The Lax Braided Structure of Streaming I/O

Alan Jeffrey[1] and Julian Rathke*[2]

1 **Alcatel–Lucent Bell Labs** ajeffrey@bell-labs.com
2 **University of Southampton** jr2@ecs.soton.ac.uk

Abstract

We investigate and implement a model of typed streaming I/O. Each type determines a language of traces analogous to regular expressions on strings, and programs are modelled by certain monotone functions on these traces. We show that sequential composition forms a lax braided monoid in the category of types and programs. This lax braided structure allows programs to be represented diagrammatically using Joyal and Street's string diagrams in 3D space. Monotone functions over traces cannot be executed efficiently, so we present an equivalent monoidal category of transducers. We demonstrate that transducers can be executed efficiently, theoretically by showing that programs with diagrams embedded in the plane can be executed in $O(1)$ space, and experimentally by an implementation in the Agda dependently typed functional language. Agda supports machine-assisted proof: we have mechanically verified that the transducer implementation and the I/O model form lax braided monoidal categories.

1998 ACM Subject Classification F.3.2 Semantics of Programming Languages, D.3.3 Language Constructs and Features

Keywords and phrases Semantics, Categorical Models, Streaming I/O, Agda I/O Library

1 Introduction

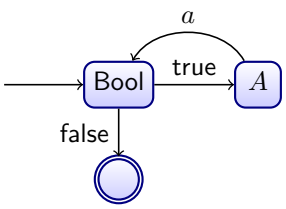
1.1 Semantics

There are many models of streaming I/O, such as Kahn's dataflow networks [22] and Milner's [26] and Hoare's [16] process models. In these models, streams are *stateless*, for example a stream might be given the type Byte^ω, and a consumer is allowed to read a Byte from such a stream at any time.

To motivate the use of *stateful* streams, consider a typical Java program which consumes a steam of data:

```
Iterator<A> stream = ...;
while (stream.hasNext()) { A a = stream.next(); ... }
```

The contract for using an `Iterator<A>` stream is that `hasNext` is called, and depending on its value the stream is either terminated, or `next` can be called, and the stream's contract is back to its initial state:



* Support for Julian Rathke provided by NSF Grant CCF-0916741.

In Java, such contracts are enforced dynamically; they are enforced statically by systems of *typestates* [9]. Typestates are modelled as automata, and so come with the usual definition of *sequential composition*. We will write $T \,\&\, U$ for the sequential composition of T and U; a typical member is given by concatenating a member of T with a member of U. There is a matching notion of sequential composition of functions on traces, and so we investigate *monoidal categories*. There has been work on formal models of computation over stateful streams, notably *session* types [17] and *games* models [4, 18]. These models emphasise the *concurrent*, rather than sequential, composition of streams, for example in games models $T \otimes U$ is modelled by interleaving.

This paper provides the first categorical model for typed streaming I/O with a monoidal structure for sequential composition. We will show that this category has *lax braided* [8] structure (§2.5), and so has a dataflow presentation as *string diagrams* in three dimensions (§2.6). Braided monoidal categories are common in mathematical physics [5]; it is surprising that they also come up in the setting of streaming.

1.2 Pragmatics

This paper grew from an attempt to provide an I/O library for the Agda [1] dependently typed functional programming language. Since Agda compiles to Haskell [3], it is possible to link against Haskell's *lazy I/O* model. Unfortunately, lazy I/O does not respect Agda's semantics. Consider:

```
hello1 []       = putStr "Hello"          hello2 xs = putStr "Hello"
hello1 (x : xs) = putStr "Hello"
```

Agda includes a mechanized proof assistant, in which it is routine to prove that `hello1` and `hello2` are extensionally equivalent. Unfortunately, executing these programs using Haskell lazy I/O (`main = getContents >>= hello*`) results in `hello1` blocking waiting for input, and `hello2` immediately printing `"Hello"`. Kiselyov [23] has proposed an alternative to lazy I/O: the *iteratee* model; Millikin [25] has written a good introduction to the topic. Iteratees are similar to transducers [24] or resumptions [15]. Similar to this, we present a streamlined process model and show that processes are equivalent to functions over traces (§3.1).

We give a characterization of the *regular* functions on streams, which can be executed in $O(1)$ space. We show (§3.2) that regular programs can be presented as *planar* dataflow diagrams. We provide an implementation of processes as an Agda library (§3.3) that links processes against the Haskell I/O library, and verify experimentally that our implementation of a simple `wc` program runs in constant space. We have used Agda to mechanically validate many of the theorems in this paper. This verification is of the I/O library implementation, not just its model, and caught some corner case buffering bugs. This is the first mechanical verification of the categorical structure of an I/O library.

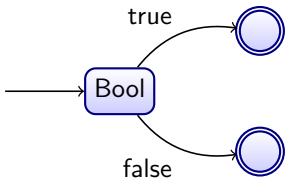
2 Semantics

2.1 Types

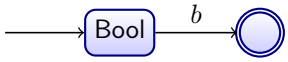
We model *stateful types* as the grammar:

$$T \overset{\nu}{::=} I \mid \Sigma(a : A)\, T_a$$

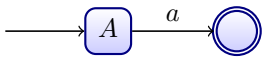
I is the *unit* type and $\Sigma(a : A)\, T_a$ is a *sum* type, where A is a set[1], and T is an A-indexed family of types. Types are defined *coinductively*, that is each type can be viewed as a (possibly infinitely deep and infinitely wide) tree, where each node is either: an I node, with no children, or a $\Sigma(A)$ node, with A-indexed children. We indicate that the grammar of types is to be interpreted coinductively by the annotation $\overset{\nu}{::=}$; we will annotate inductive grammars by $\overset{\mu}{::=}$. For example, the type: $\langle \mathsf{Bool} \rangle \overset{\mathsf{def}}{=} \Sigma(b : \mathsf{Bool})\, I$ has tree representation:



which can be viewed as the tree unfolding of the graph:

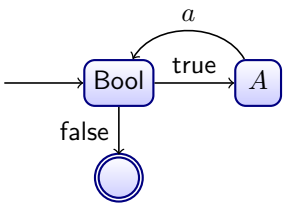


In general, the *character* type $\langle A \rangle$ is:



