

Examples

Besides its type system, delta's design differs from that of most stream processing languages in another important respect. In languages like Flink [29], Beam[34], and Spark [32], streaming programs must be written using a handful of provided combinators like `map` `filter` and `fold` (or possibly as SQL-style queries, in languages derived from CQL [6]). By contrast, delta programs are written in the style of functional list processors. Instead of working to cram complex program behaviors into `map`s, `filter`s, and `fold`s, programmers can express their intent more directly in the form of more general recursive functional programs. Of course, this does not preclude the use of the aforementioned combinators: they are directly implementable in delta.

Map. Given a transformer from `s` to `t`, we can lift it to a transformer from `s*` to `t*` with a `map` operation. The code for this function is essentially identical to the familiar functional program, but its type is more general than the standard `map` function on homogeneous streams, which has type `(a → b) → (Stream a → Stream b)`: the types s and t here can be arbitrary stream types: they need not be singletons.

```
fun map [s,t] <f : s -> t> (xs : s*) : t* =
  case xs of
    nil => nil
  | y :: ys => f(y) :: map(ys)
```

Filter. Similarly, given a "predicate" function f from s to $t + \varepsilon$ (the streaming version of t `option`), we can transform an incoming stream of $s^\star$ to include just the transformed elements which pass the filter.

We can then recover a traditional predicate-based filter by lifting a predicate `f` that takes an in-memory `s` to `Bool` to a streaming function `s -> s + Eps` with `liftP`. This program simply waits for its argument to arrive, then applies the predicate to the in-memory `s`.

```
fun mapMaybe[s,t]<f : s -> t + Eps> (xs : s*) : t*=
  case xs of
    nil => nil
  | y :: ys => case f y of
                 | inl(t) => t :: mapMaybe(ys)
                 | inr(_) => mapMaybe(ys)

fun liftP[s]<f : {s}(Eps) -> Bool>(x : s) : s + Eps =
  wait x, f{x}(sink) as b do
      if {b} then inl({x}) else inr(sink)
  end

fun filter<f : {s}(Eps) -> Bool>(xs : s*) : s* =
  mapMaybe[s,s]<liftPred<f>>(xs)
```

Fold. λ^{ST} can express both *running folds*, which output a stream of all their intermediate states, and *functional folds*, which output only the final state. Since functional folds that return only the final state cannot be given this rich type in traditional stream processing languages (for the same reason as the `head` and `tail` functions), we present one here. See Appendix B.1 for discussion of a running fold; the code is similar except that it outputs `y` at every step.

```
fun fold [s,t] <f : {t}(s) -> t>{acc : t}(xs : s*) : t =
  case xs of
    nil => {acc}
  | y :: ys => wait f{acc}(y) as acc' do
                 fold{acc'}(ys)
               end
```

The `fold` transducer maintains an in-memory accumulator of type t; this gets updated by a streaming step function `f : {t}(s) -> t` that takes the state t and the new element s and produces a t. The whole fold takes a stream `xs` of type $s^\star$ and an initial accumulator value $y : t$, and it eventually produces the final state t. As for `map`, the code for `fold` is very similar to the traditional functional program: the only distinction is the inclusion of `wait`s to marshal data into memory.

Singletons, Head, Tail. In the homogeneous model, stream types are always conceptually unbounded. But in many practical situations, a stream will only be expected to contain a single element—a

constraint that cannot be expressed with homogeneous streams. Using stream types, we can write stream transformers that 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, as shown on the right.

(Exercise: try writing the term for `tail` on star streams. This requires a use of `wait` and an accumulator argument like in `fold`.)

```
fun head [s] (xs : s*) : Eps + s =
  case xs of nil => inl(sink)
           | y :: _ => inr(y)
```

Brightness Levels. The structured communication protocol from the brightness-levels example in Section 2 can be encoded as the type $(\texttt{Int} \cdot \texttt{Int}^\star)^\star$: a stream of nonempty sequences of `Int`s, representing "runs" of light levels greater than some threshold. Given such a stream, writing a program to compute the averages is easy: we just `map` an average operation—taking $\texttt{Int} \cdot \texttt{Int}^\star$ to `Int`—over the incoming stream to produce a stream $\texttt{Int}^\star$ of averages.

```
fun averageSingle (run : Int . Int*) : Int =
  let (x;xs) = run in
  let (sm,len) = (sum(xs), length(xs)) in
  wait x,sm,len do
    {(x + sm) / (1 + len)}
  end

fun averageAbove{t : Int}(xs : Int*) : Int* =
  map<averageSingle>(thresh{t}(xs))
```

The per-run average operation, `averageSingle`, is defined by computing its sum and length in parallel, waiting for the results, then dividing the sum (plus the first element) by the length (plus one).

Notice that, since each run is statically known to have at least one element, `averageSingle` can omit the error handling that, with a homogeneous stream type, would be needed to avoid a potential divide by zero. By contrast, with a homogeneous stream type like $(\texttt{Start} + \texttt{Int} + \texttt{End})^\star$, this operation would need to be written in a low-level, more stateful manner, remembering the current run of `Int`s until an `End` event arrives, averaging, and handling the divide-by-zero error which could in principle (although not in practice) occur if no `Int`s arrived between a `Start` and an `End`.

The thresholding operation `thresh`, which takes $\texttt{Int}^\star$ and produces the runs of elements above the threshold $(\texttt{Int} \cdot \texttt{Int}^\star)^\star$, is straightforward. Whenever the incoming stream goes above the threshold `t`, we collect all of the subsequent elements into a run, emit it, and recurse down the rest of the stream. This uses an operation `spanGt : {Int} (Int*) -> Int . Int*` that returns the initial "span" of elements above `t`, followed by the rest of the stream. It's important to note that this program is completely non-blocking: as soon as the first element above `t` arrives, it is forwarded along, as are all subsequent elements until the stream drops below `t`. By contrast with homogeneously typed streaming languages, delta's type safety guarantees that `thresh` *does in fact* output a stream that adheres to the protocol, and (2) any downstream transformer does not have to replicate this parsing logic.

```
fun thresh{t : Int}(xs : Int*) : (Int . Int*)* =
  case xs of
    nil => nil
  | y :: ys => wait y do
                 if {y > t} then
                   let (run;rest) = spanGt{t}(ys) in
                   ({y};run) :: thresh{t}(rest)
                 else
                   thresh{t}(ys)
               end
```

The complete program, first calling `thresh`, and then mapping `averageSingle` over the stream of runs, is `averageAbove`.

Side Outputs & Error Handling. A common streaming idiom is the use of "side outputs" for reporting errors. In languages that support this idiom, operations include extra output streams where error messages are sent as they arise at runtime. These side outputs are always a second-class mechanism:

the error streams cannot be transformed or used in a manner other than dumping them to a log somewhere. λ^{ST} provides a first-class account of side outputs, 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 functional way, 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 that are routed to different downstream nodes in the dataflow graph, thus exposing parallelism and increasing potential throughput. Appendix B.2 shows how two different partitioning strategies can be implemented in λ^{ST}. First is 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 sending the first element to the left branch, the second to the right, and so on. Second is a *decision-based partitioner*, which routes a stream of type $s^\star$ one direction or another into an output stream of type $t^\star \| r^\star$ based on the result of a function from s to $t + r$. Like with filter, we can similarly recover a *predicate-based* partitioner by lifting a predicate to a function into sums.

Windowing and Punctuation. Windowing is another core concept in stream processing systems, where aggregation operations like moving averages or sums are defined over "windows"—groupings of consecutive events, gathered together into a set. In λ^{ST}, these transformers are just `maps` over a stream whose elements are windows. Given a per-window aggregation transformer `f` from 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*`. Appendix B.3 defines 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 RELATED WORK

Streams as a programming abstraction have their sources in early work in the programming languages [17, 48, 68, 70] and database [1, 2, 5–7, 21, 55] communities. Though streams have mostly been viewed as homogeneous sequences, more interesting treatments have also been proposed. For example, 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 in the PL community as coinductive sequences [17]: a stream of `A` has a single (co)constructor, `cocons` : `Stream`$A \to (A \times$ `Stream` $A)$ and acts as a lazily evaluated infinite list. In particular, this is the setting of traditional *dataflow programming* [68]. One 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" [48] (including their restriction to synchronous networks [12, 54, 70]) avoid this problem by allowing only *blocking reads* of messages on FIFO queues. In contrast, the semantics of λ^{ST} leverages its type structure to guarantee deterministic parallel processing *without* blocking in many cases. For example, in the context of a T-Let 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 [18, 29], Spark Streaming [32, 75], Samza [31, 59], Arc [**?**], and Storm [33] support *dynamic partitioning*: a stream type can define one stream with many parallel substreams, where the number of substreams and assignment of data to substreams is determined at runtime.

The type `Stream t` in these systems is implicitly a parallel composition of homogeneous streams: $\texttt{t}^\star \| \cdots \| \texttt{t}^\star$. Unlike in λ^{ST}, these parallel substreams cannot have more general types.

Some which papers which attempt to build very general compile targets for stream processing support parallelism in only restricted ways. For example, Brooklet [66] and the DON Calculus [24] 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 [41, 56, 64]. While λ^{ST} does not support dynamic partitioning, we hope to address it in future work; see Section 7.

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 [21] and CACQ [55], 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, 45]. Time-varying relations can be expressed in λ^{ST} using Kleene star and concatenation: a relation of tuples of type `T` timestamped by `Time` can be expressed as $(\texttt{T}^\star \cdot \texttt{Time})^\star$. We can also express the common pattern where parallel streams are synchronized by a single timestamp (again, modulo dynamic partitioning) with types like $((\texttt{T}^\star \| \texttt{T}^\star) \cdot \texttt{Time})^\star$. Each `Time` event is a punctuation mark containing the timestamp of the prior set of tuples [47, 73]. 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 found in prior work treating streams as *pomsets* (partially ordered multisets) [3, 49–51, 56], inspired by work in concurrency theory [25, 57]. In a pomset, data items may be completely ordered (a sequence), completely unordered (a bag), or somewhere in between. Some recent works have proposed pomset-based types for streams [3, 56], but their types do not support concatenation and do not come with type *systems*—programs must be shown to be well typed semantically, rather than via syntactic typing rules.

Functional reactive programming (FRP) [26] 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, 19, 53] 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, our incremental semantics demonstrates that λ^{ST}'s 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 that 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 must eventually produce new outputs), but in Section 7 we discuss how to remedy this by imposing guardedness conditions on recursive calls.

Jeffrey [46] permits the type of a signal to vary over time, using dependent types inspired by Linear Temporal Logic [62]. This 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., parallel signals all present new values together, at the same time. 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: the type of data cannot change over time. Lastly, **?**] develops a modal type system

which expresses low-level event handlers. These are also purely synchronous, and the programs are written as event handlers as opposed to high-level "batch" processors.

Stream Runtime Verification (SRV) aims, broadly, to monitor streams at runtime and provide boolean or numerical "triggers" that fire when they satisfy some specification. Many RV projects like LOLA [22], HLola [20], RTLola [27], Striver [40], HStriver [39] also provide high-level, declarative specification languages for writing such monitors. Because these languages often use regular expressions or LTL as a formalism, they often bear a resemblance to our stream types. Despite this similarity, our goals and methods are quite different. Unlike the dynamically-checked specifications of SRV, the types in delta are static guarantees: a stream program of type s necessarily produces a stream of type s.

Streaming with Laziness. It is folklore in the Haskell community that a "sufficiently lazy" list program can be run as a streaming program using a clever trick with lazy IO [52] [71]. This "sufficient laziness" condition is syntactically brittle, and requires an expert Haskell programmer to carefully ensure that all functions involved are lazy in the just the right way. Indeed, many Haskell programmers instead reach for combinator libraries like Pipes [37] FoldL [38], Conduit [65], Streamly [69], and others to ensure their programs have a streaming semantics. In delta, the type system takes care of this for you: all well-typed programs can be given a streaming semantics. Moreover, the λ^{ST} semantics gives a direct account of how pure functions execute incrementally as state machines, as opposed to the way that Haskell's non-strict semantics incidentally yields streaming behavior when combined with Lazy IO.

Session types and process calculi. Another large body of work with similar vision is session types for process calculi [43], 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 λ^{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: all events in a channel are ordered relative to each other. Recently, Frumin et al. [36] proposed a session-types interpretation of BI that uses the bunched structure very differently from λ^{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, while, in λ^{ST}, $s \cdot t$ and $s \| t$ describe very different streams.

Concurrent Kleene Algebras and regular expression types. Stream types are partly inspired by Concurrent Kleene Algebras (CKAs) [42] and related syntaxes for pomset languages [51], but 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 [74] use KA to model *programs*, whereas in λ^{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 taken inspiration from languages for programming with XML data [13, 35, 44, etc.] using types based on regular expressions.

7 CONCLUSIONS AND FUTURE WORK

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

In the future, we hope to add more types to λ^{ST}. Adding a support for *b*ags — unbounded parallelism, the parallel analog of Kleene star — would enable dynamic partitioning. λ^{ST} also lacks

function types. The proof theory of BI would imply that there should be two (one for each context former), but we have yet to investigate what these functions might mean in the streaming setting.

Further theoretical investigations include (1) alternate semantics for stream types, including a denotational semantics as pomset morphisms, Kahn Process Networks [48], or some category of state machines, (2) eliminating the inertness restriction on let-bindings, and (3) adding a *guardedness* condition on recursive calls to ensure termination and hence productivity.

On the applied side, we plan to build a distributed implementation of delta by compiling λ^{ST} terms to programs for an existing stream processing system like Apache Storm [33], thus inheriting its desirable fault-tolerance and delivery guarantees. We hope to build such a compiler and use it as a platform for experimenting with type-enabled optimizations and resource usage analysis.

ACKNOWLEDGMENTS

We thank the reviewers for their feedback. We also thank Justin Lubin for feedback on drafts of this paper, and Alex Kavvos, Andrew Hirsch, Mae Milano, and Michael Arntzenius for helpful discussions about early versions of this work.

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] Gérard Berry and Georges Gonthier. 1992. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming* 19, 2 (1992), 87–152. https://doi.org/10.1016/0167-6423(92)90005-V

[15] 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

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

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

[18] 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).

[19] 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

[20] Martín Ceresa, Felipe Gorostiaga, and César Sáchez. 2020. Declarative Stream Runtime Verification (hLola). In *Proc. of the 18th Asian Symposium on Programming Languages and Systems (APLAS'20) (LNCS, Vol. 12470)*. Springer, 25–43. https://doi.org/10.1007/978-3-030-64437-6_2

[21] 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.

[22] B. D'Angelo, S. Sankaranarayanan, C. Sanchez, W. Robinson, B. Finkbeiner, H.B. Sipma, S. Mehrotra, and Z. Manna. 2005. LOLA: runtime monitoring of synchronous systems. In *12th International Symposium on Temporal Representation and Reasoning (TIME'05)*. 166–174. https://doi.org/10.1109/TIME.2005.26

[23] 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

[24] 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.

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

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

[27] Peter Faymonville, Bernd Finkbeiner, Malte Schledjewski, Maximilian Schwenger, Marvin Stenger, Leander Tentrup, and Hazem Torfah. 2019. StreamLAB: Stream-based Monitoring of Cyber-Physical Systems. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, 421–431.

[28] 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.

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

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

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

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

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

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

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

[36] 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

[37] Gabriella Gonzalez. 2022. Pipes. https://hackage.haskell.org/package/pipes.

[38] Gabriella Gonzalez. 2024. FoldL. https://hackage.haskell.org/package/foldl.

[39] Felipe Gorostiaga and César Sánchez. 2021. HStriver: A Very Functional Extensible Tool for the Runtime Verification of Real-Time Event Streams. In *Proc. of the 24th Int'l Symp. on Formal Methods (FM'21) (LNCS, Vol. 13047)*. Springer, 563–580. https://doi.org/10.1007/978-3-030-90870-6_30

[40] Felipe Gorostiaga and César Sánchez. 2021. Stream runtime verification of real-time event streams with the Striver language. *International Journal on Software Tools for Technology Transfer* 23 (2021), 157–183. https://doi.org/10.1007/s10009-021-00605-3

[41] 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).

[42] 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.

[43] 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.

[44] 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.

[45] 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).

[46] 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

[47] 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.

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

[49] 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).

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

[51] 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

[52] Oleg Kiselyov. 2012. Iteratees. In *Functional and Logic Programming*, Tom Schrijvers and Peter Thiemann (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 166–181.

[53] 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

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

[55] 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

[56] 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)*.

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

[58] Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* 17, 3 (1978), 348–375. https://doi.org/10.1016/0022-0000(78)90014-4

[59] 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).

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

[61]]event-driven Jennifer Paykin, Neelakantan R. Krishnaswami, and Steve Zdancewic. [n. d.]. The Essence of Event-Driven Programming. ([n. d.]).

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

[63] 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.

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

[65] Michael Snoyman. 2023. Conduit. https://hackage.haskell.org/package/conduit.

[66] 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.

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

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

[69] Composewell Technologies. 2023. StreamLy. https://hackage.haskell.org/package/streamly-core.

[70] William Thies, Michal Karczmarek, and Saman Amarasinghe. 2002. StreamIt: A language for streaming applications. In *International Conference on Compiler Construction*. Springer.
[71] Jose Manuel Calderon Trilla. 2024. personal communication.
[72] 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
[73] 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).
[74] 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.
[75] 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 DELTA IMPLEMENTATION

The delta implementation is available at http://www.github.com/anonymous-github-user/delta, and has been tested with GHC version 9.2.7 and Stack version 2.9.3.

Table 1. Overview of the delta Implementation

Name	Description	Location
Var	Various kinds of variables	`Var.hs`
Values	Prefixes and environments	`Values.hs`
Types	Types and contexts	`Types.hs`
HistPgm	Types and semantics for historical programs	`HistPgm.hs`
CoreSyntax	Syntax of λ^{ST} terms and operations on them	`CoreSyntax.hs`
SurfaceSyntax	ASTs for the surface syntax	`Frontend/SurfaceSyntax.hs`
Parser	Parser for the surface syntax	`Frontend/Parser.y`
ElabSyntax	Elaborated syntax, and the elaborator code	`Frontend/ElabSyntax.hs`
Typechecker	Typechecker from elab syntax to core terms	`Frontend/Typecheck.hs`
Template	Macros and Monomorphization for λ^{ST} terms	`Backend/Monomorphizer.hs`
EnvSemantics	Implementation of the λ^{ST} semantics	`Backend/EnvSemantics.hs`
ErrUtil	Error handling utilities	`Util/ErrUtil.hs`
PartialOrder	A partial order data structure	`Util/PartialOrder.hs`
PrettyPrint	Pretty printer typeclass	`Util/PrettyPrint.hs`

B EXAMPLES

This appendix collects some additional examples of programming with delta

B.1 Running Fold

We can also define a *running* fold operation on star streams, which outputs its partial results as it goes.

```
fun runningFold[s,t]<f : {t}(s) -> t>{acc : t} (xs : s*) : t* =
  case xs of
    nil => nil
  | y :: ys => wait f{acc}(y) as acc' do
                 {acc'} :: runningFold{acc'}(ys)
               end
```

B.2 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 λ^{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 λ^{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.

```
fun roundRobin[s]{b : Bool}(xs : s*) : s* || s* =
  case xs of
    nil => (nil,nil)
  | y::ys =>
      let (zs,ws) = roundRobin{!b}(ys) in
      if {b} then (y::zs,ws) else (zs,y::ws)
```

Decision-Based Partitioning. A decision-based partitioner routes stream elements based on the result of a predicate.

```
fun decPartition[s,t,r]<f : (s) -> t + r>(xs : s*) : t* || r* =
  case xs of
    nil => (nil,nil)
  | y::ys =>
         let (ts,rs) = decPartition(ys) in
         case f(y) of
              inl t => (t::ts,rs)
            | inr r => (ts,r::rs)
```

B.3 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 λ^{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 `(Int*)*` 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 λ^{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, by grouping together the first k elements, and recursing down the rest of the stream.

```
fun firstN[s]{n : Int}(xs : s*) : s* . s* =
  case xs of
    nil => (nil;nil)
   | y::ys => if {n > 0} then
                let (predN;rest) = firstN{n-1}(ys) in
                (y::predN;rest)
              else (nil;y::ys)

fun tumble[s]{k : Int}(xs : s*) : s** =
  let (first;rest) = firstN[s]{k}(xs) in first :: tumble{k}(rest)
```

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.

```
fun parsepairs[s](xs : s*) : (s . s)* =
  case xs of
    nil => nil
  | y :: ys => case ys of
                 nil => nil
               | z :: zs => (y;z) :: parsepairs(zs)
```

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.

```
fun slidingWindower(acc : s*; xs : s*) : s** =
  case xs of
    nil => acc :: nil
```

```
  | y::ys => wait y do
              let next = {if |acc| < k then y :: acc else y :: (init acc)} in
              next :: slidingWindower(next;xs)
            end
```

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 $\left(s^\star\right)^\star$ from a punctuated stream $(\varepsilon + s)^\star$ by emitting windows which are the (potentially empty) runs of ss between punctuation marks.

```
fun tilFirstPunc[s](xs : (Eps + s)*) : s* . (Eps + s)* =
  case xs of
    nil => (nil;nil)
  | y::ys => case y of
               inl _ => (nil;ys)
             | inr s => let (cur;rest) = tilFirstPunc(ys) in
                        (s::cur;rest)

fun puncWindow[s](xs : (Eps + s)*) : s** =
  let (run;rest) = tilFirstPunc[s](xs) in
  run :: puncWindow(rest)
```

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.

```
fun sync[s,t](xs : s*, ys : t*) : (s || t)* =
case xs of
  nil => nil
| x'::xs' => case ys of
               nil => nil
             | y'::ys' => wait x',y' do
                            {(x',y')} :: sync(xs',ys')
                          end
```

Semantically, this program `wait`s 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 λ^{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 $\left(s^\star\right)^\star$ and $\left(t^\star\right)^\star$, we can synchronize to get $\left(s^\star \| t^\star\right)^\star$, and then apply a join operation to each parallel pair of windows.

C TECHNICALITIES

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

C.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 ε, and the type of streams consisting of a single integer, $\texttt{Int}$. Larger types include the concatenation type $s \cdot t$, the sum type $s + t$, the parallel stream type $s \| t$, and the star type $s^\star$.

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

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 *null* if it includes no data. Null types are parallel combinations of εs.

Definition C.1 (Nullable). We define a judgment $s\ \texttt{null}$ as follows:

$$\frac{}{\varepsilon\ \texttt{null}} \qquad \frac{s\ \texttt{null} \qquad t\ \texttt{null}}{s \| t\ \texttt{null}}$$

We extend to contexts pointwise.

$$\frac{}{\cdot\ \texttt{null}} \qquad \frac{s\ \texttt{null}}{x : s\ \texttt{null}} \qquad \frac{\Gamma\ \texttt{null} \qquad \Gamma'\ \texttt{null}}{\Gamma \,,\, \Gamma'\ \texttt{null}} \qquad \frac{\Gamma\ \texttt{null} \qquad \Gamma'\ \texttt{null}}{\Gamma \,;\, \Gamma'\ \texttt{null}}$$

Prefixes are also like in the main paper, with a definition $p\ \texttt{maximal}$ for "complete" prefixes, and a typing relation $p\ :\ \texttt{prefix}\ (s)$.

