# Hyperfunctions: Communicating Continuations

DONNACHA OISÍN KIDNEY, Imperial College London, United Kingdom
NICOLAS WU, Imperial College London, United Kingdom

A hyperfunction is a continuation-like construction that can be used to implement communication in the context of concurrency. Though it has been reinvented many times, it remains somewhat obscure: since its definition by Launchbury et al. [2000], hyperfunctions have been used to implement certain algebraic effect handlers [Kammar et al. 2013], coroutines [Spivey 2017], and breadth-first traversals [Berger et al. 2019]; however, in each of these examples, the hyperfunction type went unrecognised.

We identify the hyperfunctions hidden in all of these algorithms, and we exposit the common pattern between them, building a framework for working with and reasoning about hyperfunctions. We use this framework to solve a long-standing problem: giving a fully-abstract continuation-based semantics for a concurrent calculus, the Calculus of Communicating Systems. Finally, we use hyperfunctions to build a monadic Haskell library for efficient first-class coroutines.

**ACM Reference Format:**
Donnacha Oisín Kidney and Nicolas Wu. 2025. Hyperfunctions: Communicating Continuations. 1, 1 (November 2025), 37 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 Introduction

While continuations and concurrency have a long and happy history together [Haynes et al. 1986; Hieb and Dybvig 1990; Todoran 2000], occasionally the combination of these two patterns can result in complex and intricate programs that resist comprehension. As is often the case in partnerships, we think that the crux of the problem lies with communication: in particular, communication between continuations. This paper is interested in *hyperfunctions* [Launchbury et al. 2000], a type of continuation with a rich algebraic structure that facilitates communication.

Perhaps the best example of the problems that arise when continuations tangle with concurrency comes from the field of program semantics. There, despite the widespread use of continuations, it has proved difficult to find a continuation-based semantics for concurrent languages like the Calculus of Communicating Systems (CCS) [Milner et al. 1980] and other process calculi.

> Although continuation-passing style is sometimes regarded as a standard style to use for denotational semantics, it is inadequate for describing languages that involve non-determinism or concurrent processes. [Mosses 2010]

Though Ciobanu and Todoran have made significant progress on this problem [2018], there is currently *no* fully-abstract continuation-based model for a concurrent language like CCS. However, as we will show, hyperfunctions provide the principles to solve this long-standing problem.

Communicating continuations show up outside of program semantics, also. Coroutines, for example, are a general control abstraction where communication plays a fundamental role; in continuation-based implementations [Haynes et al. 1986; Shivers and Might 2006; Spivey 2017] this

Authors' Contact Information: Donnacha Oisín Kidney, Imperial College London, London, United Kingdom, o.kidney21@imperial.ac.uk; Nicolas Wu, Imperial College London, London, United Kingdom, n.wu@imperial.ac.uk.

ACM XXXX-XXXX/2025/11-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

communication becomes much more difficult to implement. A similar problem can show up even in simple list algorithms like *zip* or *interleave*: when lists are represented with continuations [e.g. Gill et al. 1993] the merging of two lists becomes communication between parallel processes [Launchbury et al. 2000]. Perhaps surprisingly, hyperfunctions encapsulate a pattern common to all of these problems, and they provide a formalism for building algorithms to solve them.

On our way to proving full abstraction for CCS, we will take a tour through the literature, spotting unrecognised hyperfunctions in the wild; from Hofmann's algorithm for breadth-first traversal in 1993, through Shivers and Might's transducers in 2006 and Kammar et al.'s handlers of algebraic effects in 2013, up to Spivey's coroutine pipelines in 2017. Along the way, we will build a toolbox for working with hyperfunctions, and a framework for reasoning about them. All of this will equip us to define our eventual model for CCS. Finally, we will look at some novel uses for hyperfunctions in real, practical applications: first in optimising some Haskell libraries, and finally in building a monadic library for first-class asymmetric coroutines backed by continuations.

## Contributions

- We identify and catalogue a number of appearances of hyperfunctions in the literature, including Hofmann [1993]; Kammar et al. [2013]; Shivers and Might [2006]; Spivey [2017]. To the best of our knowledge, this is the first work to connect these appearances to the hyperfunction definition of Launchbury et al. [2000].
- We describe how hyperfunctions behave through a handful of examples of using hyperfunctions to solve simple programming problems (Section 2).
- We characterise the expressive power of hyperfunctions, by showing that they can form a fully-abstract model (which we call the Communicator model) for the Calculus of Communicating Systems (Section 3), thereby showing that hyperfunctions are capable of expressing at least the model of concurrency captured by CCS.
- We use hyperfunctions to implement monadic concurrency constructions, including LogicT for backtracking [Kiselyov et al. 2005] and Claessen's concurrency monad [1999] (Section 4).
- Finally, we demonstrate that hyperfunctions underlie certain optimisations to coroutine libraries [Gonzalez 2012; Spivey 2017], and we use this understanding to implement a new Haskell library for asymmetric coroutines which allows for first-class transfer of control, and solve the stable marriage problem using this library (Section 5).

One common feature among the works that have rediscovered hyperfunctions is that the authors often comment on how difficult it was to figure out the hyperfunction-like structure they needed. So, while the scientific and technical contribution of this paper is in its study of hyperfunctions and in the development of a new model for CCS, we hope that the broader impact will be in saving future programmers from having to reinvent this tricky type on their own.

## 2 Basic Hyperfunctions

Let's start by actually defining the hyperfunction type itself. A hyperfunction of type $a \looparrowright b$ is an infinitely left-nested function of the following form:

$$a \looparrowright b = (((\ldots \to a) \to b) \to a) \to b$$

Cardinality restrictions prevent this type from having a set-theoretic interpretation. It *does* have a domain-theoretic interpretation, however (as the solution to $X \cong (X \Rightarrow A) \Rightarrow B$), as explained by Krstić et al., who also show how to interpret hyperfunctions as final coalgebras [2001a; 2001b].

For now, though, we won't concern ourselves with the details of the foundational setting of the hyperfunction type (although we will return to the question in Remark 3.8). Happily, most programming languages do not impose strict cardinality restrictions on type definitions: as a result, hyperfunctions can be defined as a simple (but strange) recursive type. In Haskell:

$$\textbf{newtype}\ a \looparrowright b = \mathsf{Hyp}\ \{\iota :: (b \looparrowright a) \to b\} \tag{1}$$

In isolation, this definition can be a little perplexing: however, it is possible to build an understanding of this type by *using* it to implement concrete algorithms. Over the next few pages we will do just that, using hyperfunctions to implement functions on church-encoded numbers, *zip* on lists, and breadth-first traversal. Each of these examples will reveal some capability of the type; by the end of this section we will have enough tools to attack the problem of modelling CCS.

**Code.** This paper uses code examples in Haskell throughout. We do not, however, use any special features unique to the language; the algorithms we present can be translated to any general-purpose language with higher-order functions. One caveat for strict-by-default languages is that the hyperfunction type must be encoded as a *lazy* function ($a \looparrowright b := (() \to b \looparrowright a) \to b$).

In addition to the Haskell code, we have also mechanised the proofs in Section 3 using Agda [Norell 2009]. This mechanisation is explained in more detail in Remark 3.17.

A brief note on syntax: we will use *copatterns* [Abel et al. 2013] to define hyperfunctions. A copattern is a way to define an instance of a record type by defining each of its fields, instead of using a constructor. The following two code snippets define the constant hyperfunction $k$, where $k\ x$ is a hyperfunction that always returns $x$.

$$\begin{array}{ll} k :: b \to (a \looparrowright b) & \qquad k :: b \to (a \looparrowright b) \\ k\ x = \mathsf{Hyp}\ \{\iota = \lambda\_ \to x\} & \qquad \iota\ (k\ x)\ \_ = x \end{array}$$

The snippet on the left uses Haskell's record syntax, the version on the right uses copatterns.

## 2.1 Church Encoding

As we will see shortly, hyperfunctions tend to show up to solve problems that arise when working with Church encodings. Church encoding is a way to encode inductive data types using only functions; it is occasionally used for optimisation. Let's quickly refresh our memory on Church encoding, starting with the natural numbers, here encoded in the standard (unary) inductive way.

$$\textbf{data}\ \mathbb{N} = \mathsf{Z} \mid \mathsf{S}\ \mathbb{N}$$

The fundamental function for processing this type is its *fold*:

$$\begin{array}{ll} \mathit{fold} :: \mathbb{N} \to (a \to a) \to a \to a & \\ \mathit{fold}\ (\mathsf{S}\ n)\ s\ \ z = s\ (\mathit{fold}\ n\ s\ z) & \qquad \mathit{fold}\ n\ s\ z\ =\ (\underbrace{s \circ \cdots \circ s}_{n})\ z \\ \mathit{fold}\ \mathsf{Z}\qquad \_\ z = z & \end{array}$$

For some $n : \mathbb{N}$, *fold* $n\ s\ z$ applies the function $s$ to $z$ $n$ times. For instance, *fold* $3\ s\ z = s\ (s\ (s\ z))$.

The Church encoding of the naturals (given below as the type $\mathcal{N}$, which also has a constructor named $\mathcal{N}$) is effectively the partial application of this *fold* function.

$$\begin{array}{ll} \textbf{newtype}\ \mathcal{N} = & \qquad \mathit{church} :: \mathbb{N} \to \mathcal{N} \\ \quad \mathcal{N}\ \{\mathit{nat} :: \forall a.(a \to a) \to a \to a\} & \qquad \mathit{church}\ n = \mathcal{N}\ (\mathit{fold}\ n) \end{array}$$

Often Church encoding is used as an optimisation technique. Church-encoded lists, for instance, underpin GHC's list fusion machinery [Gill et al. 1993; Harper 2011; Hinze et al. 2011]. Here is an example of how Church encoding can improve the performance of addition on $\mathbb{N}$ and $\mathcal{N}$:

$$\begin{array}{ll} \mathsf{Z}\quad + m = m & \qquad n + m = \\ \mathsf{S}\ n + m = \mathsf{S}\ (n + m) & \qquad\quad \mathcal{N}\ (\lambda s\ z \to \mathit{nat}\ n\ s\ (\mathit{nat}\ m\ s\ z)) \end{array}$$

Because addition on $\mathbb{N}$ always destructs and then reconstructs the left-hand argument, left-nested sums $(((\ldots + x) + y) + z)$ will evaluate in quadratic time. On $\mathcal{N}$, in contrast, + is always linear, regardless of whether it's left- or right-nested. This removal of intermediate data structures—*deforestation*—is one of the chief benefits of church encodings; *foldr*, *map*, and ++ on lists can benefit in much the same way that addition benefited above.

However, not every function adapts easily to a Church encoded variant. The predecessor function (*pred*), for instance, is infamously tricky to write, and asymptotically slow: *pred* on $\mathcal{N}$ is $O(n)$. It seems to suffer from the problem that Church encoding solved on addition: it has to traverse all of its input and then rebuild it to produce a result.

This pattern of performance suggests that there is some class of functions that work well on Church encodings: addition, ++, and *foldr*; and there is another class that does not benefit from Church encoding: *pred*, *tail*, etc. We are interested in a third class of functions which we will call *lateral* functions. Lateral functions are things like subtraction, comparison, and zipping; they process multiple structures in parallel, and they *seem* like they should be pathological cases for Church encoding (subtraction, after all, is just iterative application of *pred*). There *is* a technique to implement these functions efficiently, however, and it uses hyperfunctions as the core unit of computation. Over the rest of this section, we will explore this technique, and we will build a language of hyperfunctions that will enable the more complex examples in the rest of the paper.

## 2.2 Lateral Church Encoding

A simple example of a lateral function is $\leqslant$. On $\mathbb{N}$ it has the following implementation:

$$(\leqslant) :: \mathbb{N} \to \mathbb{N} \to \mathsf{Bool} \qquad \begin{aligned} \mathsf{S}\ n \leqslant \mathsf{S}\ m &= n \leqslant m \\ \mathsf{S}\ n \leqslant \mathsf{Z}\ \ &= \mathsf{False} \end{aligned} \qquad \mathsf{Z} \leqslant m = \mathsf{True}$$

The recursive call takes the subterm of *both* of the inputs. This is what makes Church encoding the function difficult: while we can fold over one of the arguments, as is shown below, it is difficult to see how we might fold over both.

$$\begin{array}{lll} n \leqslant m = \mathit{fold}\ n\ \mathit{ns}\ \mathit{nz}\ m & \mathit{ns} :: (\mathbb{N} \to \mathsf{Bool}) \to \mathbb{N} \to \mathsf{Bool} & \mathit{nz} :: \mathbb{N} \to \mathsf{Bool} \\ \quad \textbf{where} & \mathit{ns}\ \mathit{nk}\ (\mathsf{S}\ m) = \mathit{nk}\ m & \mathit{nz}\ m = \mathsf{True} \\ & \mathit{ns}\ \mathit{nk}\ \mathsf{Z} \qquad = \mathsf{False} & \end{array}$$

Notice that we can derive definitions like the above mechanically: the S case is replaced by the *ns* function, and the Z case by *nz*.

We can try to proceed by applying the same transformation to the *ns* function:

$$\begin{array}{ll} \mathit{ns}\ \mathit{nk} = \mathit{fold}\ m\ \mathit{ms}\ \mathit{mz}\ \mathit{nk} & \mathit{ms}\ \mathit{mk}\ \mathit{nk} = \mathit{nk}\ \mathit{mk} \\ \quad \textbf{where} & \mathit{mz}\qquad \mathit{nk} = \mathsf{False} \end{array}$$

But the *ms* case doesn't work. We can't apply *nk mk*, because *nk* expects an $\mathbb{N}$, not the fold structure built by *ms*. We need to rewrite the fold on *n* to receive a fold on *m*.

$$\begin{array}{lll} n \leqslant m = \mathit{fold}\ n\ \mathit{ns}\ \mathit{nz}\ (\mathit{fold}\ m\ \mathit{ms}\ \mathit{mz}) & \mathit{ns}\ \mathit{nk}\ \mathit{mk} = \mathit{mk}\ \mathit{nk} & \mathit{ms}\ \mathit{mk}\ \mathit{nk} = \mathit{nk}\ \mathit{mk} \\ \quad \textbf{where} & \mathit{nz}\qquad \mathit{mk} = \mathsf{True} & \mathit{mz}\qquad \mathit{nk} = \mathsf{False} \end{array}$$

The insight here is that we treat each fold as a coroutine. The fold on *n* checks if its input is Z, returning True if so (the *nz* function), otherwise it transfers control to the fold on *m*, named *mk*.