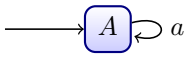
$$\langle A \rangle \overset{\mathsf{def}}{=} \Sigma(a : A)\, I$$

The *iterator* character type $\langle A \rangle^*$ is:



$$\langle A \rangle^* \overset{\mathsf{def}}{=} \Sigma(b : \mathsf{Bool})\, T_b$$
$$T_{\mathsf{true}} \overset{\mathsf{def}}{=} \Sigma(a : A)\, \langle A \rangle^*$$
$$T_{\mathsf{false}} \overset{\mathsf{def}}{=} I$$

The *stream* character type $\langle A \rangle^\omega$ is:



$$\langle A \rangle^\omega \overset{\mathsf{def}}{=} \Sigma(a : A)\, \langle A \rangle^\omega$$

As these examples show, types can be viewed as (potentially infinite state) automata, where the only acceptor is I.

▶ **Theorem 1.** *Types are in one-to-one correspondence with minimal deterministic automata where every acceptor is a sink state.*

2.2 Traces

Since types are automata, we can look at their languages. Define the transition relation on types and the language of a type:

$$\Sigma(a : A)\, T_a \xrightarrow{b} T_b \quad (b \in A) \qquad\qquad \mathcal{L}(T) = \{\, a_1 \cdots a_n \mid T \xrightarrow{a_1} \cdots \xrightarrow{a_n} I \,\}$$

[1] For readers who care about cardinality, let A range over a universe of small sets, and let the set of types be a large set.

Following [29], we will call the elements of this set *complete traces* of T. Equivalently, we can present the complete traces as a grammar:

$$t \overset{\mu}{::=} \varepsilon \mid a \cdot t$$

together with a type judgement $t : \checkmark T$:

$$\frac{}{\varepsilon : \checkmark I} \qquad \frac{a \in A \quad t : \checkmark T_a}{a \cdot t : \checkmark \Sigma(a : A)\, T_a}$$

We can also define the language of (potentially *incomplete*) *traces* as:

$$\mathcal{T}(T) = \{\, a_1 \cdots a_n \mid \exists U \,.\, T \xrightarrow{a_1} \cdots \xrightarrow{a_n} U \,\}$$

or equivalently as a type judgement $t : T$:

$$\frac{}{\varepsilon : T} \qquad \frac{a \in A \quad t : T_a}{a \cdot t : \Sigma(a : A)\, T_a}$$

We are interested in incomplete traces, because we will view programs as functions from input traces to output traces. If we only recorded complete traces, then every program has an equivalent program which blocks waiting for its input to complete.

2.3 Categories

We have discussed our model of types as languages of traces, and now consider our model of programs as functions on traces. For any function $f : \mathcal{T}(T) \to \mathcal{T}(U)$, define:

- f is *monotone* whenever $t \leq u$ implies $f(t) \leq f(u)$, where $\leq$ is prefix order, and
- f *respects completion* whenever t is complete implies $f(t)$ is complete.

Monotonicity is a standard requirement for trace models, for example [22], as it expresses that a program must commit to its output. Respecting completion is a termination property. This leads us to our first category of functions over traces. **Tr** is the category with:

- Objects are types.
- Morphisms $f : T \to U$ are monotone functions $f : \mathcal{T}(T) \to \mathcal{T}(U)$ which respect completion.
- Identity and composition are as expected.

It turns out that we will use two other conditions on functions in Sections 2.4 and 2.5. Define:

- f *reflects completion* whenever $f(t)$ is complete implies t is complete, and
- f is *strict* whenever $f(\varepsilon) = \varepsilon$.

We can then define three subcategories of **Tr**, all with the same objects:

- in **RTr**, morphisms reflect completion,
- in **STr**, morphisms are strict, and
- in **RSTr**, morphisms reflect completion and are strict.

It is routine to verify that identity and composition preserve monotonicity, respecting completion, reflecting completion, and strictness, and so form categories.

▶ **Theorem 2.** **Tr**, **RTr**, **STr** *and* **RSTr** *are categories.*

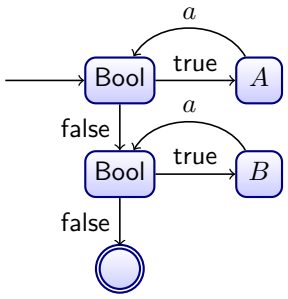
Proof. Mechanically verified [19]. ◀

2.4 Monoidal structure

Since types are automata, they come equipped with a monoidal action: *sequential composition*. We define the type $T \,\&\, U$ by tree substitution :

$$I \,\&\, U \stackrel{\text{def}}{=} U \qquad (\Sigma(a : A)\, T_a) \,\&\, U \stackrel{\text{def}}{=} \Sigma(a : A)\, (T_a \,\&\, U)$$

This is the usual definition of composition of automata, replacing any moves to the acceptor in T by a move to the initial state of U, for example:



$\langle A \rangle^* \,\&\, \langle B \rangle^*$

It is easy to check that on types, & forms a monoid with unit I. To define the action of & on morphisms, we first define concatenation of traces: given $t : T$ and $u : U$, we define $t \frown u : T \,\&\, U$. If t is complete, the definition is as expected:

$$\varepsilon \frown u \stackrel{\text{def}}{=} u \qquad\qquad (a \cdot t) \frown u \stackrel{\text{def}}{=} a \cdot (t \frown u)$$