Definition C.2 (Prefixes). The grammar of prefixes is given by:

$$\begin{aligned} p ::= \texttt{oneEmp} \mid \texttt{oneFull} \mid \texttt{epsEmp} \mid \texttt{parPair}(p, p') \\ \mid \texttt{catFst}(p) \mid \texttt{catBoth}(p, p') \\ \mid \texttt{sumEmp} \mid \texttt{sumInl}(p) \mid \texttt{sumInr}(p) \\ \mid \texttt{starEmp} \mid \texttt{starDone} \\ \mid \texttt{starFirst}(p) \mid \texttt{starRest}(p, p') \end{aligned}$$

Definition C.3 (Maximal Prefix).

$$\frac{}{\texttt{epsEmp maximal}} \qquad \frac{}{\texttt{oneFull maximal}} \qquad \frac{p_1\ \texttt{maximal} \qquad p_2\ \texttt{maximal}}{\texttt{parPair}(p_1, p_2)\ \texttt{maximal}}$$

$$\frac{p_1\ \texttt{maximal} \qquad p_2\ \texttt{maximal}}{\texttt{catBoth}(p_1, p_2)\ \texttt{maximal}} \qquad \frac{p\ \texttt{maximal}}{\texttt{sumInl}(p)\ \texttt{maximal}}$$

$$\frac{p\ \texttt{maximal}}{\texttt{sumInr}(p)\ \texttt{maximal}} \qquad \frac{}{\texttt{starDone maximal}} \qquad \frac{p\ \texttt{maximal} \qquad p'\ \texttt{maximal}}{\texttt{starRest}(p, p')\ \texttt{maximal}}$$

Definition C.4 (Well-Typed Prefixes).

$$\frac{}{\texttt{epsEmp} : \texttt{prefix}\,(\varepsilon)} \qquad \frac{}{\texttt{oneEmp} : \texttt{prefix}\,(1)} \qquad \frac{}{\texttt{oneFull} : \texttt{prefix}\,(1)}$$

$$\frac{p_1 : \texttt{prefix}\,(s) \qquad p_2 : \texttt{prefix}\,(t)}{\texttt{parPair}(p_1, p_2) : \texttt{prefix}\,(s \| t)} \qquad \frac{p : \texttt{prefix}\,(s)}{\texttt{catFst}(p) : \texttt{prefix}\,(s \cdot t)}$$

$$\frac{p_1 : \texttt{prefix}\,(s) \qquad p_1\ \texttt{maximal} \qquad p_2 : \texttt{prefix}\,(t)}{\texttt{catBoth}(p_1, p_2) : \texttt{prefix}\,(s \cdot t)} \qquad \frac{}{\texttt{sumEmp} : \texttt{prefix}\,(s + t)}$$

$$\frac{p : \texttt{prefix}\,(s)}{\texttt{sumInl}(p) : \texttt{prefix}\,(s + t)} \qquad \frac{p : \texttt{prefix}\,(t)}{\texttt{sumInr}(p) : \texttt{prefix}\,(s + t)} \qquad \frac{}{\texttt{starEmp} : \texttt{prefix}\,(s^\star)}$$

$$\frac{}{\texttt{starDone} : \texttt{prefix}\,(s^\star)} \qquad \frac{p : \texttt{prefix}\,(s)}{\texttt{starFirst}(p) : \texttt{prefix}\,(s^\star)}$$

$$\frac{p : \texttt{prefix}\,(s) \qquad p\ \texttt{maximal} \qquad p' : \texttt{prefix}\,(s^\star)}{\texttt{starRest}(p, p') : \texttt{prefix}\,(s^\star)}$$

For each type s, we define the "empty" prefix $\texttt{emp}_s$ inductively on the structure of s.

Definition C.5 (Empty Prefix). The empty prefix is defined as follows:

$$\begin{aligned}
(\varepsilon)\ \ & \texttt{emp}_\varepsilon = \texttt{epsEmp} \\
(1)\ \ & \texttt{emp}_1 = \texttt{oneEmp} \\
(s \| t)\ \ & \texttt{emp}_{s\|t} = \texttt{parPair}(\texttt{emp}_s, \texttt{emp}_t) \\
(s + t)\ \ & \texttt{emp}_{s+t} = \texttt{sumEmp} \\
(s \cdot t)\ \ & \texttt{emp}_{s\cdot t} = \texttt{catFst}(\texttt{emp}_s) \\
(s^\star)\ \ & \texttt{emp}_{s^\star} = \texttt{starEmp}
\end{aligned}$$

We lift this to contexts in the natural way, with $\texttt{emp}_\cdot = \texttt{epsEmp}$, and $\texttt{emp}_{\Gamma;\Delta} = \texttt{catFst}(\texttt{emp}_\Gamma)$, and $\texttt{emp}_{\Gamma,\Delta} = \texttt{parPair}(\texttt{emp}_\Gamma, \texttt{emp}_\Delta)$.

Theorem C.6 (Empty Prefix is Well-Typed). $\texttt{emp}_s : \texttt{prefix}\,(s)$

Definition C.7 (Prefix is Empty).

$$\frac{}{\texttt{epsEmp empty}} \qquad \frac{}{\texttt{oneEmp empty}} \qquad \frac{p_1\ \texttt{empty} \qquad p_2\ \texttt{empty}}{\texttt{parPair}(p_1, p_2)\ \texttt{empty}}$$

$$\frac{p\ \texttt{empty}}{\texttt{catFst}(p)\ \texttt{empty}} \qquad \frac{}{\texttt{sumEmp empty}} \qquad \frac{}{\texttt{starEmp empty}}$$

Theorem C.8 (Empty Prefix Is Empty). $\texttt{emp}_s\ \texttt{empty}$

Proof. Induction on s. □

Theorem C.9 (Empty And Maximal Means Nullable). *If* $p : \texttt{prefix}\,(s)$, *and simultaneously* $p\ \texttt{empty}$ *and* $p\ \texttt{maximal}$, *then* $s\ \texttt{null}$.

Proof. By induction on $p : \texttt{prefix}\,(s)$ □

C.2 Derivatives

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

Definition C.10 (Derivatives).

$$\frac{}{\delta_{\mathtt{epsEmp}}\,(\varepsilon) \sim \varepsilon} \qquad \frac{}{\delta_{\mathtt{oneEmp}}\,(1) \sim 1} \qquad \frac{}{\delta_{\mathtt{oneFull}}\,(1) \sim \varepsilon} \qquad \frac{\delta_{p_1}\,(s) \sim s' \qquad \delta_{p_2}\,(t) \sim t'}{\delta_{\mathtt{parPair}(p_1,p_2)}\,(s \| t) \sim s' \| t'}$$

$$\frac{\delta_p\,(s) \sim s'}{\delta_{\mathtt{catFst}(p)}\,(s \cdot t) \sim s' \cdot t} \qquad \frac{\delta_{p_2}\,(t) \sim t'}{\delta_{\mathtt{catBoth}(p_1,p_2)}\,(s \cdot t) \sim t'} \qquad \frac{}{\delta_{\mathtt{sumEmp}}\,(s + t) \sim s + t}$$

$$\frac{\delta_p\,(s) \sim s'}{\delta_{\mathtt{sumInl}(p)}\,(s + t) \sim s'} \qquad \frac{\delta_p\,(t) \sim t'}{\delta_{\mathtt{sumInl}(p)}\,(s + t) \sim t'} \qquad \frac{}{\delta_{\mathtt{starEmp}}\,(s^\star) \sim s^\star} \qquad \frac{}{\delta_{\mathtt{starDone}}\,(s^\star) \sim \varepsilon}$$

$$\frac{\delta_p\,(s) \sim s'}{\delta_{\mathtt{starFirst}(p)}\,(s^\star) \sim s' \cdot s^\star} \qquad \frac{\delta_{p'}\,(s^\star) \sim s'}{\delta_{\mathtt{starRest}(p,p')}\,(s^\star) \sim s'}$$

Definition C.11 (Context Derivatives).

$$\frac{}{\delta_\eta\,(\cdot) \sim \cdot} \qquad \frac{\eta(x) \mapsto p \qquad \delta_p\,(s) \sim s'}{\delta_\eta\,(x : s) \sim x : s'} \qquad \frac{\delta_\eta\,(\Gamma) \sim \Gamma' \qquad \delta_\eta\,(\Delta) \sim \Delta'}{\delta_\eta\,(\Gamma\,, \Delta) \sim \Gamma'\,, \Delta'}$$

$$\frac{\delta_\eta\,(\Gamma) \sim \Gamma' \qquad \delta_\eta\,(\Delta) \sim \Delta'}{\delta_\eta\,(\Gamma\,; \Delta) \sim \Gamma'\,; \Delta'}$$

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

Theorem C.12 (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. Induction on the derivation of $\delta_p\,(s) \sim s'$ for uniqueness, and $p : \mathtt{prefix}\,(s)$ for existence. □

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 C.13 (Empty Prefix Derivative). $\delta_{\mathsf{emp}_s}\,(s) = s$.

Theorem C.14 (Context Derivatives Function). *There is at most one Γ' such that $\delta_\eta\,(\Gamma) \sim \Gamma'$, and the Γ' exists when $\eta : \mathtt{env}\,(\Gamma)$.*

Proof. Uniqueness by induction on the derivation of $\delta_\eta\,(\Gamma) \sim \Gamma'$, existence by induction on the derivation of $\eta : \mathtt{env}\,(\Gamma)$. □

Theorem C.15 (Maximal Derivative iff Nullable). *If $\delta_p\,(s) \sim s'$ then p $\mathtt{maximal}$ if and only if s' $\mathtt{null}$*

Theorem C.16 (Only Prefix of a Null Type is Empty). *If $p : \mathtt{prefix}\,(s)$ and s $\mathtt{null}$, then $p = \mathsf{emp}_s$*

C.3 Environments

Definition C.17 (Environments and Typing). An environment is a partial map $\eta : \mathtt{Var} \to \mathtt{Prefix}$. We write $\eta : \mathtt{env}\,(\Gamma)$ to mean that η is a well-typed environment for Γ.

$$\frac{}{\eta \;:\; \mathsf{env}\,(\cdot)} \qquad \frac{\eta(x) \mapsto p \qquad p \;:\; \mathtt{prefix}\,(s)}{\eta \;:\; \mathsf{env}\,(x : s)} \qquad \frac{\eta \;:\; \mathsf{env}\,(\Gamma) \qquad \eta \;:\; \mathsf{env}\,(\Delta)}{\eta \;:\; \mathsf{env}\,(\Gamma, \Delta)}$$

$$\frac{\eta \;:\; \mathsf{env}\,(\Gamma) \qquad \eta \;:\; \mathsf{env}\,(\Delta) \qquad \eta\ \mathtt{emptyOn}\ \Delta \vee \eta\ \mathtt{maximalOn}\ \Gamma}{\eta \;:\; \mathsf{env}\,(\Gamma\,;\Delta)}$$

Definition C.18 (All Maximal, All Empty, Agreement). For a set S, we say $\eta\ \mathtt{emptyOn}\ S$ if for all $x \in S$, there is some p such that $\eta(x) \mapsto p$, and $p\ \mathtt{empty}$. We say $\eta\ \mathtt{maximalOn}\ S$ if for all $x \in S$, there is some p such that $\eta(x) \mapsto p$, and $p\ \mathtt{maximal}$. We write $\eta\ \mathtt{emptyOn}\ \Gamma$ and $\eta\ \mathtt{maximalOn}\ \Gamma$ to mean $\eta\ \mathtt{emptyOn}\ \mathrm{Dom}\,(\Gamma)$ and $\eta\ \mathtt{maximalOn}\ \mathrm{Dom}\,(\Gamma)$, respectively. We also write $\eta\ \mathtt{emptyOn}\ e$ and $\eta\ \mathtt{maximalOn}\ e$ to mean $\eta\ \mathtt{emptyOn}\ \mathrm{fv}(e)$ and $\eta\ \mathtt{maximalOn}\ \mathrm{fv}(e)$, respectively.

Finally, we say that η and η' agree on Δ and Δ', written $\mathtt{agree}\,(\eta, \eta', \Delta, \Delta')$ if $\eta\ \mathtt{maximalOn}\ \Delta \implies \eta'\ \mathtt{maximalOn}\ \Delta'$, and $\eta\ \mathtt{emptyOn}\ \Delta \implies \eta'\ \mathtt{maximalOn}\ \Delta'$

An environment is also an environment for every subcontext.

Theorem C.19 (Environment Subcontext Lookup). *If* $\eta \;:\; \mathsf{env}\,(\Gamma(\Delta))$*, then* $\eta \;:\; \mathsf{env}\,(\Delta)$

Proof. Induction on $\Gamma(-)$. □

Moreover, replacing a the environment $\eta|_\Delta$ for a subcontext Δ with another environment η' for another context Δ' yields a well-typed context, so long as η and η' agree on Δ and Δ'. If η was maximal (on Δ) then η' must also be (on Δ'), and if η was empty (on Δ), then η' must also be empty (on Δ').

Theorem C.20 (Environment Subcontext Bind). *If* $\eta \;:\; \mathsf{env}\,(\Gamma(\Delta))$ *and* $\eta' \;:\; \mathsf{env}\,(\Delta')$ *such that* $\mathtt{agree}\,(\eta, \eta', \Delta, \Delta')$ *then* $\eta \cdot \eta' \;:\; \mathsf{env}\,(\Gamma(\Delta'))$

Proof. Induction on the structure of $\Gamma(-)$, inverting everything in sight. □

Lastly, the structure of the above subcontext replacement operation is compatible with derivatives. Taking the derivative of $\Gamma(\Delta)$ by η yields $\Gamma'(\delta_\eta\,(\Delta))$ for some $\Gamma'(-)$, and for *any* other filler Δ_0 and environment $\eta_0 \;:\; \mathsf{env}\,(\Delta_0)$, the outer derivative bit of the derivative remains unchanged: $\delta_{\eta \cup \eta_0}\,(\Gamma(\Delta_0))$ is $\Gamma'(\delta_{\eta_0}\,(\Delta_0))$

Theorem C.21 (Environment Subcontext Bind Derivative). *If* $\delta_\eta\,(\Gamma(\Delta)) \sim \Gamma_0$ *then there is some* $\Gamma'(-)$ *such that for all* Δ' *and* Δ'' *and* η'*, if* $\delta_{\eta'}\,(\Delta') \sim \Delta''$ *and* $\mathtt{agree}\,(\eta, \eta', \Delta, \Delta')$ *then* $\delta_{\eta \cup \eta'}\,(\Gamma(\Delta')) \sim \Gamma'(\Delta'')$

Proof. Induction on $\Gamma(-)$. □

Theorem C.22 (Environment Lookup). *For any* η*, there is at most one* p *so that* $\eta(x) \mapsto p$*. When* $\eta \;:\; \mathsf{env}\,(\Gamma(x : s))$*, this* p *exists, and* $p \;:\; \mathtt{prefix}\,(s)$*.*

Proof. The "at most one" p is immediate from the fact that η is a deterministic partial function. If $\eta \;:\; \mathsf{env}\,(\Gamma(x : s))$ then $\eta \;:\; \mathsf{env}\,(x : s)$ by Theorem C.19. By inversion, there is some $p \;:\; \mathtt{prefix}\,(s)$ such that $\eta(x) \mapsto p$. □

Theorem C.23 (Environment Lookup Derivative). *Suppose:*

(1) $\eta(x) = p$
(2) $\eta \;:\; \mathsf{env}\,(\Gamma(x : s))$
(3) $\delta_p\,(s) \sim s'$
(4) $\delta_\eta\,(\Gamma(x : s)) \sim \Gamma_0$

Then there is some $\Gamma'(-)$ *such that* $\Gamma_0 = \Gamma'(x : s')$.

Proof. Immediate by Theorem C.21 □

C.4 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 3-place, type-indexed relation.

Definition C.24 (Prefix Concatenation). We define a relation $p \cdot p' \sim p''$.

$$\frac{}{\mathtt{epsEmp} \cdot \mathtt{epsEmp} \sim \mathtt{epsEmp}}$$

$$\frac{p : \mathtt{prefix}(1)}{\mathtt{oneEmp} \cdot p \sim p} \qquad \frac{}{\mathtt{oneFull} \cdot \mathtt{epsEmp} \sim \mathtt{oneFull}}$$

$$\frac{p_1 \cdot p_1' \sim p_1'' \qquad p_2 \cdot p_2' \sim p_2''}{\mathtt{parPair}(p_1, p_2) \cdot \mathtt{parPair}(p_1', p_2') \sim \mathtt{parPair}(p_1'', p_2'')}$$

$$\frac{p \cdot p' \sim p''}{\mathtt{catFst}(p) \cdot \mathtt{catFst}(p') \sim \mathtt{catFst}(p'')} \qquad \frac{p \cdot p' \sim p''}{\mathtt{catFst}(p) \cdot \mathtt{catBoth}(p', p''') \sim \mathtt{catBoth}(p'', p''')}$$

$$\frac{p' \cdot p'' \sim p'''}{\mathtt{catBoth}(p, p') \cdot p'' \sim \mathtt{catBoth}(p, p''')}$$

$$\frac{}{\mathtt{sumEmp} \cdot p \sim p} \qquad \frac{p' \cdot p'' \sim p''}{\mathtt{sumInl}(p) \cdot p' \sim \mathtt{sumInl}(p'')} \qquad \frac{p \cdot p' \sim p''}{\mathtt{sumInr}(p) \cdot p' \sim \mathtt{sumInr}(p'')}$$

$$\frac{}{\mathtt{starEmp} \cdot p \sim p} \qquad \frac{}{\mathtt{starDone} \cdot \mathtt{epsEmp} \sim \mathtt{starDone}}$$

$$\frac{p \cdot p' \sim p''}{\mathtt{starFirst}(p) \cdot \mathtt{catFst}(p') \sim \mathtt{starFirst}(p'')}$$

$$\frac{p \cdot p' \sim p''}{\mathtt{starFirst}(p) \cdot \mathtt{catBoth}(p', p''') \sim \mathtt{starRest}(p'', p''')}$$

$$\frac{p' \cdot p'' \sim p'''}{\mathtt{starRest}(p, p') \cdot p'' \sim \mathtt{starRest}(p, p''')}$$

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

Theorem C.25 (Prefix Concatenation Function). *For all* p, p' *and* s*, there is at most one* p'' *such that* $p \cdot p' \sim p''$*. If* $p : \mathtt{prefix}(s)$ *and* $p' : \mathtt{prefix}\left(\delta_p(s)\right)$*, then such a* p'' *exists, and satisfies:*

(1) $p'' : \mathtt{prefix}(s)$
(2) $\delta_{p''}(s) = \delta_{p'}\left(\delta_p(s)\right)$

Proof. Existence, (1), and (2) follow by induction on the derivation of $p : \mathtt{prefix}(s)$. Uniqueness is immediate by the fact that the relation is a function. □

Theorem C.26 (Prefix Concatenation Empty). *If* $p : \texttt{prefix}\,(s)$, *then* $\texttt{emp}_s \cdot p \sim p$ *and* $p \cdot \texttt{emp}_{\delta_p(s)} \sim p$

Proof. Induction on the derivation of $p : \texttt{prefix}\,(s)$. □

Theorem C.27 (Maximal Prefix Concatenation). *Suppose* $p \cdot p' \sim p''$. *If* p'' *is maximal, then* p' *is maximal. If* p *or* p' *is maximal, then* p'' *is maximal. Moreover, if* p *is maximal, then* $p'' = p$.

Proof. Induction on the derivation of $p \cdot p' \sim p''$. □

Theorem C.28 (Prefix Concatenation Associativity). $p \cdot (p' \cdot p'') = (p \cdot p') \cdot p''$, *when defined.*

Proof. Induction on derivations of concatenation. □

Definition C.29 (Environment Concatenation). We write $\eta \cdot \eta' \sim \eta''$ to mean that η'' is the function defined on the largest subset S of $\text{Dom}\,(\eta) \cap \text{Dom}\,(\eta')$ such that for all $x \in S$, the prefix concatenation $\eta(x) \cdot \eta'(x) \sim p$ exists, and $\eta''(x) = p$, for all $x \in S$.

Theorem C.30 (Environment Concatenation Function). *For any* η *and* η', *there is at most one* η'' *such that* $\eta \cdot \eta' \sim \eta''$, *and such an* η'' *exists when* $\eta : \texttt{env}\,(\Gamma)$ *and* $\delta_\eta\,(\Gamma) \sim \Gamma'$ *and* $\eta' : \texttt{prefix}\,(\Gamma')$.

Proof. Uniqueness by the "greatest" property, existence by Theorem C.25. □

Theorem C.31 (Environment Concatenation Correctness). *If* $\eta : \texttt{env}\,(\Gamma)$ *and* $\delta_\eta\,(\Gamma) \sim \Gamma'$ *and* $\eta' : \texttt{env}\,(\Gamma')$, *and* $\eta \cdot \eta' \sim \eta''$, *then* $\eta'' : \texttt{env}\,(\Gamma)$, *and if* $\delta_{\eta'}\,(\Gamma') \sim \Gamma''$ *then* $\delta_{\eta''}\,(\Gamma) \sim \Gamma''$.

Theorem C.32 (Environment Concatenation Empty). *If* $\eta : \texttt{env}\,(\Gamma)$ *and* $\eta \cdot \eta' \sim \eta''$, *then:*

- *If* η $\texttt{emptyOn}\,S$ *and then* $\eta''|_S = \eta'|_S$
- *If* η' $\texttt{emptyOn}\,S$, *then* $\eta''|_S = \eta|_S$

Proof. Induction on the derivation of $\eta : \texttt{env}\,(\Gamma)$, using Theorem C.26. □

Theorem C.33 (Maximal Environment Concatenation). *If* $\eta \cdot \eta' \sim \eta''$, *then* η $\texttt{maximalOn}\,S$ *or* η' $\texttt{maximalOn}\,S$ *if and only if* η'' $\texttt{maximalOn}\,S$.

Proof. Immediate corollary of Theorem C.27 □

Theorem C.34 (Prefix Concatenation Associativity). $\eta \cdot (\eta' \cdot \eta'') = (\eta \cdot \eta') \cdot \eta''$, *when defined.*

Proof. Corollary of Theorem C.28 □

C.5 Historical Contexts

Definition C.35 (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 C.36 (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 Γ 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 $\texttt{toPrefix}_s\,(v)$ for the maximal prefix of type s that it corresponds to. Dually, for a maximal prefix $p : \texttt{prefix}\,(s)$, we write $\langle p \rangle$ for the STLC value of type $\langle s \rangle$ it corresponds to.

Definition C.37 (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 $\Omega' \vdash M[\theta] : A$. We lift this substitution action to λ^{ST} terms compositionally, substituting into all historical terms. We write a list of such terms as $\overline{M}$, and lift the typing relation and semantics to lists of terms, written $\Omega \vdash \overline{M} : \overline{A}$ and $\overline{M} \downarrow \theta$.

C.6 Context Subtyping

The following is a full listing of subtyping rules.

Definition C.38 (Subtyping).

$$\textsc{Sub-Cong}\ \frac{\Delta \mathrel{<:} \Delta'}{\Gamma(\Delta) \mathrel{<:} \Gamma(\Delta')} \qquad \textsc{Sub-Refl}\ \frac{}{\Gamma \mathrel{<:} \Gamma} \qquad \textsc{Sub-Comma-Exc}\ \frac{}{\Gamma , \Delta \mathrel{<:} \Delta , \Gamma}$$

$$\textsc{Sub-Sng-Wkn}\ \frac{}{x : s \mathrel{<:} \cdot} \qquad \textsc{Sub-Comma-Wkn}\ \frac{}{\Gamma , \Delta \mathrel{<:} \Gamma} \qquad \textsc{Sub-Semic-Wkn-1}\ \frac{}{\Gamma ; \Delta \mathrel{<:} \Gamma} \qquad \textsc{Sub-Semic-Wkn-2}\ \frac{}{\Gamma ; \Delta \mathrel{<:} \Delta}$$

$$\textsc{Sub-Comma-Unit}\ \frac{}{\Gamma \mathrel{<:} \Gamma , \cdot} \qquad \textsc{Sub-Semic-Unit-1}\ \frac{}{\Gamma \mathrel{<:} \Gamma ; \cdot} \qquad \textsc{Sub-Semic-Unit-2}\ \frac{}{\Gamma \mathrel{<:} \cdot ; \Gamma}$$

Environment typing is preserved by subtyping, and derivatives preserve subtyping relations between contexts.

Theorem C.39 (Subtyping Preserves Environments). *If* $\eta : \mathsf{env}\,(\Gamma)$ *and* $\Gamma \mathrel{<:} \Delta$ *then* $\eta : \mathsf{env}\,(\Delta)$

Proof. By induction on $\Gamma \mathrel{<:} \Delta$, and inversion on $\eta : \mathsf{env}\,(\Gamma)$. □

Theorem C.40 (Derivatives Preserve Subtyping). *Suppose* $\Gamma \mathrel{<:} \Delta$ *and* $\delta_\eta\,(\Gamma) \sim \Gamma'$ *and* $\delta_\eta\,(\Delta) \sim \Delta'$. *Then,* $\Gamma' \mathrel{<:} \Delta'$.