Ignoring types for a moment, this function does compute. But, of course, this is Haskell: we can't ignore the types. Plug the above function into GHC and you will receive the following complaint:

```
Could not construct infinite type t ~ (t -> Bool) -> Bool
```

This is a similar error to the one you will encounter if you try to write the Y-combinator in Haskell (without newtypes). While the function is correct in an untyped world, Haskell's type system cannot unify the types `t` and `(t -> Bool) -> Bool`. But while GHC can't construct a type that satisfies that equation, *we* can. It is, in fact, a hyperfunction: `t` above is inhabited by Bool $\looparrowright$ Bool.

$$\begin{array}{lll} n \leqslant m = \iota\ (nat\ n\ ns\ nz) & \iota\ (ns\ nk)\ \ mk = \iota\ mk\ nk & \iota\ (ms\ mk)\ \ nk = \iota\ nk\ mk \\ \quad (nat\ m\ ms\ mz)\ \textbf{where} & \iota\ nz \qquad mk = \mathsf{True} & \iota\ mz \qquad nk = \mathsf{False} \end{array}$$

This implementation follows the recursion pattern of the direct-style $\leqslant$ exactly: as a result, we know that it has the same asymptotic performance. This is a well-typed, linear implementation of $\leqslant$ on Church-encoded naturals, using hyperfunctions.

## 2.3 Hyperfunctions as Streams

In this example we will implement another lateral function: subtraction. We will also introduce another concept here that can aid in reasoning about hyperfunctions: the *stream model* [Launchbury et al. 2013]. While hyperfunctions themselves are just functions of a particular form, it can be difficult to build a mental model for how they behave, especially when they are deeply nested and intricately combined. However, it is possible to visualise hyperfunctions as *streams*, which we have found to be much easier to reason about.

The stream model treats a hyperfunction of type $a \looparrowright b$ as a stream of functions of type $a \to b$.

$$\textbf{data}\ \mathsf{Stream}\ a = a \triangleleft \mathsf{Stream}\ a \qquad\qquad (a \looparrowright b) \approx \mathsf{Stream}\ (a \to b)$$

Think of the original hyperfunction type (Eq. (1)) as the low-level implementation, and the stream version above as a high-level mental model. Note that this model is an *approximation*, not a one-to-one representation. Many hyperfunctions are not streams, and so there are many situations when the correspondence between the two representations breaks down.

However, when we confine ourselves to using only the interface below (Eqs. (2) to (4)), the behaviour of the two representations is indistinguishable. In particular, all equalities on the stream model will hold on the hyperfunction model as well. In this way, we can write code with the stream model in mind, and have it "compile" to the continuation model of Eq. (1).

The interface in question consists of three combinators: $\triangleleft$, which pushes a function onto a stream; $\odot$, which zips two streams together; and *run*, which collapses a stream into a single value (these combinators were present in Launchbury et al.'s original work on hyperfunctions [2000]).

$$\begin{array}{lll} (\triangleleft)\ ::\ (a \to b) \to (a \looparrowright b) & (\odot)\ ::\ (b \looparrowright c) \to (a \looparrowright b) & run :: a \looparrowright a \to a \\ \quad \to (a \looparrowright b) & \quad \to (a \looparrowright c) & \end{array} \tag{2}$$

The $\triangleleft$ function is the stream constructor, so the expression $f \triangleleft g \triangleleft h \triangleleft \ldots$ constructs a stream with $f$ at the head, followed by $g$, then $h$, and so on. The semantics of $\odot$ (zipping) and *run* are as follows:

$$(f \triangleleft fs) \odot (g \triangleleft gs) = (f \circ g) \triangleleft (fs \odot gs) \quad (3) \qquad\qquad run\ (f \triangleleft fs) = f\ (run\ fs) \quad (4)$$

With this small toolbox of functions, we can build algorithms and prove things about them. For instance, *rep* lifts a function $a \to b$ into a hyperfunction $a \looparrowright b$. Using Eqs. (3) and (4) we can show that *rep* is homomorphic through $\circ$ and $\odot$.

$$\begin{array}{l} rep :: (a \to b) \to a \looparrowright b \\ rep\ ab = ab \triangleleft rep\ ab \end{array} \quad (5) \qquad \begin{array}{ll} rep\ f \odot rep\ g & \\ \quad \equiv (f \triangleleft rep\ f) \odot (g \triangleleft rep\ g) & \{\text{Eq. (5)}\} \\ \quad \equiv (f \circ g) \triangleleft (rep\ f \odot rep\ g) & \{\text{Eq. (3)}\} \\ \quad \equiv rep\ (f \circ g) & \end{array}$$

Let's now look at subtraction. To implement $n - m$, our strategy will be to convert both $n$ and $m$ to hyperfunctions, zip them together using $\odot$, and then run the result to get the answer. Our

$$
\begin{aligned}
n - m &= \mathcal{N}\ \lambda\ s\ z \longrightarrow \mathit{run}\ (\mathit{nat}\ n\ (\mathit{id}\ \triangleleft)\ (\mathit{rep}\ (\mathit{const}\ z)) \odot \mathit{nat}\ m\ (\mathit{id}\ \triangleleft)\ (\mathit{rep}\ s)) \\
\{\ \mathit{nat}\ n, m\ \} &= \mathcal{N}\ \lambda\ s\ z \longrightarrow \mathit{run}\ ((\overbrace{\mathit{id}\ \triangleleft \cdots \triangleleft\ \mathit{id}\ \triangleleft\ \mathit{id}\ \triangleleft\ \mathit{id}}^{n} \triangleleft\ \mathit{const}\ z\ \triangleleft \cdots) \\
&\qquad\qquad\qquad\qquad \odot\ (\underbrace{\mathit{id}\ \triangleleft \cdots \triangleleft\ \mathit{id}}_{m}\ \triangleleft\ s\ \triangleleft\ s\ \triangleleft\ s\ \triangleleft\ s\ \triangleleft \cdots)) \\
\{\ \text{Apply} \odot\ \} &= \mathcal{N}\ \lambda\ s\ z \longrightarrow \mathit{run}\ (\overbrace{\underbrace{\mathit{id} \circ \mathit{id}\ \triangleleft \cdots \triangleleft\ \mathit{id} \circ \mathit{id}}_{m}\ \triangleleft\ \underbrace{\mathit{id} \circ s\ \triangleleft \cdots \triangleleft\ \mathit{id} \circ s}_{n-m}}^{n}\ \triangleleft\ \mathit{const}\ z \circ s\ \triangleleft \cdots) \\
\{\ \text{Apply all} \circ\ \} &= \mathcal{N}\ \lambda\ s\ z \longrightarrow \mathit{run}\ (\overbrace{\underbrace{\mathit{id}\ \triangleleft \cdots \triangleleft\ \mathit{id}}_{m}\ \triangleleft\ \underbrace{s\ \triangleleft \cdots \triangleleft\ s}_{n-m}}^{n}\ \triangleleft\ \mathit{const}\ z\ \triangleleft \cdots) \\
\{\ \text{Apply } \mathit{run}\ \} &= \mathcal{N}\ \lambda\ s\ z \longrightarrow \overbrace{\underbrace{\mathit{id}\ (\ldots (\mathit{id}}_{m}\ \underbrace{(s\ (\ldots (s}_{n-m}}^{n}\ (\mathit{const}\ z\ (\ldots))))))) \\
\{\ \text{Apply } \mathit{id}, \mathit{const}\ \} &= \mathcal{N}\ \lambda\ s\ z \longrightarrow \underbrace{s\ (\ldots (s}_{n-m}\ z))
\end{aligned}
$$

Fig. 1. Derivation of Subtraction

implementation returns 0 when $n < m$, but we assume that $m \leqslant n$ for the rest of this explanation for simplicity's sake. The implementation is given below, and diagrammed in Fig. 1.

$$n - m = \mathcal{N}\ (\lambda s\ z \longrightarrow \mathit{run}\ (\mathit{nat}\ n\ (\mathit{id} \triangleleft)\ (\mathit{rep}\ (\mathit{const}\ z)) \odot \mathit{nat}\ m\ (\mathit{id} \triangleleft)\ (\mathit{rep}\ s)))$$

$n$ is converted into a stream of functions that starts with $n$ *id*s, followed by infinitely many *const* $z$s, and $m$ is converted to a stream starting with $m$ *id*s, followed by infinitely many $s$s.

When zipped together, the resulting stream starts by drawing the *id*s from both $n$ and $m$'s streams. Then, at the $m$th entry in the stream, the *ids* from $m$ run out, and the stream switches to $\mathit{id} \circ s$. At the $n$th entry in the stream, the *id*s from $n$ run out, and the stream switches to $\mathit{const}\ z \circ s$.

Our stream is now $m$ *ids*, followed by $n - m$ $s$s, followed by infinitely many *const* $z$s. When we *run* the stream, we discard the *ids* and anything after the first *const* $z$ (since $\mathit{const}\ z\ (\mathit{const}\ z\ \ldots) = z$), leaving behind $n - m$ applications of $s$s applied to $z$. Subtraction is done!

Let's now leave the stream model, and return to the continuation-based model from Eq. (1). We swap out the implementations of $\triangleleft$, $\odot$, and *run* for the following:

$$
\begin{aligned}
&(\triangleleft) :: (a \longrightarrow b) \longrightarrow \\
&\quad (a \looparrowright b) \longrightarrow (a \looparrowright b) \quad (6) \\
&\iota\ (f \triangleleft h)\ k = f\ (\iota\ k\ h)
\end{aligned}
\qquad
\begin{aligned}
&(\odot) :: (b \looparrowright c) \longrightarrow \\
&\quad (a \looparrowright b) \longrightarrow (a \looparrowright c) \quad (7) \\
&\iota\ (f \odot g)\ h = \iota\ f\ (g \odot h)
\end{aligned}
\qquad
\begin{aligned}
&\mathit{run} :: a \looparrowright a \longrightarrow a \\
&\mathit{run}\ h = \iota\ h\ (\mathsf{Hyp}\ \mathit{run}) \\
&\qquad\qquad\qquad\quad (8)
\end{aligned}
$$

As promised, the implementation of subtraction above still works, with all equalities preserved.

One final point to make is that for the stream model, these three combinators seem to be the most "primitive" operations, from which other operations are derived. On the $\looparrowright$ type, however, the primitive operation is $\iota$. We can relate this operation to the stream model via the following identity:

$$\iota\ f\ g = \mathit{run}\ (f \odot g) \tag{9}$$

## 2.4 Message Passing

The original motivation for hyperfunctions, and perhaps their most well-known use, is in implementing *zip* [Launchbury et al. 2000]. An important optimisation in Haskell is *foldr*-fusion [Gill et al. 1993], which uses a continuation-based encoding of lists to eliminate intermediate data structures in list-processing code. Gill et al. demonstrated how to apply this optimisation to a library of standard list functions (*map*, *filter*, *sum*, etc.); however, *zip* proved to be more difficult. This is because, like subtraction, *zip* is a lateral function, which processes two structures in parallel. Launchbury et al. [2000] were the first to figure out how to apply *foldr*-fusion to *zip*, using hyperfunctions.

Zipping employs an additional feature of hyperfunctions that we have not yet seen: message passing. To explain this feature, we will model hyperfunctions as processes that can communicate.

We will treat a hyperfunction $a \looparrowright a$ as a kind of process with some result domain $a$. In this context, the *run* function runs the process, extracting the final result, and $f \triangleleft P$ prefixes a process $P$ with some action $f :: a \to a$. $\odot$ performs a parallel merge of processes.

Adding a parameter $i$ to the domain of $a \looparrowright a$ gives a process which takes an $i$ as input at every step; $(a, i) \looparrowright a$. We can curry this type to arrive at $a \looparrowright (i \to a)$, which we call a Consumer.

$$\textbf{type}\ \mathsf{Consumer}\ i\ a = a \looparrowright (i \to a) \tag{10}$$

The *cons* function prefixes a process with an action $a \to a$ that can rely on some input $i$.

$$\begin{aligned}&\mathit{cons} :: (i \to a \to a) \to \mathsf{Consumer}\ i\ a \to \mathsf{Consumer}\ i\ a\\ &\iota\ (\mathit{cons}\ f\ p)\ q\ i = f\ i\ (\iota\ q\ p)\end{aligned}$$

The inverse of a consumer is a producer; we derive it simply by flipping the hyperfunction arrow.

$$\textbf{type}\ \mathsf{Producer}\ o\ a = (o \to a) \looparrowright a \qquad \begin{aligned}&\mathit{prod} :: o \to \mathsf{Producer}\ o\ a \to \mathsf{Producer}\ o\ a\\ &\iota\ (\mathit{prod}\ o\ p)\ q = \iota\ q\ p\ o\end{aligned} \tag{11}$$

Finally, a pair of a producer and consumer can be run together with $\iota$.

$$\iota :: \mathsf{Producer}\ m\ a \to \mathsf{Consumer}\ m\ a \to a$$

We will use this model of hyperfunctions to implement *zip* with folds on lists. To zip two lists, *xs* and *ys*, we convert *xs* to a producer and *ys* to a consumer, and run both of them together with $\iota$.

$$\begin{aligned}&\mathit{zip} :: [\,a\,] \to [\,b\,] \to [\,(a, b)\,]\\ &\mathit{zip}\ \mathit{xs}\ \mathit{ys} = \iota\ (\mathit{foldr}\ \mathit{xf}\ \mathit{xb}\ \mathit{xs})\ (\mathit{foldr}\ \mathit{yf}\ \mathit{yb}\ \mathit{ys})\end{aligned}$$

The conversion of *xs* is simple: on an empty list (*xb*), we return a process which ignores its input and returns an empty list. On a non-empty list (*xf*), we produce one item: the head of the list.

$$\begin{aligned}&\mathit{xf} :: a \to \mathsf{Producer}\ a\ [\,(a, b)\,]\\ &\qquad\to \mathsf{Producer}\ a\ [\,(a, b)\,]\\ &\mathit{xf}\ x\ \mathit{xk} = \mathit{prod}\ x\ \mathit{xk}\end{aligned} \qquad \begin{aligned}&\mathit{xb} :: \mathsf{Producer}\ a\ [\,(a, b)\,]\\ &\iota\ \mathit{xb}\ \_ = [\,]\end{aligned}$$

On *ys*, the conversion is slightly more complex. In the empty case (*yb*), we also just return an empty list. However, in the non-empty case (*yf*), we consume one message, using the *cons* function. This message is the $x$, sent from *xf*: we pair it up with the $y$ we have, and cons it on to the output.

$$\begin{aligned}&\mathit{yf} :: b \to \mathsf{Consumer}\ a\ [\,(a, b)\,]\\ &\qquad\to \mathsf{Consumer}\ a\ [\,(a, b)\,]\\ &\mathit{yf}\ y\ \mathit{yk} = \mathit{cons}\ (\lambda x\ \mathit{xys} \to (x, y) : \mathit{xys})\ \mathit{yk}\end{aligned} \qquad \begin{aligned}&\mathit{yb} :: \mathsf{Consumer}\ a\ [\,(a, b)\,]\\ &\iota\ \mathit{yb}\ \_\ \_ = [\,]\end{aligned}$$

This defines *zip* on lists, entirely with folds, and without any performance penalty.