We cannot, however, use this definition when t is incomplete, as it does not typecheck; instead we define:

$$t \frown u \stackrel{\text{def}}{=} t \text{ when } t \text{ is incomplete}$$

Given $t : T \,\&\, U$, we define $\mathsf{front}_T(t) : T$ and $\mathsf{back}_T(t) : U$ as:

$$\begin{aligned}
\mathsf{front}_I(t) &\stackrel{\text{def}}{=} \varepsilon & \mathsf{back}_I(t) &\stackrel{\text{def}}{=} t \\
\mathsf{front}_{\Sigma(a:A)\, T_a}(\varepsilon) &\stackrel{\text{def}}{=} \varepsilon & \mathsf{back}_{\Sigma(a:A)\, T_a}(\varepsilon) &\stackrel{\text{def}}{=} \varepsilon \\
\mathsf{front}_{\Sigma(a:A)\, T_a}(a \cdot t) &\stackrel{\text{def}}{=} a \cdot \mathsf{front}_{T_a}(t) & \mathsf{back}_{\Sigma(a:A)\, T_a}(a \cdot t) &\stackrel{\text{def}}{=} \mathsf{back}_{T_a}(t)
\end{aligned}$$

We will often elide the types from front and back. From concatenation and projection, we can define the action of & on morphisms. Given $f : T \to U$ and $g : T' \to U'$, define $f \,\&\, g : T \,\&\, T' \to U \,\&\, U'$ as:

$$(f \,\&\, g)(t) \stackrel{\text{def}}{=} f(\mathsf{front}(t)) \frown g(\mathsf{back}(t))$$

Unfortunately, this is *not* a monoid on **Tr**, as it is not a functor, for which we would need:

$$(f_1; g_1) \,\&\, (f_2; g_2) = (f_1 \,\&\, f_2); (g_1 \,\&\, g_2)$$

This fails for morphisms (of type $\langle \mathsf{Bool} \rangle \to \langle \mathsf{Bool} \rangle$):

$$f_1(t) \stackrel{\text{def}}{=} g_2(t) \stackrel{\text{def}}{=} t \qquad f_2(t) \stackrel{\text{def}}{=} g_1(t) \stackrel{\text{def}}{=} \mathsf{true} \cdot \varepsilon \qquad \mathit{lhs}(\varepsilon) = \mathsf{true} \cdot \mathsf{true} \cdot \varepsilon \neq \mathsf{true} \cdot \varepsilon = \mathit{rhs}(\varepsilon)$$

To show functoriality, it is sufficient to show:

- g_1 reflects completion, or
- f_1 respects completion, and f_2 is strict.

▶ **Theorem 3.** **RTr**, **STr** *and* **RSTr** *are monoidal categories.*

Proof. Mechanically verified [19]. ◀

2.5 Lax braided structure

We can define a family of morphisms:

$$\mathsf{swap} : T \,\&\, U \to U \,\&\, T \qquad \mathsf{swap}(t) \stackrel{\mathsf{def}}{=} \mathsf{back}(t) \frown \mathsf{front}(t)$$

Unfortunately, swap does *not* form a symmetry, as this would require $\mathsf{swap}(\mathsf{swap}(t)) = t$, which is only true (for non-I types) when t is complete:

$$\mathsf{swap}(\mathsf{swap}(t)) = \varepsilon \text{ when } t \text{ is incomplete}$$

The problem is that swap has to *buffer* the front of the input until the the input is complete. We will discuss this form of space usage in Section 3.2.

We have shown that swap is not a symmetry, and in fact it is it not even an isomorphism. For example, for $\mathsf{swap} : \langle \mathsf{Bool} \rangle \,\&\, U \to U \,\&\, \langle \mathsf{Bool} \rangle$ we have, for any f:

$$f(\mathsf{swap}(\mathsf{true} \cdot \varepsilon)) = f(\varepsilon) = f(\mathsf{swap}(\mathsf{false} \cdot \varepsilon))$$

and so f cannot be the inverse of swap.

Categories in which swap is an isomorphism have been studied in depth: they are *braided monoidal categories*, and have applications in mathematical physics, as surveyed, for example, by Baez and Stay [5]. Braided monoidal categories can be regarded as the categorical version of *braid groups*, where every braiding has an inverse. *Positive braid monoids* [10] drop this requirement; their categorical equivalent, *lax braided monoidal categories* have only recently been investigated by Day *et al.* [8], see Figures 2–3.

We will see that **RSTr** is a lax braided monoidal category. Unfortunately, **RTr** and **STr** are *not* lax braided, as swap is not natural, for which we need:

$$(f \,\&\, g); \mathsf{swap} = \mathsf{swap}; (g \,\&\, f)$$

To see that reflecting completion without strictness is not enough to guarantee naturality, consider:

$$f(t) \stackrel{\mathsf{def}}{=} t \qquad g(t) \stackrel{\mathsf{def}}{=} a \cdot t \qquad \mathit{lhs}(\varepsilon) = \varepsilon \neq a \cdot \varepsilon = \mathit{rhs}(\varepsilon)$$

To see that strictness without reflecting completion is not enough, consider, for any complete $t \neq \varepsilon$, incomplete $u \neq \varepsilon$ and complete s:

$$f(t) \stackrel{\mathsf{def}}{=} t \qquad g(t) \stackrel{\mathsf{def}}{=} \begin{cases} \varepsilon & \text{if } t = \varepsilon \\ s & \text{otherwise} \end{cases} \qquad \mathit{lhs}(t \frown u) = s \frown t \neq s = \mathit{rhs}(t \frown u)$$

Naturality of swap can be shown when f respects completion, and g is strict and reflects completion.

▶ **Theorem 4.** **RSTr** *is a lax braided monoidal category.*

Proof. Mechanically verified [19]. ◀

2.6 String diagrams

Braided monoidal categories have an associated graphical language of *string diagrams*, as shown by Joyal and Street [20]. String diagrams have been used in various guises for quite some time now and we refer the reader to Baez and Stay [5] or Selinger [27] up to date surveys of graphical languages.

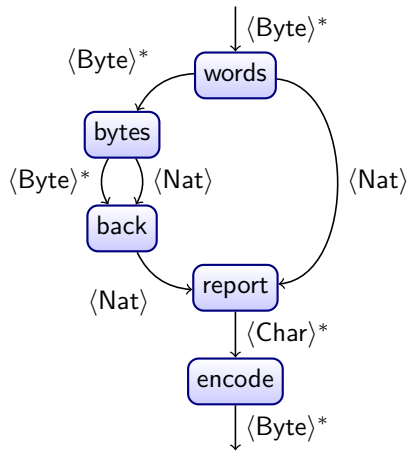


Figure 1 Example dataflow diagram.