Proof. By cases on $\Gamma \mathrel{<:} \Delta$, inverting the derivations of $\delta_\eta\,(\Gamma) \sim \Gamma'$ and $\delta_\eta\,(\Delta) \sim \Delta'$, and using the determinism of the derivative relation. □

C.7 Type System

C.7.1 Inertness. Most terms, like variables or case expressions, require some non-empty amount of input to arrive for them to produce a non-empty output. However, this is not true of all terms:

constants like `()` and `nil`, (some) sequential terms like $((); e)$ and $\texttt{sink} :: e$, and sum terms $\texttt{inl}(e)$ produce nonempty output even when given an entirely empty input prefix. Terms like these contain "information" that they are always ready to produce, even if there is no input to drive them forward. We call terms that are always ready to produce output *jumpy*, and terms that are not *inert*.

As described in Section 4.3, the type system requires that let-bound terms are always inert to guarantee soundness of the semantics. In particular, inertness is what guarantees that the "agreement" (Definition C.18) requirement in Theorem C.20 to hold in the soundness case for T-Let. For arbitrary terms, the maximality component of agreement always holds (this by Lemma C.49), but the emptiness component of agreement requires inertness.

To enforce that the bodies of let-bindings are inert, we track a syntactic over-approximation of inertness with the type system, essentially as an *effect*. This is accomplished by giving every typing judgment an *inertness annotation*, $i ::= \texttt{I} \mid \texttt{J}$, and we ensure that if e is typed with annotation `Inert`, then e produces empty output when given an empty input. This invariant is proved as an additional consequence to the soundness theorem.

We note that the choice to include inertness in the type system itself, as opposed to a predicate on (typed) terms, is essentially an arbitrary one: we choose the former to minimize the number of assumptions running around in our proofs.

For the most part, the inertness analysis is straightforward. Constants like `()` and `nil`, and injections like $\texttt{inl}\,(e)$, $\texttt{inr}\,(e)$, and $e_1 :: e_2$ (secretly the right injection into $\varepsilon + s \cdot s^\star$) all have annotation J. Non-buffering elimination forms have the same inertness as their bodies, and variables and `sink` are inert. The most important ones are in the rules T-Cat-R and T-Plus-L (and the similar ones in T-Star-L and T-Plus-L).

The inertness requirement for T-Cat-R says that if the resulting term $(e_1; e_2)$ is to be typed as inert, e_1 must be inert, and the type of e_1 must not be `null`. Otherwise, $(e_1; e_2)$ could produce a maximal .

The rule for T-Plus-L says that it is inert when the buffer environment does not yet include a decision for which way to go ($\eta(z) = \texttt{sumEmp}$). Note that in practice, this is always satisfied. At the beginning of execution, η maps all variables to empty prefixes, and as soon as $\eta(z)$ gets either $\texttt{sumInl}(p)$ or $\texttt{sumInr}(p)$, we step to the corresponding branch. In fact, the result of *every* step is inert: otherwise we would've output a larger prefix in that step!

Definition C.41 (Typing Rules). Figure 7 and Figure 8 present the full typing rules.

Definition C.42 (Recursion Signature). A recursion signature Σ is either empty (signaling that typechecking is not in the body of a recursive function), or the signature $\Omega \mid \Gamma \longrightarrow s \mathbin{@} i$ of a sequent which defines the recursive function we are currently checking the body of. $\Sigma ::= \emptyset \mid (\Omega \mid \Gamma \longrightarrow s \mathbin{@} i)$

These typing rules are mutually defined with another typing judgment $\Omega \mid \Gamma \vdash_\Sigma A : \Gamma' \mathbin{@} i$, meaning that A is a well-typed set of arguments (hence A) for a recursive call to a function accepting inputs Γ'. Here, A is an *tree* of terms, with either comma or semicolon nodes. This judgment ensures that $e_{\Gamma'}$ has well-typed bindings for every variable $x : s$ in Γ', and that the variables that $e_{\Gamma'}$ uses are used in accordance with Γ, its context.

Definition C.43 (Recursive Argument Typing).

$$A ::= \cdot \mid e \mid (A, A') \mid (A; A') \mid (\cdot; A)$$

$$\text{T-Eps-R}\ \frac{}{\Omega \mid \Gamma \vdash_\Sigma \mathtt{sink} : \varepsilon \,@\, i} \qquad \text{T-One-R}\ \frac{}{\Omega \mid \Gamma \vdash_\Sigma () : 1 \,@\, \mathtt{J}} \qquad \text{T-Var}\ \frac{}{\Omega \mid \Gamma(x : s) \vdash_\Sigma x : s \,@\, i}$$

$$\text{T-Sub}\ \frac{\Omega \mid \Delta \vdash_\Sigma e : s \,@\, i \qquad \Gamma <: \Delta}{\Omega \mid \Gamma \vdash_\Sigma e : s \,@\, i}$$

$$\text{T-Par-R}\ \frac{\Omega \mid \Gamma \vdash_\Sigma e_1 : s \,@\, i \qquad \Omega \mid \Gamma \vdash_\Sigma e_2 : t \,@\, i}{\Omega \mid \Gamma \vdash_\Sigma (e_1, e_2) : s \| t \,@\, i}$$

$$\text{T-Cat-R}\ \frac{\Omega \mid \Gamma \vdash_\Sigma e_1 : s \,@\, i_1 \qquad \Omega \mid \Delta \vdash_\Sigma e_2 : t \,@\, i_2 \qquad i_3 = \mathtt{I} \implies i_1 = \mathtt{I} \wedge \neg\,(s\ \mathtt{null})}{\Omega \mid \Gamma\,;\Delta \vdash_\Sigma (e_1; e_2) : s \cdot t \,@\, i_3}$$

$$\text{T-Par-L}\ \frac{\Omega \mid \Gamma(x : s, y : t) \vdash_\Sigma e : r \,@\, i}{\Omega \mid \Gamma(z : s \| t) \vdash_\Sigma \mathtt{let}\ (x, y)\ =\ z\ \mathtt{in}\ e : r \,@\, i} \qquad \text{T-Cat-L}\ \frac{\Omega \mid \Gamma(x : s\,; y : t) \vdash_\Sigma e : r \,@\, i}{\Omega \mid \Gamma(z : s \cdot t) \vdash_\Sigma \mathtt{let}_t\ (x; y)\ =\ z\ \mathtt{in}\ e : r \,@\, i}$$

$$\text{T-Plus-R-1}\ \frac{\Omega \mid \Gamma \vdash_\Sigma e : s \,@\, i}{\Omega \mid \Gamma \vdash_\Sigma \mathtt{inl}(e) : s + t \,@\, \mathtt{J}} \qquad \text{T-Plus-R-2}\ \frac{\Omega \mid \Gamma \vdash_\Sigma e : t \,@\, i}{\Omega \mid \Gamma \vdash_\Sigma \mathtt{inr}(e) : s + t \,@\, i}$$

$$\text{T-Plus-L}\ \frac{\begin{array}{c}\eta\ :\ \mathsf{env}\,(\Gamma(z : s + t)) \qquad \delta_\eta\,(\Gamma(z : s + t)) \sim \Gamma' \\ \Omega \mid \Gamma(x : s) \vdash_\Sigma e_1 : r \,@\, i_1 \qquad \Omega \mid \Gamma(y : t) \vdash_\Sigma e_2 : r \,@\, i_2 \qquad i = \mathtt{I} \implies \eta(z) = \mathtt{sumEmp}\end{array}}{\Omega \mid \Gamma' \vdash_\Sigma \mathtt{case}_r\,(\eta; z, x.e_1, y.e_2) : r \,@\, i}$$

Fig. 7. Full Typing Rules (Part 1)

$$\text{T-Args-Emp}\ \frac{}{\Omega \mid \Gamma \vdash_\Sigma \cdot : \cdot \,@\, i} \qquad \text{T-Args-Sng}\ \frac{\Omega \mid \Gamma \vdash_\Sigma e : s \,@\, i}{\Omega \mid \Gamma \vdash_\Sigma e : (x : s)\ @\, i} \qquad \text{T-Args-Semic-1}\ \frac{\Omega \mid \Gamma \vdash_\Sigma A : \Delta \,@\, i_1 \qquad \Omega \mid \Gamma' \vdash_\Sigma A' : \Delta' \,@\, i_2}{\Omega \mid \Gamma\,;\Gamma' \vdash_\Sigma (A; A') : \Delta\,;\Delta' \,@\, i_3}$$

$$\text{T-Args-Semic-2}\ \frac{\Omega \mid \Gamma' \vdash_\Sigma A : \Delta' \,@\, i \qquad \Delta\ \mathtt{null}}{\Omega \mid \Gamma\,;\Gamma' \vdash_\Sigma (\cdot; A) : \Delta\,;\Delta' \,@\, i} \qquad \text{T-Args-Comma}\ \frac{\Omega \mid \Gamma \vdash_\Sigma A : \Delta \,@\, i \qquad \Omega \mid \Gamma \vdash_\Sigma A' : \Delta' \,@\, i}{\Omega \mid \Gamma \vdash_\Sigma (A, A') : \Delta\,,\Delta' \,@\, i}$$

Buffering Rules. 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 η, which we gather until it includes a maximal prefix of $x : s$.

$$\text{T-Wait}\ \frac{\begin{array}{c}\eta\ :\ \mathsf{env}\,(\Gamma(x : s)) \\ \delta_\eta\,(\Gamma(x : s)) \sim \Gamma' \qquad \Omega, x : \langle s \rangle \mid \Gamma(\cdot) \vdash_\Sigma e : s \,@\, ii' = \mathtt{I} \implies \neg\,(\eta(z)\ \mathtt{maximal}) \wedge \neg\,(s\ \mathtt{null})\end{array}}{\Omega \mid \Gamma' \vdash_\Sigma \mathtt{wait}_{\eta,t}(x)\,(e) : t \,@\, i'}$$

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 η of the input, the term is expecting the rest of the context. Users of

$$\textsc{T-Star-R-1}\ \frac{}{\Omega \mid \Gamma \vdash_\Sigma \mathtt{nil} : s^\star \mathbin{@} \mathtt{J}} \qquad \textsc{T-Star-R-2}\ \frac{\Omega \mid \Gamma \vdash_\Sigma e_1 : s \mathbin{@} i_1 \qquad \Omega \mid \Delta \vdash_\Sigma e_2 : s^\star \mathbin{@} i_2}{\Omega \mid \Gamma ; \Delta \vdash_\Sigma e_1 :: e_2 : s^\star \mathbin{@} \mathtt{J}}$$

$$\textsc{T-Star-L}\ \frac{\begin{array}{c}\eta \ :\ \mathsf{env}\left(\Gamma(z : s^\star)\right) \qquad \delta_\eta\left(\Gamma(z : s^\star)\right) \sim \Gamma' \\ \Omega \mid \Gamma(\cdot) \vdash_\Sigma e_1 : r \mathbin{@} i_1 \qquad \Omega \mid \Gamma(x : s; xs : s^\star) \vdash_\Sigma e_2 : r \mathbin{@} i_2 \qquad i = \mathtt{I} \implies \eta(z) = \mathtt{starEmp}\end{array}}{\Omega \mid \Gamma' \vdash_\Sigma \mathsf{case}_{s,r}\left(\eta; z, e_1, x.xs.e_2\right) : r \mathbin{@} i}$$

$$\textsc{T-HistPgm}\ \frac{\Omega \vdash M : \langle s \rangle}{\Omega \mid \Gamma \vdash_\Sigma \langle M : s \rangle : s \mathbin{@} \mathtt{J}}$$

$$\textsc{T-Wait}\ \frac{\begin{array}{c}\eta \ :\ \mathsf{env}\left(\Gamma(x : s)\right) \qquad \delta_\eta\left(\Gamma(x : s)\right) \sim \Gamma' \\ \Omega, x : \langle s \rangle \mid \Gamma(\cdot) \vdash_\Sigma e : t \mathbin{@} i \qquad i' = \mathtt{I} \implies \neg\left(\eta(z)\ \mathtt{maximal}\right) \wedge \neg\left(s\ \mathtt{null}\right)\end{array}}{\Omega \mid \Gamma' \vdash_\Sigma \mathtt{wait}_{\eta,t}(x)\ (e) : t \mathbin{@} i'}$$

$$\textsc{T-Let}\ \frac{\Omega \mid \Delta \vdash_\Sigma e_1 : s \mathbin{@} \mathtt{I} \qquad \Omega \mid \Gamma(x : s) \vdash_\Sigma e_2 : t \mathbin{@} i}{\Omega \mid \Gamma(\Delta) \vdash_\Sigma \mathtt{let}\, x = e_1\, \mathtt{in}\, e_2 : t \mathbin{@} i} \qquad \textsc{T-Rec}\ \frac{\Omega' \mid \Gamma' \vdash_{\Omega \mid \Gamma \to s \mathbin{@} i} A : \Gamma \mathbin{@} i \qquad \Omega' \vdash \overline{M} : \Omega}{\Omega' \mid \Gamma' \vdash_{\Omega \mid \Gamma \to s \mathbin{@} i} \mathtt{rec}\left\{\overline{M}\right\}(A) : s \mathbin{@} i}$$

$$\textsc{T-Fix}\ \frac{\Omega \mid \Gamma \vdash_{\Omega \mid \Gamma \to s \mathbin{@} i} e : s \mathbin{@} i \qquad \Omega' \vdash \overline{M} : \Omega \qquad \Omega' \mid \Gamma' \vdash_\Sigma A : \Gamma \mathbin{@} i}{\Omega' \mid \Gamma' \vdash_\Sigma \mathtt{fix}\left\{\overline{M}\right\}(A)\,.\,(e) : s \mathbin{@} i}$$

$$\textsc{T-ArgsLet}\ \frac{\Omega \mid \Gamma' \vdash_\Sigma A : \Gamma \mathbin{@} i \qquad \Omega \mid \Gamma \vdash_\Sigma e : s \mathbin{@} i}{\Omega \mid \Gamma' \vdash_\Sigma \mathtt{let}\, \Gamma = A\, \mathtt{in}\, e : s \mathbin{@} i}$$

Fig. 8. Full Typing Rules (Part 2)

the calculus need not worry about this detail: when writing programs and when the program starts running, the buffer is empty: $\eta = \mathsf{emp}_{\Gamma(\Delta)}$, and since $\delta_\eta\left(\Gamma(\Delta)\right) = \Gamma(\Delta)$, this 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.

C.8 Sink Terms

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

Definition C.44 (Sink Terms). We define a term sink_p by induction on p.

- $\mathsf{sink}_{\mathsf{epsEmp}} = \mathsf{sink}$
- $\mathsf{sink}_{\mathsf{oneEmp}} = \mathsf{sink}$
- $\mathsf{sink}_{\mathsf{oneFull}} = \mathsf{sink}$
- $\mathsf{sink}_{\mathsf{parPair}(p_1,p_2)} = \left(\mathsf{sink}_{p_1}, \mathsf{sink}_{p_2}\right)$
- $\mathsf{sink}_{\mathsf{catFst}(p)} = \mathsf{sink}_p$
- $\mathsf{sink}_{\mathsf{catBoth}(p_1,p_2)} = \mathsf{sink}_{p_2}$
- $\mathsf{sink}_{\mathsf{sumEmp}} = \mathsf{sink}$

- $\mathsf{sink}_{\mathtt{sumInl}(p)} = \mathsf{sink}_p$
- $\mathsf{sink}_{\mathtt{sumInr}(p)} = \mathsf{sink}_p$
- $\mathsf{sink}_{\mathtt{starEmp}} = \mathsf{sink}$
- $\mathsf{sink}_{\mathtt{starDone}} = \mathsf{sink}$
- $\mathsf{sink}_{\mathtt{starFirst}(p)} = \mathsf{sink}_p$
- $\mathsf{sink}_{\mathtt{starRest}(p,p')} = \mathsf{sink}_{p'}$

Note that (because it's easier to have this be a function rather than a relation) sink terms are defined for *all* prefixes rather than just the maximal ones.

Sink terms are closed, and have the type we expect for a stream transformer that has just emitted an maximal p of type s.

Theorem C.45 (Sink Terms Typing). *If* p $\mathtt{maximal}$ *and* $p : \mathtt{prefix}\,(s)$ *and* $\delta_p\,(s) \sim s'$, *then* $\cdot \mid \Gamma \vdash_\emptyset \mathsf{sink}_p : s' \,@\, \mathtt{I}$

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

Theorem C.46 (Sink Term Concatenation). *If* $p \cdot p' \sim p''$, *then* $\mathsf{sink}_{p'} = \mathsf{sink}_{p''}$.

Proof. By induction on $p \cdot p' \sim p''$. □

Theorem C.47 (Fixpoint Substitution). *For* e, e' *terms, we define* $e\,[e'/\mathtt{rec}]$ *compositionally over the structure of* e, *with the only two interesting cases being:*

$$\left(\mathtt{rec}\left\{\overline{M}\right\}(A)\right)[e'/\mathtt{rec}] = \mathtt{fix}\left\{\overline{M}\right\}(A\,[e'/\mathtt{rec}]) \,.\, (e')$$

and

$$\left(\mathtt{fix}\left\{\overline{M}\right\}(A) \,.\, (e)\right)[e'/\mathtt{rec}] = \mathtt{fix}\left\{\overline{M}\right\}(A\,[e'/\mathtt{rec}]) \,.\, (e)$$

We define this mutually with a substitution for arguments A, *with* $A\,[e'/\mathtt{rec}]$ *defined compositionally over the structure of* A.

Then if $\Omega \mid \Gamma \vdash_{\Omega|\Gamma \to s \,@\, i'} e' : s \,@\, i'$, *we have:*

(1) *If* $\Omega' \mid \Delta \vdash_{\Omega|\Gamma \to s \,@\, i'} e : t \,@\, i$ *then* $\Omega' \mid \Delta \vdash_\cdot e\,[e'/\mathtt{rec}] : t \,@\, i$

(2) *If* $\Omega' \mid \Delta \vdash_{\Omega|\Gamma \to s \,@\, i'} A : \Gamma' \,@\, i$, *then* $\Omega' \mid \Delta \vdash_\cdot A\,[e'/\mathtt{rec}] : \Gamma' \,@\, i$

Proof. (1) and (2) are proved by a routine simultaneous induction on typing derivations. □

C.9 Semantics

Definition C.48 (Semantics). We define relations $p \Rightarrow e \downarrow^n e' \Rightarrow p'$ (as shown in Figures 9 and 10), and $\eta \Rightarrow A@\Gamma \downarrow^n A' \Rightarrow \eta'$ (as shown in Figure 11).

Recursive Argument Semantics. The arguments semantics $\eta \Rightarrow A@\Gamma \downarrow^n A' \Rightarrow \eta'$ accepts an environment η and runs it through A to produce an environment $\eta' : \mathsf{env}\,(\Gamma)$. This relation is essentially the same as evaluating a large nested tree T-Cat-R T-Par-R terms, structured like the context Γ. The only difference is that, because context derivatives do not remove the left component of a semicolon context (the Γ in $\Gamma\,;\Delta$) after a maximal prefix has arrived, we have a special term former $(\cdot;A')$ for cat-pair terms $(A;A')$ that have crossed over. The context is required in the semantics so we can compute the empty environment in S-Args-Semic-1-1 and S-Args-Semic-2.

$$\text{S-Eps-R}\ \frac{}{\eta \Rightarrow \mathtt{sink} \downarrow^{n} \mathtt{sink} \Rightarrow \mathtt{epsEmp}} \qquad \text{S-One-R}\ \frac{}{\eta \Rightarrow () \downarrow^{n} \mathtt{sink} \Rightarrow \mathtt{oneFull}} \qquad \text{S-Var}\ \frac{\eta(x) \mapsto p}{\eta \Rightarrow x \downarrow^{n} x \Rightarrow p}$$

$$\text{S-Par-R}\ \frac{\eta \Rightarrow e_1 \downarrow^{n_1} e_1' \Rightarrow p_1 \qquad \eta \Rightarrow e_2 \downarrow^{n_2} e_2' \Rightarrow p_2}{\eta \Rightarrow (e_1, e_2) \downarrow^{n_1+n_2} (e_1', e_2') \Rightarrow \mathtt{parPair}(p_1, p_2)} \qquad \text{S-Cat-R-1}\ \frac{\eta \Rightarrow e_1 \downarrow^{n} e_1' \Rightarrow p \qquad \neg\,(p\ \mathtt{maximal})}{\eta \Rightarrow (e_1; e_2) \downarrow^{n} (e_1'; e_2) \Rightarrow \mathtt{catFst}(p)}$$

$$\text{S-Cat-R-2}\ \frac{\eta \Rightarrow e_1 \downarrow^{n_1} e_1' \Rightarrow p_1 \qquad p_1\ \mathtt{maximal} \qquad \eta \Rightarrow e_2 \downarrow^{n_2} e_2' \Rightarrow p_2}{\eta \Rightarrow (e_1; e_2) \downarrow^{n_1+n_2} e_2' \Rightarrow \mathtt{catBoth}(p_1, p_2)}$$

$$\text{S-Par-L}\ \frac{\eta(z) \mapsto \mathtt{parPair}(p_1, p_2) \qquad \eta[x \mapsto p_1, y \mapsto p_2] \Rightarrow e \downarrow^{n} e' \Rightarrow p'}{\eta \Rightarrow \mathtt{let}\ (x, y)\ =\ z\ \mathtt{in}\ e \downarrow^{n} \mathtt{let}\ (x, y)\ =\ z\ \mathtt{in}\ e \Rightarrow p}$$

$$\text{S-Cat-L-1}\ \frac{\eta(z) \mapsto \mathtt{catFst}(p) \qquad \eta[x \mapsto p, y \mapsto \mathtt{emp}_t] \Rightarrow e \downarrow^{n} e' \Rightarrow p'}{\eta \Rightarrow \mathtt{let}_t\ (x; y)\ =\ z\ \mathtt{in}\ e \downarrow^{n} \mathtt{let}_t\ (x; y)\ =\ z\ \mathtt{in}\ e' \Rightarrow p'}$$

$$\text{S-Cat-L-2}\ \frac{\eta(z) \mapsto \mathtt{catBoth}(p_1, p_2) \qquad \eta[x \mapsto p_1, y \mapsto p_2] \Rightarrow e \downarrow^{n} e' \Rightarrow p}{\eta \Rightarrow \mathtt{let}_t\ (x; y)\ =\ z\ \mathtt{in}\ e \downarrow^{n} \mathtt{let}\ x = \mathtt{sink}_{p_1}\ \mathtt{in}\ e'[z/y] \Rightarrow p}$$

$$\text{S-Plus-R-1}\ \frac{\eta \Rightarrow e \downarrow^{n} e' \Rightarrow p}{\eta \Rightarrow \mathtt{inl}(e) \downarrow^{n} e' \Rightarrow \mathtt{inl}(p)} \qquad \text{S-Plus-R-2}\ \frac{\eta \Rightarrow e \downarrow^{n} e' \Rightarrow p}{\eta \Rightarrow \mathtt{inr}(e) \downarrow^{n} e' \Rightarrow \mathtt{inr}(p)}$$

$$\text{S-Plus-L-1}\ \frac{\eta' \cdot \eta \sim \eta'' \qquad \eta''(z) = \mathtt{sumEmp}}{\eta \Rightarrow \mathtt{case}_r\ (\eta'; z, x.e_1, y.e_2) \downarrow^{n} \mathtt{case}_r\ (\eta''; z, x.e_1, y.e_2) \Rightarrow \mathtt{emp}_r}$$

$$\text{S-Plus-L-2}\ \frac{\eta' \cdot \eta \sim \eta'' \qquad \eta''(z) = \mathtt{sumInl}(p) \qquad \eta''[x \mapsto p] \Rightarrow e_1 \downarrow^{n} e_1' \Rightarrow p'}{\eta \Rightarrow \mathtt{case}_r\ (\eta'; z, x.e_1, y.e_2) \downarrow^{n} e_1'[z/x] \Rightarrow p'}$$

$$\text{S-Plus-L-3}\ \frac{\eta' \cdot \eta \sim \eta'' \qquad \eta''(z) = \mathtt{sumInr}(p) \qquad \eta''[y \mapsto p] \Rightarrow e_2 \downarrow^{n} e_2' \Rightarrow p'}{\eta \Rightarrow \mathtt{case}_r\ (\eta'; z, x.e_1, y.e_2) \downarrow^{n} e_2'[z/y] \Rightarrow p'}$$