The Producer and Consumer types are not just useful for implementing *zip*: the pattern displayed here, of passing messages between continuations executed in lock-step, shows up repeatedly in implementations of coroutines (where "coroutine" here refers to structures like the kind defined in Gonzalez's Pipes library [2012]). We will discuss this in more detail in Section 5.1, but for now, we will note that types almost identical to the Producer and Consumer types (save for some rearranging of parameters) appear in both Spivey's optimised implementation of coroutines [2017], and in Kammar et al.'s deep handlers for coroutines [2013].

### 2.5 Breadth-First Traversals

The first occurrence of a hyperfunction-like type we were able to find is in an email to the TYPES mailing list [1993], where Hofmann uses the following type to implement breadth-first traversal.

$$\textbf{data}\ \mathsf{Rou}\ a\ b = \mathsf{Over} \mid \mathsf{Next}\ ((\mathsf{Rou}\ a\ b \to a) \to b) \tag{12}$$

This type differs from the hyperfunction type we have above in two ways: first, it unfolds the recursive definition by one step, making the type *regular* (i.e. its parameters don't change in the recursive occurrence); secondly, Rou includes the Over constructor, which is used in Hofmann's algorithm to signify termination of the traversal.

Without the Over constructor, it becomes necessary to pass an extra parameter around to track recursion depth. This technique can be seen clearly in Allison's implementation [1989] of breadth-first traversal (or Smith's translation of those ideas to Haskell [2009]); both of these works develop algorithms quite similar to Hofmann's, though they don't quite arrive at the hyperfunction type.

Notwithstanding the extra constructor, the structure of Hofmann's algorithm shares some elements with the implementation of *zip* above (Section 2.4). While we won't present Hofmann's original algorithm here, we will say that it works by building a hyperfunction for each path into the tree, and then zipping those hyperfunctions together. The hyperfunction structure handles the separation of levels; as a result, the final algorithm resembles the level-wise algorithms in Gibbons et al. [2022]; Jones and Gibbons [1993].

## 3 Modelling CCS

Though continuations are widely used in denotational semantics, they can cause meta-theoretical problems when used to model concurrent languages. This section will describe how we solved some of those problems in developing a hyperfunction model of the Calculus of Communicating Systems (CCS) [Milner et al. 1980]. The existence of this model shows that hyperfunctions are powerful enough to express the essential components of concurrency; or at least the kind of concurrency encapsulated by CCS.

### 3.1 CCS

CCS is a process calculus which supports concurrency, nondeterminism, and communication between processes. Its syntax is given in Fig. 2a. A term $p : \mathsf{P}\ n$ represents a process with names of type $n$. The operational semantics of CCS, given in Fig. 2b, is a labelled transition system. Each transition is labelled with an action $\mathsf{Act}\ n$, where an action can be silent, $\tau$, an input $\underline{n}$ or an output $\overline{n}$ of some name $n$. A trace for a process $p$ is a list of actions $[a_1, \ldots, a_n]$ that label a sequence of transitions $p \xrightarrow{a_1} p_1 \xrightarrow{a_2} \cdots \xrightarrow{a_n} p_n$. A process can have multiple possible traces.

**Actions.** The term $a \cdot p$ represents a process consisting of an action (Act) $a$, followed by a process $p$. The process $a \cdot p$ can emit the action $a$ and reduce to $p$, according to the Act rule.

**Nondeterminism.** The $\oplus$ operator represents nondeterministic choice, and $\mathbb{0}$ represents the empty or finished process. A process $p \oplus q$ can proceed by stepping through the left hand process (Sum$_\mathrm{L}$) or the right (Sum$_\mathrm{R}$). Notice that when one branch of a $\oplus$ expression is chosen, the other branch is discarded. So the process $a \cdot b \cdot \mathbb{0} \oplus c \cdot d \cdot \mathbb{0}$ has only two traces: $[a, b]$ or $[c, d]$. There is no rule related to $\mathbb{0}$, so the finished process cannot reduce.

**Parallelism.** The term $p \parallel q$ represents a parallel merge between the processes $p$ and $q$, which may communicate with each other. The rules Step$_\mathrm{L}$ and Step$_\mathrm{R}$ allow either side of $\parallel$ to step, without discarding the other. So the process $a \cdot b \cdot \mathbb{0} \parallel c \cdot \mathbb{0}$ has the traces $[a, b, c]$, $[a, c, b]$, and $[c, a, b]$. Parallel processes can also communicate: if an output from one process matches an

$$\textbf{data}\ \mathsf{Act}\ n = \tau \mid \underline{n} \mid \overline{n} \qquad \textbf{data}\ \mathsf{P}\ n = \mathsf{Act}\ n \cdot \mathsf{P}\ n \mid \mathbb{0} \mid \mathsf{P}\ n \oplus \mathsf{P}\ n \mid \mathsf{P}\ n \parallel \mathsf{P}\ n \mid \nu n \cdot \mathsf{P}\ n \mid !\ (\mathsf{P}\ n)$$

(a) CCS Syntax

$$\frac{}{a{\cdot}P \xrightarrow{a} P}\ \textsc{Act} \qquad \frac{P \xrightarrow{a} P'}{P \oplus Q \xrightarrow{a} P'}\ \textsc{Sum}_\textsc{L} \qquad \frac{Q \xrightarrow{a} Q'}{P \oplus Q \xrightarrow{a} Q'}\ \textsc{Sum}_\textsc{R} \qquad \frac{P \xrightarrow{a} P'}{P \parallel Q \xrightarrow{a} P' \parallel Q}\ \textsc{Step}_\textsc{L}$$

$$\frac{Q \xrightarrow{a} Q'}{P \parallel Q \xrightarrow{a} P \parallel Q'}\ \textsc{Step}_\textsc{R} \qquad \frac{P \xrightarrow{\underline{n}} P' \qquad Q \xrightarrow{\overline{n}} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'}\ \textsc{Sync}_\textsc{IO} \qquad \frac{P \xrightarrow{\overline{n}} P' \qquad Q \xrightarrow{\underline{n}} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'}\ \textsc{Sync}_\textsc{OI}$$

$$\frac{P \xrightarrow{a} P' \qquad a \notin \{\underline{n}, \overline{n}\}}{\nu n \cdot P \xrightarrow{a} \nu n \cdot P'}\ \textsc{Res} \qquad \frac{P \parallel !P \xrightarrow{a} P'}{!P \xrightarrow{a} P'}\ \textsc{Rep}$$

(b) CCS Operational Semantics

Fig. 2. CCS

input to the other, both processes reduce one step and the silent action is emitted ($\textsc{Sync}_\textsc{IO}$ and $\textsc{Sync}_\textsc{OI}$). So the process $\underline{n} \cdot \mathbb{0} \parallel \overline{n} \cdot \mathbb{0}$ has the traces $[\underline{n}, \overline{n}]$, $[\overline{n}, \underline{n}]$, and $[\tau]$.

**Restriction.** The term $\nu n \cdot p$ hides the name $n$ from anything outside of the process. As per the Res rule, a process under a $\nu n \cdot$ term can only reduce if the emitted action does not contain $n$. This can be used to enforce private communication: recall that the possible traces from $\underline{n} \cdot \mathbb{0} \parallel \overline{n} \cdot \mathbb{0}$ included $[\underline{n}, \overline{n}]$; if we instead wrap the term with $\nu n \cdot$, then we enforce communication, so the only valid trace is $[\tau]$.

**Iteration.** The term $!\,p$ represents the infinitely replicated process $p$. The rule Rep means that the expression $!\,p$ is equivalent to $p \parallel !\,p$.

Our treatment of CCS is standard: we take the same syntax and operational semantics as is in Chappe et al. [2023], which is a slight variant of the versions present in Veltri and Vezzosi [2023] and Bruni and Montanari [2017].

## 3.2 CCS Algebras

A CCS algebra is a way to interpret some CCS expression into a denotational domain. Concretely, we capture the notion of a CCS algebra with a class, CCSAlg, where a type $c$ is a carrier for a CCS algebra if there is an instance CCSAlg $c$.

$$\begin{array}{ll}
\textbf{class}\ \mathsf{CCSAlg}\ c\ \textbf{where} & \\
\quad \textbf{type}\ \mathsf{Name}\ c :: \mathsf{Type} & \\
\quad (\cdot)\ :: \mathsf{Act}\ (\mathsf{Name}\ c) \to c \to c & \nu \cdot \cdot :: \mathsf{Name}\ c \to c \to c \\
\quad \mathbb{0}\ \ :: c & (\parallel) :: c \to c \to c \\
\quad (\oplus) :: c \to c \to c & !\ \ :: c \to c
\end{array} \tag{13}$$

This class has one method for each of the syntactic constructors of CCS. It also includes an associated type Name, where Name $c$ represents the type of names that the CCS algebra on $c$ supports.

Using this class, we define $[\![\_]\!]$, which interprets a syntax tree P (Name $c$) into $c$ [Hutton 1998].

$$[\![\_]\!] :: \mathsf{CCSAlg}\ c \Rightarrow \mathsf{P}\ (\mathsf{Name}\ c) \to c \tag{14}$$

This $[\![\_]\!]$ function maps each syntactic construction to its corresponding method in CCSAlg. Note that this style of defining denotational semantics means they are automatically compositional.

*Remark 3.1.* There are three "interpretation"-like functions in this section: $[\![\_]\!]$ above (Eq. (14)), and $[\![\_]\!]\!\downarrow$ (Fig. 3b) and $[\![\_]\!]\!\uparrow$ (Fig. 4b) below. These functions are polymorphic in their return types, which can be difficult to infer from context, so we will occasionally use subscripts to disambiguate. In Eq. (14) above, for example, we might write $[\![\_]\!]_c$ to indicate that it has type $\mathsf{P}\ (\mathsf{Name}\ c) \to c$.

Instances of the CCSAlg class are expected to follow the following laws.

$$\mathbb{0} \oplus p = p \quad (p \oplus q) \oplus r = p \oplus (q \oplus r) \quad p \oplus q = q \oplus p \quad p \oplus p = p \tag{15}$$

$$\mathbb{0} \parallel p = p \quad (p \parallel q) \parallel r = p \parallel (q \parallel r) \quad p \parallel q = q \parallel p \tag{16}$$

$$\nu n \cdot \mathbb{0} = \mathbb{0} \quad \nu n \cdot (p \oplus q) = \nu n \cdot p \oplus \nu n \cdot q \tag{17}$$

$$!\,p = p \parallel !\,p \tag{18}$$

Under these laws, $\oplus$ forms a semilattice (a commutative idempotent monoid) with identity $\mathbb{0}$ (Eq. (15)), $\parallel$ forms a commutative monoid with identity $\mathbb{0}$ (Eq. (16)), $\nu\cdot$ is homomorphic on the $\oplus$ monoid (Eq. (17)), and ! expands to perform replication (Eq. (18)).

Structural congruence of CCS terms, an equivalence relation on P denoted by $\approx_s$, is defined as the equivalence closure of the relation generated by the above rules (with the addition of congruence rules). Any lawful implementation of CCSAlg therefore satisfies the property:

$$p \approx_s q \implies [\![p]\!] \equiv [\![q]\!]$$

All models of CCS are expected to be lawful instances of the CCSAlg class. Furthermore the syntax of CCS, when quotiented by $\approx_s$, also forms a lawful instance, where $[\![\_]\!]_{\mathsf{P}} \equiv id$.

We will note at this point that while these laws are *sound* (i.e. structurally congruent processes are semantically equivalent) they are not *complete* (bisimilar processes need not be structurally congruent). In fact, there is *no* finite set of laws (with this particular set of operators) that has this completeness property: this is explained in more detail in Section 3.4.

## 3.3 A Hyperfunction Model of CCS

Let's now turn back to hyperfunctions, and to the hyperfunction model of CCS. The entirety of this model is contained in Fig. 3: it consists of the carrier type (Communicator, Fig. 3a), a way to *interpret* this type into another model of CCS (Fig. 3b), and an implementation of the CCS operations (Fig. 3c).

**The Communicator Type.** The carrier type of our hyperfunction model is Communicator (Fig. 3a). We have taken some structure from Section 2.4: a Communicator is a process with result type $r$, that passes messages of type Message $n$. A Message is either a $\mathbb{q}$uery or an $\mathbb{a}$nswer. A $\mathbb{q}$uery is like a prompt: by passing a $\mathbb{q}$uery to a Communicator, we are asking "what is your top-level action?" The Communicator then responds with an $\mathbb{a}$nswer containing that top-level action.

**Interpretation.** It can be difficult to understand some of the functions in Fig. 3c in isolation: their implementations only really make sense when we keep the *interpretation* of a Communicator (Fig. 3b) in mind. For that reason, we'll go over interpretation first.

In this context, interpreting a Communicator $n\ r$ means evaluating that Communicator to its result type, $r$, via the function $[\![\_]\!]\!\downarrow$ :: Communicator $n\ r \to r$ (Fig. 3b). This evaluation translates the actions and nondeterministic operations on the Communicator to their analogous operations on $r$. In this way, $[\![\_]\!]\!\downarrow$ is a translation between two different representations of a CCS process.