Such diagrams are often used to represent the *dataflow* of programs, for example in Kahn networks [22]. An example dataflow program is shown in Figure 1: a simple `wc` program, which counts the number of bytes and the number of words in an input stream. (We draw diagrams flowing from top to bottom in line with Baez and Stay rather than Selinger).

The categorical structure of such dataflow diagrams is well-known: they form a symmetric monoidal category, with combinators shown in Figure 2. It is routine to verify that dataflow graphs up to isomorphism satisfy the properties given in Figures 3 and 4, which are the defining equations of a (strict) symmetric monoidal category. Joyal and Street have shown that not only are these equations sound, but they are also complete for graph isomorphism [20, Thm. 2.3], as discussed by Selinger [27, Thm. 3.12].

In a braided monoidal category, Figure 4 is replaced by Figure 5. This is accompanied by a matching change in the interpretation of diagrams; rather than graph isomorphism, we consider equivalence in three dimensional space. It is routine to verify that string diagrams up to isotopy[2] satisfy the properties given in Figures 3 and 5, which are the defining equations of a (strict) braided monoidal category. Again, Joyal and Street have shown that not only are these equations sound, but they are also complete for isotopy [20, Thm. 3.7], as discussed by Selinger [27, Thm. 3.7].

In string diagrams, swap is interpreted as a left-over-right crossing, and so has an inverse right-over-left crossing. In a lax braided monoidal category, the requirement that swap has an inverse is dropped, and we are left with the equations presented in Figure 3. This provides us with a graphical language of a lax braided monoidal category: we conjecture that this graphical language is sound and complete.

▶ Conjecture 2.1. A well-formed equation between morphisms in the language of lax braided monoidal categories follows from the axioms of lax braided monoidal categories if and only if it holds in the graphical language up to isotopy in 3 dimensions.

Proof sketch. Given Joyal and Street's results, it is enough to show that the equational theory of a braided monoidal category is a conservative extension of the equational theory

[2] More precisely, progressive isotopy of smooth string diagrams in three dimensions, see Joyal and Street [20].

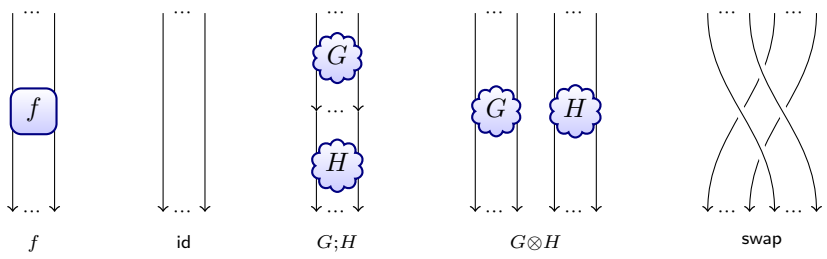


Figure 2 Combinators of a (strict) symmetric monoidal category.

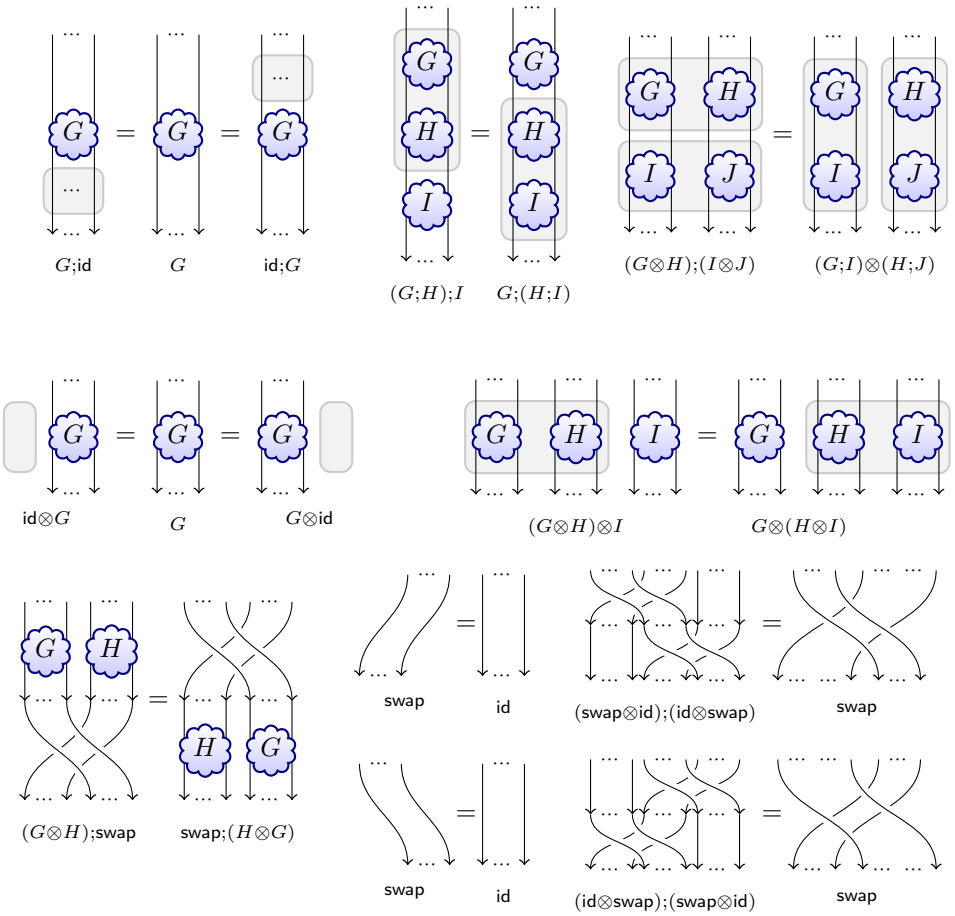


Figure 3 The equations of a (strict) lax braided monoidal category

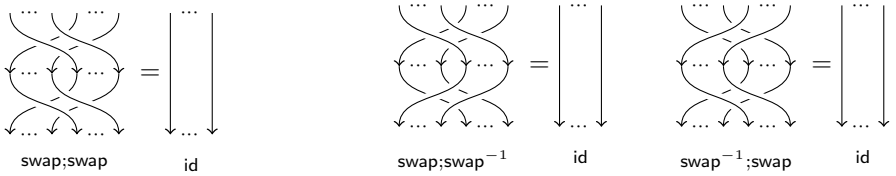


Figure 4 Symmetry

Figure 5 Braiding

of a lax braided monoidal category. That is, given any two morphisms f and g in the free lax braided monoidal category over a given signature, if $f =_B g$ then $f =_L g$ (where $=_B$ is the theory of a braid, and $=_L$ is the theory of a lax braid). In the case without generating morphisms, this problem collapses to the case of showing that the free positive braid monoid embeds into the free braid group, which was shown by Garside [10][3].