Fig. 9. Semantics (part 1)

Semantics of Buffering. The 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 (S-Wait-1 and S-Wait-2 in Figure 10). In both cases, we take the incoming environment η, and concatenate it onto the buffer η', to get the combined prefix η''. We then dispatch on whether η'' is enough input to run the continuation e. In this case, "enough" means that η'' contains a maximal prefix p of $x : s$. If it does (P-Wait-2), we run the continuation, substituting the maximal prefix in for the (historical) occurrences of x. If it does not,

S-Star-R-1
$$\frac{}{\eta \Rightarrow \mathtt{nil} \downarrow^{n} \mathtt{sink} \Rightarrow \mathtt{starDone}}$$

S-Star-R-2-1
$$\frac{\eta \Rightarrow e_1 \downarrow^{n} e_1' \Rightarrow p \qquad \neg\,(p\ \mathtt{maximal})}{\eta \Rightarrow e_1 :: e_2 \downarrow^{n} (e_1'; e_2) \Rightarrow \mathtt{starFirst}(p)}$$

S-Star-R-2-2
$$\frac{\eta \Rightarrow e_1 \downarrow^{n_1} e_1' \Rightarrow p_1 \qquad p_1\ \mathtt{maximal} \qquad \eta \Rightarrow e_2 \downarrow^{n_2} e_2' \Rightarrow p_2}{\eta \Rightarrow e_1 :: e_2 \downarrow^{n_1+n_2} e_2' \Rightarrow \mathtt{starRest}(p_1, p_2)}$$

S-Star-L-1
$$\frac{\eta' \cdot \eta \sim \eta'' \qquad \eta''(z) = \mathtt{starEmp}}{\eta \Rightarrow \mathtt{case}_{s,r}\,(\eta'; z, e_1, x.xs.e_2) \downarrow^{n} \mathtt{case}_{s,r}\,(\eta''; z, e_1, x.xs.e_2) \Rightarrow \mathtt{emp}_r}$$

S-Star-L-2
$$\frac{\eta' \cdot \eta \sim \eta'' \qquad \eta''(z) = \mathtt{starDone} \qquad \eta'' \Rightarrow e_1 \downarrow^{n} e_1' \Rightarrow p}{\eta \Rightarrow \mathtt{case}_{s,r}\,(\eta'; z, e_1, x.xs.e_2) \downarrow^{n} e_1' \Rightarrow p}$$

S-Star-L-3
$$\frac{\eta' \cdot \eta \sim \eta'' \qquad \eta''(z) = \mathtt{starFirst}(p) \qquad \eta''[x \mapsto p, y \mapsto \mathtt{emp}_{s^\star}] \Rightarrow e_2 \downarrow^{n} e_2' \Rightarrow p'}{\eta \Rightarrow \mathtt{case}_{s,r}\,(\eta'; z, e_1, x.xs.e_2) \downarrow^{n} \mathtt{let}_{s^\star}\ (x; y)\ =\ z\ \mathtt{in}\ e_2' \Rightarrow p'}$$

S-Star-L-4
$$\frac{\eta' \cdot \eta \sim \eta'' \qquad \eta''(z) = \mathtt{starRest}(p, p') \qquad \eta''[x \mapsto p, y \mapsto p'] \Rightarrow e_2 \downarrow^{n} e_2' \Rightarrow p''}{\eta \Rightarrow \mathtt{case}_{s,r}\,(\eta'; z, e_1, x.xs.e_2) \downarrow^{n} \mathtt{let}\ x = \mathtt{sink}_p\ \mathtt{in}\ e_2'[z/xs] \Rightarrow p''}$$

S-Let
$$\frac{\eta \Rightarrow e_1 \downarrow^{n_1} e_1' \Rightarrow p \qquad \eta[x \mapsto p] \Rightarrow e_2 \downarrow^{n_2} e_2' \Rightarrow p'}{\eta \Rightarrow \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \downarrow^{n_1+n_2} \mathtt{let}\ x = e_1'\ \mathtt{in}\ e_2' \Rightarrow p'}$$

S-HistPgm
$$\frac{M \downarrow v \qquad p = \mathtt{toPrefix}_s\,(v)}{\eta \Rightarrow \langle M : s \rangle \downarrow^{n} \mathtt{sink}_p \Rightarrow p}$$

S-Wait-1
$$\frac{\eta' \cdot \eta \sim \eta'' \qquad \eta''(x) = p \qquad \neg\,(p\ \mathtt{maximal})}{\eta \Rightarrow \mathtt{wait}_{\eta',t}(x)\,(e) \downarrow^{n} \mathtt{wait}_{\eta'',t}(x)\,(e) \Rightarrow \mathtt{emp}_t}$$

S-Wait-2
$$\frac{\eta' \cdot \eta \sim \eta'' \qquad \eta''(x) = p \qquad p\ \mathtt{maximal} \qquad \eta'' \Rightarrow e[\langle p \rangle / x] \downarrow^{n} e' \Rightarrow p'}{\eta \Rightarrow \mathtt{wait}_{\eta',t}(x)\,(e) \downarrow^{n} e' \Rightarrow p'}$$

S-Fix
$$\frac{\overline{M} \downarrow \theta \qquad \eta \Rightarrow \mathtt{let}\ \Gamma = A\ \mathtt{in}\ e\ [e/\mathtt{rec}]\ [\theta] \downarrow^{n} e' \Rightarrow p}{\eta \Rightarrow \mathtt{fix}\left\{\overline{M}\right\}\,(A)\,.\,(e) \downarrow^{n+1} e' \Rightarrow p}$$

S-ArgsLet
$$\frac{\eta \Rightarrow A@\Gamma \downarrow^{n_1} A' \Rightarrow \eta' \qquad \eta' \Rightarrow e \downarrow^{n_2} e' \Rightarrow p \qquad \delta_{\eta'}\,(\Gamma) \sim \Gamma'}{\eta \Rightarrow \mathtt{let}\ \Gamma = A\ \mathtt{in}\ e \downarrow^{n_1+n_2} \mathtt{let}\ \Gamma' = A'\ \mathtt{in}\ e' \Rightarrow p}$$

Fig. 10. Semantics (part 2)

we simply save η'' as the new buffer in the resulting `wait` term, and return the empty prefix in P-Wait-1.

Fig. 11. Arguments Semantics

$$\frac{}{\eta \Rightarrow \cdot @ \cdot \downarrow^{n} \cdot \Rightarrow \{\}}\ \textsc{S-Args-Emp} \qquad \frac{\eta \Rightarrow e \downarrow^{n} e' \Rightarrow p}{\eta \Rightarrow e @ (x : s) \downarrow^{n} e' \Rightarrow \{x \mapsto p\}}\ \textsc{S-Args-Sng}$$

$$\frac{\eta \Rightarrow A_1 @ \Gamma \downarrow^{n_1} A_1' \Rightarrow \eta_1 \qquad \eta \Rightarrow A_2 @ \Gamma' \downarrow^{n_2} A_2' \Rightarrow \eta_2}{\eta \Rightarrow (A_1, A_2)\, @ \Gamma, \Gamma' \downarrow^{n_1+n_2} (A_1', A_2') \Rightarrow \eta_1 \cup \eta_2}\ \textsc{S-Args-Comma}$$

$$\frac{\eta \Rightarrow A_1 @ \Gamma \downarrow^{n_1} A_1' \Rightarrow \eta_1 \qquad \neg\, (\eta_1 \,\texttt{maximalOn}\, \Gamma)}{\eta \Rightarrow (A_1; A_2)\, @ \Gamma; \Gamma' \downarrow^{n_1} (A_1'; A_2) \Rightarrow \eta_1 \cup \mathsf{emp}_{\Gamma'}}\ \textsc{S-Args-Semic-1-1}$$

$$\frac{\eta \Rightarrow A_1 @ \Gamma \downarrow^{n_1} A_1' \Rightarrow \eta_1 \qquad \eta_1 \,\texttt{maximalOn}\, \Gamma \qquad \eta \Rightarrow A_2 @ \Gamma' \downarrow^{n_2} A_2' \Rightarrow \eta_2}{\eta \Rightarrow (A_1; A_2)\, @ \Gamma; \Gamma' \downarrow^{n_1+n_2} (\cdot;)\, A_2' \Rightarrow \eta_1 \cup \eta_2}\ \textsc{S-Args-Semic-1-2}$$

$$\frac{\eta \Rightarrow A @ \Gamma' \downarrow^{n} A' \Rightarrow \eta'}{\eta \Rightarrow (\cdot; A)\, @ \Gamma; \Gamma' \downarrow^{n} (\cdot; A') \Rightarrow \mathsf{emp}_{\Gamma} \cup \eta'}\ \textsc{S-Args-Semic-2}$$

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.

Maximal Semantics Theorem. If all input prefixes are maximal and the step terminates, then the output prefixes are maximal. The contrapositive of this fact is crucial: if the output of a step is not maximal, than some stream in the input must still be sending more data.

Lemma C.49 (Maximal Semantics Auxiliary).

(1) *If* $\eta \Rightarrow e \downarrow e' \Rightarrow p$ *and* η `maximalOn` e *we have that* p `maximal`.
(2) *If* $\eta \Rightarrow A @ \Gamma \downarrow e' \Rightarrow \eta'$ *and* η `maximalOn` A *then* η' `maximalOn` Γ

By mutual induction on the derivation of $\eta \Rightarrow e \downarrow e' \Rightarrow p$ and $\eta \Rightarrow A @ \Gamma \downarrow e' \Rightarrow \eta'$.

► **Case 1:** *S-Var.*

Immediate.

► **Case 2:** *S-Eps-R.*

Immediate.

► **Case 3:** *S-One-R.*

Immediate.

► **Case 4:** *S-Par-R.*

► **Given:**
(1) $\eta \Rightarrow e_1 \downarrow e_1' \Rightarrow p_1$
(2) $\eta \Rightarrow e_2 \downarrow e_2' \Rightarrow p_2$
(3) η `maximalOn` (e_1, e_2)

► **Goal:**
`parPair`(p_1, p_2) `maximal`

By (3) and the definition of fv

(4) For $i = 1, 2$, for all $x \in \mathrm{fv}(e_i)$, there is some p `maximal` such that $\eta(x) \mapsto p$.

By IH on (1)

(5) p_1 is maximal

By IH on (2)

(6) p_1 is maximal

The goal follows by (5) and (6).

► **Case 5:** *S-Cat-R-1.*

► **Given:**

(1) $\eta \Rightarrow e_1 \downarrow e_1' \Rightarrow p_1$

(2) $\neg\,(p_1$ `maximal`$)$

(3) η `maximalOn` $(e_1; e_2)$

► **Goal:**

`catFst`(p) `maximal`

By (3) and the definition of fv

(4) For all $x \in \mathrm{fv}(e_1)$, there is some p `maximal` such that $\eta(x) \mapsto p$.

By IH on (1)

(5) p_1 is maximal

(2) and (5) are a contradiction

► **Case 6:** *S-Cat-R-2.*

► **Given:**

(1) $\eta \Rightarrow e_1 \downarrow e_1' \Rightarrow p_1$

(2) p_1 `maximal`

(3) $\eta \Rightarrow e_2 \downarrow e_2' \Rightarrow p_2$

(5) η `maximalOn` e_i for $i = 1, 2$.

► **Goal:**

`catBoth`(p_1, p_2) `maximal`

By (4) and the definition of fv

(5) For all $i = 1, 2$, for all $x \in \mathrm{fv}(e_1)$, there is some p `maximal` such that $\eta(x) \mapsto p$.

By IH on (3)

(6) p_2 `maximal`

The conclusion follows by (2) and (6)

► **Case 7:** *S-Par-L.*

► **Given:**

(1) $\eta(z) \mapsto$ `parPair`(p_1, p_2)

(2) $\eta[x \mapsto p_1, y \mapsto p_2] \Rightarrow e \downarrow e' \Rightarrow p$

(3) η `maximalOn let` (x, y) $=$ z `in` e

► **Goal:**

p `maximal`

By (1) and (3), using the fact that $\mathrm{fv}($`let` (x, y) $=$ z `in` $e) = \{z\} \cup \mathrm{fv}(e) \setminus \{x, y\}$

(4) `parPair`(p_1, p_2) `maximal`

By inversion on (4)

(5) p_1 `maximal`

(6) p_2 `maximal`

By (3), (5), and (6)

(7) For all $u \in \text{fv}(e)$, there is some p' `maximal` such that $\eta[x \mapsto p_1, y \mapsto p_2](u) \mapsto p'$.

The conclusion follows by IH on (2), using (7)

► **Case 8:** *S-Cat-L-1.*

► **Given:**

(1) $\eta(z) \mapsto \texttt{catFst}(p)$

(2) $\eta[x \mapsto p_1, y \mapsto \texttt{emp}_t] \Rightarrow e \downarrow e' \Rightarrow p'$

(3) η `maximalOn` $\texttt{let}_t\ (x; y)\ =\ z\ \texttt{in}\ e$

► **Goal:**

p' `maximal`

By (3), since $z \in \text{fv}(\texttt{let}_t\ (x; y)\ =\ z\ \texttt{in}\ e)$, we have

(4) $\texttt{catFst}(p)$ `maximal`

But (4) is a contradiction, so the conclusion follows.

► **Case 9:** *S-Cat-L-2.*

► **Given:**

(1) $\eta(z) \mapsto \texttt{catBoth}(p_1, p_2)$

(2) $\eta[x \mapsto p_1, y \mapsto p_2] \Rightarrow e \downarrow e' \Rightarrow p'$

(3) η `maximalOn` $\texttt{let}_t\ (x; y)\ =\ z\ \texttt{in}\ e$

► **Goal:**

p' `maximal`

By (3), since $z \in \text{fv}(\texttt{let}_t\ (x; y)\ =\ z\ \texttt{in}\ e)$, we have

(4) $\texttt{catBoth}(p_1, p_2)$ `maximal`

Inverting (4)

(5) p_1 `maximal`

(6) p_2 `maximal`

By (2), (5), and (6)

(7) For all $u \in \text{fv}(e)$, there is some p' `maximal` such that $\eta[x \mapsto p_1, y \mapsto p_2](u) \mapsto p'$.

The goal follows by IH on (2), using (7)

► **Case 10:** *S-Plus-R-1.*

Immediate by IH, using the fact that $\text{fv}(\texttt{inr}(e)) = \text{fv}(e)$

► **Case 11:** *S-Plus-R-2.*

Identical to previous

► **Case 12:** *S-Plus-L-1.*

► **Given:**

(1) $\eta' \cdot \eta \sim \eta''$

(2) $\eta''(z) = \texttt{sumEmp}$

(3) η `maximalOn` $\texttt{case}_z\ (\eta'; r, x.e_1, y.e_2)$

► **Goal:**

$\texttt{emp}_r$ `maximal`

By (3), there exists some p such that:

(4) p `maximal`
(5) $\eta(z) = p$.
By Theorem C.33 on (1), (4) and (5)
(6) $\eta''(z)$ `maximal`
But (2) and (6) are contradictory, since `sumEmp` is not maximal.

► **Case 13:** *S-Plus-L-2.*

► **Given:**
(1) $\eta' \cdot \eta \sim \eta''$
(2) $\eta''(z) = \texttt{sumInl}(p)$
(3) $\eta''[x \mapsto p] \Rightarrow e_1 \downarrow e_1' \Rightarrow p'$
(4) η `maximalOn` $\texttt{case}_z\ (\eta'; r, x.e_1, y.e_2)$
► **Goal:**
p' `maximal`
By Theorem C.33 on (1) and (4), we have
(5) For all $x \in \text{fv}(\texttt{case}_z\ (\eta'; r, x.e_1, y.e_2))$, there is some p `maximal` such that $\eta''(x) \mapsto p$.
By (5) and (2)
(6) $\texttt{sumInl}(p)$ `maximal`
Inverting (6)
(7) p `maximal`
By (5) and (7)
(8) For all $u \in \text{fv}(e_1)$, there is some p' `maximal` such that $\eta''[x \mapsto p](u) \mapsto p'$.
The goal follows immediately by IH.

► **Case 14:** *S-Plus-L-3.*

Identical to previous

► **Case 15:** *S-Star-R-1.*

Immediate.

► **Case 16:** *S-Star-R-2-1.*

Identical to S-Cat-R-1

► **Case 17:** *S-Star-R-2-2.*

Identical to S-Cat-R-2

► **Case 18:** *S-Star-L-1.*

Identical to S-Plus-L-1

► **Case 19:** *S-Star-L-2.*

► **Given:**
(1) $\eta' \cdot \eta \sim \eta''$
(2) $\eta''(z) = \texttt{starDone}$
(3) $\eta'' \Rightarrow e_1 \downarrow^n e_1' \Rightarrow p$
(4) η `maximalOn` $\texttt{case}_{s,r}\ (\eta'; z, e_1, x.xs.e_2)$
► **Goal:**

p maximal

By Theorem C.33 on (1) and (4), and specializing to $\mathrm{fv}(e_1)$ have

(5) For all $x \in \mathrm{fv}(e_1)$, there is some p maximal so that $\eta''(x) = p$.

The goal follows immediately by (5) and IH on (3).

► **Case 20:** *S-Star-L-3.*

► **Given:**

(1) $\eta' \cdot \eta \sim \eta''$
(2) $\eta''(z) = \mathtt{starFirst}(p)$
(3) $\eta'' \Rightarrow e_1 \downarrow^n e_1' \Rightarrow p$
(4) η maximalOn $\mathsf{case}_{s,r}\ (\eta'; z, e_1, x.xs.e_2)$

► **Goal:**

p maximal

By Theorem C.33 on (1) and (4), we have

(5) η'' maximalOn $\mathsf{case}_{s,r}\ (\eta'; z, e_1, x.xs.e_2)$

In particular, $\eta''(z)$ maximal, but this is a contradiction with (2).

► **Case 21:** *S-Star-L-3.*

Identical to S-Plus-L-2

► **Case 22:** *S-Let.*

► **Given:**

(1) $\eta \Rightarrow e_1 \downarrow^{n_1} e_1' \Rightarrow p$
(2) $\eta[x \mapsto p] \Rightarrow e_2 \downarrow^{n_2} e_2' \Rightarrow p'$
(4) η maximalOn $\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$

► **Goal:**

p' maximal

By IH on (1), using the fact that $\mathrm{fv}(e_1) \subseteq \mathrm{fv}(\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2)$, we have

(4) p maximal

By (3), using the fact that $\mathrm{fv}(e_2) \setminus \{x\} \subseteq \mathrm{fv}(\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2)$

(5) For all $y \in \mathrm{fv}(e_2)$, there exists p' maximal so that $\eta[x \mapsto p](y) \mapsto p'$

The goal follows immediately y IH on (2) and (5)

► **Case 23:** *S-HistPgm.*

Immediate by Definition C.36

► **Case 24:** *S-Wait-1.*

Identical to Plus-L-1.

► **Case 24:** *S-Wait-2.*

► **Given:**

(1) $\eta' \cdot \eta \sim \eta''$
(2) $\eta''(x) = p$
(3) p maximal
(4) $\eta'' \Rightarrow e[\langle p \rangle / x] \downarrow^n e' \Rightarrow p'$
(4) η maximalOn $\mathtt{wait}_{\eta,t}(x)\ (e)$

► **Goal:**

$\boxed{\text{p' } \texttt{maximal}}$

By Theorem C.33 on (5) and (1):

(6) For all $y \in \mathrm{fv}(\texttt{wait}_{\eta,t}(x)\ (e))$, there is some p `maximal` such that $\eta''(y) \mapsto p$.

But since $\mathrm{fv}(e[\langle p \rangle / x]) \subset \mathrm{fv}(\texttt{wait}_{\eta,t}(x)\ (e))$, we have:

(7) For all $y \in \mathrm{fv}(e[\langle p \rangle / x])$, there is some p `maximal` such that $\eta''(y) \mapsto p$.

The goal follows immediately by IH on (4) with (7)

► **Case 25:** *S-Fix.*

► **Given:**

(1) $\overline{M} \downarrow \theta$

(2) $\eta \Rightarrow \texttt{let}\ \Gamma = A\ \texttt{in}\ e\ [e/\texttt{rec}]\ [\theta] \downarrow^n e' \Rightarrow p$

(3) $\eta\ \texttt{maximalOn}\ \texttt{fix}\ \left\{\overline{M}\right\}\ (A)\ .\ (e)$

► **Goal:**

$\boxed{p\ \texttt{maximal}}$

By noting that $\mathrm{fv}(\texttt{fix}\ \left\{\overline{M}\right\}\ (A)\ .\ (e)) = \mathrm{fv}(A)$ and $\mathrm{fv}(\texttt{let}\ \Gamma = A\ \texttt{in}\ e\ [e/\texttt{rec}]\ [\theta]) = \mathrm{fv}(A)$:

4 $\eta\ \texttt{maximalOn}\ \texttt{let}\ \Gamma = A\ \texttt{in}\ e\ [e/\texttt{rec}]\ [\theta]$

Goal follows immediately by IH on (1) and (3)

► **Case 26:** *S-ArgsLet.*

Immediate by two uses of IH.

► **Case 27:** *S-Args-Emp.*

Immediate.

► **Case 28:** *S-Args-Sng.*

Immediate by IH

► **Case 29:** *S-Args-Comma.*

Same as S-Par-R

► **Case 30:** *S-Args-Semic-1-1.*

► **Given:**

(1) $\eta \Rightarrow A_1 @ \Gamma \downarrow^{n_1} A_1' \Rightarrow \eta_1$

(2) $\neg\ (\eta_1\ \texttt{maximalOn}\ \Gamma)$

(3) $\eta\ \texttt{maximalOn}\ (A_1; A_2)$

► **Goal:**

$\boxed{\eta_1 \cup \texttt{emp}_{\Gamma'}\ \texttt{maximalOn}\ \Gamma, \Gamma'}$

By (3)

(4) $\eta\ \texttt{maximalOn}\ A_1$

By IH on (1) and (4)

(5) $\eta_1\ \texttt{maximalOn}\ \Gamma$

(2) and (5) are a contradiction.

► **Case 31:** *S-Args-Semic-1-2.*

► **Given:**
(1) $\eta \Rightarrow A_1@\Gamma \downarrow^{n_1} A_1' \Rightarrow \eta_1$
(2) η_1 `maximalOn` Γ
(3) $\eta \Rightarrow A_2@\Gamma' \downarrow^{n_2} A_2' \Rightarrow \eta_2$
(4) η `maximalOn` $(A_1; A_2)$
► **Goal:**
$\eta_1 \cup \eta_2$ `maximalOn` Γ, Γ'
By IH on (3) and (4)
(5) η_2 `maximalOn` Γ'
The goal follows by (2) and (5)

► **Case 32:** *S-Args-Semic-2-1.*

Immediate by IH.

□

Theorem C.50 (Maximal Semantics). *Suppose:*
(1) $\eta \Rightarrow e \downarrow^n e' \Rightarrow p$
(2) $\cdot \mid \Gamma \vdash_\emptyset e : s @ i$
(3) η `maximalOn` Γ
Then, p `maximal`.