The $[\![\_]\!]\!\downarrow$ function works by taking a Communicator $p$, and passing it two arguments: $\mathbb{1}$ and $\mathbb{q}$. Recall that passing a $\mathbb{q}$uery to a Communicator as its second argument prompts it to respond with its top-level action: in this case, the Communicator $p$ will respond by passing its top-level action to its first argument, $\mathbb{1}$. $\mathbb{1}$ is a special Communicator that translates Messages into actions on $r$: when supplied with an $\mathbb{a}$nswer containing some action $a$, it emits that action by using action prefixing

$$
\begin{aligned}
&\textbf{type}\ \mathsf{Communicator}\ n\ r\\
&\quad = (\mathsf{Message}\ n \to r) \looparrowright (\mathsf{Message}\ n \to r)\\
&\textbf{data}\ \mathsf{Message}\ n = \mathbb{q} \mid \mathbb{a}\ (\mathsf{Act}\ n)
\end{aligned}
$$

(a) The Communicator Type

$$
\begin{aligned}
&[\![p]\!]\!\downarrow\ = \iota\ p\ \mathbb{1}\ \mathbb{q}\\
&\iota\ \mathbb{1}\ p\ (\mathbb{a}\ a) = a \cdot [\![p]\!]\!\downarrow\\
&\iota\ \mathbb{1}\ \_\ \mathbb{q} \qquad = \mathbb{0}
\end{aligned}
$$

(b) Interpretation

$$
\begin{aligned}
&\textbf{instance}\ (\mathsf{Semilattice}\ r, \mathsf{Eq}\ n) \Rightarrow \mathsf{CCSAlg}\ (\mathsf{Communicator}\ n\ r)\ \textbf{where}\\
&\quad \textbf{type}\ \mathsf{Name}\ (\mathsf{Communicator}\ n\ r) = n
\end{aligned}
$$

$$
\begin{aligned}
&\iota\ (a \cdot p)\ q\ \mathbb{q} &&= \iota\ q\ p\ (\mathbb{a}\ a)\\
&\iota\ (\overline{n} \cdot p)\ q\ (\mathbb{a}\ \underline{n}) &&= \iota\ q\ p\ (\mathbb{a}\ \tau) \qquad (19)\\
&\iota\ (\_ \cdot \_)\ \_\ \_ &&= \mathbb{0}
\end{aligned}
$$

$$
\begin{aligned}
&\iota\ (\nu n \cdot p)\ q\ (\mathbb{a}\ \overline{n}) = \mathbb{0}\\
&\iota\ (\nu n \cdot p)\ q\ (\mathbb{a}\ \underline{n}) = \mathbb{0} \qquad (22)\\
&\iota\ (\nu n \cdot p)\ q\ m = \iota\ p\ (\nu n \cdot q)\ m
\end{aligned}
$$

$$\iota\ \mathbb{0}\ \_\ \_ = \mathbb{0} \qquad (20)$$

$$p \parallel q = (p \mathbin{|\!\lfloor} q) \oplus (q \mathbin{|\!\lfloor} p) \qquad (23)$$

$$\iota\ (p \oplus q)\ k\ m = \iota\ p\ k\ m \oplus \iota\ q\ k\ m \qquad (21)$$

$$!\ p = p \mathbin{|\!\lfloor}\ !\ p \qquad (24)$$

$$\iota\ (p \mathbin{|\!\lfloor} q)\ r = \iota\ p\ (q \parallel r) \qquad (25)$$

(c) The CCS Algebra on the Communicator Model

Fig. 3. The Communicator Model

on $r$, and then continues the interpretation of the rest of the process ($\mathbb{1}$ will never be supplied a $\mathbb{q}$uery in the context of this section, so for the clause $\iota\ \mathbb{1}\ p\ \mathbb{q}$ we simply terminate and return $\mathbb{0}$).

**The CCSAlg Instance.** While $r$ needs to support both action prefixing and nondeterminism for interpretation, only nondeterminism is needed for the CCSAlg instance on Communicator (Fig. 3c).

Theorem 3.2. *A Communicator $n$ $r$ is a CCS algebra for any semilattice* $(r, \oplus, \mathbb{0})$.

We prove this theorem by providing below instantiations for each of the methods; these proofs are straightforward, and provided in our mechanisation. Notice that we use the same symbols ($\oplus$ and $\mathbb{0}$) for the semilattice on $r$ and the semilattice on Communicator.

**Actions on Communicators.** The implementation of action prefixing on a Communicator is given in Eq. (19). Recall that we define hyperfunctions with copattern syntax; in the context of a Communicator, this means we define a Communicator by specifying what happens when it interacts with another process and message. So, to define the process generated by prepending the action $a$ to a process $p$, we specify what happens when this process $a \cdot p$ is merged with the process $q$ and some Message $m$. There are three clauses:

- In the first clause, $\iota\ (a \cdot p)\ q\ \mathbb{q}$, the incoming message is a $\mathbb{q}$uery, so we respond by transferring control to $q$ (by calling $\iota\ q$), passing it the rest of the current process ($p$), and the action being prefixed ($\mathbb{a}\ a$).
- In the second clause, $\iota\ (\overline{n} \cdot p)\ q\ (\mathbb{a}\ \underline{n})$, the incoming message $\mathbb{a}\ \underline{n}$ matches the action being prefixed $\overline{n}$, so we transfer control to $q$, passing it the message $\tau$. This "emits" the silent transition $\tau$ on a communication match.
- Finally, if neither of those cases match, we end, returning the empty process.

As mentioned above, this makes more sense when we bear interpretation (Fig. 3b) in mind. Consider the following example of stepping through the interpretation of $a \cdot p$:

$$\begin{array}{lll}
[\![a \cdot p]\!]\!\downarrow & \equiv & \{\text{ Apply the definition of } [\![\_]\!]\!\downarrow \text{ (Fig. 3b) }\} \\
\iota\ (a \cdot p)\ \mathbb{1}\ \mathbb{q} & \equiv & \{\text{ Match the first clause of Eq. (19) }\} \\
\iota\ \mathbb{1}\ p\ (\mathbb{a}\ a) & \equiv & \{\text{ Apply the definition of } \mathbb{1} \text{ (Fig. 3b) }\} \\
a \cdot [\![p]\!]\!\downarrow & &
\end{array}$$

In this way, we can see that $[\![\_]\!]\!\downarrow$ is a homomorphism for action prefixing. We could also step through a communication match (although we do not include the full trace here):

$$[\![\underline{n} \cdot p \parallel \overline{n} \cdot q]\!]\!\downarrow \equiv \tau \cdot [\![p \parallel q]\!]\!\downarrow \oplus \underline{n} \cdot [\![p \parallel \overline{n} \cdot q]\!]\!\downarrow \oplus \overline{n} \cdot [\![\underline{n} \cdot p \parallel q]\!]\!\downarrow$$

**Nondeterminism.** Nondeterminism in CCS comprises the operators $\mathbb{0}$ and $\oplus$. These are implemented on the Communicator model in Eqs. (20) and (21). Since these two operators are simple algebraic operators, they can be lifted pointwise into a hyperfunction. The proofs of the laws are also simple: they follow directly from the semilattice instance on $r$.

**Restriction.** Restricting a process $\nu n \cdot p$ (Eq. (22)) makes it so that the process $p$ cannot communicate the name $n$ with anything outside of $p$. A Communicator can both send and receive messages: to restrict a Communicator, we censor incoming and outgoing messages to kill processes which mention the restricted name. For example, the process $\iota\ (\nu n \cdot p)\ q\ m$ receives the message $m$, and can send messages to $q$, all while restricting the name $n$. If the incoming message contains the restricted name (i.e. when $m \coloneqq \mathbb{a}\ \underline{n}$) the whole process is equal to $\mathbb{0}$. If the incoming message does not contain the restricted name, we continue by transferring control to the next process, $q$. To censor outgoing messages, we censor the *incoming* messages of the *recipient* process ($q$), by calling $\nu$ recursively ($\nu n \cdot q$).

**Parallel Merge.** Parallel merge is given in Eq. (23). We can use hyperfunction composition (Eq. (7)) as a starting point for this implementation. However, while composition allows processes to interleave and communicate, $\parallel$ needs to also produce all possible orderings between its two arguments. To add this behaviour, we might first attempt something like $p \parallel q = (p \circ q) \oplus (q \circ p)$, but this only permutes the top level arguments. Instead, we need to replace $\circ$ with a kind of composition which continues reordering recursively: here we rely on a helper function, $\parallel\!\lfloor$ (Eq. (25)). This performs one layer of composition before calling back to $\parallel$ to permute all later arguments.

*Remark 3.3.* Later, when we prove that hyperfunctions form a model for CCS, we will use a variant of the CCS syntax which includes the $\parallel\!\lfloor$ operator. A similar technique is used in Bergstra and Klop [1984], which also adds additional operators similar in semantics to the $\parallel\!\lfloor$ operator here, or $step_l$ and $sync_{io}$ later (Eqs. (33) and (34)).

**Replication.** Replication (Eq. (24)) should have the semantics $!\,p = p \parallel !\,p$. Unfortunately, using that equation as a definition is not well-founded: it would not give a productive definition in our implementation. However, we can derive another identity, $!\,p = p \parallel\!\lfloor\ !\,p$ (Lemma C.5), which *does* yield a productive definition.

*Remark 3.4.* We can combine $[\![\_]\!]\!\downarrow$ with $[\![\_]\!]$ to interpret CCS syntax into a Communicator and then interpret that Communicator into the underlying CCS algebra.

$$[\![\_]\!]\!\downarrow \circ [\![\_]\!] :: \mathsf{CCSAlg}\ r \Rightarrow \mathsf{P}\ (\mathsf{Name}\ r) \to r$$

However, notice that nowhere in Fig. 3 do we make use of $\parallel$, $\nu\cdot$, or ! on $r$. This means that the above function rewrites a CCS process into one that uses only $\oplus$, $\mathbb{0}$, and $\cdot$ (action prefixing). When this conversion is semantic-preserving (proven below in Lemma 3.10), it amounts to a constructive proof of Theorem 11.10 from Bruni and Montanari [2017].

## 3.4 Proving that Communicator is a Model

We now turn to the task of proving that the model established above (Fig. 3) is fully abstract.

**The Plan.** We will start by defining bisimilarity and full abstraction (Section 3.4.1). Then, we will discuss why full abstraction is difficult for continuation-based models in particular, and summarise the progress made by Ciobanu and Todoran [2017, 2018] on this problem (Remark 3.6). From there, we will introduce the Proc model [Veltri and Vezzosi 2023]: this is a standard fully-abstract model for CCS which we can use to prove full abstraction for the Communicator via a pair of homomorphisms between the Proc and Communicator model (Section 3.4.2). Then, we will give our foundational model of hyperfunctions, based on the categorical model of Krstić et al. [2001a], and briefly give our argument for well-foundedness (Remark 3.8). Finally, in Sections 3.4.3 and 3.4.4, we will briefly summarise the detailed proof (present in full in Appendix C), and we will end with a short discussion of formalisation (Remark 3.17).

### *3.4.1 Bisimilarity and Full Abstraction.*

Equivalence between CCS processes is captured by (strong) bisimilarity, denoted by $\sim$. Other notions of equivalence, like trace equivalence or weak bisimilarity (a version of bisimilarity where emitted $\tau$s are ignored), fail to capture important aspects of CCS's semantics: Bruni and Montanari give a good summary of the problems [2017].

A model of CCS is an algebra that *respects* this bisimilarity. A fully abstract model is a model where equality in the denotational domain corresponds precisely to bisimilarity of CCS processes.

*Definition 3.5 (Full Abstraction).* A model $m$ of CCS is *fully abstract* when:

$$\forall p, q.\ p \sim q \iff [\![p]\!]_m \equiv [\![q]\!]_m$$

The structural congruence laws stated in Section 3.2 are not sufficient to prove this property, nor even a weakening like $p \sim q \implies [\![p]\!] \equiv [\![q]\!]$. In fact, there is no finite set of laws that is sufficient. To be precise, there is no finite axiomatisation of CCS that corresponds to the bisimulation equivalence derived from the operational semantics in Fig. 2b [Moller 1990a,b]. Our proof will have to take a different route.

*Remark 3.6 (Why Full Abstraction is Difficult for Continuation Models).* Continuation-based models tend to be *large*, where the denotational domain contains more values than there are denotations of the source language. So, for some language with terms of type $\mathcal{T}$, and an interpretation into a denotational domain of type $\mathcal{D}$, if the domain is large then there are values $v$ of type $\mathcal{D}$ for which there are no terms that interpret to those values ($\exists(v : \mathcal{D}).\nexists(t : \mathcal{T}).[\![t]\!] = v$).

This alone isn't a showstopper: while a large domain can't be isomorphic to the denotations, full abstraction is a little weaker than isomorphism. Notice that the definition of full abstraction above (Definition 3.5) only refers to values from the denotational domain that are generated from the syntax of CCS: the fact that there might be extra "junk" in the denotational domain doesn't matter.

For continuation-based models like the Communicator, however, this "junk" causes other problems. To understand why, consider the type of Communicators that *are* generated from syntax trees. This is a subset of the Communicator type; Ciobanu and Todoran [2017] call their version of this type the "denotable" continuations. While at first glance it might seem viable to work with this subset type alone, remember that a Communicator is a function *which takes another Communicator as an argument.*

$$\mathsf{Communicator}\ n\ r \equiv \mathsf{Communicator}\ n\ r \to \mathsf{Message}\ n \to r$$

So any property we prove about the denotable Communicators will not necessarily apply to the Communicator passed recursively. This breaks all but the simplest proofs that rely on (co)recursion. We *could* amend the definition of Communicator to only accept denotable Communicators, but that turns a simple subset type into something much more restricted and complex altogether. Furthermore, the $\mathbb{1}$ Communicator (Fig. 3b) is decidedly not denotable, so we would have lost our ability to interpret a Communicator with this change.

Ciobanu and Todoran identified this problem, and defined *weak abstraction* to better capture a notion of correctness that applies to continuation-based models [2017]. Weak abstraction takes into account the idea of "denotable" continuations: informally, a proof of weak abstraction is very similar to one of full abstraction, but the former proof only considers denotable continuations. Weak abstraction still gives strong correctness guarantees, and it may well be the case that some useful continuation models can only ever be weakly abstract; Ciobanu and Todoran's model of CCS [2018] may be one such model.

Our proof of full abstraction does not contain any particularly clever trick or conceptual leap to sidestep this problem of denotable continuations. Instead, some proofs apply to Communicators generally, and others apply only to those Communicators that are denotations. It is only by the careful design of the inductive hypotheses of Lemmas 3.15 and 3.16 that these restrictions line up with available premises in the right places, yielding our eventual proof.

*3.4.2 The Proc Model.* As mentioned in Section 3.4.1, we cannot prove full abstraction via the laws of the CCS algebra alone. We will instead prove full abstraction by relating the Communicator to another fully-abstract model: the Proc model (Fig. 4). The denotational domain for this model is given as the Haskell type in Fig. 4a. CCS processes are represented by forests of coinductive rose trees, with internal nodes labelled by Acts: for example, the process $\underline{a} \cdot \underline{b} \cdot \mathbb{0} \parallel \overline{a} \cdot \mathbb{0}$ is represented by the tree in Fig. 4c.

While this is a standard model for CCS, our specific iteration is based on the presentation in Veltri and Vezzosi [2023], with some notable differences. Firstly, our type is not indexed by the number of free names (Veltri and Vezzosi's Proc has kind $\mathbb{N} \rightarrow$ Type). Secondly, our type contains no special constructions to handle the coinduction in Proc: these constructions are needed in Agda, where inductive and coinductive types are distinguished; Haskell is less precise in this area, allowing us to write coinductive types without ceremony.

Finally, our Proc type is built out of nested lists, where Veltri and Vezzosi's Proc type is built out of nested "countable powersets". As it happens, the full generality of the countable powerset type is not needed: Proc implemented with finite sets is also a fully abstract model of CCS. Unfortunately, current Haskell does not have quotients (although projects like Hewer and Hutton [2024] are beginning to remedy this), so even finite sets are unavailable to us. The usual trick in this situation is to *mimic* quotients, by pretending that the desired equalities hold, and by carefully implementing only functions which respect those desired equalities. The additional equalities on Proc are given in Fig. 4d; when they hold, they imply the validity of identities like the following:

$$\mathsf{Proc}\ [\,(a, \mathsf{Proc}\ [\,]), (b, \mathsf{Proc}\ [\,])\,] \equiv \mathsf{Proc}\ [\,(b, \mathsf{Proc}\ [\,]), (a, \mathsf{Proc}\ [\,])\,]$$

$$\mathsf{Proc}\ [\,(a, \mathsf{Proc}\ [\,]), (a, \mathsf{Proc}\ [\,])\,] \equiv \mathsf{Proc}\ [\,(a, \mathsf{Proc}\ [\,])\,]$$

**The CCS Algebra on Proc.** The implementation of the CCS algebra on Proc is given in Fig. 4e. It implements the following methods:

- **Action prefixing, Eq. (26).** $a \cdot p$ creates a new tree with root $a$, and a single child $p$.
- **Nondeterminism, Eqs. (27) and (28).** $\oplus$ concatenates the root levels of trees; $\mathbb{0}$ is represented by the empty tree.
- **Restriction, Eq. (29).** $\nu n \cdot p$ recursively traverses $p$, deleting any branches with $n$ at the root.