Let w range over *wiring morphisms* (that is, morphisms without generators) and p range over *planar morphisms* (that is, morphisms without swap). Define a *stratified* morphism to be one of the form: $w_0; p_1; w_1; \ldots; p_n; w_n$ such that if $p_i; w_i =_L w; f$ for some w, then $w =_L \mathsf{id}$, and if $w_i; p_{i+1} =_L p; f$ for some p, then $p =_L \mathsf{id}$. Now, if we can show that:

- every morphism can be stratified up to $=_L$,
- stratified terms are a normal form for $=_B$, and
- Conjecture 3.4 of Selinger [27],

then we can prove our conjecture. Assume morphisms f and g in the free lax braided monoidal category, such that $f =_B g$. Stratify each of them to get $f =_L w_0; f_1; w_1; \ldots; f_n; w_n$ and $g =_L v_0; g_1; v_1; \ldots; g_b; w_n$. Since these are normal forms for $=_B$, we have: $v_i =_B w_i$ and $f_i =_B g_i$. Each $v_i =_L w_i$ from Garside [10]. Since $f_i =_B g_i$, we have that f_i and g_i are isotopic as string diagrams in three dimensions, so Selinger's conjecture implies $f_i =_L g_i$. The result then follows. ◀

We leave the full proof of this conjecture as future work. If this conjecture is true, it provides a powerful, and unexpected, proof technique for equivalence of streaming programs: draw the dataflow diagrams for the programs as string diagrams in three dimensions, and check that an isotopy exists. Isotopies can often be checked "by eye", so this technique would allow many routine rewiring steps in a proof to be elided.

3 Pragmatics

3.1 Processes

One of the major drawbacks of functions on partial traces as a model of streaming I/O is that they cannot easily be executed directly. Imagine an execution of f after receiving input t: when a new input symbol a arrives, we need to know the matching output, so we have to apply f to the new history $t \cdot a$. This is inefficient in both time (potentially $O(n^2)$ rather than $O(n)$) and space (potentially $O(n)$ rather than $O(1)$). Delimited call/cc [28] would avoid this, at the cost of sophisticated language features.

Therefore, as a move towards an implementation we find it useful to introduce a syntactic representation of programs as a small language of *transducer processes*:

$$S \overset{\nu}{::=} \mathsf{inp}(a : A)\, P_a \mid \mathsf{done} \qquad P \overset{\mu}{::=} S \mid \mathsf{out}\, a\, P$$

In this definition:

- $\mathsf{inp}(a : A)\, P_a$ is an *input* process, A is a set, and P is an A-indexed family of processes,
- done is the *terminated* process, and
- $\mathsf{out}\, a\, P$ is an *output* process.

Notice that the two levels of grammar define *strict* processes S and (*lazy*) processes P. Laziness here refers to not being reliant on an input in order to generate output. Also note

[3] Thanks to Ross Street for discussions on this topic, and for pointing us to Garside's work.

the coinductive annotation on the grammar of strict processes, and the inductive annotation on P, which ensures that there are no infinite sequences of output actions.

The type rules for processes are given coinductively:

$$\frac{}{\mathsf{done} : I \to I} \qquad \frac{P_a : T_a \to U \text{ for all } a \in A}{\mathsf{inp}(a : A)\, P_a : \Sigma(a : A)\, T_a \to U} \qquad \frac{a : A \qquad P : T \to U_a}{\mathsf{out}\, a\, P : T \to \Sigma(a : A)\, U_a}$$

Note that since there are no infinite sequences of output actions, well=typed processes respect completion.

We can define[4] the operation of composition on processes $\gg$ as (in order):

$$\begin{aligned} P \gg \mathsf{out}\, a\, Q &\stackrel{\mathsf{def}}{=} \mathsf{out}\, a\, (P \gg Q) \\ \mathsf{inp}(a : A)\, P_a \gg Q &\stackrel{\mathsf{def}}{=} \mathsf{inp}(a : A)\, (P_a \gg Q) \\ \mathsf{out}\, a\, P \gg \mathsf{inp}(b : B)\, Q_b &\stackrel{\mathsf{def}}{=} P \gg Q_a \\ \mathsf{done} \gg Q &\stackrel{\mathsf{def}}{=} Q \\ P \gg \mathsf{done} &\stackrel{\mathsf{def}}{=} P \end{aligned}$$

The identity processes for this composition are:

$$\mathsf{id}_I = \mathsf{done} \qquad \mathsf{id}_{\Sigma(a:A)\, T_a} = \mathsf{inp}(a : A)\, \mathsf{out}\, a\, \mathsf{id}_{T_a}$$

This leads us to a category of processes. **Pr** has:

- Objects are types.
- Morphisms $P : T \to U$ are processes.
- Identity and composition are id and $\gg$.

We have already defined the strict processes. A process *reflects completion* when it can be typed with an additional side-condition on the type rule for input:

$$\frac{P_a : T_a \to U \text{ for all } a \in A}{\mathsf{inp}(a : A)\, P_a : \Sigma(a : A)\, T_a \to U} (U \neq I)$$

We can then define three subcategories of **Pr**, all with the same objects:

- in **RPr**, processes reflect completion,
- in **SPr**, processes are strict, and
- in **RSPr**, processes reflect completion and are strict.