Proof. Because $\cdot \mid \Gamma \vdash_\emptyset e : s @ \mathrm{i}$, we have that $\mathrm{fv}(e) \subseteq \mathrm{Dom}(\Gamma)$. Thus, η `maximalOn` e, and so the goal follows by Lemma C.49 □

Theorem C.51 (Maximal Semantics Extension).
(1) *If* $\eta \Rightarrow e \downarrow^n e' \Rightarrow p$ *and* p `maximal` *and* $\eta \cdot \eta' \sim \eta''$, *and* $\eta'' \Rightarrow e \downarrow^{n'} e'' \Rightarrow p'$, *then* $p = p'$.
(2) *If* $\eta_1 \Rightarrow A@\Gamma \downarrow^n A' \Rightarrow \eta_2$ *and* η_2 `maximalOn` Γ *and* $\eta_1 \cdot \eta_1' \sim \eta_1''$, *and* $\eta_1'' \Rightarrow A@\Gamma \downarrow^{n'} A'' \Rightarrow \eta_2'$, *then* $\eta_2|_\Gamma = \eta_2|_\Gamma$.

Proof. Mutual induction on $\eta \Rightarrow e \downarrow^n e' \Rightarrow p$ and $\eta_1 \Rightarrow A@\Gamma \downarrow^n A' \Rightarrow \eta_2$, using Theorem C.27. □

Semantics Theorems.

Theorem C.52 (Semantics Inputs Determine Outputs).
(1) *If* $\eta \Rightarrow e \downarrow^n e' \Rightarrow p'$ *and* $\eta \Rightarrow e \downarrow^{n'} e'' \Rightarrow p''$, *then* $e' = e''$, *and* $p' = p''$.
(2) *If* $\eta \Rightarrow A@\Gamma \downarrow^n A' \Rightarrow \eta'$ *and* $\eta \Rightarrow A@\Gamma \downarrow^{n'} A'' \Rightarrow \eta''$ *then* $A' = A''$ *and* $\eta' = \eta''$.

Proof. By inspection. □

Theorem C.53 (Semantics Monotonicity).
(1) *If* $\eta \Rightarrow e \downarrow^n e' \Rightarrow p'$ *and* $n' \geq n$, *then* $\eta \Rightarrow e \downarrow^{n'} e' \Rightarrow p'$.
(2) *If* $\eta \Rightarrow A@\Gamma \downarrow^n A' \Rightarrow \eta'$ *and* $n' \geq n$, *then* $\eta \Rightarrow A@\Gamma \downarrow^{n'} A' \Rightarrow \eta'$.

Proof. Mutual induction. □

Theorem C.54 (Soundness).
(1) *Suppose*
(a) $\cdot \mid \Gamma \vdash_\emptyset e : s @ i$
(b) $\eta \Rightarrow e \downarrow^n e' \Rightarrow p$

(c) η : env (Γ)
Then,
(a) p : prefix (s)
(b) *If* $\delta_\eta\,(\Gamma) \sim \Gamma'$ *and* $\delta_p\,(s) \sim s'$, *then* $\cdot \mid \Gamma' \vdash_\emptyset e' : s' \,@\, \mathtt{I}$
(c) *If* $i = \mathtt{I}$ *and* η emptyOn e *then* p empty
(2) *Suppose*
(a) $\cdot \mid \Gamma_1 \vdash_\emptyset A : \Gamma_1 \,@\, i$
(b) $\eta \Rightarrow A@\Gamma_2 \downarrow^n A' \Rightarrow \eta'$
(c) η : env (Γ_1)
Then,
(a) η' : prefix (Γ_2)
(b) *If* $\delta_\eta\,(\Gamma_1) \sim \Gamma_1'$ *and* $\delta_{\eta'}\,(\Gamma_2) \sim \Gamma_2'$, *then* $\cdot \mid \Gamma_1' \vdash_\emptyset A' : \Gamma_2' \,@\, \mathtt{I}$
(c) *If* $i = \mathtt{I}$ *and* η emptyOn A *then* η' emptyOn Γ_2

By mutual induction on the semantics. In the cases for the term (non-argument) semantics, we also do an inner induction on the typing derivation $\cdot \mid \Gamma \vdash_\emptyset e : s \,@\, i$. All of these inner inductions have two cases: one for the corresponding syntax-directed rule, and one for T-Sub. We handle all of the cases with T-Sub simultaneously, in the first case of this proof.

► **Case 1:** *T-Sub.*

► **Given:**
(1) $\cdot \mid \Delta \vdash_\emptyset e : s \,@\, i$
(2) $\Gamma <: \Delta$
(3) $\eta \Rightarrow e \downarrow e' \Rightarrow p$
(4) η : env (Γ)

► **Goal A:**
p : prefix (s)

► **Goal B:**
If $\delta_\eta\,(\Gamma) \sim \Gamma'$ and $\delta_p\,(s) \sim s'$, then $\cdot \mid \Gamma' \vdash_\emptyset e' : s' \,@\, \mathtt{I}$

► **Goal C:**
If $i = \mathtt{I}$ and η emptyOn e then p empty

By Theorem C.39 on (2) and (4)
(5) η : env (Δ)
By IH on (1), using (3) and (5)
(6) p : prefix (s)
(7) If $\delta_\eta\,(\Delta) \sim \Delta'$ and $\delta_p\,(s) \sim s'$, then $\cdot \mid \Delta' \vdash_\emptyset e' : s' \,@\, \mathtt{I}$
(8) If $i = I$ and η emptyOn e then p empty
Goal A is complete by (6), and **Goal C** by (8). To prove **Goal B**, we suppose there are Γ' and s' so that:
(9) $\delta_\eta\,(\Gamma) \sim \Gamma'$
(10) $\delta_p\,(s) \sim s'$
By Theorem C.14 on (5), there is some Δ' so that:
(11) $\delta_\eta\,(\Delta) \sim \Delta'$
By (7), (10), and (11) we have:
(12) $\cdot \mid \Delta' \vdash_\emptyset e' : s' \,@\, \mathtt{I}$
By Theorem C.40,

(13) $\Gamma' <: \Delta'$

Goal B follows by T-Sub on (12) and (13)

► **Case 2:** *S-Eps-R.*

Immediate.

► **Case 3:** *S-One-R.*

Immediate.

► **Case 4:** *S-Var.*

► **Given:**

(1) $\cdot \mid \Gamma(x : s) \vdash_\emptyset x : s \mathbin{@} i$

(2) $\eta : \mathsf{env}(\Gamma(x : s))$

(3) $\eta(x) \mapsto p$

► **Goal A:**

$p : \mathtt{prefix}(s)$

► **Goal B:**

If $\delta_\eta(\Gamma(x : s)) \sim \Gamma'$ and $\delta_p(s) \sim s'$, then $\cdot \mid \Gamma' \vdash_\emptyset x : s' \mathbin{@} \mathtt{I}$

► **Goal C:**

If $i = \mathtt{I}$ and $\eta\ \mathtt{emptyOn}\ x$ then $p\ \mathtt{empty}$

Goal A follows immediately by Theorem C.22 on (2) and (3). **Goal C** is immediate. For **Goal B**, we assume:

(4) $\delta_\eta(\Gamma(x : s)) \sim \Gamma'$

(5) $\delta_p(s) \sim s'$

By Theorem C.23 on (3), (4), and (5), there is some $\Gamma''(-)$ so that:

(6) $\Gamma' = \Gamma''(x : s')$

Then **Goal B** follows by (6) and T-Var.

► **Case 4:** *S-Par-R.*

► **Given:**

(1) $\cdot \mid \Gamma \vdash_\emptyset e_1 : s \mathbin{@} i$

(2) $\cdot \mid \Gamma \vdash_\emptyset e_2 : t \mathbin{@} i$

(3) $\eta : \mathsf{env}(\Gamma)$

(4) $\eta \Rightarrow e_1 \downarrow e_1' \Rightarrow p_1$

(5) $\eta \Rightarrow e_2 \downarrow e_2' \Rightarrow p_2$

► **Goal A:**

$\mathtt{parPair}(p_1, p_2) : \mathtt{prefix}(s \| t)$

► **Goal B:**

If $\delta_\eta(\Gamma) \sim \Gamma'$ and $\delta_{\mathtt{parPair}(p_1,p_2)}(s \| t) \sim s_0$, then $\cdot \mid \Gamma_0 \vdash_\emptyset (e_1', e_2') : s_0 \mathbin{@} \mathtt{I}$

► **Goal C:**

If $i = \mathtt{I}$ and $\eta\ \mathtt{emptyOn}\ (e_1, e_2)$ then $\mathtt{parPair}(p_1, p_2)\ \mathtt{empty}$

By IH on (1), (3), and (4):

(6) $p_1 : \mathtt{prefix}(s)$

(7) If $\delta_\eta(\Gamma) \sim \Gamma'$ and $\delta_{p_1}(s) \sim s'$, then $\cdot \mid \Gamma' \vdash_\emptyset e_1' : s' \mathbin{@} \mathtt{I}$

(8) If $i = \mathtt{I}$ and $\eta\ \mathtt{emptyOn}\ e_1$ then $p_1\ \mathtt{empty}$

By IH on (2), (4), and (5):

(9) $p_2 : \texttt{prefix}\,(t)$
(10) If $\delta_\eta\,(\Gamma) \sim \Gamma'$ and $\delta_{p_2}\,(t) \sim t'$, then $\cdot \mid \Gamma' \vdash_\emptyset e_2' : t'$ @
(11) If $i = \mathtt{I}$ and $\eta\ \texttt{emptyOn}\ e_1$ then $p_1\ \texttt{empty}$

Goal A follows by (6) and (9). **Goal C** follows by (8) and (11). For **Goal B**, we assume:
(12) $\delta_\eta\,(\Gamma) \sim \Gamma'$
(13) $\delta_{\texttt{parPair}(p_1,p_2)}\,(s \| t) \sim s_0$

Inverting (13), we have s', and t' so that $s_0 = s' \| t'$, and:
(14) $\delta_{p_1}\,(s) \sim s'$
(15) $\delta_{p_2}\,(t) \sim t'$

By applying (7) to (12) and (14), we have:
(16) $\cdot \mid \Gamma' \vdash_\emptyset e_1' : s'$ @ $\mathtt{I}$

By applying (10) to (12) and (15), we have:
(17) $\cdot \mid \Gamma' \vdash_\emptyset e_2' : t'$ @ $\mathtt{I}$

Goal B follows by T-Par-R on (16) and (18)

► **Case 5:** *S-Cat-R-1.*

► **Given:**
(1) $\eta \Rightarrow e_1 \downarrow e_1' \Rightarrow p$
(2) $\neg\,(p\ \texttt{maximal})$
(3) $\eta : \texttt{env}\,(\Gamma)$
(4) $\eta : \texttt{env}\,(\Delta)$
(5) $(\exists x \in \mathrm{Dom}\,(\Delta)\,.\neg\eta(x)\ \texttt{empty}) \implies (\forall x \in \mathrm{Dom}\,(\Gamma)\,.\eta(x)\ \texttt{maximal})$
(6) $\cdot \mid \Gamma \vdash_\emptyset e_1 : s$ @ i_1
(7) $\cdot \mid \Delta \vdash_\emptyset e_2 : t$ @ i_2
(8) $i_3 = \mathtt{I} \implies i_1 = \mathtt{I} \wedge \neg\,(s\ \texttt{null})$

► **Goal A:**
$\texttt{catFst}(p) : \texttt{prefix}\,(s \cdot t)$

► **Goal B:**
If $\delta_\eta\,(\Gamma\,;\Delta) \sim \Gamma_0$ and $\delta_{\texttt{catFst}(p)}\,(s \cdot t) \sim s_0$, then $\cdot \mid \Gamma_0 \vdash_\emptyset (e_1'; e_2) : s_0$ @ $\mathtt{I}$

► **Goal C:**
If $i_3 = \mathtt{I}$ and $\eta\ \texttt{emptyOn}\ (e_1; e_2)$, then $\texttt{catFst}(p)\ \texttt{empty}$

By IH on (1), (3), and (6)
(9) $p : \texttt{prefix}\,(s)$
(10) If $\delta_\eta\,(\Gamma) \sim \Gamma'$ and $\delta_p\,(s) \sim s'$, then $\cdot \mid \Gamma' \vdash_\emptyset e_1' : s'$ @ $\mathtt{I}$
(11) If $i_1 = \mathtt{I}$ and $\eta\ \texttt{emptyOn}\ e_1$ then $p\ \texttt{empty}$

Goal A follows by (9). For **Goal B**, we assume:
(12) $\delta_\eta\,(\Gamma\,;\Delta) \sim \Gamma_0$
(13) $\delta_{\texttt{catFst}(p)}\,(s \cdot t) \sim s_0$

Inverting (12) and (13), we have Γ', Δ', and s' so that $\Gamma_0 = \Gamma'\,;\Delta'$ and $s_0 = s' \cdot t$, and:
(14) $\delta_\eta\,(\Gamma) \sim \Gamma'$
(15) $\delta_\eta\,(\Delta) \sim \Delta'$
(16) $\delta_p\,(s) \sim s'$

Applying (11) to (14) and (16)
(17) $\cdot \mid \Gamma' \vdash_\emptyset e_1 : s'$ @ $\mathtt{I}$

By Theorem C.50 on (1) and (6), using (2)
(18) $\neg\,(\eta\ \texttt{maximalOn}\ \Gamma)$

Therefore, by (18) and (5)
(19) $\eta\ \texttt{emptyOn}\ \Delta$
Then by Theorem **??** and Theorem C.14 on (17) and (13)
(20) $\Delta = \Delta'$
Then, **Goal B** follows by T-Cat-R on (15) and (7)
For **Goal C**, assume:
(21) $i_3 = \mathtt{I}$
(22) $\eta\ \texttt{emptyOn}\ (e_1; e_2)$
By (8) with (21)
(23) $i_1 = \mathtt{I}$
Goal C follows by (11), with (22) and (23).

► **Case 6:** *S-Cat-R-2.*

► **Given:**
(1) $\eta \Rightarrow e_1 \downarrow e_1' \Rightarrow p_1$
(2) $p_1\ \texttt{maximal}$
(3) $\eta \Rightarrow e_2 \downarrow e_2' \Rightarrow p_2$
(4) $\eta\ :\ \texttt{env}\,(\Gamma)$
(5) $\eta\ :\ \texttt{env}\,(\Delta)$
(6) $(\exists x \in \mathrm{Dom}\,(\Delta)\,.\neg\eta(x)\ \texttt{empty}) \implies (\forall x \in \mathrm{Dom}\,(\Gamma)\,.\eta(x)\ \texttt{maximal})$
(7) $\cdot \mid \Gamma \vdash_\emptyset e_1 : s\ @\ i_1$
(8) $\cdot \mid \Delta \vdash_\emptyset e_2 : t\ @\ i_2$
(9) $i_3 = \mathtt{I} \implies i_1 = \mathtt{I} \wedge \neg\,(s\ \texttt{null})$

► **Goal A:**
$\boxed{\texttt{catBoth}(p_1, p_2)\ :\ \texttt{prefix}\,(s \cdot t)}$

► **Goal B:**
$\boxed{\text{If } \delta_\eta\,(\Gamma\,;\Delta) \sim \Gamma_0 \text{ and } \delta_{\texttt{catBoth}(p_1,p_2)}\,(s \cdot t) \sim s_0 \text{ then } \cdot \mid \Gamma_0 \vdash_\emptyset e_2' : s_0\ @\ \mathtt{I}}$

► **Goal C:**
$\boxed{\text{If } i_3 = \mathtt{I} \text{ and } \eta\ \texttt{emptyOn}\ (e_1; e_2) \text{ then } \texttt{catBoth}(p_1, p_2)\ \texttt{empty}}$

By IH on (1), (4), and (7)
(10) $p_1\ :\ \texttt{prefix}\,(s)$
(11) If $\delta_\eta\,(\Gamma) \sim \Gamma'$ and $\delta_{p_1}\,(s) \sim s'$, then $\cdot \mid \Gamma' \vdash_\emptyset e_1 : s'\ @\ \mathtt{I}$
(12) if $i_1 = \mathtt{I}$ and $\eta\ \texttt{emptyOn}\ e_1$ then $p_1\ \texttt{empty}$
By IH on (3), (5), and (8)
(12) $p_2\ :\ \texttt{prefix}\,(t)$
(13) If $\delta_\eta\,(\Delta) \sim \Delta'$ and $\delta_{p_2}\,(t) \sim t'$, then $\cdot \mid \Delta' \vdash_\emptyset e_2 : t'\ @\ \mathtt{I}$
Goal A follows by (2), (9), and (11). For **Goal B**, we assume
(14) $\delta_\eta\,(\Gamma\,;\Delta) \sim \Gamma_0$
(15) $\delta_{\texttt{catBoth}(p_1,p_2)}\,(s \cdot t) \sim s_0$
By inversion on (14) and (15), there are Γ', Δ', and t' so that $\Gamma_0 = \Gamma'\,;\Delta'$, and $s_0 = t'$, and:
(16) $\delta_\eta\,(\Gamma) \sim \Gamma'$
(17) $\delta_\eta\,(\Delta) \sim \Delta'$
(18) $\delta_{p_2}\,(t) \sim t'$
By (13) on (17) and (18)
(18) $\cdot \mid \Delta' \vdash_\emptyset e_2' : t'\ @\ \mathtt{I}$
Goal B follows by T-Sub with $\Gamma'\,;\Delta'\ <:\ \Delta'$ For **Goal B**, assume:

(19) $i_3 = \mathtt{I}$
(20) $\eta\ \mathtt{emptyOn}\ (e_1; e_2)$
By (9) with (19) and (20)
(21) $i_1 = \mathtt{I}$
(22) $\neg\,(s\ \mathtt{null})$
By (12), with (21) and (20)
(23) $p_1\ \mathtt{empty}$
By Theorem C.9 with (2), (23), and (10)
(24) $s\ \mathtt{null}$
But (22) and (24) are contradictory.

► **Case 7:** *S-Par-L.*

► **Given:**
(1) $\eta(z) \mapsto \mathtt{parPair}(p_1, p_2)$
(2) $\eta[x \mapsto p_1, y \mapsto p_2] \Rightarrow e \downarrow e' \Rightarrow p$
(3) $\eta\ :\ \mathtt{env}\,(\Gamma(z : s \| t))$
(4) $\cdot \mid \Gamma(x : s, y : t) \vdash_\emptyset e : r\ @\ i$

► **Goal A:**
$p'\ :\ \mathtt{prefix}\,(r)$

► **Goal B:**
If $\delta_\eta\,(\Gamma(z : s \| t)) \sim \Gamma_0$ and $\delta_p\,(r) \sim r'$, then $\cdot \mid \Gamma_0 \vdash_\emptyset \mathtt{let}\ (x, y)\ =\ z\ \mathtt{in}\ e' : r'\ @\ \mathtt{I}$

► **Goal C:**
If $i = \mathtt{I}$ and $\eta\ \mathtt{emptyOn}\ \mathtt{let}\ (x, y)\ =\ z\ \mathtt{in}\ e$ then $p\ \mathtt{empty}$

By Theorem C.22 on (1) and (3):
(5) $\mathtt{parPair}(p_1, p_2)\ :\ \mathtt{prefix}\,(s \| t)$
Inverting (5)
(6) $p_1\ :\ \mathtt{prefix}\,(s)$
(7) $p_2\ :\ \mathtt{prefix}\,(t)$
By Theorem **??** on (3), (6) and (7)
(8) $\eta[x \mapsto p_1, y \mapsto p_2]\ :\ \mathtt{env}\,(\Gamma(x : s, y : t))$
By IH on (2), (4), and (8)
(9) $p\ :\ \mathtt{prefix}\,(r)$
(10) For all Γ_0' and r', if $\delta_{\eta[x \mapsto p_1, y \mapsto p_2]}\,(\Gamma(x : s, y : t)) \sim \Gamma_0'$ and $\delta_p\,(r) \sim r'$ then $\cdot \mid \Gamma_0' \vdash_\emptyset e' : r'\ @\ \mathtt{I}$
(11) If $i = \mathtt{I}$ and $\eta[x \mapsto p_1, y \mapsto p_2]\ \mathtt{emptyOn}\ e$ then $p\ \mathtt{empty}$
Goal A is complete by (9). For **Goal B**, we assume that there are Γ_0 and r' so that:
(12) $\delta_\eta\,(\Gamma(z : s \| t)) \sim \Gamma_0$
(13) $\delta_p\,(r) \sim r'$
By two uses of Theorem C.12, we have s' and t' so that
(14) $\delta_{\mathtt{parPair}(p_1, p_2)}\,(s \| t) \sim s' \| t'$
By Theorem C.23 on (1), (3), and (14), we have $\Gamma'(-)$ so that:
(15) $\Gamma_0 = \Gamma'(z : s' \| t')$
By Theorem **??** on (1), (12), and (15)
(16) $\delta_{\eta[x \mapsto p_1, y \mapsto p_2]}\,(\Gamma(x : s, y : t)) \sim \Gamma'(x : s', y : t')$
By (10) on (13) and (16), we have:
(17) $\cdot \mid \Gamma'(x : s', y : t') \vdash_\emptyset e' : r'\ @\ \mathtt{I}$

Goal B follows by T-Par-L on (17)
For **Goal C**, assume:
(18) $i = \mathtt{I}$
(19) $\eta\ \mathtt{emptyOn}\ \mathtt{let}\ (x, y)\ =\ z\ \mathtt{in}\ e$
In particular with (19), since $z \in \mathrm{fv}(\mathtt{let}\ (x, y)\ =\ z\ \mathtt{in}\ e)$, we have $\eta(z)\ \mathtt{empty}$ and so:
(20) $p_1\ \mathtt{empty}$
(21) $p_2\ \mathtt{empty}$
By (19), (20), and (21)
(22) $\eta[x \mapsto p_1, y \mapsto p_2]\ \mathtt{emptyOn}\ e$
Goal C follows by (11), with (18) and (22)

► **Case 8:** *S-Cat-L-1.*

► **Given:**
(1) $\cdot \mid \Gamma(x : s ; y : t) \vdash_\emptyset e : r\ @\ i$
(2) $\eta\ :\ \mathsf{env}\,(\Gamma(z : s \cdot t))$
(3) $\eta(z) \mapsto \mathtt{catFst}(p)$
(4) $\eta[x \mapsto p, y \mapsto \mathsf{emp}_t] \Rightarrow e \downarrow e' \Rightarrow p'$
► **Goal A:**
$p'\ :\ \mathtt{prefix}\,(r)$
► **Goal B:**
$\forall \Gamma_0, r'$ if $\delta_\eta\ (\Gamma(z : s \cdot t)) \sim \Gamma_0$ and $\delta_{p'}\ (r) \sim r'$, then $\cdot \mid \Gamma_0 \vdash_\emptyset \mathtt{let}_t\ (x; y)\ =\ z\ \mathtt{in}\ e' : r'\ @\ \mathtt{I}$
► **Goal C:**
If $i = \mathtt{I}$ and $\eta\ \mathtt{emptyOn}\ \mathtt{let}_t\ (x; y)\ =\ z\ \mathtt{in}\ e$ then $p'\ \mathtt{empty}$
By Theorem C.22 on (2) and (3):
(5) $\mathtt{catFst}(p)\ :\ \mathtt{prefix}\,(s \cdot t)$
Inverting (5)
(6) $p\ :\ \mathtt{prefix}\,(s)$
By Theorem **??** on (2) and (6)
(7) $\eta[x \mapsto p, y \mapsto \mathsf{emp}_t]\ :\ \mathsf{env}\,(\Gamma(x : s ; y : t))$
By IH on (2), (4), and (7)
(8) $p'\ :\ \mathtt{prefix}\,(r)$
(9) For all Γ_0 and r', if $\delta_{\eta[x \mapsto p, y \mapsto \mathsf{emp}_t]}\ (\Gamma(x : s ; y : t)) \sim \Gamma_0$ and $\delta_{p'}\ (r) \sim r'$, then $\cdot \mid \Gamma_0 \vdash_\emptyset e' : r'\ @\ \mathtt{I}$
(10) If $i = \mathtt{I}$ and $\eta[x \mapsto p, y \mapsto \mathsf{emp}_t]\ \mathtt{emptyOn}\ e$ then $p'\ \mathtt{empty}$
Goal A is completed by (8). For **Goal B**, we assume that there are Γ_0 and r' such that:
(11) $\delta_\eta\ (\Gamma(z : s \cdot t)) \sim \Gamma_0$
(12) $\delta_{p'}\ (r) \sim r'$
By Theorem C.12 on (6), there is some s' so that
(13) $\delta_p\ (s) \sim s'$
And so by definition from (13)
(14) $\delta_{\mathtt{catFst}(p)}\ (s \cdot t) \sim s' \cdot t$
By Theorem C.23 on (3), (11), and (14), there is some $\Gamma'(-)$ so that
(15) $\Gamma_0 = \Gamma'(z : s' \cdot t)$
By Theorem **??** on (3) and (11) using (15)
(16) $\delta_{\eta[x \mapsto p, y \mapsto \mathsf{emp}_t]}\ (\Gamma(x : s ; y : t)) \sim \Gamma'(x : s' ; y : t)$
By (9), with (12) and (16)

(17) $\cdot \mid \Gamma'(x : s' ; y : t) \vdash_\emptyset e' : r' \,@$

Goal B follows by T-Cat-L on (17).