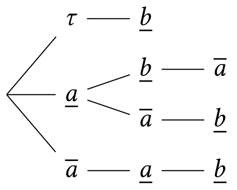
$$\textbf{newtype}\ \mathsf{Proc}\ n = \mathsf{Proc}\ \{\, root :: [\,(\mathsf{Act}\ n, \mathsf{Proc}\ n)\,]\,\}$$

(a) The Proc Type

$$\begin{aligned}
[\![\mathsf{Proc}\ [\,]]\!]\!\uparrow &= \mathbb{0} \\
[\![\mathsf{Proc}\ ((a,p) : q)]\!]\!\uparrow &= a \cdot [\![p]\!]\!\uparrow \oplus [\![\mathsf{Proc}\ q]\!]\!\uparrow
\end{aligned}$$

(b) Interpretation

$$[\![\underline{a}\cdot\underline{b}\cdot\mathbb{0} \parallel \overline{a}\cdot\mathbb{0}]\!]_{\mathsf{Proc}} = \begin{array}{l}
\mathsf{Proc}\ [\,(\tau, \mathsf{Proc}\ [\,(\underline{b}, \mathsf{Proc}\ [\,])\,])\\
\quad ,(\underline{a}, \mathsf{Proc}\ [\,(\underline{b}, \mathsf{Proc}\ [\,(\overline{a}, \mathsf{Proc}\ [\,])\,])\\
\qquad\qquad ,(\overline{a}, \mathsf{Proc}\ [\,(\underline{b}, \mathsf{Proc}\ [\,])\,])\,])\\
\quad ,(\overline{a}, \mathsf{Proc}\ [\,(\underline{a}, \mathsf{Proc}\ [\,(\underline{b}, \mathsf{Proc}\ [\,])\,])\,])\,]
\end{array} =$$


$\tau$ — $\underline{b}$
$\underline{a}$ — $\underline{b}$ — $\overline{a}$
$\underline{a}$ — $\overline{a}$ — $\underline{b}$
$\overline{a}$ — $\underline{a}$ — $\underline{b}$

(c) The Proc Representation of a CCS process

$$\forall p, q.\ \mathsf{Proc}\ (p +\!\!+\ q) \equiv \mathsf{Proc}\ (q +\!\!+\ p) \qquad \forall p.\ \mathsf{Proc}\ (p +\!\!+\ p) \equiv \mathsf{Proc}\ p$$

(d) Quotients on Proc

$$\begin{array}{ll}
\multicolumn{2}{l}{\textbf{instance}\ \mathsf{Eq}\ n \Rightarrow \mathsf{CCSAlg}\ (\mathsf{Proc}\ n)\ \textbf{where}}\\
\multicolumn{2}{l}{\quad \textbf{type}\ \mathsf{Name}\ (\mathsf{Proc}\ n) = n}\\
\quad a \cdot p = \mathsf{Proc}\ [\,(a, p)\,] & (26)\\
\quad \mathbb{0} = \mathsf{Proc}\ [\,] & (27)\\
\quad p \oplus q = \mathsf{Proc}\ (root\ p +\!\!+\ root\ q) & (28)\\
\quad \nu n \cdot p = \mathsf{Proc}\ [\,(a, \nu n \cdot p') \mid (a, p') \leftarrow root\ p,\ a \not\equiv \underline{n}, a \not\equiv \overline{n}] & (29)\\
\quad p \parallel q = (p \mathbin{\|\!\lfloor} q) \oplus (q \mathbin{\|\!\lfloor} p) & (30)\\
\quad !\ p = step_l\ (p \oplus sync_{io}\ p\ p)\ (!\ p) & (31)
\end{array}$$

(e) The CCSAlg Instance

$$\begin{array}{ll}
p \mathbin{\|\!\lfloor} q = sync_{io}\ p\ q \oplus step_l\ p\ q & (32)\\
step_l\ p\ q = \mathsf{Proc}\ [\,(a, p' \parallel q) \mid (a, p') \leftarrow root\ p] & (33)\\
sync_{io}\ p\ q = \mathsf{Proc}\ [\,(\tau, p' \parallel q') \mid (\underline{a}, p') \leftarrow root\ p,\ (\overline{b}, q') \leftarrow root\ q,\ a \equiv b] & (34)
\end{array}$$

(f) Helper Functions

Fig. 4. The Proc Model

**Parallel Merge, Eq.(30).** $\parallel$ is the most complicated method. Similarly to $\parallel$ on the Communicator, this method is implemented as nondeterministic choice between two applications of the left-biased parallel merge, $\mathbin{\|\!\lfloor}$ (Eq.(32)). When we expand out the definition of $\mathbin{\|\!\lfloor}$, we see that $\parallel$ has the implementation $p \parallel q = step_l\ p\ q \oplus sync_{io}\ p\ q \oplus step_l\ q\ p \oplus sync_{io}\ q\ p$: in other words, it is a nondeterministic choice between all four possible operational rules (Fig. 2b) that apply to $\parallel$. The two helper functions $step_l$ (Eq.(33)) and $sync_{io}$ (Eq.(34)) correspond to the rules $\textsc{Step}_{\textsc{L}}$ and $\textsc{Sync}_{\textsc{IO}}$. $step_l$ allows the left-hand-side argument to perform one action, and then merges the subsequent processes ($step_l\ (a \cdot p)\ q = a \cdot (p \parallel q)$). $sync_{io}\ p\ q$ pulls an input from $p$, and a corresponding output from $q$, and merges the rest of the processes ($sync_{io}\ (\underline{a} \cdot p)\ (\overline{a} \cdot q) = \tau \cdot (p \parallel q)$). These two rules are grouped together in the function $\mathbin{\|\!\lfloor}$. The other two rules—$\textsc{Step}_{\textsc{R}}$ and $\textsc{Sync}_{\textsc{OI}}$—are just symmetric variants of the first two, so they can be applied by flipping the arguments to $\mathbin{\|\!\lfloor}$.

**Replication, Eq. (31).** We cannot use the identity Eq. (18) as a definition, because it is not productive. The implementation given here, however, *is* productive, and is also bisimilar to Eq. (18). This definition exploits the idempotency of $\oplus$ to define an equivalent expression that does not diverge.

These methods are all adapted from their implementations in Veltri and Vezzosi [2023]. The only real change is our definition of $\parallel$, where Veltri and Vezzosi's implementation is:

$$p \parallel q = step_l\ p\ q \oplus step_r\ p\ q \oplus synch\ p\ q$$

$step_r$ is a variant of $step_l$ with the arguments reversed, and $synch$ is a commutative variant of $sync_{io}$, where $synch\ p\ q = sync_{io}\ p\ q \oplus sync_{io}\ q\ p$. Some rearranging shows that the difference is superficial.

All laws given by our structural congruence are proven in Veltri and Vezzosi [2023], with the exception of the idempotency of $\oplus$, though that law is implied by the operational semantics, and so is proven indirectly. Also proven in Veltri and Vezzosi [2023] is the following:

Theorem 3.7 (Proc is fully abstract). $\forall p, q.\ p \sim q \iff [\![p]\!]_{\mathsf{Proc}} \equiv [\![q]\!]_{\mathsf{Proc}}$

We will use this to prove full abstraction for the Communicator.

*Remark 3.8 (Foundations and Well-Foundedness).* For cardinality reasons, the hyperfunction type does not have a set-theoretic interpretation (there is no set that corresponds to the type $a \looparrowright b$). Hyperfunctions follow a standard domain-theoretic [Abramsky and Jung 1995] interpretation, however, as described by Krstić et al. [2001b]. The base category here is some (cartesian closed) category of pointed domains, closed under bilimits. Under this interpretation, hyperfunctions of type $A \looparrowright B$ are the canonical solution of the equation $X \cong (X \Rightarrow A) \Rightarrow B$. This interpretation characterises the recursively-defined hyperfunctions and hyperfunction operations of Launchbury et al. [2000]. Krstić et al. also gave an account of hyperfunctions as final coalgebras [2001a], and showed that the recursive definitions correspond to this coalgebraic interpretation.

Our proofs go through without issue in this setting; because our proofs proceed by induction on the syntax of CCS, we do not need to use the more sophisticated tools of "hyperfunction induction" from Krstić et al. [2001b]; function extensionality is sufficient.

However, because CCS processes can be infinite, we do need to address the issue of corecursion and well-foundedness in our proofs. We don't necessarily need to consider corecursion on hyperfunctions directly: to prove equality of Communicators, we need only prove the equality of the underlying CCS processes that they produce. In fact, we can simplify further; instead of referring to CCS processes in general, we can specialise to the Proc model. Because the Proc model is fully-abstract, we can perform this specialisation without loss of generality. This means that all proofs of equality in this section eventually resolve to proofs of equality on Proc objects.

The well-foundedness of our proofs, then, corresponds to the well-foundedness of proofs of equality on coinductive Proc trees. This notion is well-defined: indeed, Veltri and Vezzosi's formalisation of Proc contains a detailed exploration in the context of guarded cubical Agda and Ticked Cubical Type Theory [Møgelberg and Veltri 2019]. We have not formalised our well-foundedness argument (see Remark 3.17 for a discussion); instead this argument will be made in prose, and will be based on *syntactic guarded* coinduction [Coquand 1994].

In this section, a CCS process is *guarded* if it is syntactically "under" some action. For example, in the expression $(a \cdot p) \oplus q$, the process $p$ is guarded ("$p$ is guarded by $a$"), whereas the process $q$ is unguarded. Corecursive calls are permitted only if they are guarded; so the infinite process $p \coloneqq a \cdot p$, which consists of a stream of $a$s, is well-founded, whereas the definition $p \coloneqq p \oplus p$ is not.

This notion of guardedness extends to proofs of equality: if a proof relies on some coinductive call, that call must be guarded under an action. So, to prove the equality of two processes $a \cdot p$

and $b \cdot q$, the proof must prove equality of $a$ and $b$ inductively, but once those are proven some coinductive call is permitted to prove the equality of $p$ and $q$.

All of the proofs and definitions in this section are well-founded according to this guardedness condition. In particular, the proofs of full abstraction all proceed via induction on the syntax of CCS: all recursive calls either reduce the size of one argument (i.e. they are "terminating" in the normal/inductive sense), or the recursive call is guarded under an action. As might be expected, we only need to employ this guardedness argument when the CCS process is infinite, i.e. it includes the ! operator. Without that operator, our proofs are well-founded by induction on syntax.

### 3.4.3 *Relating Communicator to Proc.*

Our strategy is to rely on the fact that Proc is fully abstract, and prove full abstraction for the Communicator model via a relation between Communicator and Proc. Let's now define precisely what that relation is.

We have already seen a way to convert a Communicator to any CCSAlg, including Proc: the $[\![\_]\!]\!\downarrow$ function (Fig. 3b). To go the other direction we use $[\![\_]\!]\!\uparrow$ (Fig. 4b). A Proc represents a CCS process as nested sums-of-acts, so to convert that structure into another CCS algebra we just apply $\oplus$ and $\cdot$ in the right places.

If, at this point, we could show that these functions form the two halves of an isomorphism, we would have our proof of full abstraction. And, indeed, $[\![\_]\!]\!\downarrow$ is a retraction of $[\![\_]\!]\!\uparrow$:

Lemma 3.9. $\forall (p : \text{Proc}\ n).\ [\![ [\![p]\!]\!\uparrow_{\text{Communicator}\ n\ (\text{Proc}\ n)} ]\!]\!\downarrow_{\text{Proc}\ n} \equiv p$

However the inverse is not true in general ($[\![\_]\!]\!\uparrow \circ [\![\_]\!]\!\downarrow \not\equiv id$). As described in Remark 3.6, we do not have an isomorphism; but we do not need a full isomorphism for full abstraction. Instead, the following two lemmas are sufficient to prove full abstraction for the Communicator model:

Lemma 3.10. $\forall (p : P\ n).\ [\![ [\![p]\!]_{\text{Communicator}} ]\!]\!\downarrow \equiv [\![p]\!]_{\text{Proc}}$

Lemma 3.11. $\forall (p : P\ n).\ [\![ [\![p]\!]_{\text{Proc}} ]\!]\!\uparrow \equiv [\![p]\!]_{\text{Communicator}}$

The first of these, Lemma 3.10, says that, for any CCS term $p$, if we interpret that term into a Communicator, and then interpret that Communicator into a Proc, that is the same as interpreting the term $p$ directly into a Proc. The second (Lemma 3.11) says the inverse. We can combine these with Proc's full abstraction to prove the following:

Theorem 3.12 (Communicator is fully abstract).

$$\forall p, q.\ p \sim q \iff [\![p]\!]_{\text{Communicator}} \equiv [\![q]\!]_{\text{Communicator}}$$

Proof. Recall first that Proc is fully abstract (Theorem 3.7; $\forall p, q.\ p \sim q \iff [\![p]\!]_{\text{Proc}} \equiv [\![q]\!]_{\text{Proc}}$). To prove full abstraction for Communicator, then, we need to show:

$$\forall p, q.\ [\![p]\!]_{\text{Proc}} \equiv [\![q]\!]_{\text{Proc}} \iff [\![p]\!]_{\text{Communicator}} \equiv [\![q]\!]_{\text{Communicator}}$$

Here we prove the bi-implication in both directions, for all $p$ and $q$:

$$[\![p]\!]_{\text{Proc}} \equiv [\![q]\!]_{\text{Proc}} \implies [\![p]\!]_{\text{Communicator}} \equiv [\![q]\!]_{\text{Communicator}}$$

$$\begin{aligned} & [\![p]\!]_{\text{Communicator}} && \equiv \{\text{Lemma 3.11}\} \\ & [\![ [\![p]\!]_{\text{Proc}} ]\!]\!\uparrow && \equiv \{\text{Given}\} \\ & [\![ [\![q]\!]_{\text{Proc}} ]\!]\!\uparrow && \equiv \{\text{Lemma 3.11}\} \\ & [\![q]\!]_{\text{Communicator}} && \square \end{aligned}$$

$$[\![p]\!]_{\text{Communicator}} \equiv [\![q]\!]_{\text{Communicator}} \implies [\![p]\!]_{\text{Proc}} \equiv [\![q]\!]_{\text{Proc}}$$

$$\begin{aligned} & [\![p]\!]_{\text{Proc}} && \equiv \{\text{Lemma 3.10}\} \\ & [\![ [\![p]\!]_{\text{Communicator}} ]\!]\!\downarrow && \equiv \{\text{Given}\} \\ & [\![ [\![q]\!]_{\text{Communicator}} ]\!]\!\downarrow && \equiv \{\text{Lemma 3.10}\} \\ & [\![q]\!]_{\text{Proc}} && \square \end{aligned}$$

□

*Remark 3.13.* Notice that the proof above refers specifically to a Communicator specialised to Communicator $n$ (Proc $n$). However, because Proc itself is fully abstract, we haven't lost any generality via this specialisation: a Proc can be interpreted (via $[\![\_]\!]\uparrow$, Fig. 4b) into any other model of CCS while preserving semantics.