We can define a sequential composition operator $P \,\&\, Q$ on typed processes as (in order):

$$\begin{aligned} P \,\&\, \mathsf{out}\, b\, Q &\stackrel{\mathsf{def}}{=} \mathsf{out}\, b\, (P \,\&\, Q) \quad \text{if } P : T \to I \\ \mathsf{done} \,\&\, Q &\stackrel{\mathsf{def}}{=} Q \\ \mathsf{inp}(a : A)\, P_a \,\&\, Q &\stackrel{\mathsf{def}}{=} \mathsf{inp}(a : A)\, (P_a \,\&\, Q) \\ \mathsf{out}\, a\, P \,\&\, Q &\stackrel{\mathsf{def}}{=} \mathsf{out}\, a\, (P \,\&\, Q) \end{aligned}$$

This definition is straightforward except for the first clause in which a process that has completed its output will not block immediate output from Q.

[4] Some care is required to ensure that this definition is well formed, since it uses a mix of induction and coinduction. This has been mechanically verified [19].

The lax braided structure on processes is given by the $\mathsf{swap}_{T,U}$ process defined (in order):

$$\begin{aligned}
\mathsf{swap}_{I,U} &\stackrel{\mathsf{def}}{=} \mathsf{id}_U \\
\mathsf{swap}_{T,I} &\stackrel{\mathsf{def}}{=} \mathsf{id}_T \\
\mathsf{swap}_{T,U} &\stackrel{\mathsf{def}}{=} \mathsf{swap}_{T,U}(\varepsilon) \\
\mathsf{swap}_{I,I}(u) &\stackrel{\mathsf{def}}{=} \mathsf{out}\, u\, \mathsf{done} \\
\mathsf{swap}_{I,\Sigma(a:A)\, U_a}(u) &\stackrel{\mathsf{def}}{=} \mathsf{inp}(a:A)\, \mathsf{out}\, a\, \mathsf{swap}_{I,U_a} \\
\mathsf{swap}_{\Sigma(a:A)\, T_a,U}(u) &\stackrel{\mathsf{def}}{=} \mathsf{inp}(a:A)\, \mathsf{swap}_{T_a,U}(u \cdot a)
\end{aligned}$$

$$\begin{aligned}
\mathsf{out}\, \varepsilon\, P &\stackrel{\mathsf{def}}{=} P \\
\mathsf{out}\, (u \cdot a)\, P &\stackrel{\mathsf{def}}{=} \mathsf{out}\, u\, \mathsf{out}\, a\, P
\end{aligned}$$

This process explicitly maintains a buffer u of actions which are output after its input has completed.

▶ **Theorem 5.**

1. **Pr** *is a category.*
2. **RPr** *and* **SPr** *are monoidal categories.*
3. **RSPr** *is a lax braided monoidal category.*

Proof. Mechanically verified [19]. ◀

In order to show that our transducer processes accurately represent our model, we show equivalences of categories. Given a morphism $P : T \to U$ in **Pr** we give a morphism $[\![P]\!] : T \to U$ in **Tr** as follows:

$$\begin{aligned}
[\![\mathsf{done}]\!](\varepsilon) &\stackrel{\mathsf{def}}{=} \varepsilon \\
[\![\mathsf{out}\, a\, P]\!](t) &\stackrel{\mathsf{def}}{=} a \cdot [\![P]\!](t) \\
[\![\mathsf{inp}(a:A)\, P_a]\!](\varepsilon) &\stackrel{\mathsf{def}}{=} \varepsilon \\
[\![\mathsf{inp}(a:A)\, P_a]\!](b \cdot t) &\stackrel{\mathsf{def}}{=} [\![P_b]\!](t)
\end{aligned}$$

On traces, define $t \xrightarrow{u} t'$ as:

$$\frac{}{t \xrightarrow{\varepsilon} t} \qquad \frac{t \xrightarrow{u} t'}{a \cdot t \xrightarrow{a \cdot u} t'}$$

that is, whenever t can be partitioned into a prefix u and suffix t'. On morphisms, define:

$$f \xrightarrow{t/u} f' \text{ whenever } s \xrightarrow{t} s' \text{ implies } f(s) \xrightarrow{u} f'(s')$$

This allows us to view morphisms as (possibly infinite state) *transducers* [24]: f responds to input t by producing output u and changing state to f'. Given a strict morphism $f : T \to U$, define the strict process $(\!|f|\!)^s_T : T \to U$ as:

$$\begin{aligned}
(\!|f|\!)^s_I &\stackrel{\mathsf{def}}{=} \mathsf{done} \\
(\!|f|\!)^s_{\Sigma(a:A)\, T_a} &\stackrel{\mathsf{def}}{=} \mathsf{inp}(a:A)\, \mathsf{out}\, t\, (\!|f'|\!)^s_T \text{ where } f \xrightarrow{a/t} f'
\end{aligned}$$

Given a morphism $f : T \to U$, define the process $(\!|f|\!)_T : T \to U$ as:

$$(\!|f|\!)_T \stackrel{\mathsf{def}}{=} \mathsf{out}\, t\, (\!|f'|\!)^s_T \text{ where } f \xrightarrow{\varepsilon/t} f'$$

We can show that $(\!|\cdot|\!)$ and $[\![\cdot]\!]$ are inverses and respect categorical structure, and so form equivalences.

▶ **Theorem 6.** $(\!|\cdot|\!)$ *and* $[\![\cdot]\!]$ *form equivalences:*

1. **Tr** *and* **Pr** *as categories.*
2. **RTr** *and* **RPr** *as monoidal categories.*
3. **STr** *and* **SPr** *as monoidal categories.*
4. **RSTr** *and* **RSPr** *as lax braided monoidal categories.*

Proof. Mechanically verified [19]. ◀

3.2 Space usage

In Section 2.5, we discussed the fact that swap introduces buffering: we will now discuss space usage more formally.