For **Goal C**, assume:

(18) $i = \mathtt{I}$

(19) $\eta\ \mathtt{emptyOn}\ \mathtt{let}_t\ (x; y)\ =\ z\ \mathtt{in}\ e$

In particular with (19), since $z \in \mathrm{fv}(\mathtt{let}_t\ (x; y)\ =\ z\ \mathtt{in}\ e)$, we have $\eta(z)\ \mathtt{empty}$ and so:

(20) $\mathtt{catFst}(p)\ \mathtt{empty}$

Inverting (20)

(21) $p\ \mathtt{empty}$

By (21), (19) and Theorem C.8

(22) $\eta[x \mapsto p, y \mapsto \mathtt{emp}_t]\ \mathtt{emptyOn}\ e$

Goal C follows by (10), with (18) and (22)

► **Case 9:** *S-Cat-L-2.*

► **Given:**

(1) $\cdot \mid \Gamma(x : s ; y : t) \vdash_\emptyset e : r \,@\, i$

(2) $\eta\ :\ \mathsf{env}\,(\Gamma(z : s \cdot t))$

(3) $\eta(z) \mapsto \mathtt{catBoth}(p_1, p_2)$

(4) $\eta[x \mapsto p_1, y \mapsto p_2] \Rightarrow e \downarrow e' \Rightarrow p$

► **Goal A:**

$p\ :\ \mathtt{prefix}\,(r)$

► **Goal B:**

$\forall \Gamma_0, r'$ if $\delta_\eta\,(\Gamma(z : s \cdot t)) \sim \Gamma_0$ and $\delta_p\,(r) \sim r'$ then $\cdot \mid \Gamma_0 \vdash_\emptyset \mathtt{let}\ x = \mathtt{sink}_{p_1}\ \mathtt{in}\ e'[z/y] : r' \,@\, \mathtt{I}$

► **Goal C:**

If $i = \mathtt{I}$ and $\eta\ \mathtt{emptyOn}\ \mathtt{let}_t\ (x; y)\ =\ z\ \mathtt{in}\ e$ then $p\ \mathtt{empty}$

By Theorem C.22 on (2) and (3):

(5) $\mathtt{catBoth}(p_1, p_2)\ :\ \mathtt{prefix}\,(s \cdot t)$

Inverting (5)

(6) $p_1\ :\ \mathtt{prefix}\,(s)$

(7) $p_1\ \mathtt{maximal}$

(8) $p_2\ :\ \mathtt{prefix}\,(t)$

By Theorem **??** on (2), (6), (7), and (8),

(9) $\eta[x \mapsto p_1, y \mapsto p_2]\ :\ \mathsf{env}\,(\Gamma(x : s ; y : t))$

By IH on (1), (9), and (4)

(10) $p\ :\ \mathtt{prefix}\,(r)$

(11) For all Γ_0 and r', if $\delta_{\eta[x \mapsto p_1, y \mapsto p_2]}\,(\Gamma(x : s ; y : t)) \sim \Gamma_0$ and $\delta_p\,(r) \sim r'$, then $\cdot \mid \Gamma_0 \vdash_\emptyset e' : r' \,@\, \mathtt{I}$

(12) If $i = \mathtt{I}$ and $\eta\ \mathtt{emptyOn}\ \mathtt{let}_t\ (x; y)\ =\ z\ \mathtt{in}\ e$ then $p\ \mathtt{empty}$

Goal A is complete by (10). For **Goal B**, we assume that there are Γ_0 and r' such that:

(13) $\delta_\eta\,(\Gamma(z : s \cdot t)) \sim \Gamma_0$

(14) $\delta_p\,(r) \sim r'$

By Theorem C.12 on (8), there is some t' so that

(15) $\delta_{p_2}\,(t) \sim t'$

By definition from (15)

(16) $\delta_{\mathtt{catBoth}(p_1, p_2)}\,(s \cdot t) \sim t'$

By Theorem C.23 on (13), (3), and (16), there is some $\Gamma'(-)$ so that:

(17) $\Gamma_0 = \Gamma'(z : t')$

By Theorem **??** on (13) and (3), there is some s' so that

(18) $\delta_{\eta[x \mapsto p_1, y \mapsto p_2]}\ (\Gamma(x : s\,; y : t)) \sim \Gamma'(x : s'\,; y : t')$

(19) $\delta_{p_1}\ (s) \sim s'$

By (11) on (14) and (18), we have:

(20) $\cdot \mid \Gamma'(x : s'\,; y : t') \vdash_\emptyset e' : r' \mathbin{@} \mathtt{I}$

By substitution on (20), we have:

(21) $\cdot \mid \Gamma'(x : s'\,; z : t') \vdash_\emptyset e'[z/y] : r' \mathbin{@} \mathtt{I}$

By Theorem C.45 on (7), (6), and (19)

(22) $\cdot \mid \cdot \vdash_\emptyset \mathtt{sink}_{p_1} : s' \mathbin{@} \mathtt{I}$

By T-Let on (21), (22).

(23) $\cdot \mid \Gamma'(\cdot\,; z : t') \vdash_\emptyset \mathtt{let}\, x = \mathtt{sink}_{p_1}\, \mathtt{in}\, e'[z/y] : r' \mathbin{@} \mathtt{I}$

Goal B follows by (23) with the T-Sub using $\Gamma'(z : t') <: \Gamma'(\cdot\,; z : t')$.

For **Goal C**, assume:

(24) $i = \mathtt{I}$

(25) $\eta\ \mathtt{emptyOn}\ \mathtt{let}_t\ (x; y) = z\ \mathtt{in}\ e$

In particular with (25), since $z \in \mathrm{fv}(\mathtt{let}_t\ (x; y) = z\ \mathtt{in}\ e)$, we have $\eta(z)$ $\mathtt{empty}$ and so:

(26) $\mathtt{catBoth}(p_1, p_2)\ \mathtt{empty}$

But (26) is impossible

► **Case 10:** *S-Plus-R-1.*

► **Given:**

(1) $\cdot \mid \Gamma \vdash_\emptyset e : s \mathbin{@} i$

(2) η : $\mathsf{env}\,(\Gamma)$

(3) $\eta \Rightarrow e \downarrow e' \Rightarrow p$

► **Goal A:**

$\mathtt{sumInl}(p)$: $\mathtt{prefix}\,(s + t)$

► **Goal B:**

For all Γ_0 and r, if $\delta_\eta\ (\Gamma) \sim \Gamma_0$ and $\delta_{\mathtt{sumInl}(p)}\ (s + t) \sim r$ then $\cdot \mid \Gamma_0 \vdash_\emptyset e' : r \mathbin{@} \mathtt{I}$

► **Goal C:**

If $\mathtt{I} = \mathtt{J}$ and $\eta\ \mathtt{emptyOn}\ \mathtt{inl}\ (e)$ then $\mathtt{sumInl}(p)\ \mathtt{empty}$

By IH on (1), (2), and (3)

(4) p : $\mathtt{prefix}\,(s)$

(5) For all Γ_0 and s', if $\delta_\eta\ (\Gamma) \sim \Gamma_0$ and $\delta_p\ (s) \sim s'$ then $\cdot \mid \Gamma_0 \vdash_\emptyset e' : s' \mathbin{@} \mathtt{I}$

Goal A follows from (4), and **Goal B** follows from (5), inverting the derivation of $\delta_{\mathtt{sumInl}(p)}\ (s + t) \sim r$. The premise of **Goal C** is absurd.

► **Case 11:** *S-Plus-R-2.*

Identical to previous.

► **Case 12:** *S-Plus-L-1.*

► **Given:**

(1) η : $\mathsf{env}\,(\Gamma(z : s + t))$

(2) $\delta_\eta\ (\Gamma(z : s + t)) \sim \Gamma'$

(3) $\cdot \mid \Gamma(x : s) \vdash_\emptyset e_1 : r \mathbin{@} i_1$

(4) $\cdot \mid \Gamma(y : t) \vdash_\emptyset e_2 : r \mathbin{@} i_2$

(5) $\eta' : \mathsf{env}(\Gamma')$
(6) $\eta' \cdot \eta \sim \eta''$
(7) $\eta''(z) = \mathtt{sumEmp}$

► **Goal A:**
$\mathsf{emp}_r : \mathtt{prefix}(r)$

► **Goal B:**
For all Γ'' and r', if $\delta_{\eta'}(\Gamma') \sim \Gamma''$ and $\delta_{\mathsf{emp}_r}(r) \sim r'$ then $\cdot \mid \Gamma'' \vdash_\emptyset \mathsf{case}_r(\eta''; z, x.e_1, y.e_2) : r' @ \mathtt{I}$

► **Goal C:**
If $i = \mathtt{I}$ and $\eta\ \mathtt{emptyOn}\ \mathsf{case}_r(\eta'; z, x.e_1, y.e_2)$ then $\mathsf{emp}_r\ \mathtt{empty}$

Goal A is complete by Theorem C.6, and **Goal C** is immediate by Theorem C.8. For **Goal B**, assume there are Γ'' and r' so that

(8) $\delta_{\eta'}(\Gamma') \sim \Gamma''$
(9) $\delta_{\mathsf{emp}_r}(r) \sim r'$

By Theorem C.13
(10) $r = r'$

By Theorem C.31 on (1), (2), (5), (6), and (8)
(11) $\eta'' : \mathsf{env}(\Gamma(z : s + t))$
(12) $\delta_{\eta''}(\Gamma(z : s + t)) \sim \Gamma''$

Goal B follows immediately by T-Plus-L on (11), (12), (3), (4), and (7).

► **Case 13:** *S-Plus-L-2.*

► **Given:**
(1) $\eta : \mathsf{env}(\Gamma(z : s + t))$
(2) $\delta_\eta(\Gamma(z : s + t)) \sim \Gamma'$
(3) $\cdot \mid \Gamma(x : s) \vdash_\emptyset e_1 : r @ i_1$
(4) $\cdot \mid \Gamma(y : t) \vdash_\emptyset e_2 : r @ i_2$
(5) $i = \mathtt{I} \implies \eta(z) = \mathtt{sumEmp}$
(6) $\eta' : \mathsf{env}(\Gamma')$
(7) $\eta \cdot \eta' \sim \eta''$
(8) $\eta''(z) \mapsto \mathtt{sumInl}(p)$
(9) $\eta''[x \mapsto p] \Rightarrow e_1 \downarrow e_1' \Rightarrow p'$

► **Goal A:**
$p' : \mathtt{prefix}(r)$

► **Goal B:**
If $\delta_{\eta'}(\Gamma') \sim \Gamma''$ and $\delta_{p'}(r) \sim r'$ then $\cdot \mid \Gamma'' \vdash_\emptyset e_1'[x/z] : r' @ \mathtt{I}$

If $i = \mathtt{I}$ and $\eta'\ \mathtt{emptyOn}\ \mathsf{case}_r(\eta; z, x.e_1, y.e_2)$ then $p'\ \mathtt{empty}$

By Theorem C.31 on (1), (2), (5), (6), then:
(10) $\eta'' : \mathtt{prefix}(\Gamma(z : s + t))$
(11) If $\delta_{\eta'}(\Gamma') \sim \Gamma''$, then $\delta_{\eta''}(\Gamma(z : s + t)) \sim \Gamma''$

By Theorem C.22 on (10)
(12) $\mathtt{sumInl}(p) : \mathtt{prefix}(s + t)$

By inversion on (12)
(13) $p : \mathtt{prefix}(s)$

By Theorem C.20 on (10) and (13)
(14) $\eta''[x \mapsto p] : \mathtt{prefix}(\Gamma(x : s))$

By IH on (3), (14), and (9)

(15) $p' : \texttt{prefix}\,(r)$
(16) If $\delta_{\eta''[x \mapsto p]}\,(\Gamma(x : s)) \sim \Gamma_0$ and $\delta_{p'}\,(r) \sim r'$, then $\cdot \mid \Gamma_0 \vdash_\emptyset e_1' : r'$ @

Goal A is complete by (15). For **Goal B**, suppose that there are Γ'' and r' such that:
(17) $\delta_{p'}\,(r) \sim r'$
(18) $\delta_{\eta'}\,(\Gamma') \sim \Gamma''$

By (11) and (18)
(19) $\delta_{\eta''}\,(\Gamma(z : s + t)) \sim \Gamma''$

By Theorem C.12, there is some s' so that
(20) $\delta_p\,(s) \sim s'$

By Theorem C.21, on (19), there is some $\Gamma'''(-)$ so that:
(21) For all Δ' and Δ'' and η', if $\delta_{\eta'}\,(\Delta') \sim \Delta''$ and $\texttt{agree}\,(\eta, \eta', \Delta, \Delta')$ then $\delta_{\eta \cdot \eta'}\,(\Gamma(\Delta')) \sim \Gamma'''(\Delta'')$

By (20) and (21), taking $\eta' = \{z \mapsto \texttt{starFirst}(p)\}$, we have:
(22) $\Gamma'' = \Gamma'''(z : s')$

Also by (21):
(23) $\delta_{\eta''[x \mapsto p]}\,(\Gamma(x : s)) \sim \Gamma'''(x : s')$

By (16) with (18), and (23)
(24) $\cdot \mid \Gamma'''(x : s') \vdash_\emptyset e_1' : r'$ @ $\texttt{I}$

Goal B follows immediately by substituting x for z in (23).

(For **Goal C**, assume:)
(25) $i = \texttt{I}$
(26) $\eta'\ \texttt{emptyOn}\ \texttt{case}_r\,(\eta; z, x.e_1, y.e_2)$

In particular with 26, since $z \in \text{fv}(\texttt{case}_r\,(\eta; z, x.e_1, y.e_2))$
(27) $\eta'(z)\ \texttt{empty}$

By (5) with (25)
(28) $\eta(z) = \texttt{sumEmp}$

By Definition of environment concatenation, using (7), (8), (28)
(29) $\texttt{sumEmp} \cdot \eta'(z) \sim \texttt{sumInl}(p)$

But by inversion, we note that the only case has $\eta'(z) = \texttt{sumInl}(p_0)$ for some p_0, which contradicts (27). This completes **Goal C**.

► **Case 14:** *S-Plus-L-3.*

Identical to previous.

► **Case 15:** *S-Star-R-1.*

Immediate.

► **Case 16:** *S-Star-R-2-1.*

Identical to S-Cat-R-1.

► **Case 17:** *S-Star-R-2-2.*

Identical to S-Cat-R-2.

► **Case 18:** *S-Star-L-1.*

Identical to S-Plus-L-1

► **Case 19:** *S-Star-L-2.*

Identical to S-Plus-L-2

► **Case 20:** *S-Star-L-3.*

Identical to S-Plus-L-2

▶ **Case 21:** *S-Star-L-4.*

Identical to S-Plus-L-2

▶ **Case 22:** *S-Let.*

▶ **Given:**

(1) $\cdot \mid \Delta \vdash_\emptyset e_1 : s \mathbin{@} \mathtt{I}$
(2) $\cdot \mid \Gamma(x : s) \vdash_\emptyset e_2 : t \mathbin{@} i$
(3) $\eta \ : \ \mathsf{env}\,(\Gamma(\Delta))$
(4) $\eta \Rightarrow e_1 \downarrow e_1' \Rightarrow p$
(5) $\eta[x \mapsto p] \Rightarrow e_2 \downarrow e_2' \Rightarrow p'$

▶ **Goal A:**

$p' \ : \ \mathtt{prefix}\,(t)$

▶ **Goal B:**

For all Γ_0 and t', if $\delta_\eta\,(\Gamma(\Delta)) \sim \Gamma_0$ and $\delta_{p'}\,(t) \sim t'$ then $\cdot \mid \Gamma_0 \vdash_\emptyset \mathtt{let}\, x = e_1' \,\mathtt{in}\, e_2' : t' \mathbin{@} \mathtt{I}$

If $i = \mathtt{I}$ and $\eta\ \mathtt{emptyOn}\ \mathtt{let}\, x = e_1 \,\mathtt{in}\, e_2$ then $p'\ \mathtt{empty}$

By Theorem C.19 on (3)
(6) $\eta \ : \ \mathsf{env}\,(\Delta)$

By IH on (1), (6), and (4)
(7) $p \ : \ \mathtt{prefix}\,(s)$
(8) For all Δ' and s', if $\delta_\eta\,(\Delta) \sim \Delta'$ and $\delta_p\,(s) \sim s'$ then $\Delta' \mid e_1' \vdash_{s'} \mathtt{I} : \ @$
(9) If $\mathtt{I} = \mathtt{I}$ and $\eta\ \mathtt{emptyOn}\ e_1$ then $p\ \mathtt{empty}$

By Theorem C.49 and (9) on (5)
(10) $\mathtt{agree}\,(\eta, \{x \mapsto p\}, \Delta, x : s)$

By Theorem C.20 on (4), (8), (10):
(11) $\eta[x \mapsto p] \ : \ \mathsf{env}\,(\Gamma(x : s))$

By IH on (3), (6), and (11)
(12) $p' \ : \ \mathtt{prefix}\,(t)$
(13) For all Γ_0' and t', if $\delta_{\eta[x \mapsto p]}\,(\Gamma(x : s)) \sim \Gamma_0'$ and $\delta_{p'}\,(t) \sim t'$, then $\Gamma_0' \mid e_2' \vdash_{t'} \mathtt{I} : \ @$
(14) If $i = \mathtt{I}$ and $\eta[x \mapsto p]\ \mathtt{emptyOn}\ e_2$ then $p'\ \mathtt{empty}$

Goal A is complete by (12). For **Goal B**, we assume that there are Γ_0 and t' such that:
(15) $\delta_\eta\,(\Gamma(\Delta)) \sim \Gamma_0$
(16) $\delta_{p'}\,(t) \sim t'$

By Theorem C.21 on (15), there is some $\Gamma'(-)$ such that
(17) For all Δ' and Δ'', if $\delta_{\eta'}\,(\Delta') \sim \Delta''$ then $\delta_{\eta \cdot \eta'}\,(\Gamma(\Delta')) \sim \Gamma'(\Delta'')$

By Theorem C.14 on (7), there is some Δ' so that
(18) $\delta_\eta\,(\Delta) \sim \Delta'$

By the uniqueness in Theorem C.14 and (17) and (18)
(19) $\Gamma_0 = \Gamma'(\Delta')$

By Theorem C.12 on (8), there is some s' so that
(20) $\delta_p\,(s) \sim s'$

By (9) on (18) and (20)
(21) $\cdot \mid \Delta' \vdash_\emptyset e_1' : s' \mathbin{@} \mathtt{I}$

By (17) on (20)
(22) $\delta_{\eta[x \mapsto p]}\,(\Gamma(x : s)) \sim \Gamma'(x : s')$

By (13) on (22) and (16)
(23) $\cdot \mid \Gamma'(x : s') \vdash_\emptyset e_2' : t' \mathbin{@} \mathtt{I}$
Goal B follows by T-Let on (21) and (23)
For **Goal C**, assume:
(24) $i = \mathtt{I}$
(25) $\eta\ \mathtt{emptyOn}\ \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$
By (25), applying (9)
(26) $p\ \mathtt{empty}$
By (25) and (26)
(27) $\eta[x \mapsto p]\ \mathtt{emptyOn}\ e_2$
Goal C follows by (14) applied to (24) and (27)

► **Case 23:** *S-HistPgm.*

► **Given:**
(1) $\Omega \vdash M : \langle s \rangle$
(2) $M \downarrow v$
(3) $p = \mathtt{toPrefix}_s\,(v)$
► **Goal A:**
$p : \mathtt{prefix}\,(s)$
► **Goal B:**
If $\delta_\eta\,(\Gamma) \sim \Gamma'$ and $\delta_p\,(s) \sim s'$, then $\cdot \mid \Gamma' \vdash_\emptyset \mathtt{sink}_p : s' \mathbin{@} \mathtt{I}$
► **Goal C:**
If $\mathtt{J} = \mathtt{I}$ and $\eta\ \mathtt{emptyOn}\ \emptyset$ then $p\ \mathtt{empty}$
By Definition C.37:
(4) $v : \langle s \rangle$
By Definition C.36 on (4):
(5) $p : \mathtt{prefix}\,(s)$
(6) $p\ \mathtt{maximal}$
Goal A follows by (5). For **Goal B**, assume:
(7) $\delta_\eta\,(\Gamma) \sim \Gamma'$
(8) $\delta_p\,(s) \sim s'$
Then **Goal B** follows by Theorem C.45 on (5), (6), and (8). The premise of **Goal C** is absurd.

► **Case 24:** *S-Wait-1.*

Identical to S-Plus-L-1

► **Case 25:** *S-Wait-2.*

► **Given:**
(1) $\eta' : \mathtt{env}\,(\Gamma(x : s))$
(2) $\delta_{\eta'}\,(\Gamma(x : s)) \sim \Gamma'$
(3) $x : \langle s \rangle \mid \Gamma(\cdot) \vdash_\emptyset e : t \mathbin{@} i'$
(4) $i = \mathtt{I} \implies \neg\,(\eta(x)\ \mathtt{maximal}) \wedge \neg\,(s\ \mathtt{null})$
(5) $\eta : \mathtt{env}\,(\Gamma')$
(6) $\eta' \cdot \eta \sim \eta''$
(7) $\eta''(x) = p$
(8) $p\ \mathtt{maximal}$

(9) $\eta'' \Rightarrow e[\langle p \rangle / x] \downarrow^n e' \Rightarrow p'$

► **Goal A:**

$p' : \texttt{prefix}\,(t)$

► **Goal B:**

If $\delta_\eta\,(\Gamma') \sim \Gamma''$ and $\delta_{p'}\,(t) \sim t'$, then $\cdot \mid \Gamma'' \vdash_\emptyset e' : t'$ @

► **Goal C:**

If $i = \texttt{I}$ and $\eta\ \texttt{emptyOn}\ \texttt{wait}_{\eta',t}(x)\,(e)$ then $p'\ \texttt{empty}$

By Theorem C.31 on (1), (2), (5), and (6)

(10) $\eta'' : \texttt{env}\,(\Gamma(x : s))$

(11) If $\eta' : \texttt{env}\,(\Gamma')$ and $\delta_{\eta'}\,(\Gamma') \sim \Gamma''$, then $\delta_{\eta''}\,(\Gamma(x : s)) \sim \Gamma''$

By Theorem C.22 on (10) and (7)

(12) $p : \texttt{prefix}\,(s)$

By Definition C.36 on (12) and (8)

(13) $\langle p \rangle : \langle s \rangle$

By Definition C.37 with (3) and (13)

(14) $\cdot \mid \Gamma(\cdot) \vdash_\emptyset e[\langle p \rangle / x] : t$ @ i'

By Theorem C.39 on (10), using $\Gamma(x : s) \leq \Gamma(\cdot)$

(15) $\eta'' : \texttt{env}\,(\Gamma(\cdot))$

By IH on (9) with (14) and (15)

(16) $p' : \texttt{prefix}\,(t)$

(17) For all Γ'', if $\delta_{\eta''}\,(\Gamma(\cdot)) \sim \Gamma''$ and $\delta_{p'}\,(t) \sim t'$, then $\cdot \mid \Gamma'' \vdash_\emptyset e' : t'$ @ I

(16) completes **Goal A**. For **Goal B**, suppose:

(18) $\delta_\eta\,(\Gamma') \sim \Gamma''$

(19) $\delta_{p'}\,(t) \sim t'$

By (18) and (11)

(20) $\delta_{\eta''}\,(\Gamma(x : s)) \sim \Gamma''$

By Theorem C.21, on (20) using (6), there is some $\Gamma'''(-)$ so that:

(21) For all Δ' and Δ'' and η_0, if $\delta_{\eta_0}\,(\Delta') \sim \Delta''$ and $\texttt{agree}\,(\eta'', \eta_0, \Delta, \Delta')$ then $\delta_{\eta'' \cdot \eta_0}\,(\Gamma(\Delta')) \sim \Gamma'''(\Delta'')$

By (21), taking $\Delta' = \cdot$, $\Delta'' = \big(x : \delta_p\,(s)\big)$, and $\eta_0 = \{x \mapsto p\}$

(22) $\Gamma'' = \Gamma'''(x : \delta_p\,(s))$h

By (21) again, taking $\Delta' = \Delta'' = \cdot$

(23) $\delta_{\eta''}\,(\Gamma(\cdot)) \sim \Gamma'''(\cdot)$

By (17) on (23) and (19)

(24) $\cdot \mid \Gamma'''(\cdot) \vdash_\emptyset e' : t'$ @

Goal B follows from (23), using $\Gamma'''(x : \delta_p\,(s)) \leq \Gamma'''(\cdot)$

For **Goal C**, assume:

(25) $i = \texttt{I}$

(26) $\eta\ \texttt{emptyOn}\ \texttt{wait}_{\eta',t}(x)\,(e)$

Since $x \in \text{fv}(\texttt{wait}_{\eta',t}(x)\,(e))$, we have

(27) $\eta(x)\ \texttt{empty}$

Applying (4) to (25)

(27) $\neg\,(\eta'(x)\ \texttt{maximal})$

(28) $\neg\,(s\ \texttt{null})$

By the definition of environment concatenation, on (6) and (7)

(29) $\eta'(x) \cdot \eta(x) \sim p$

By Theorem C.27 on (29) and (8)
(30) $\eta(x)$ `maximal`
But by Theorem C.9, (27), (30), and (28) are contradictory.