*3.4.4 Proving Lemmas 3.10 and 3.11.* The main theorem of this section, Theorem 3.12, relies on Lemmas 3.10 and 3.11. These lemmas establish that there is a relation between the Communicator and Proc models, and this relation is homomorphic through the CCS algebra. Proving these lemmas is where the bulk of the work of proving full abstraction for Communicator occurs.

The proofs of both lemmas follow the same pattern: we will give a brief outline of that pattern here (full proofs are present in Appendix C). Unfortunately, it is not possible to prove either of these lemmas by proving individual homomorphisms for each operator. While such homomorphisms do hold for some operators:

$$[\![p]\!]\downarrow \oplus [\![q]\!]\downarrow \equiv [\![p \oplus q]\!]\downarrow \qquad\qquad [\![\mathbb{0}]\!]\downarrow \equiv \mathbb{0}$$

They do not hold for others, with $\|$ being the most problematic.

Recall the problem of not being able to finitely axiomatise CCS, discussed above. Though we no longer rely on such an axiomatisation, solutions (or, rather, workarounds) to this problem from the literature will provide insights that we can use in our own proof.

Bergstra and Klop [1984] describe the Algebra of Communicating Processes (ACP), a similar calculus to CCS that *can* be finitely axiomatised. The key change in ACP that allows this axiomatisation is the addition of two new operators: a left-biased operator supporting the Step$_\text{L}$ rule, and a commutative operator that allows for communication. The original $\|$ can then be defined in terms of these operators. Unfortunately, the Communicator type does not implement ACP; but their decomposition of $\|$ is similar to our decomposition.

We have defined $\|\!\lfloor$, on both Communicator (Eq. (25)) and Proc (Eq. (32)), and $\|$ can be defined in terms of it. Furthermore, on Proc, the $\|\!\lfloor$ operator is defined in terms of two even more fundamental operators: $step_l$ (Eq. (33)) and $sync_{io}$ (Eq. (34)). The $\|\!\lfloor$ operator on Communicator can *almost* be decomposed in a similar way with the following definitions:

$$\iota\ (step_l\ p\ q)\ o = \iota\ p\ (o \|\!\lfloor q) \qquad\qquad \iota\ (sync_{io}\ p\ q)\ o = \iota\ p\ (q \|\!\lfloor o)$$

However, the identity $p \|\!\lfloor q \equiv sync_{io}\ p\ q \oplus step_l\ p\ q$ does not hold in general. The problem is that we cannot distribute a $\oplus$ under $\iota\ p$; however this equality *does* hold (definitionally) in the situation where $p \coloneqq a \cdot p'$. We will use this fact to prove homomorphism for Communicator.

Lemma 3.14. $\forall a, p, q.\ a \cdot p \|\!\lfloor q \equiv sync_{io}\ (a \cdot p)\ q \oplus step_l\ (a \cdot p)\ q$

The strategy for this proof, then, is to rewrite the term $p$ into a form where Lemma 3.14 and similar lemmas can apply. One other thing to note about the proof is that we add the $\|\!\lfloor$ operator to the syntax of CCS; this allows us to easier track when a term stays the same size or gets smaller. It also does not lose any generality: any term $p$ can be converted to a term that contains $\|\!\lfloor$.

The bulk of the work of this proof is accomplished in Lemmas 3.15 and 3.16.

Lemma 3.15. $\forall n, p, q.\ [\![[\![\nu_s n.(p \|\!\lfloor q)]\!]_{\text{Communicator}}]\!]\downarrow \equiv [\![\nu_s n.(p \|\!\lfloor q)]\!]_{\text{Proc}}$

Lemma 3.16. $\forall n, p, q.\ [\![[\![\nu_s n.(p \|\!\lfloor q)]\!]_{\text{Proc}}]\!]\uparrow \equiv [\![\nu_s n.(p \|\!\lfloor q)]\!]_{\text{Communicator}}$

Lemmas 3.15 and 3.16 are effectively special cases of Lemmas 3.10 and 3.11; they prove that $[\![\_]\!]\downarrow$ and $[\![\_]\!]\uparrow$ are homomorphisms on terms of the form $\nu_s n.(p \|\!\lfloor q)$. The operator $\nu_s$ here is a variant of $\nu$ that takes a list of names rather than a single name, where

$$\nu_s [\,].p = p \qquad\qquad \nu_s (n : ns).p = \nu_s ns.\nu n \cdot p$$

To use Lemma 3.15 we notice that all terms $t$ can be rewritten into this form $(\nu_s n.(p \parallel\!\lfloor\, q))$, because (by Lemma C.4), $t \equiv \nu_s[].(t \parallel\!\lfloor\, \mathbb{0})$. This identity holds on both Proc and Communicator, so we can apply it to both sides of the equation, meaning that Lemma 3.15 proves the homomorphism.

*Remark 3.17 (Mechanisation).* Accompanying this paper, we have provided a mechanisation of our proofs of full abstraction. This mechanisation is encoded in Agda [Norell 2009]. It follows the same structure as our prose proofs of Theorem 3.12, and of the proofs in Appendix C.

This mechanisation is necessarily partial, because there are aspects of the nature of hyperfunctions and the Communicator model that are not expressible in current Agda (without significant extensions to the underlying type theory which are beyond the scope of this work).

The first roadblock to full formalisation is that hyperfunctions (and specifically the Communicator type) are not (currently) admissible in Agda. As discussed in Remark 3.8, the hyperfunction type itself is somewhat exotic, and as such does not exist in all foundational settings (set theory, in particular, does not support the definition of hyperfunctions). Agda's type theory is another setting which does not admit hyperfunctions, however the problem here is *positivity*. Since the hyperfunction type contains recursion to the left of a function arrow, it is not *positive*. The presence of such types can allow for proofs of Curry's paradox [1942].

There are some possible routes around the positivity restriction. For example, while the type $a \looparrowright b$ is not "strictly" positive, if we were able to restrict $a$ to being contravariant, and $b$ to covariant, then the whole definition would become *positive* (albeit not strictly so). There is some evidence that Agda could admit these positive types (with the co/contra-variant restrictions) without sacrificing soundness [Coquand 2013; Sjöberg 2015].

Another route to admissibility comes from Berger et al.'s formalisation [2019] of Hofmann's breadth-first traversal [1993]. Berger et al. give several different verifications of the algorithm which use the Rou type (Eq. (12)); we believe the techniques of embedding the Rou type could also apply to the Communicator type.

The second assumption our mechanisation makes concerns well-foundedness. We have already given our argument for well-foundedness in the text (Remark 3.8); unfortunately, this argument relies on mixing notions of guardedness and continuations in ways that are currently beyond the capabilities of Agda's productivity checker. Certainly, the work of Veltri and Vezzosi [2020, 2023] paves the way for a future formalisation: however, adapting these techniques to work with a continuation-based representation would require extension to Agda itself.

It is worth emphasising that our foundational setting in this work is the domain-theoretic setting established by Krstić et al. [2001b]. This is different from the setting of our mechanisation, and as such the mechanisation should be regarded as supplementary to the proofs in this paper. Because the proofs can get quite intricate and dense, we think that the mechanisation gives some valuable reassurance that all cases/parameters have been handled.

The code is rendered online at doisinkidney.com/code/hyperfunctions/README.html. Alternatively, the code is available to download from doisinkidney.com/artifacts/popl-2025-hyperfunctions-agda.tar.gz; it has been typechecked with Agda version 2.8.0, and the cubical library version 0.8.

## 4 Hyperfunctions and Monads

So far, we have seen hyperfunctions model various aspects of concurrency, culminating in an implementation of CCS. In this section, we will show how hyperfunctions interact with *monads* [Wadler 1995], and in particular how they can be used to build concurrency monads. This section will demonstrate that hyperfunctions can serve a useful role in implementing efficient monadic library code, especially when concurrency or concurrency-like patterns are involved.

### 4.1 Adding Monads Simply

We will warm up with a simple example of combining monads and hyperfunctions. Recall the implementation of *zip* using hyperfunctions (Section 2.4): there, hyperfunctions allowed us to write *zip* on two Church-encoded lists without the usual $O(n^2)$ slowdown that comes from repeated applications of *tail*. We can use a similar technique to efficiently implement disjunction on the LogicT type [Kiselyov et al. 2005], a type for Prolog-style logic programming.

On LogicT, disjunction is implemented by interleaving ($[1,2,3] \mathbin{\langle|\rangle} [4,5,6] = [1,4,2,5,3,6]$). On the CPS-encoded version of LogicT, interleaving runs into the same problems as *zip*, because *interleave* is a *lateral* function. However, we are armed with a toolbox of hyperfunctions and hyperfunction combinators. As a result, implementing *interleave* is not difficult, following the pattern of *zip*:

```
interleave :: [a] → [a] → [a]
interleave xs ys = let xz = foldr (λx xk → (x:) ◂ xk) (Hyp (const [ ])) xs
                       yz = foldr (λy yk → (y:) ◂ yk) (Hyp (const [ ])) ys
                   in  ι xz yz
```

In fact, it is a little simpler than *zip*, since no message-passing is needed.

The LogicT type is not just a Church-encoded list, however. It is a CPS-encoded list *transformer.*

```
newtype LogicT m a = LogicT {runLogicT :: ∀b.(a → m b → m b) → m b → m b}
```

This type is similar to a Church-encoded list, but it allows effects—drawn from $m$—to be interleaved with the elements of the list. The following function, for instance, converts a list to a LogicT list, interleaving each element with an IO effect that prints that element to `stdout`.

```
printed :: Show a ⇒ [a] → LogicT IO a
printed xs = LogicT (λc n → foldr (λx xs → do putStr (show x); c x xs) n xs)
```

We can evaluate a LogicT with the following function:

```
evalLogicT :: Monad m
           ⇒ LogicT m a → m [a]
evalLogicT ls = runLogicT
  ls (λx → fmap (x:)) (return [ ])
```

```
>>> evalLogicT (printed [1,2,3])
123
[1,2,3]
```

Luckily, many of the hyperfunction combinators can be adapted to this monadic setting. For instance, the ◂ function (Eq.(6)) has the following monadic variant:

```
(◂ₘ) :: Monad m ⇒ (m a → b) → m (m a ↬ b) → (m a ↬ b)
ι (f ◂ₘ h) k = f (ι k =≪ h)
```

Notice that this function preserves the ordering of effects: $h$ is executed before $\iota\ k$. This can be used as a drop-in replacement for ◂, resulting in the following function:

```
interleaveT :: Monad m ⇒ LogicT m a → LogicT m a → LogicT m a
interleaveT xs ys = LogicT (λc n →
  do xz ← runLogicT xs (λx xk → return (c x ◂ₘ xk)) (return (Hyp (const n)))
     yz ← runLogicT ys (λy yk → return (c y ◂ₘ yk)) (return (Hyp (const n)))
     ι xz yz)
```

And again, the effect order is preserved.

```
>>> evalLogicT (interleaveT (printed [1,2]) (printed [3,4]))
1324
[1,3,2,4]
```

### 4.2 A Monadic Language for Concurrency

We now know that hyperfunctions and monads can interface without much ceremony. Let's next look at using hyperfunctions to build an actual monad transformer for concurrency.

We will use Claessen's concurrency monad [1999] for this example, given below by the type *C*.

```
type C m = Cont (Action m)        newtype Cont r a = Cont {runCont :: (a → r) → r}

data Action m = Atom (m (Action m)) | Fork (Action m) (Action m) | Stop
```

A term C *m a* is a concurrent computation that draws effects from *m*. It is built on top of the Cont monad, and has the following interface:

```
atom :: Functor m ⇒ m a → C m a             fork :: C m a → C m ()
atom m = Cont (λk → Atom (fmap k m))        fork m = Cont (λk → Fork (action m) (k ()))
```

*atom* lifts an atomic action into *C*; *fork* runs a process in the background.

The following simple program draws effects from the Writer monad, which allows us to log output, via the *tell* :: String → Writer () function.

```
prog :: C (Writer String) ()
prog = do atom (tell "go!"); fork (forever (atom (tell "to"))); forever (atom (tell "fro"))
```

This program first lifts an action that outputs the string `"go!"`, then, in the background, it repeatedly outputs `"to"`, and then, on the main thread, it repeatedly outputs the string `"fro"`.

We can interpret this language into the underlying effect using *run*:

```
run_c :: Monad m ⇒ C m a → m ()           round :: Monad m ⇒ [Action m] → m ()
run_c c = round [action c]                 round [] = return ()
                                           round (x : xs) = case x of
                                             Atom a_m → a_m ≫= (λa → round (xs ++ [a]))
>>> take 15 (execWriter (run prog))          Fork a_1 a_2 → round (xs ++ [a_1, a_2])
"go!tofrotofroto"                            Stop → round xs
```

*round* here implements round-robin scheduling. However, notice that this function follows the pattern of *foldr* on lists: if we proceed by mechanically fusing away the intermediate list (similarly to our approach in Section 2.4), we arrive at a hyperfunction-based implementation. Below, we have packaged up that implementation into a type called Conc.

```
type Conc r m = Cont (m r ↬ m r)                 atom_h :: Monad m ⇒ m a → Conc r m a
                                                 atom_h a_m = Cont (λk → id ◃_m (k <$> a_m))
fork_h :: Conc r m a → Conc r m ()
fork_h m =                                       run_h :: Conc r m a → m r
  Cont (λk → runCont m (const id) ∘ k ())        run_h c = run (runCont c (const id))
```

This language has the same operations as C. It demonstrates how hyperfunctions can be a building block for a "concurrency monad", when used in combination with the continuation monad. This monad is a monad transformer [Jones 1995], where *atom* corresponds to the *lift* function.

## 5 Coroutines

We have now seen a few small examples of how hyperfunctions might be used in a functional programming language to implement concurrency as a monadic effect. This section will explore a larger example: we will see that hyperfunctions underpin important optimisations in practical coroutine libraries, and then we will see how to use hyperfunctions to build a new, powerful library for asymmetric coroutines.