An *online algorithm* is one which can be implemented by a multi-tape Turing Machine where only rightward moves are allowed on the input and output tapes. The space usage of such an implementation is the space usage of the scratch tapes: for example the identity function is considered to be in $O(1)$ space. It is routine to show that if f and g can be implemented in $O(1)$ space, then so can $f; g$ and $f \,\&\, g$. The only constructor of a lax braided monoidal category to introduce $O(n)$ space usage is swap.

Another way to characterize the space usage is by analogy with regular trees. In our setting, we are interested in the infinite trees generated by the coinductive definition of types. A type is *regular* whenever: $\{\, T' \mid T \xrightarrow{t} T' \,\}$ is finite and a morphism is *regular* whenever: $\{\, f' \mid f \xrightarrow{t/u} f' \,\}$ is finite It is routine to see that regular types (resp. morphisms) can be implemented as deterministic finite-state automata (resp. sequential finite-state transducers [24]), and so can be executed in $O(1)$ space (provided the alphabet can be represented in $O(1)$ space).

The identity function is regular, and regularity of morphisms is preserved by $f; g$, and $f \,\&\, g$, so we can form the subcategory (resp. strict monoidal subcategory) of **Tr** (resp. **RTr**, **STr** and **RSTr**) where morphisms are regular.

We can now show that swap is irregular. Consider the type $N \stackrel{\mathsf{def}}{=} 1.N + 0.I$ whose complete traces are of the form 1^n0, and whose incomplete traces are of the form 1^n. Now, for any m and n, if (at type $N \,\&\, N$):

$$\mathsf{swap} \xrightarrow{1^m/\varepsilon} f \qquad \mathsf{swap} \xrightarrow{1^n/\varepsilon} f$$

then we must have: $01^m0 = \mathsf{swap}(1^m00) = f(00) = \mathsf{swap}(1^n00) = 01^n0$ and hence $m = n$. Thus there must be infinitely many such f, and so swap is irregular. This argument is essentially a replay of the Pumping Lemma in the setting of transducers rather than automata.

Since any plane dataflow graph can be expressed without swap [27], this gives a surprising method for showing that a function can be implemented in $O(1)$ space: check that its dataflow diagram is embedded in a plane. Moreover, if Conjecture 2.1 is true, then this means that planarity only has to hold up to isotopy in three dimensions.

▶ **Theorem 7.** *Any plane dataflow diagram whose generating morphisms can be implemented in online* $O(1)$ *space determines a morphism which can be implemented in online* $O(1)$ *space.*

3.3 Implementation

Agda [1] is a dependently typed functional programming language, which supports mechanical theorem proving. Its core language is similar to that of Coq [2], although it does not

have a separate language of tactics. It has a compiler to Haskell [3], and a foreign function interface, which allows calls out to Haskell code.

The transducer process language and type system discussed in Section 3.1 is implemented as an Agda library. It is linked against the Haskell I/O monad, and compiles to simple pipes from standard input to standard output. The implementation allows us to verify experimentally that regular programs run in constant space, for example the `wc` program from Figure 1 can word count 29Mb of data (1M lines of XML) in 120k heap, with 22k live:

```
head -1000000 h_sapiens.xml | WC +RTS -A50k -M120k -s
29294872 1339150
  16,796,509,664 bytes allocated in the heap
     477,860,144 bytes copied during GC
          22,112 bytes maximum residency (1 sample(s))
          32,512 bytes maximum slop
```

Many of the theorems in this paper have been mechanically verified, which shows correctness not just of the model **Tr**, but also of its implementation in **Pr**. This proof of correctness caught some subtle bugs, for example in the definition of $P \,\&\, Q$ there is a clause:

$$P \,\&\, \mathsf{out}\, b\, Q \stackrel{\mathsf{def}}{=} \mathsf{out}\, b\, (P \,\&\, Q) \quad \text{if } P : T \to I$$

This clause was originally not present, which causes subtle buffering errors in the corner case where P's output has completed even though its input has not. For unit testing to catch this bug, a test harness would be required which supports mocking an I/O library to allow testing with incomplete traces. Such mock libraries are difficult to construct, which in turn makes unit testing of I/O-bound programs difficult.

In summary, we have provided an executable library of streaming I/O, together with a mechanically verified proof of its categorical structure. This is the first such library.

4 Related work

Our model is based on monotone functions over traces, and so is strongly related to Kahn dataflow networks [22]. Kahn networks are for streams of type A^ω, and so do not support a notion of stream termination, or concatenation of streams. The main difference between Kahn's model and ours is that we require streams to be consumed in left-to-right order, that is a function of type $f : T \,\&\, U \to V$ must consume all of its T input before consuming any of its U input.

Games models [4, 18] have a similar structure to our model: arenas are (essentially) automata with additional structure, and strategies are (essentially) transducers. Games, however, are designed to have symmetric monoidal structure, rather than braided monoidal structure, since the tensor on types is an interleaving rather than a sequencing of automata. Games models are also compact closed, since they have a dualising action $\cdot^\perp$. This corresponds to the *bidirectional* nature of strategies in a game: a morphism $f : T \to U$ supports input on the right and output on the left, as well as the left-to-right communication allowed by our model.

Session types [17] also provide a bidirectional extension of the types considered here. We can define the first-order output-only sessions in our system as:

$$\mathsf{end} \stackrel{\mathsf{def}}{=} I \quad ![A].T \stackrel{\mathsf{def}}{=} \Sigma(a : A)\, T \quad \oplus\{\ell_i : T_i\} \stackrel{\mathsf{def}}{=} \Sigma(a : \{\ell_1, \ldots, \ell_n\})\, U_a \text{ where } U_{\ell_i} \stackrel{\mathsf{def}}{=} T_i$$

Session types *do* support a notion of terminated session, and so could support a sequential composition operator, but this has not been investigated. Moreover, much of the work on

session types has been in the context of processes rather than functions, with the exception of recent work by Gay and Vasconcelos [11]. There has been no work on categorical session models. Linear logic [14] has symmetric monoidal structure, which has been generalized to the non-commutative case (up to cyclic permutations) by Yetter [30]. Kiselyov's *iteratees* [23] provide a similar model to our transducer process model, although they also support impure iteratees and exceptions. The more fundamental difference between the models is output, which in Millikin's [25] notation is: $\mathsf{Yield}(\mathsf{Chunk}[\vec{b}])[\vec{a}]$. Here, an iteratee is being built with input type A, and output type U; it is outputting $\vec{b}$, and also returning the unconsumed input $\vec{a}$. In our terms, this is a process of type $\langle A\rangle^* \to \langle B\rangle^* \,\&\, \langle A\rangle^*$, that is iteratees are given by the state transformer construction applied to processes. A similar syntactic model for stream processing is provided by Ghani *et al.* [12], and has been generalized to arbitrary coinductive datatypes [13].