► **Case 26:** *S-Fix.*

► **Given:**
(1) $\Omega \mid \Gamma \vdash_{\Omega \mid \Gamma \to s \,@\, i} e : s \,@\, i$
(2) $\cdot \mid \Gamma' \vdash_{\emptyset} A : \Gamma \,@\, i$
(3) $\cdot \vdash \overline{M} : \Omega$
(4) η : $\mathsf{env}\,(\Gamma')$
(5) $\overline{M} \downarrow \theta$
(6) $\eta \Rightarrow \mathtt{let}\,\Gamma = A\,\mathtt{in}\,e\,[e/\mathtt{rec}]\,[\theta] \downarrow^{n} e' \Rightarrow p$

► **Goal A:**
p : $\mathtt{prefix}\,(s)$

► **Goal B:**
If $\delta_\eta\,(\Gamma') \sim \Gamma''$ and $\delta_p\,(s) \sim s'$, then $\cdot \mid \Gamma'' \vdash_{\emptyset} e' : s' \,@\, \mathtt{I}$

► **Goal C:**
If $i = \mathtt{I}$ and η `emptyOn` $\mathtt{let}\,\Gamma = A\,\mathtt{in}\,e$ then p `empty`

By Theorem C.47 on (1)
(7) $\Omega \mid \Gamma \vdash_{\emptyset} e\,[e/\mathtt{rec}] : s \,@\, i$
By Definition C.37 on (4)
(8) $\theta : \Omega'$
Then again by Definition C.37 on (7) and (8)
(9) $\cdot \mid \Gamma \vdash_{\emptyset} e\,[e/\mathtt{rec}]\,[\theta] : s \,@\, i$
By T-ArgsLet on (2) and (9)
(10) $\cdot \mid \Gamma' \vdash_{\emptyset} \mathtt{let}\,\Gamma = A\,\mathtt{in}\,e\,[e/\mathtt{rec}]\,[\theta] : s \,@\, i$
Goal follows immediately by IH on (6), with (4) and (10).

► **Case 27:** *S-ArgsLet.*

► **Given:**
(1) $\cdot \mid \Gamma_i \vdash_{\emptyset} A : \Gamma_o \,@$
(2) $\cdot \mid \Gamma_o \vdash_{\emptyset} e : s \,@$
(3) η : $\mathsf{env}\,(\Gamma_i)$
(4) $\eta \Rightarrow A @ \Gamma_o \downarrow^{n_1} A' \Rightarrow \eta'$
(5) $\eta' \Rightarrow e \downarrow^{n_2} e' \Rightarrow p$
(6) $\delta_{\eta'}\,(\Gamma_o) \sim \Gamma_o'$

► **Goal A:**
p : $\mathtt{prefix}\,(s)$

► **Goal B:**
For all Γ_i' and s', if $\delta_\eta\,(\Gamma_i) \sim \Gamma_i'$ and $\delta_{p'}\,(s) \sim s'$ then $\cdot \mid \Gamma_i' \vdash_{\emptyset} \mathtt{let}\,\Gamma_o' = A'\,\mathtt{in}\,e' : s' \,@$

► **Goal C:**
If $i = \mathtt{I}$ and η `emptyOn` $\mathtt{let}\,\Gamma_o = A\,\mathtt{in}\,e$ then p `empty`

By IH on (4), with (1) and (3)
(7) η' : $\mathsf{env}\,(\Gamma_o)$
(8) For all Γ_i', if $\delta_\eta\,(\Gamma_i) \sim \Gamma_i'$ then $\cdot \mid \Gamma_i' \vdash_{\emptyset} A' : \Gamma_o' \,@$
(9) If $i = \mathtt{I}$ and η `emptyOn` A then η' `emptyOn` Γ_o

By IH on (5), with (2) and (7)
(10) p : prefix (s)
(11) For all s', if $\delta_p(s) \sim s'$, then $\cdot \mid \Gamma_o' \vdash_\emptyset e' : s'$ @
(12) If $i = \mathtt{I}$ and η' emptyOn e then p empty
Goal A is (9). **Goal B** follows by assuming Γ_i' and s', specializing (8) and (10), and applying T-ArgsLet.
For **Goal C**, assume:
(13) $i = \mathtt{I}$
(14) η emptyOn let $\Gamma_o = A$ in e
Because $\mathrm{fv}(\mathtt{let}\,\Gamma_o = A\,\mathtt{in}\,e) = \mathrm{fv}(A)$
(15) η emptyOn A
Applying (9) to (13) and (15)
(16) η' emptyOn Γ_o
Because of (2), $\mathrm{fv}(e) \subseteq \Gamma_o$, and so (16) implies
(17) η' emptyOn e
Goal C is complete by applying (12) to (13) and (17)

► **Case 28:** *S-Args-Emp.*

Immediate.

► **Case 29:** *S-Args-Sng.*

Immediate by IH.

► **Case 30:** *S-Args-Comma.*

Like T-Par-R

► **Case 31:** *S-Args-Semic-1-1.*

Like T-Cat-R-1

► **Case 32:** *S-Args-Semic-1-2.*

Like T-Cat-R-2

► **Case 33:** *S-Args-Semic-2-1.*

Immediate by IH

□

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

Theorem C.55 (Sink Term Semantics Characterization). *If* p : prefix (s), *and* p maximal *then for all* n *and* η, *we have* $\eta \Rightarrow \mathtt{sink}_p \downarrow^n \mathtt{sink}_p \Rightarrow \mathtt{emp}_{\delta_p(s)}$.

Theorem C.56 (Homomorphism Theorem). (1) *Suppose:*
- $\cdot \mid \Gamma \vdash_\emptyset e : s$ @ i
- η : env (Γ)
- η' : env $\left(\delta_\eta(\Gamma)\right)$
- $\eta \Rightarrow e \downarrow^{n_1} e' \Rightarrow p$
- $\eta' \Rightarrow e' \downarrow^{n_2} e'' \Rightarrow p'$
- $\eta \cdot \eta' \Rightarrow e \downarrow^{n_1+n_2} e''' \Rightarrow p''$

Then $e'' = e'''$ *and* $p \cdot p' \sim p''$.

(2) *Suppose:*
- $\cdot \mid \Gamma_i \vdash_\emptyset A : \Gamma_i \mathbin{@} i$
- $\eta_i : \mathsf{env}(\Gamma_i)$
- $\eta_i' : \mathsf{env}(\delta_{\eta_i}(\Gamma))$
- $\eta_i \Rightarrow A@\Gamma_o \downarrow^{n_1} A' \Rightarrow \eta_o$
- $\eta_i' \Rightarrow A'@\delta_{\eta_o}(\Gamma_o) \downarrow^{n_2} A'' \Rightarrow \eta_o'$
- $\eta_i \cdot \eta_i' \Rightarrow A@\Gamma_o \downarrow^{n_1+n_2} A''' \Rightarrow \eta_o''$

Then $A'' = A'''$, *and* $\eta_o \cdot \eta_o' \sim \eta_o''$.

By mutual induction on the semantics judgments, then inverting all other judgments. To reduce clutter, we will omit the typing premises that simply go along for the ride in each case. We name the cases by the rule used for the step of e, and then if they are not uniquely determined, the step for e' an then the step of e on $\eta \cdot \eta'$.

► **Case 1:** *S-Eps-R.*

Immediate.

► **Case 2:** *S-One-R.*

Immediate.

► **Case 3:** *S-Var.*

Immediate.

► **Case 4:** *S-Par-R.*

► **Given:**
(1) $\eta_1 \Rightarrow e_1 \downarrow^{n_1} e_1' \Rightarrow p_1$
(2) $\eta_1 \Rightarrow e_2 \downarrow^{n_2} e_2' \Rightarrow p_2$
(3) $\eta_2 \Rightarrow e_1' \downarrow^{n_1'} e_1'' \Rightarrow p_1'$
(4) $\eta_2 \Rightarrow e_2' \downarrow^{n_2'} e_2'' \Rightarrow p_2'$
(5) $\eta_1 \cdot \eta_2 \Rightarrow e_1 \downarrow^{n_1''} e_1''' \Rightarrow p_1''$
(6) $\eta_1 \cdot \eta_2 \Rightarrow e_2 \downarrow^{n_2''} e_2''' \Rightarrow p_2''$

► **Goal A:**
$\boxed{(e_1'', e_2'') = (e_1''', e_2''')}$

► **Goal B:**
$\boxed{\mathtt{parPair}(p_1, p_2) \cdot \mathtt{parPair}(p_1', p_2') \sim \mathtt{parPair}(p_1'', p_2'')}$

By IH on (1), (3), and (5)
(7) $e_1'' = e_1'''$
(8) $p_1 \cdot p_1' \sim p_1''$

By IH on (2), (4), and (6)
(9) $e_2'' = e_2'''$
(10) $p_2 \cdot p_2' \sim p_2''$

Goal A is immediate by (7) and (9)
Goal B is immediate by (8) and (10)

► **Case 5:** *S-Cat-R-1, S-Cat-R-1, S-Cat-R-1.*

► **Given:**
(1) $\eta_1 \Rightarrow e_1 \downarrow^{n} e_1' \Rightarrow p_1$

(2) $\neg\,(p_1\ \texttt{maximal})$
(3) $\eta_2 \Rightarrow e_1' \downarrow^{n'} e_1'' \Rightarrow p_1'$
(4) $\neg\,(p_1'\ \texttt{maximal})$
(5) $\eta_1 \cdot \eta_2 \Rightarrow e_1 \downarrow^{n} e_1''' \Rightarrow p_1''$
(6) $\neg\,(p_1''\ \texttt{maximal})$

► **Goal A:**

$$(e_1''; e_2) = (e_1'''; e_2)$$

► **Goal B:**

$$\texttt{catFst}(p_1) \cdot \texttt{catFst}(p_1') \sim \texttt{catFst}(p_1'')$$

Immediate by IH on (1), with (3) and (5).

► **Case 6:** *S-Cat-R-1, S-Cat-R-1, S-Cat-R-2.*

► **Given:**
(1) $\eta_1 \Rightarrow e_1 \downarrow^{n} e_1' \Rightarrow p_1$
(2) $\neg\,(p_1\ \texttt{maximal})$
(3) $\eta_2 \Rightarrow e_1' \downarrow^{n'} e_1'' \Rightarrow p_1'$
(4) $\neg\,(p_1'\ \texttt{maximal})$
(5) $\eta_1 \cdot \eta_2 \Rightarrow e_1 \downarrow^{n} e_1''' \Rightarrow p_1''$
(6) $p_1''\ \texttt{maximal}$
(7) $\eta_1 \cdot \eta_2 \Rightarrow e_2 \downarrow^{n} e_2' \Rightarrow p_2$

► **Goal A:**

$$(e_1''; e_2) = e_2'$$

► **Goal B:**

$$\texttt{catFst}(p_1) \cdot \texttt{catFst}(p_1') \sim \texttt{catBoth}(p_1'', p_2)$$

By IH on (1), with (3) and (5):
(8) $e_1'' = e_1'''$
(9) $p_1 \cdot p_1' \sim p_1''$

But (9) is impossible by Theorem C.27, since p_1'' is maximal, but neither p_1 nor p_1' are.

► **Case 7:** *S-Cat-R-1, S-Cat-R-2, S-Cat-R-1.*

► **Given:**
(1) $\eta_1 \Rightarrow e_1 \downarrow^{n} e_1' \Rightarrow p_1$
(2) $\neg\,(p_1\ \texttt{maximal})$
(3) $\eta_2 \Rightarrow e_1' \downarrow^{n_1} e_1'' \Rightarrow p_1'$
(4) $p_1'\ \texttt{maximal}$
(5) $\eta_2 \Rightarrow e_2 \downarrow^{n_1} e_2' \Rightarrow p_2$
(5) $\eta_1 \cdot \eta_2 \Rightarrow e_1 \downarrow^{n} e_1''' \Rightarrow p_1''$
(6) $\neg\,(p_1''\ \texttt{maximal})$

► **Goal A:**

$$e_2' = (e_1''; e_2)$$

► **Goal B:**

$$\texttt{catFst}(p_1) \cdot \texttt{catBoth}(p_1', p_2) \sim \texttt{catFst}(p_1'')$$

By IH on (1), with (3) and (5):
(8) $e_1'' = e_1'''$
(9) $p_1 \cdot p_1' \sim p_1''$

But (9) is impossible by Theorem C.27, since p_1' is maximal, p_1'' is not.

► **Case 8:** *S-Cat-R-1, S-Cat-R-2, S-Cat-R-2.*

► **Given:**

(1) $\cdot \mid \Gamma \vdash_\emptyset e_1 : s \mathbin{@} i_1$
(2) $\cdot \mid \Delta \vdash_\emptyset e_2 : t \mathbin{@} i_2$
(3) $\eta_1 \,\texttt{maximalOn}\, \Gamma \vee \eta_2 \,\texttt{maximalOn}\, \Delta$
(4) $\eta_1 \Rightarrow e_1 \downarrow^n e_1' \Rightarrow p_1$
(5) $\neg\,(p_1 \,\texttt{maximal})$
(6) $\eta_2 \Rightarrow e_1' \downarrow^{n_1} e_1'' \Rightarrow p_1'$
(7) $p_1' \,\texttt{maximal}$
(8) $\eta_2 \Rightarrow e_2 \downarrow^{n_1} e_2' \Rightarrow p_2$
(9) $\eta_1 \cdot \eta_2 \Rightarrow e_1 \downarrow^n e_1''' \Rightarrow p_1''$
(10) $p_1'' \,\texttt{maximal}$
(11) $\eta_1 \cdot \eta_2 \Rightarrow e_2 \downarrow^n e_2'' \Rightarrow p_2'$

► **Goal A:**

$\boxed{e_2' = e_2''}$

► **Goal B:**

$\boxed{\texttt{catFst}(p_1) \cdot \texttt{catBoth}(p_1', p_2) \sim \texttt{catBoth}(p_1'', p_2')}$

By IH on (4), with (6) and (8):

(12) $e_1'' = e_1'''$
(13) $p_1 \cdot p_1' \sim p_1''$

By the contrapositive of Theorem C.50 on (1), (3), and (5)

(14) $\neg\,(\eta_1 \,\texttt{maximalOn}\, \Gamma)$

By (3) and (14)

(15) $\eta_1 \,\texttt{emptyOn}\, \Delta$

Because Dom $(Delta) \geq \text{fv}(e_2)$, (15) implies

(16) $\eta_1 \,\texttt{emptyOn}\, e_2$

By Theorem C.32 on (16)

(17) $(\eta_1 \cdot \eta_2)\,|_{e_2} = \eta_2|_{e_2}$

So by (8) and (16)

(18) $\eta_1 \cdot \eta_2 \Rightarrow e_2 \downarrow^{n_1} e_2'' \Rightarrow p_2$

By (18) and Theorem C.52

(19) $p_2 = p_2'$
(20) $e_2' = e_2''$

Goal A is immediate by (20), and **Goal B** follows by (19) and (13)

► **Case 9:** *S-Cat-R-2.*

► **Given:**

(1) $\eta_1 \Rightarrow e_1 \downarrow^{n_1} e_1' \Rightarrow p_1$
(2) $p_1 \,\texttt{maximal}$
(3) $\eta_1 \Rightarrow e_2 \downarrow^{n_2} e_2' \Rightarrow p_2$
(4) $\eta_2 \Rightarrow e_2' \downarrow^{n'} e_2'' \Rightarrow p_2'$
(5) $\eta_1 \cdot \eta_2 \Rightarrow (e_1; e_2) \downarrow^{n''} e_0 \Rightarrow p_0$

► **Goal A:**

$\boxed{e_2'' = e_0}$

► **Goal B:**

$\boxed{\texttt{catBoth}(p_1, p_2) \cdot p_2' \sim p_0}$

We begin by inverting (5). The case of S-Cat-R-1 is impossible, because with Theorem C.27, this would contradict the maximality of p_1. Thus,

(6) $\eta_1 \cdot \eta_2 \Rightarrow e_1 \downarrow^{n_1'} e_1'' \Rightarrow p_1'$

(7) p_1' `maximal`

(8) $\eta_1 \cdot \eta_2 \Rightarrow e_2 \downarrow^{n_2'} e_2''' \Rightarrow p_2''$

The inversion also tells us that $p_0 = \texttt{catBoth}(p_1'', p_2'')$, and $e_0 = e_2'''$

By IH on (3), (4), and (8)

(9) $p_2 \cdot p_2' \sim p_2''$

(10) $e_2''' = e_2''$

Goal A is complete by (10)

By (9):

(11) $\texttt{catBoth}(p_1'', p_2) \cdot p_2' \sim \texttt{catBoth}(p_1'', p_2'')$

By Theorem C.51 on (1) and (6)

(12) $p_1' = p_1''$

Goal B follows by (11) and (12).

► **Case 10:** *S-Par-L.*

Immediate by two uses of IH

► **Case 11:** *S-Cat-L-1, S-Cat-L-1.*

► **Given:**

(1) $\eta_1(z) \mapsto \texttt{catFst}(p_1)$

(2) $\eta_1[x \mapsto p_1, y \mapsto \texttt{emp}_t] \Rightarrow e \downarrow^{n} e' \Rightarrow p_1'$

(3) $\eta_2(z) \mapsto \texttt{catFst}(p_2)$

(4) $\eta_2[x \mapsto p_2, y \mapsto \texttt{emp}_t] \Rightarrow e' \downarrow^{n'} e'' \Rightarrow p_2'$

(5) $\eta_1 \cdot \eta_2 \Rightarrow \texttt{let}_t\ (x; y)\ =\ z\ \texttt{in}\ e \downarrow^{n''} e_0 \Rightarrow p_0$

► **Goal A:**

$\boxed{\texttt{let}_t\ (x; y)\ =\ z\ \texttt{in}\ e'' = e_0}$

► **Goal B:**

$\boxed{p_1' \cdot p_2' \sim p_0}$

By Definition,

(6) $(\eta_1 \cdot \eta_2)(z) = \texttt{catFst}(p_1 \cdot p_2)$

By (6), inverting (5) can only conclude with S-Cat-L-1, and so $e_0 = \texttt{let}_t\ (x; y)\ =\ z\ \texttt{in}\ e'''$ such that

(7) $(\eta_1 \cdot \eta_2)\ [x \mapsto p_1 \cdot p_2, y \mapsto \texttt{emp}_t] \Rightarrow e \downarrow^{n''} e''' \Rightarrow p_0$

By definition:

(8) $(\eta_1 \cdot \eta_2)\ [x \mapsto p_1 \cdot p_2, y \mapsto \texttt{emp}_t] = \eta_1[x \mapsto p_1, y \mapsto \texttt{emp}_t] \cdot \eta_2[x \mapsto p_2, y \mapsto \texttt{emp}_t]$

By IH on (2), with (4) and (7), rewriting by (8)

(9) $e''' = e''$

(10) $p_1' \cdot p_2' \sim p_0$

(9) and (10) complete **Goal A** and **Goal B**, respectively.

► **Case 12:** *S-Cat-L-1, S-Cat-L-2.*

► **Given:**

(1) $\eta_1(z) \mapsto \mathtt{catFst}(p_1)$
(2) $\eta_1[x \mapsto p_1, y \mapsto \mathtt{emp}_t] \Rightarrow e \downarrow^n e' \Rightarrow p_1'$
(3) $\eta_2(z) \mapsto \mathtt{catBoth}(p_1'', p_2,)$
(4) $\eta_2[x \mapsto p_1'', y \mapsto p_2] \Rightarrow e' \downarrow^{n'} e'' \Rightarrow p_2'$
(5) $\eta_1 \cdot \eta_2 \Rightarrow \mathtt{let}_t\ (x;y)\ =\ z\ \mathtt{in}\ e \downarrow^{n''} e_0 \Rightarrow p_0$

► **Goal A:**

$$\boxed{\mathtt{let}\ x = \mathtt{sink}_{p_1''}\ \mathtt{in}\ e''[z/y] = e_0}$$

► **Goal B:**

$$\boxed{p_1' \cdot p_2' \sim p_0}$$

By Definition,

(6) $(\eta_1 \cdot \eta_2)(z) = \mathtt{catFst}(p_1) \cdot \mathtt{catBoth}(p_1'', p_2) = \mathtt{catBoth}(p_1 \cdot p_1'', p_2)$

By (6), inverting (5) can only conclude with S-Cat-L-2, and so $e_0 = \mathtt{let}\ x = \mathtt{sink}_{p_1 \cdot p_1''}\ \mathtt{in}\ e'''$, such that:

(7) $\eta_2[x \mapsto p_1 \cdot p_1'', y \mapsto p_2] \Rightarrow e \downarrow^{n''} e''' \Rightarrow p_0$

By IH on (2), with (4) and (7)

(8) $e''' = e''$
(9) $p_1' \cdot p_2' \sim p_0$

Goal B follows by (9). By Theorem C.46:

(10) $\mathtt{sink}_{p_1''} = \mathtt{sink}_{p_1 \cdot p_1''}$

Goal A follows by (8) and (10).