### 5.1 Pipes

Coroutines are a broad concept, with many different implementations; in Haskell alone, the machines [Kmett 2025], conduit [Snoyman 2011], and Pipes [Gonzalez 2012] libraries all present different interpretations of the abstraction. Though they differ in their details, these libraries are all built around a central coroutine-like object, which is a kind of computation that can be paused and resumed, and can communicate by sending and receiving data. For our purposes, we will take the specific interface described by Gonzalez [2012] and Blažević [2011].

$$\textbf{newtype}\ \mathsf{Pipe}\ r\ i\ o\ m\ a \qquad \mathit{yield} :: o \to \mathsf{Pipe}\ r\ i\ o\ m\ () \qquad \mathit{halt} :: m\ r \to \mathsf{Pipe}\ r\ i\ o\ m\ x$$

$$\mathit{await} :: \mathsf{Pipe}\ r\ i\ o\ m\ i \qquad \mathit{merge} :: \mathsf{Pipe}\ r\ i\ x\ m\ \bot \to \mathsf{Pipe}\ r\ x\ o\ m\ \bot \to \mathsf{Pipe}\ r\ i\ o\ m\ a$$

A value of type $\mathsf{Pipe}\ r\ i\ o\ m\ a$ is a coroutine that takes input of type $i$, outputs $o$s, performs effects in $m$, has a final result type $r$, and intermediate result of type $a$. *yield* produces an output; *await* requests an input; *halt* ends the computation; and *merge* joins two Pipes, connecting corresponding *yield*s and *await*s.

Early implementations of this interface were written in direct style: the Pipe type was represented by an inductive, tree-like data type (a variant of the free monad), and each function was defined by pattern-matching on that type. However, as Spivey noted [2017], this direct-style implementation can suffer from a slow-down when pipes are deeply nested. Unfortunately, the usual trick of CPS-encoding everything turns out to be much more difficult to apply than it might first appear. The problem lies with the *merge* function. Just like the *zip* function on lists, *merge* processes two sequences in lock-step, and also just like *zip*, it becomes much more difficult to implement when those sequences are CPS-encoded: *merge* is a *lateral* function.

Spivey's solution (further explained by Pieters and Schrijvers [2019]) uses the following encoding of a Pipe that is an intricate variant of the Cont monad (Section 4.2), given below.

$$\begin{aligned}&\textbf{newtype}\ \mathsf{Pipe}\ r\ i\ o\ m\ a = \mathsf{MkPipe} \\ &\quad ((a \to \mathsf{Result}\ (m\ r)\ i\ o) \to \mathsf{Result}\ (m\ r)\ i\ o)\end{aligned} \qquad \begin{aligned}&\textbf{type}\ \mathsf{Result}\ r\ i\ o \\ &\quad = \mathsf{InCont}\ r\ i \to \mathsf{OutCont}\ r\ o \to r\end{aligned}$$

A Result takes two continuations before returning the final computation $m\ r$: the InCont is called when the Pipe requests input (of type $i$, with *await*), and the OutCont when the Pipe emits some $o$ (with *yield*).

$$\begin{aligned}&\textbf{newtype}\ \mathsf{InCont}\ r\ i &&= \mathsf{MkInCont}\ &&\{\mathit{resumeIn} :: \mathsf{OutCont}\ r\ i \to r\} \\ &\textbf{newtype}\ \mathsf{OutCont}\ r\ o &&= \mathsf{MkOutCont}\ &&\{\mathit{resumeOut} :: o \to \mathsf{InCont}\ r\ o \to r\}\end{aligned}$$

It is not difficult to see that, after flipping the arguments to *resumeOut*, these types are structurally identical to a specialisation of hyperfunctions.

$$\begin{aligned}&\mathsf{OutCont}\ r\ o \\ &\quad \equiv o \to \mathsf{InCont}\ r\ o \to r \\ &\quad \simeq \mathsf{InCont}\ r\ o \to o \to r \\ &\quad \equiv (\mathsf{OutCont}\ r\ i \to r) \to o \to r \\ &\quad \equiv r \looparrowright (o \to r)\end{aligned} \qquad \begin{aligned}&\mathsf{InCont}\ r\ i \\ &\quad \equiv \mathsf{OutCont}\ r\ i \to r \\ &\quad \equiv (i \to \mathsf{InCont}\ r\ i \to r) \to r \\ &\quad \simeq (\mathsf{InCont}\ r\ i \to i \to r) \to r \\ &\quad \equiv (i \to r) \looparrowright r\end{aligned}$$

In fact, we can see that these two constructions are actually instances of the Consumer and Producer types (Eqs. (10) and (11)), where

$$\mathsf{OutCont}\ r\ o \simeq \mathsf{Consumer}\ o\ r \qquad \mathsf{InCont}\ r\ i \simeq \mathsf{Producer}\ i\ r$$

Much like how these types enabled us to implement message-passing in a CPS-encoded *zip* in Section 2.4, they allowed Spivey to implement message-passing for CPS-encoded Pipes.

Spivey was not the only author to independently rediscover the hyperfunction type while working with Pipe-like abstractions. Shivers and Might's encoding of transducers [2006] includes the same structure (although a large portion of their work is untyped, so the hyperfunction structure is a little more difficult to see). Furthermore, Kammar et al.'s work on handlers for algebraic effects [2013] used the following types to implement a handler for Pipes:

$$\textbf{data}\ \mathsf{Prod}\ s\ r = \mathsf{Prod}\ (() \to \mathsf{Cons}\ s\ r \to r) \qquad \textbf{data}\ \mathsf{Cons}\ s\ r = \mathsf{Cons}\ (s \to \mathsf{Prod}\ s\ r \to r)$$

Like InCont and OutCont above, both Prod and Cons are simple rearrangements of the Producer and Consumer types.

## 5.2 First-class Coroutines

The Pipe implementation above has a significant shortcoming: the only way to communicate with a Pipe is to merge it with another Pipe. From inside a Pipe, we can *yield* and *await* to send and receive values, but there are no corresponding functions to communicate from *outside* a Pipe.

$$\begin{aligned} &\mathit{send} :: i \to \mathsf{Pipe}\ r\ i\ o\ m\ a \to \\ &\qquad m\ (\mathsf{Pipe}\ r\ i\ o\ m\ a) \end{aligned} \qquad \begin{aligned} &\mathit{receive} :: \mathsf{Pipe}\ r\ i\ o\ m\ a \to \\ &\qquad m\ (\mathsf{Maybe}\ (o, \mathsf{Pipe}\ r\ i\ o\ m\ a)) \end{aligned}$$

The above putative interface would allow us to pass Pipes around as first-class values, while still communicating with them. *send* passes a value a Pipe, and advances its execution to the next *await*. *receive* "pops" a value from a Pipe. These two functions are necessary for many standard patterns in coroutine programming: if we want to store a pool of coroutines, for instance, and receive one value from each entry, we cannot accomplish this with merging alone.

To build the solution we will take some inspiration from Shivers and Might [2006]. One of the coroutine implementations in their work is built on `Channel`, an SML type:

```
type α cont (* Continuations. *)
datatype (α,β) Channel = Chan of (α * (β,α) Channel) cont
```

Without the continuation machinery of SML we cannot translate this type directly to Haskell; we can, however, adapt it using the Cont monad:

$$\textbf{type}\ \mathsf{Channel}\ r\ \alpha\ \beta \approx \mathsf{Cont}\ r\ (\alpha, \mathsf{Channel}\ r\ \beta\ \alpha) \approx (((\alpha, \mathsf{Channel}\ r\ \beta\ \alpha) \to r) \to r)$$

Notice that the type on the right-hand-side above resembles the Producer hyperfunction (Eq.(11)): it "produces" $\alpha$, and the parameters to Channel swap on recursion, just like a hyperfunction. It's not a perfect match, but it seems like the Haskell analogue of the Channel type is the following:

$$\textbf{type}\ \mathsf{Channel}\ r\ i\ o = (o \to r) \looparrowright (i \to r)$$

We can turn this type into a monad by wrapping it in a continuation:

$$\textbf{newtype}\ \mathsf{Co}\ r\ i\ o\ m\ a = \mathsf{Co}\ \{\mathit{route} :: (a \to \mathsf{Channel}\ (m\ r)\ i\ o) \to \mathsf{Channel}\ (m\ r)\ i\ o\}$$

This type has a lot in common with Pipe from the previous section, with one significant difference: instead of using separate producer and consumer continuations, it has one continuation which both produces and consumes. This means that every input is accompanied by an output: in terms of the interface to this type, this means that *yield* and *await* are combined into one function that outputs a value and waits for an input at the same time.

$$\begin{aligned} &\mathit{yield} :: o \to \mathsf{Co}\ r\ i\ o\ m\ i \\ &\mathit{yield}\ x = \mathsf{Co}\ (\lambda k \to \mathsf{Hyp}\ (\lambda h\ i \to \iota\ h\ (k\ i)\ x)) \end{aligned}$$

The statement *yield* $x$ suspends execution, outputs the value $x$, and awaits input of some type $i$.

We also have the *merge* and *halt* functions from the Pipe interface, and we can also run a coroutine to produce a result. We will not include the implementations for brevity's sake.

So far, so familiar. However, we have not yet implemented *send*. To do so, we turn back to Shivers and Might [2006], where control operators are used to implement a function they call `switch`:

```
val switch : α * (α,β) Channel -> β * (α,β) Channel
fun switch(x, Chan k) = callcc (fn k’ => throw k (x, Chan k’))
```

This function is analogous to *send*: it takes a value of type $\alpha$, and a channel, sends the value to the channel, and returns a response $\beta$ along with the new channel. It does so by using `callcc` (**call** with **c**urrent **c**ontinuation): `callcc (fn k => e)` binds `k` to the continuation that `callcc` was called from. The `throw` function invokes a continuation; so `switch` binds the current continuation to `k'`, and then `throw`s to the continuation contained in the supplied channel, with the current continuation embedded in the new channel.

Unfortunately, Haskell doesn’t have first-class continuations. It does have the continuation *monad*, however, and the MonadCont typeclass [Jones 1995], which supplies a variant of call/cc.

$$\mathit{callCC} :: \mathsf{MonadCont}\ m \Rightarrow ((a \to m\ b) \to m\ a) \to m\ a$$

Using this, we can build a combinator to send values to a coroutine from outside the coroutine.

$$\mathit{send} :: \mathsf{MonadCont}\ m \Rightarrow \mathsf{Co}\ r\ i\ o\ m\ r \to i \to m\ (\mathsf{Either}\ r\ (o, \mathsf{Co}\ r\ i\ o\ m\ r))$$

The function *send* $c\ v$ send a value $v : i$ to the coroutine $c : \mathsf{Co}\ r\ i\ o\ m\ r$, and returns an effectful computation $m\ (\mathsf{Either}\ r\ (o, \mathsf{Co}\ r\ i\ o\ m\ r))$. The returned value can be Left if the coroutine terminates (either by running out of *yields*, or by encountering a *halt*), or it is Right containing the *yielded* value along with the rest of the coroutine.

The implementation of *send* is as follows:

$$\begin{aligned} \mathit{send}\ c\ v = \mathit{callCC}\ \$\ \lambda k \to \mathsf{Left} \mathbin{\langle\$\rangle} \iota\ (&\mathit{route}\ c\ (\lambda x \to \mathsf{Hyp}\ (\lambda\_\ \_ \to \mathit{return}\ x))) \\ &(\mathsf{Hyp}\ (\lambda r\ o \to k\ (\mathsf{Right}\ (o, \mathsf{Co}\ (\mathit{const}\ r)))))\ v \end{aligned}$$

*callCC* supplies a continuation, $k : \mathsf{Either}\ r\ (o, \mathsf{Co}\ r\ i\ o\ m\ r) \to m\ \_$, which can be called to “return” from the computation. Above, it is called from inside a hyperfunction, where it returns the next value supplied to the consumer, and wraps the rest of the hyperfunction. The other branch, the *halt* branch, is called when there are no more values to return. This branch is represented by *return* $x$.

**Sending Without Return.** Given a coroutine of type $\mathsf{Co}\ \bot\ i\ o\ m\ \bot$, we know that it cannot return or exit, because there is no value of type $\bot$ to return or exit with. A variant of *send* makes use of this fact to avoid the need for Either.

$$\begin{aligned} &\mathit{send}' :: \mathsf{MonadCont}\ m \Rightarrow \mathsf{Co}\ \bot\ i\ o\ m\ \bot \to i \to m\ (o, \mathsf{Co}\ \bot\ i\ o\ m\ \bot) \\ &\mathit{send}'\ c\ v = \mathit{either}\ \mathit{absurd}\ \mathit{id} \mathbin{\langle\$\rangle} \mathit{send}\ c\ v \end{aligned}$$

**Execution Order.** Note that the order of execution of effects is slightly unintuitive. When a process *sends* to a coroutine, the coroutine executes up until the *previous yield* statement, and then transfers control back to the caller. Changing the execution order, so that *send* executes up until the *next yield* is not too difficult: the Channel type is replaced with Suspension ($\mathsf{Suspension}\ r\ i\ o = \mathsf{Channel}\ r\ o\ i \to r$), and the coroutine is represented by $i \to \mathsf{Co}\ r\ i\ o\ m\ a$ rather than $\mathsf{Co}\ r\ i\ o\ m\ a$.

**Coroutines with References.** The expression *send* $g\ i$ returns a pair $(o, g')$, where $g'$ is the updated generator; this is a common pattern in Haskell, often encapsulated with the state monad. In our case, we can use references (IORef) to build a clean interface with the following function:

$$\mathit{send}' :: (\mathsf{MonadCont}\ m, \mathsf{MonadIO}\ m) \Rightarrow \mathsf{IORef}\ (i \to \mathsf{Co}\ \bot\ i\ o\ m\ \bot) \to i \to m\ o$$

## 5.3 Stable Marriages

To demonstrate the power of our coroutine implementation, we will now implement the stable marriage example, following the coroutine-based implementation of Allison [1983].

The stable marriage problem [Gale and Shapley 1962] takes two groups of people–described in the original formulation as a group of men and a group of women who wish to be married–and generates a *matching*, where each member of one group is paired with a distinct member of the other. This matching should also be *stable*: given that each individual has a preference ranking for the members of the opposing group, a stable matching is one where no two individuals would prefer to be matched with each other than with their current match.

Allison's algorithm is an elegant encoding of a natural solution to the problem. A coroutine is constructed for each man and each woman, and the "men" propose to the women, in order of the men's preference. If a man's proposal is accepted, his coroutine is suspended. The "women" are coroutines awaiting proposals; if a proposal is better than their current offer they jilt their current fiancé, whose coroutine resumes and then continues to propose to his next choice.

For our encoding of the algorithm, we will have three men (Aaron, Barry, and Conor), and three women (Annie, Betty, and Ciara). Their rankings are as follows:

```
mranks = assoc [ (Aaron, [Ciara, Annie, Betty])      wranks = assoc [ (Annie, [Barry, Conor, Aaron])
               , (Barry, [Ciara, Betty, Annie])                     , (Betty, [Aaron, Barry, Conor])
               , (Conor, [Ciara, Annie, Betty]) ]                   , (Ciara, [Conor, Aaron, Barry]) ]
```

Our encoding of the algorithm has the following type:

```
stable :: Array Man [Woman] → Array Woman [Man] → IO [(Woman, Man)]
```

It takes a pair of rankings, and outputs a list of marriages.