5 Future work

Cyclic graphs are modelled categorically as *traced* monoidal categories [21], as discussed by Selinger [27]. Braided traced categories are well-known, but it is not immediately obvious what the right notion of lax braided traced category is. Cyclic graphs would allow modelling of recursive dataflow programs, although there may be a requirement that only contraction maps (with respect to an appropriate metric on traces) can be made cyclic.

Games models and session types are naturally bidirectional, so there exist duals for all types; categorically games form compact closed categories. It is not obvious how to generalize the notion of autonomous category from the braided to the lax braided setting. Compact closed categories are monoidal closed, where $T \Rightarrow U$ is defined to be $T^\perp \otimes U$. This should extend to a higher order sequential model, where $T^\perp \,\&\, U$ is a type for programs which consume all their arguments before producing any results.

There is a natural improvement order on functions, given pointwise by prefix order, for example $\mathsf{swap};\mathsf{swap} \leq \mathsf{id}$. Such an improvement order would make our category a 2-category. These come with natural minimal and maximal elements, for example the the maximal natural transform of type $T \to T$ is the identity function, and the minimal natural transform is a "delay" function that returns ε on any incomplete input, and acts as the identity on complete input. Moreover, **Tr** comes with the right combinators to form a category with finite products, but the equations are only satisfied up to two-cells, for example $\langle f, g\rangle; \pi_2 \leq g$. This may be given an interesting restriction category structure [7] or form a variant of a cartesian bicategory [6].

References

1 The Agda wiki. `http://wiki.portal.chalmers.se/agda/`.
2 The Coq proof assistant. `http://coq.inria.fr/`.
3 The Haskell programming language. `http://www.haskell.org/`.
4 S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. *Information and Computation*, 163:409–470, 1996.
5 J. Baez and M. Stay. Physics, topology, logic and computation: A Rosetta Stone. In B. Coecke, editor, *New Structures for Physics*, volume 813 of *Lecture Notes in Physics*, chapter 2, pages 95–172. Springer, 2011.
6 A. Carboni and R. F. C. Walters. Cartesian bicategories I. *J. Pure and Applied Algebra*, 49:11–32, 1987.
7 J. R. B. Cockett and S. Lack. Restriction categories I: categories of partial maps.

8 B. Day, E. Panchadcharam, and R. Street. Lax braidings and the lax centre. In L. H. Kauffman, D. E. Radford, and F. J. O. Souza, editors, *Hopf Algebras and Generalizations*, volume 441 of *Contemporary Mathematics*, pages 1–17. AMS, 2007.

9 R. Deline and M. Fähndrich. Typestates for objects. In *Proc. European Conf. Object-Oriented Programming*, pages 465–490. Springer, 2004.

10 F. A. Garside. The braid group and other groups. *Q. J. Mathematics*, 20(1):235–254, 1969.

11 S. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types. *J. Functional Programming*, 20(1):19–50, 2010.

12 N. Ghani, P. Hancock, and D. Pattinson. Continuous functions on final coalgebras. In *Proc. Coalgebraic Methods in Computer Science*, Electronic Notes in Theoretical Computer Science, 2006.

13 N. Ghani, P. Hancock, and D. Pattinson. Continuous functions on final coalgebras. *Electronic Notes in Theoretical Computer Science*, 249:3–18, 2009. Proc. Mathematical Foundations of Programming Semantics.

14 J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

15 M. Hennessy and G. D. Plotkin. Full abstraction for a simple programming language. In *Proc. Mathematical Foundations of Computer Science*, number 74 in Lecture Notes in Computer Science, pages 108–120. Springer, 1979.

16 C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

17 K. Honda. Types for dyadic interaction. In *Proc. Int. Conf. Concurrency Theory*, number 715 in Lecture Notes in Computer Science, pages 509–523. Springer, 1993.

18 J. M. E. Hyland and C.-H. Ong. On full abstraction for PCF. *Information and Computation*, 163:285–408, 2000.

19 A. S. A. Jeffrey. agda-system-io. `https://github.com/agda/agda-system-io/`, 2010.

20 A. Joyal and R. Street. The geometry of tensor calculus I. *Advances in Mathematics*, 88(1):55–112, 1991.

21 A. Joyal, R. Street, and D. Verity. Traced monoidal categories. *Math. Proc. Cambridge Philosophical Society*, 119:447–468, 1996.

22 G. Kahn. The semantics of a simple language for parallel programming. In *Proc. IFIP Congress*, pages 471–475, 1974.

23 O. Kiselyov. Streams and iteratees. `http://okmij.org/ftp/Streams.html`.

24 M. Lothaire. *Applied Combinatorics on Words*, volume 105 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 2005.

25 J. Millikin. Understanding iteratees. `http://john-millikin.com/articles/understanding-iteratees/`, 2010.

26 R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

27 P. Selinger. A survey of graphical languages for monoidal categories. In B. Coecke, editor, *New Structures for Physics*, volume 813 of *Lecture Notes in Physics*, chapter 4, pages 289–356. Springer, 2011.

28 D. Sitaram and M. Felleisen. Control delimiters and their hierarchies. *Lisp and Symbolic Computation*, 3(1):67–99, 1990.

29 R. J. van Glabbeek. The linear time – branching time spectrum. In *Proc. Int. Conf. Concurrency Theory*, volume 458 of *Lecture Notes in Computer Science*, pages 278–297. Springer, 1990.

30 D. Yetter. Quantales and (noncommutative) linear logic. *Journal of Symbolic Logic*, 55(1):41–64, 1990.