► **Case 13:** *S-Cat-L-2.*

► **Given:**

(1) $\eta_1(z) \mapsto \mathtt{catBoth}(p_1, p_2)$
(2) $\eta_1[x \mapsto p_1, y \mapsto p_2] \Rightarrow e \downarrow^n e' \Rightarrow p$
(3) $\eta_2 \Rightarrow \mathtt{let}\ x = \mathtt{sink}_{p_1}\ \mathtt{in}\ e'[z/y] \downarrow^{n_1+n_2} \mathtt{let}\ x = e_0\ \mathtt{in}\ e_0' \Rightarrow p'$
(4) $\eta_1 \cdot \eta_2 \Rightarrow \mathtt{let}_t\ (x;y)\ =\ z\ \mathtt{in}\ e \downarrow^{n'} e_1 \Rightarrow p''$

► **Goal A:**

$$\boxed{e_1 = \mathtt{let}\ x = e_0\ \mathtt{in}\ e_0'}$$

► **Goal B:**

$$\boxed{p \cdot p' \sim p''}$$

Inverting (3)

(5) $\eta_2 \Rightarrow \mathtt{sink}_{p_1} \downarrow^{n_1} e_0 \Rightarrow p_0$
(6) $\eta_2[x \mapsto p_0] \Rightarrow e'[z/y] \downarrow^{n_2} e_0' \Rightarrow p'$

By Definition, there is some p_2' such that $\eta_2(z) = p_2'$, and so

(7) $(\eta_1 \cdot \eta_2)\ (z) = \mathtt{catBoth}(p_1, p_2 \cdot p_2')$

Because of (7), inverting (4) can only end with S-Cat-L-2, and so we have that $e_1 = \mathtt{let}\ x = \mathtt{sink}_{p_1}\ \mathtt{in}\ e''[y/z]$, and

(8) $(\eta_1 \cdot \eta_2)\ [x \mapsto p_1, y \mapsto p_2 \cdot p_2'] \Rightarrow e \downarrow^{n'} e'' \Rightarrow p''$

By Theorem C.55 and Theorem C.52 on (5)

(9) $e_0 = \mathtt{sink}_{p_1}$
(10) $p_0 = \mathtt{emp}_{\delta_{p_1}(s)}$

Because $\eta_2(z) = p_2'$, we have that (6) equivalently says:

(11) $\eta_2[x \mapsto \mathtt{emp}_{\delta_{p_1}(s)}, y \mapsto p_2'] \Rightarrow e' \downarrow^{n_2} e_0' \Rightarrow p'$

Then, we can compute:

(12) $(\eta_1 \cdot \eta_2)\,[x \mapsto p_1, y \mapsto p_2 \cdot p_2'] = (\eta_1[x \mapsto p_1, y \mapsto p_2]) \cdot \left(\eta_2[x \mapsto \mathsf{emp}_{\delta_{p_1}(s)}, y \mapsto p_2']\right)$

So by IH on (2), with (11) and (8)

(13) $e'' = e_0'$

(14) $p \cdot p' \sim p''$

Goal A follows by (9) and (13), with **Goal B** immediate by (14)

► **Case 14:** *S-Plus-R-1.*

Immediate by IH.

► **Case 15:** *S-Plus-R-2.*

Immediate by IH.

► **Case 16:** *S-Plus-L-1, S-Plus-L-1.*

► **Given:**

(1) $\eta_1' \cdot \eta_1 \sim \eta_1''$

(2) $\eta_1''(z) = \mathsf{sumEmp}$

(3) $\eta_1'' \cdot \eta_2 \sim \eta_2'$

(4) $\eta_2'(z) = \mathsf{sumEmp}$

(5) $\eta_1 \cdot \eta_2 \Rightarrow \mathsf{case}_r\left(\eta_1'; z, x.e_1, y.e_2\right) \downarrow^n e_0 \Rightarrow p$

► **Goal A:**

$\boxed{e_0 = \mathsf{case}_r\left(\eta_2'; z, x.e_1, y.e_2\right)}$

► **Goal B:**

$\boxed{\mathsf{emp}_t \cdot \mathsf{emp}_t \sim p}$

By Theorem C.34

(6) $\eta_1' \cdot (\eta_1 \cdot \eta_2) = \left(\eta_1' \cdot \eta_1\right) \cdot \eta_2 = \eta_1'' \cdot \eta_2 = \eta_2'$

By (6) and (4), the only rule can conclude with (5) is S-Plus-L-1. Inverting (5) and rewriting by (6), we have that

(7) $e_0 = \mathsf{case}_r\left(\eta_2'; z, x.e_1, y.e_2\right)$

(8) $p = \mathsf{emp}_t$

Goal A is (7), and **Goal B** follows by Theorem C.26 and (8)

► **Case 17:** *S-Plus-L-1, S-Plus-L-2.*

► **Given:**

(1) $\eta_1' \cdot \eta_1 \sim \eta_1''$

(2) $\eta_1''(z) = \mathsf{sumEmp}$

(3) $\eta_1'' \cdot \eta_2 \sim \eta_2'$

(4) $\eta_2'(z) = \mathsf{sumInl}(p)$

(5) $\eta_2'[x \mapsto p] \Rightarrow e_1 \downarrow^n e_1' \Rightarrow p'$

(6) $\eta_1 \cdot \eta_2 \Rightarrow \mathsf{case}_r\left(\eta_1'; z, x.e_1, y.e_2\right) \downarrow^{n'} e_0 \Rightarrow p''$

► **Goal A:**

$\boxed{e_0 = e_1'[z/x]}$

► **Goal B:**

$\boxed{\mathsf{emp}_t \cdot p' \sim p''}$

By Theorem C.34

(7) $\eta_1' \cdot (\eta_1 \cdot \eta_2) = \left(\eta_1' \cdot \eta_1\right) \cdot \eta_2 = \eta_1'' \cdot \eta_2 = \eta_2'$

By (7) and (4), the only rule can conclude with (5) is S-Plus-L-4. Inverting (6) and rewriting by (7), we have that there is e_1''

(8) $e_0 = e_1''[z/x]$

(9) $\eta_2'[x \mapsto p] \Rightarrow e_1 \downarrow^{n'} e_1'' \Rightarrow p'$

By Theorem C.52 on (5) and (9)

(10) $e_1'' = e_1'$

(11) $p'' = p'$

Goal A follows by (8) and (10), while **Goal B** follows by Theorem C.26 and (11)

► **Case 18:** *S-Plus-L-1, S-Plus-L-3.*

Identical to Previous.

► **Case 19:** *S-Plus-L-2.*

► **Given:**

(1) $\eta_1' \cdot \eta_1 \sim \eta_1''$

(2) $\eta_1''(z) = \texttt{sumInl}(p_1)$

(3) $\eta_1''[x \mapsto p_1] \Rightarrow e_1 \downarrow^{n} e_1' \Rightarrow p$

(4) $\eta_2 \Rightarrow e_1'[z/x] \downarrow^{n'} e \Rightarrow p'$

(5) $\eta_1 \cdot \eta_2 \Rightarrow \texttt{case}_r\left(\eta_1'; z, x.e_1, y.e_2\right) \downarrow^{n''} e' \Rightarrow p''$

► **Goal A:**

$\boxed{e = e'}$

► **Goal B:**

$\boxed{p \cdot p' \sim p''}$

By Theorem C.34:

(6) $\eta_1' \cdot (\eta_1 \cdot \eta_2) = \left(\eta_1' \cdot \eta_1\right) \cdot \eta_2 = \eta_1'' \cdot \eta_2$

By (6), there is some p_1' such that:

(7) $\eta_2(z) = p_1'$

(8) $\left(\eta_1'' \cdot \eta_2\right) = \texttt{sumInl}(p_1 \cdot p_1')$

Using (7), we can rewrite (4) as:

(9) $\eta_2[x \mapsto p_1'] \Rightarrow e_1' \downarrow^{n'} e \Rightarrow p'$

By (8) and (6), inverting (5) can only lead to S-Plus-L-2. Doing so yields:

(10) $\left(\eta_1'' \cdot \eta_2\right)[x \mapsto p_1 \cdot p_1'] \Rightarrow e_1 \downarrow^{n''} e_1'' \Rightarrow p''$

(11) $e' = e_1''[z/x]$

We can compute that:

(12) $\left(\eta_1'' \cdot \eta_2\right)[x \mapsto p_1 \cdot p_1'] = \left(\eta_1''[x \mapsto p_1]\right) \cdot \left(\eta_2'[x \mapsto p_1']\right)$

So by (8), we can write (10) as

13 $\left(\eta_1''[x \mapsto p_1]\right) \cdot \left(\eta_2'[x \mapsto p_1']\right) \Rightarrow e_1 \downarrow^{n''} e_1'' \Rightarrow p''$

Both goals follow by IH on (3), with (9) and (13)

► **Case 20:** *S-Plus-L-3.*

Identical to previous.

► **Case 21:** *S-Star-R-1.*

Immediate.

► **Case 22:** *S-Star-R-2-1.*

Identical to the cases for S-Cat-R-1

► **Case 23:** *S-Star-R-2-2.*

Identical to the cases for S-Cat-R-2

▶ **Case 24:** *S-Star-L-1.*

Identical to the cases for S-Plus-L-1

▶ **Case 25:** *S-Star-L-2.*

Identical to S-Plus-L-2

▶ **Case 26:** *S-Star-L-3.*

Identical to S-Plus-L-2

▶ **Case 27:** *S-Star-L-4.*

Identical to S-Plus-L-2

▶ **Case 28:** *S-Let.*

▶ **Given:**

(1) $\eta_1 \Rightarrow e_1 \downarrow^{n_1} e_1' \Rightarrow p_1$
(2) $\eta_1[x \mapsto p_1] \Rightarrow e_2 \downarrow^{n_2} e_2' \Rightarrow p_1'$
(3) $\eta_2 \Rightarrow e_1' \downarrow^{n_1'} e_1'' \Rightarrow p_2$
(4) $\eta_2[x \mapsto p_2] \Rightarrow e_2' \downarrow^{n_2'} e_2'' \Rightarrow p_2'$
(5) $\eta_1 \cdot \eta_2 \Rightarrow e_1 \downarrow^{n_1''} e_1''' \Rightarrow p_3$
(6) $(\eta_1 \cdot \eta_2)\,[x \mapsto p_3] \Rightarrow e_2 \downarrow^{n_2''} e_2''' \Rightarrow p_3'$

▶ **Goal A:**

$\boxed{\texttt{let}\, x = e_1''\, \texttt{in}\, e_2'' = \texttt{let}\, x = e_1'''\, \texttt{in}\, e_2'''}$

▶ **Goal B:**

$\boxed{p_1' \cdot p_2' \sim p_3'}$

By IH on (1), with (3) and (4)

(7) $e_1'' = e_1'''$
(8) $p_1 \cdot p_2 \sim p_3$

By (8):

(9) $(\eta_1 \cdot \eta_2)\,[x \mapsto p_3] = (\eta_1[x \mapsto p_1]) \cdot (\eta_2[x \mapsto p_2])$

By (9), we can rewrite (6) to:

(10) $(\eta_1[x \mapsto p_1]) \cdot (\eta_2[x \mapsto p_2]) \Rightarrow e_2 \downarrow^{n_2''} e_2''' \Rightarrow p_3'$

By IH on (2), with (4) and (7)

(11) $e_2'' = e_2'''$
(12) $p_1' \cdot p_2' \sim p_3'$

Goal A is immediate by (7) and (11), and **Goal B** is (12).

▶ **Case 29:** *S-HistPgm.*

Immediate by Theorem C.55 and Theorem C.46.

▶ **Case 30:** *S-Wait-1, S-Wait-1.*

▶ **Given:**

(1) $\eta_1' \cdot \eta_1 \sim \eta_1''$
(2) $\eta_1''(x) = p$
(3) $\neg\,(p\ \texttt{maximal})$
(4) $\eta_1'' \cdot \eta_2 \sim \eta_2'$

(5) $\eta_2'(x) = p'$
(6) $\neg\,(p'\ \texttt{maximal})$
(7) $\eta_1 \cdot \eta_2 \Rightarrow \texttt{wait}_{\eta_1',t}(x)\ (e) \downarrow^{n''} e' \Rightarrow \texttt{emp}_t$

► **Goal A:**

$\boxed{e' = \texttt{wait}_{\eta_2',t}(x)\ (e)}$

► **Goal B:**

$\boxed{\texttt{emp}_t \cdot \texttt{emp}_t \sim \texttt{emp}_t}$

Note that, by Theorem C.28

(8) $\eta_1' \cdot (\eta_1 \cdot \eta_2) = (\eta_1' \cdot \eta_1) \cdot \eta_2 = \eta_1'' \cdot \eta_2 = \eta_2'$

Because of (8), when we invert (7), only S-Wait-1 can apply. Inverting (7) completes **Goal A**, and **Goal B** follows by Theorem C.26

► **Case 31:** *S-Wait-1, S-Wait-2.*

► **Given:**
(1) $\eta_1' \cdot \eta_1 \sim \eta_1''$
(2) $\eta_1''(x) = p$
(3) $\neg\,(p\ \texttt{maximal})$
(4) $\eta_1'' \cdot \eta_2 \sim \eta_2'$
(5) $\eta_2'(x) = p'$
(6) $p'\ \texttt{maximal}$
(7) $\eta_2' \Rightarrow e[\langle p' \rangle / x] \downarrow^{n} e' \Rightarrow p''$
(8) $\eta_1 \cdot \eta_2 \Rightarrow \texttt{wait}_{\eta_1',t}(x)\ (e) \downarrow^{n'} e'' \Rightarrow p'''$

► **Goal A:**

$\boxed{e'' = e'}$

► **Goal B:**

$\boxed{\texttt{emp}_t \cdot p'' \sim p'''}$

By Theorem C.28

(8) $\eta_1' \cdot (\eta_1 \cdot \eta_2) = (\eta_1' \cdot \eta_1) \cdot \eta_2 = \eta_1'' \cdot \eta_2 = \eta_2'$

Because of (8), when we invert (7), only S-Wait-2 can apply. Inverting (7) and applying Theorem C.52 completes **Goal A**. **Goal B** is complete by Theorem C.26

► **Case 32:** *S-Wait-2.*

► **Given:**
(1) $\eta_1' \cdot \eta_1 \sim \eta_1''$
(2) $\eta_1''(x) = p$
(3) $p\ \texttt{maximal}$
(4) $\eta_1'' \Rightarrow e[\langle p \rangle / x] \downarrow^{n} e' \Rightarrow p'$
(5) $\eta_2 \Rightarrow e' \downarrow^{n'} e'' \Rightarrow p''$
(6) $\eta_1 \cdot \eta_2 \Rightarrow \texttt{wait}_{\eta_1',t}(x)\ (e) \downarrow^{n''} e''' \Rightarrow p'''$

► **Goal A:**

$\boxed{e'' = e'''}$

► **Goal B:**

$\boxed{p' \cdot p'' \sim p'''}$

By Theorem C.28 on (1):

(7) $\eta_1' \cdot (\eta_1 \cdot \eta_2) = (\eta_1' \cdot \eta_1) \cdot \eta_2 = \eta_1'' \cdot \eta_2$

By definition using (7), there is some p_0 so that $\eta_2(x) = p_0$, and

(8) $\left(\eta_1'' \cdot \eta_2\right)(x) = p \cdot p_0$
By Theorem C.27 on (3) and (8)
(9) $p \cdot p_0 = p$
By (9) and (3), inverting (6) can only conclude by S-Wait-2. Doing so yields:
(10) $\eta_1'' \cdot \eta_2 \Rightarrow e[\langle p \rangle / x] \downarrow^{n''} e''' \Rightarrow p'''$
Goal A and **Goal B** follow immediately by IH on (4), with (5), (10).

► **Case 33:** *S-Fix.*

Immediate by IH and Definition C.37

► **Case 34:** *S-ArgsLet.*

Immediate by two uses of IH.

► **Case 35:** *S-Args-Emp.*

Immediate.

► **Case 36:** *S-Args-Sng.*

Immediate by IH.

► **Case 37:** *S-Args-Comma.*

Immediate by two uses of IH.

► **Case 38:** *S-Args-Semic-1-1, S-Args-Semic-1-1.*

Immediate by IH.

► **Case 39:** *S-Args-Semic-1-1, S-Args-Semic-1-2.*

Like the S-Cat-R-1, S-Cat-R-2 case.

► **Case 40:** *S-Args-Semic-1-2, S-Args-Semic-2.*

Immediate by IH.

► **Case 41:** *S-Args-Semic-2.*

Immediate by IH.

□

C.10 Determinism

Theorem C.57 (Determinism Theorem). *Suppose:*

(1) $\cdot \mid \Gamma, \Gamma' \vdash_{\emptyset} e : s \mathbin{@} i$
(1) $\eta : \mathsf{env}(\Gamma, \Gamma')$
(2) $\eta|_{\Gamma} \cup \mathsf{emp}_{\Gamma'} \Rightarrow e \downarrow^{n_1} e_1 \Rightarrow p_1$ *and* $\eta|_{\Gamma'} \cup \mathsf{emp}_{\Gamma} \Rightarrow e_1 \downarrow^{n_2} e_2 \Rightarrow p_2$
(3) $\eta|_{\Gamma'} \cup \mathsf{emp}_{\Gamma} \Rightarrow e \downarrow^{n_1'} e_1' \Rightarrow p_1'$ *and* $\eta|_{\Gamma} \cup \mathsf{emp}_{\Gamma'} \Rightarrow e_1' \downarrow^{n_2'} e_2' \Rightarrow p_2'$.
(4) $\eta \Rightarrow e \downarrow e' \Rightarrow p$

Then $e' = e_2 = e_2'$ *and* $p = p_1 \cdot p_2 = p_1' \cdot p_2'$.

Proof. By Theorem C.32, $\eta|_\Gamma \cdot \mathsf{emp}_\Gamma = \eta|_\Gamma$, and $\eta|_{\Gamma'} \cdot \mathsf{emp}_{\Gamma'} = \eta|_{\Gamma'}$. Then, we can compute the concatenation of the subsequent input environments in (2), and those in (3).

$$\begin{aligned}(\eta|_\Gamma \cup \mathsf{emp}_{\Gamma'}) \cdot (\eta|_{\Gamma'} \cup \mathsf{emp}_\Gamma) &= (\eta_\Gamma \cdot \mathsf{emp}_\Gamma) \cup (\eta|_{\Gamma'} \cdot \mathsf{emp}_{\Gamma'}) \\ &= \eta|_\Gamma \cup \eta_{\Gamma'} \\ &= \eta\end{aligned}$$

By the same argument,

$$\eta|_{\Gamma'} \cup \mathsf{emp}_\Gamma \cdot \eta|_\Gamma \cup \mathsf{emp}_{\Gamma'} = \eta$$

By two uses of Theorem C.56, we have: $e' = e_2 = e_2'$, and $p = p_1 \cdot p_2 = p_1' \cdot p_2'$, as required. □

C.11 Events

Events allow us to represent a λ^{ST} prefix as a sequence of totally ordered items, while retaining information needed to infer the rich structure of the prefix representations. In this section, we define serialization and deserialization functions from sequences of events to prefixes and back. We further prove that, for any type s, the size of the possible events that may occur on a channel sending events of type s (and its derivatives) is bounded. This section serves to justify our claim from Section 3 that λ^{ST} can be run atop a traditional stream processing system where streams are sequences.

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 C.58 (Event Typing Relation). We define a binary relation $x : \mathsf{event}\,(s)$ as follows:

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

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

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

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 C.59 (Event Derivative Relation). We define a ternary relation $\delta_x\,(s) \sim s'$.

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

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

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

Theorem C.60 (Event Derivative Function). *If $x : \mathsf{event}\,(s)$, there is a unique s' such that $\delta_x\,(s) \sim s'$.*

Because of Theorem C.60, 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 C.61 (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 C.62 (Events Derivative Function). *If $xs : \texttt{events}\,(s)$, there is a unique s' such that $\delta_{xs}\,(s) \sim s'$.*

Because of Theorem C.62, 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 C.63 (Empty List of Events). *For all s, we have $[] : \texttt{events}\,(s)$, and $\delta_{[]}\,(s) \sim s$*

Theorem C.64 (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. Induction on xs. □

Events

Events to Prefix

Definition C.65 (Event(s) to Prefix).

$$\frac{}{\texttt{oneev} \hookrightarrow^{1} \texttt{oneFull}} \qquad \frac{x \hookrightarrow^{s} p}{\texttt{parevA}\,(x) \hookrightarrow^{s \| t} \texttt{parPair}(p, \texttt{emp}_t)}$$

$$\frac{x \hookrightarrow^{t} p}{\texttt{parevB}\,(x) \hookrightarrow^{s \| t} \texttt{parPair}(\texttt{emp}_s, p)} \qquad \frac{x \hookrightarrow^{s} p}{\texttt{catevA}\,(x) \hookrightarrow^{s \cdot t} \texttt{catFst}(p)}$$

$$\frac{s\ \texttt{null} \qquad s\ \texttt{done}\ b}{{}_{\cdot\texttt{punc}} \hookrightarrow^{s \cdot t} \texttt{catBoth}(b, \texttt{emp}_t)} \qquad \frac{}{{}_{+\texttt{puncA}} \hookrightarrow^{s+t} \texttt{sumInl}(\texttt{emp}_s)} \qquad \frac{}{{}_{+\texttt{puncB}} \hookrightarrow^{s+t} \texttt{sumInr}(\texttt{emp}_t)}$$

$$\frac{}{{}_{+\texttt{puncA}} \hookrightarrow^{s^\star} \texttt{starDone}} \qquad \frac{}{{}_{+\texttt{puncB}} \hookrightarrow^{s^\star} \texttt{starFirst}(\texttt{emp}_s)}$$

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

Theorem C.66 (Event to Prefix Function). *If $x : \texttt{event}\,(s)$ then there is a unique $p : \texttt{prefix}\,(s)$ such that $x \hookrightarrow^{s} p$.*

Proof. Induction on $x : \texttt{event}\,(s)$. □

Theorem C.67 (Events to Prefix Function). *If* $xs : \texttt{events}\,(s)$, *then there is a unique* $p : \texttt{prefix}\,(s)$ *such that* $xs \hookrightarrow^s p$.

Proof. Induction on xs, using Theorem C.66. □

Prefix to Events

Definition C.68 (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.

$$\text{PToES-Eps}\ \frac{}{\texttt{epsEmp} \dagger^{\epsilon} []} \qquad \text{PToES-One-A}\ \frac{}{\texttt{oneEmp} \dagger^{1} []} \qquad \text{PToES-One-B}\ \frac{}{\texttt{oneFull} \dagger^{1} [\texttt{oneev}]}$$

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

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

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

$$\text{PToES-Star-A}\ \frac{p \dagger^{s} xs}{\texttt{starFirst}(p) \dagger^{s^{\star}} +_{\texttt{puncB}} :: \widehat{\texttt{catevA}}\,(xs)}$$

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

Theorem C.69 (PToES Empty). *If* $p \dagger^{s} xs$, *then* $xs = []$ *iff* $p = \texttt{emp}_s$

Proof. Induction on $p \dagger^{s} xs$. □

Theorem C.70 (PToES Left Total). *If* $p : \texttt{prefix}\,(s)$ *then there exists (not necessarily unique)* xs *such that* $p \dagger^{s} xs$

Proof. Induction on $p : \texttt{prefix}\,(s)$. □

Lemma C.71 (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. Induction on $p \dagger^{s} xs$. □

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 C.72 (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 C.73 (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 C.74 (Event size bound).

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

Theorem C.75 (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. Induction on □

Serialization and Deserialization

We turn now to the final result of **??**, 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 C.76 (EsToP Par Recovery). *If*

(1) $zs \in \widehat{\texttt{parevA}}\,(xs) \,\|\widehat{\texttt{parevB}}\,(ys)$ *(where $\|$ is list shuffle)*
(2) $xs \hookrightarrow^s p$

(3) $ys \hookrightarrow^{t} p'$
then $zs \hookrightarrow^{s \| t} \texttt{parPair}(p, p')$

Proof. Induction on zs. □

Lemma C.77 (EsToP Cat Recovery). *If* $xs \hookrightarrow^{s} (b)^{\circ}$ *and* $ys \hookrightarrow^{t} p$ *then*

$$\widehat{\texttt{catevA}}\,(xs) +\!\!+ \cdot_{\texttt{punc}} :: ys \hookrightarrow^{s \cdot t} \texttt{catBoth}(b, p)$$

Proof. Induction on xs. □

Lemma C.78 (EsToP Star Recovery). *If* $xs \hookrightarrow^{s} (b)^{\circ}$ *and* $ys \hookrightarrow^{s^{\star}} p$ *then*

$$+_{\texttt{puncB}} :: \widehat{\texttt{catevA}}\,(xs) +\!\!+ \cdot_{\texttt{punc}} :: ys \hookrightarrow^{s^{\star}} \texttt{starRest}(b, p)$$

Proof. Induction on xs. □

Theorem C.79 (Serialization/Deserialization Round Trip). *If* $p \dagger^{s} xs$, *then* $xs \hookrightarrow^{s} p$.

Proof. Induction on $p \dagger^{s} xs$. □