The first step of the algorithm is to initialise the array of engagements:

```
engagements ← liftIO (newArray_ (minBound, maxBound) :: IO (IOArray Woman Man))
```

This will store the current engagements while the algorithm runs. Note that we do not use this for inter-process communication; all communication is done with the *send′* and *yield* functions.

Next, we construct the array of coroutines for men and women:

```
men ← genM (λi → newIORef (man i)); women ← genM (λi → newIORef (woman i))
```

Each coroutine is stored in an array, indexed by the Man and Woman data types.

The next step is to construct a coroutine for a man:

```
man :: Man → () → Co ⊥ () () M ⊥
man me () = do for_ (mranks ! me) $ λwi → do
                    liftIO (printf "%s proposes to %s; " me wi)
                    accept ← lift (send′ (women ! wi) me)
                    when accept (yield ())
                return (error "Unreachable")
```

This function takes an index representing the man that corresponds to the coroutine. Then, it iterates through the man's ranks, and for each it sends a proposal to the corresponding woman (*send′* (*women* ! *wi*) *me*). The response to this message is a Bool saying whether or not the woman has accepted; if she does accept, the man suspends himself (*when accept* (*yield* ())). The end of this loop will never be reached if all preferences are strict total orders, but we cannot prove that in Haskell, so we need to use *error* in the return statement so that the coroutine has return type ⊥.

Then, the women. A "woman" is a coroutine that takes a Man as input (a suitor), and yields Bools as output (responses to marriage proposals).

```
woman :: Woman → Man → Co ⊥ Man Bool M ⊥
woman me suitor = do
  liftIO (printf "%s accepts %s\n" me suitor)
  liftIO (writeArray engagements me suitor)
  yield True ≫ loop (λsuitor → do
    jiltee ← liftIO (readArray engagements me)
    if elemIndex suitor (wranks ! me) < elemIndex jiltee (wranks ! me)
      then do liftIO (printf "%s jilts %s for %s\n" me jiltee suitor)
              liftIO (writeArray engagements me suitor)
              lift (send′ (men ! jiltee) ())
              yield True
      else  do liftIO (printf "%s rejects %s, stays with %s\n" me suitor jiltee)
              yield False)
```

The first suitor is always accepted (*yield* True), after that the coroutine loops, comparing the new suitor to the old, and jilting the old suitor if the new is preferable. If that does happen, the woman will modify the engagements array, notify her jiltee (*send′* (*men* ! *jiltee*)), and respond True to the marriage proposal. If the new suitor is not preferable, she will instead *yield* False.

Finally, to run the algorithm we initiate all of the men and collect the engagments:

```
forAll_ (λi → send′ (men ! i) ()); liftIO (getAssocs engagements)
```

The output of the algorithm is as follows:

```
>>> stable mranks wranks
Aaron proposes to Ciara; Ciara accepts Aaron
Barry proposes to Ciara; Ciara rejects Barry, stays with Aaron
Barry proposes to Betty; Betty accepts Barry
Conor proposes to Ciara; Ciara jilts Aaron for Conor
Aaron proposes to Annie; Annie accepts Aaron
[(Annie,Aaron),(Betty,Barry),(Ciara,Conor)]
```

The final result is [(Annie, Aaron), (Betty, Barry), (Ciara, Conor)], what a happy coincidence that their names match too.

## 6 Related Work

The first research on hyperfunctions was conducted by Launchbury et al., who defined and named the construction in a technical report [2000]. Subsequently, Krstić et al. established the formal basis for hyperfunctions, and developed the coalgebraic interpretation of the type [2001a; 2001b]. In 2013, Launchbury et al. revised and published their earlier technical report; this publication forms the basis of the research contained in this paper.

Outside of the academic literature, Kmett's Haskell library for hyperfunctions [2015] proved extremely helpful for demonstrating some of the more complex patterns of hyperfunction usage. In addition, the first occurrence of a hyperfunction-like type we were able to find was Hofmann's Rou type [1993], which was later studied in more depth by Berger et al. [2019].

The algorithms of Allison [1983, 1989] seem to be quite similarly structured to hyperfunction algorithms, although they do not contain hyperfunctions themselves. In particular, the research of

Smith [2009] on Allison's "corecursive queues" includes a lot of recursion patterns reminiscent of Hofmann's breadth-first traversal [1993].

One of the contentions of this work is that hyperfunctions are already being used throughout the functional programming world by programmers who need to combine continuations and concurrency in certain ways. While we have documented some of these usages [Hofmann 1993; Kammar et al. 2013; Shivers and Might 2006; Spivey 2017], we think it is likely that the pattern is even more widespread. In particular, while most of the examples we have documented are in Haskell, we are much less familiar with the Scheme or Lisp communities, and we think that the prevalence of continuations in those languages would increase the likelihood of rediscoveries of hyperfunctions.

One of the main patterns of usage of hyperfunctions is in efficiently implementing zip-like functions (what we have called "lateral" functions) on CPS-encoded data. The difficulty of implementing this pattern is precisely what Spivey identified in implementing CPS-encoded Pipes [2017]. Pieters and Schrijvers wrote a follow-up to this work [2019], with the intention of simplifying the exposition by systematically deriving Spivey's more efficient implementation. We think that this paper can also help clarify Spivey's intricate type by isolating the hard-to-understand part—the hyperfunction—and demonstrating its use in more simple examples.

While the original motivation for the development of hyperfunctions was in allowing fold-fusion [Gill et al. 1993] to apply to the *zip* function, these days *stream* fusion [Coutts et al. 2007] is able to perform most of the functions of fold-fusion, and has no difficulty in fusing away *zip*.

Our approach to CCS is strongly influenced by Bruni and Montanari [2017]. Early drafts of our model took inspiration (especially for the implementation of the $\|$ operator) from Bahr and Hutton [2023] and Bergstra and Klop [1985]. The canonical model we use (Proc) comes from [Veltri and Vezzosi 2023], whose work was also invaluable for understanding the well-founded implementation of the CCS operations.

Our model of CCS is similar in many ways to the model of Ciobanu and Todoran [2018]. The formal foundation for their model is in metric spaces, however, which differs from ours. While we did not need to use their weak abstractness condition [Ciobanu and Todoran 2017] for our Communicator model of CCS, it is possible that other process calculi (especially those which contain sequencing operators, like ACP Bergstra and Klop [1986], which we were not able to model using hyperfunctions) can only ever have weakly abstract continuation models.

## 7 Conclusion

> In the early history of continuations, basic concepts were independently discovered an extraordinary number of times. This was due less to poor communication among computer scientists than to the rich variety of settings in which continuations were found useful [Reynolds 1993]

Hyperfunctions, like continuations, have been rediscovered multiple times. Wherever concurrency and continuations intersect, authors have used hyperfunctions "to open up apparently closed doors" [Launchbury et al. 2013]. Despite their many uses, however, hyperfunctions have remained obscure and under-studied. This paper has demonstrated that hyperfunctions are powerful and broadly useful: we hope that our work sheds more light on hyperfunctions, facilitates their more widespread use, and spurs further research on these curious beasts.

## Acknowledgments

We would like to thank the reviewers for their feedback, which significantly improved the paper. We would also like to thank Jeremy Gibbons, whose notes were invaluable.

## References

Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. 2013. Copatterns: Programming Infinite Structures by Observations. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*. Association for Computing Machinery, New York, NY, USA, 27–38. doi:10.1145/2429069.2429075

Samson Abramsky and Achim Jung. 1995. Domain Theory. In *Handbook of Logic in Computer Science*, S Abramsky, Dov M Gabbay, and T S E Maibaum (Eds.). Oxford University PressOxford, 1–168. doi:10.1093/oso/9780198537625.003.0001

Lloyd Allison. 1983. Stable Marriages by Coroutines. *Inform. Process. Lett.* 16, 2 (Feb. 1983), 61–65. doi:10.1016/0020-0190(83)90025-X

Lloyd Allison. 1989. Circular Programs and Self-Referential Structures. *Software: Practice and Experience* 19, 2 (Feb. 1989), 99–109. doi:10.1002/spe.4380190202 arXiv:2403.01866 [cs]

Patrick Bahr and Graham Hutton. 2023. Calculating Compilers for Concurrency. 7, ICFP (Aug. 2023), 213:740–213:767. doi:10.1145/3607855

Ulrich Berger, Ralph Matthes, and Anton Setzer. 2019. Martin Hofmann's Case for Non-Strictly Positive Data Types. In *24th International Conference on Types for Proofs and Programs (TYPES 2018) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 130)*, Peter Dybjer, José Espírito Santo, and Luís Pinto (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 22. doi:10.4230/LIPIcs.TYPES.2018.1

Jan Bergstra and Jan Willem Klop. 1984. Process Algebra for Communication and Mutual Exclusion. R 8409 (Jan. 1984). https://ir.cwi.nl/pub/6504

J. A. Bergstra and J. W. Klop. 1985. Algebra of Communicating Processes with Abstraction. *Theoretical Computer Science* 37 (Jan. 1985), 77–121. doi:10.1016/0304-3975(85)90088-X

Jan A Bergstra and Jan Willem Klop. 1986. Algebra of Communicating Processes. *Mathematics and Computer Science, CWI Monograph* 1, 89-138 (1986), 9.

Mario Blažević. 2011. Coroutine Pipelines. *The Monad.Reader* 19 (Oct. 2011), 29–48. https://themonadreader.wordpress.com/wp-content/uploads/2011/10/issue19.pdf

Roberto Bruni and Ugo Montanari. 2017. CCS, the Calculus of Communicating Systems. In *Models of Computation*. Springer International Publishing, Cham, 221–270. doi:10.1007/978-3-319-42900-7_11

Nicolas Chappe, Paul He, Ludovic Henrio, Yannick Zakowski, and Steve Zdancewic. 2023. Choice Trees: Representing Nondeterministic, Recursive, and Impure Programs in Coq. *Proceedings of the ACM on Programming Languages* 7, POPL (Jan. 2023), 61:1770–61:1800. doi:10.1145/3571254

Gabriel Ciobanu and Eneia Nicolae Todoran. 2017. Abstract Continuation Semantics for Asynchronous Concurrency. In *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. 296–303. doi:10.1109/SYNASC.2017.00056

Gabriel Ciobanu and Eneia Nicolae Todoran. 2018. On the Abstractness of Continuation Semantics. In *2018 20th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. 167–174. doi:10.1109/SYNASC.2018.00036

Koen Claessen. 1999. A Poor Man's Concurrency Monad. *Journal of Functional Programming* 9, 3 (May 1999), 313–323. doi:10.1017/S0956796899003342

Thierry Coquand. 1994. Infinite Objects in Type Theory. In *Types for Proofs and Programs*, Henk Barendregt and Tobias Nipkow (Eds.). Springer, Berlin, Heidelberg, 62–78. doi:10.1007/3-540-58085-9_72

Thierry Coquand. 2013. [Agda] Defining Coinductive Types. https://lists.chalmers.se/pipermail/agda/2013/006189.html

Duncan Coutts, Roman Leshchinskiy, and Don Stewart. 2007. Stream Fusion: From Lists to Streams to Nothing at All. *ACM SIGPLAN Notices* 42, 9 (Oct. 2007), 315–326. doi:10.1145/1291220.1291199

Haskell B. Curry. 1942. The Inconsistency of Certain Formal Logics. *The Journal of Symbolic Logic* 7, 3 (Sept. 1942), 115–117. doi:10.2307/2269292

D. Gale and L. S. Shapley. 1962. College Admissions and the Stability of Marriage. *The American Mathematical Monthly* 69, 1 (1962), 9–15. doi:10.2307/2312726

Jeremy Gibbons, Donnacha Oisín Kidney, Tom Schrijvers, and Nicolas Wu. 2022. Breadth-First Traversal via Staging. In *Mathematics of Program Construction*, Ekaterina Komendantskaya (Ed.). Springer International Publishing, Cham, 1–33. doi:10.1007/978-3-031-16912-0_1

Andrew Gill. 2022. Mtl: Monad Classes for Transformers, Using Functional Dependencies. https://hackage.haskell.org/package/mtl

Andrew Gill, John Launchbury, and Simon L. Peyton Jones. 1993. A Short Cut to Deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA '93)*. Association for Computing Machinery, New York, NY, USA, 223–232. doi:10.1145/165180.165214

Gabriella Gonzalez. 2012. Pipes: Compositional Pipelines. http://hackage.haskell.org/package/pipes-3.0.0

Thomas Harper. 2011. A Library Writer's Guide to Shortcut Fusion. In *Proceedings of the 4th ACM Symposium on Haskell (Haskell '11)*. Association for Computing Machinery, New York, NY, USA, 47–58. doi:10.1145/2034675.2034682

Christopher T. Haynes, Daniel P. Friedman, and Mitchell Wand. 1986. Obtaining Coroutines with Continuations. *Computer Languages* 11, 3 (Jan. 1986), 143–153. doi:10.1016/0096-0551(86)90007-X

Brandon Hewer and Graham Hutton. 2024. Quotient Haskell: Lightweight Quotient Types for All. *Proceedings of the ACM on Programming Languages* 8, POPL (Jan. 2024), 785–815. doi:10.1145/3632869

R. Hieb and R. Kent Dybvig. 1990. Continuations and Concurrency. *ACM SIGPLAN Notices* 25, 3 (March 1990), 128–136. doi:10.1145/99164.99178

Ralf Hinze, Thomas Harper, and Daniel W. H. James. 2011. Theory and Practice of Fusion. In *Implementation and Application of Functional Languages*, Jurriaan Hage and Marco T. Morazán (Eds.). Springer, Berlin, Heidelberg, 19–37. doi:10.1007/978-3-642-24276-2_2

Martin Hofmann. 1993. Non Strictly Positive Datatypes in System F. https://www.seas.upenn.edu/~sweirich/types/archive/1993/msg00027.html

Graham Hutton. 1998. Fold and Unfold for Program Semantics. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*. ACM, Baltimore Maryland USA, 280–288. doi:10.1145/289423.289457

Geraint Jones and Jeremy Gibbons. 1993. *Linear-Time Breadth-First Tree Algorithms: An Exercise in the Arithmetic of Folds and Zips*. Technical Report 71. Dept of Computer Science, University of Auckland. http://www.cs.ox.ac.uk/people/jeremy.gibbons/publications/linear.ps.gz

Mark P. Jones. 1995. Functional Programming with Overloading and Higher-Order Polymorphism. In *Advanced Functional Programming*, Gerhard Goos, Juris Hartmanis, Jan Leeuwen, Johan Jeuring, and Erik Meijer (Eds.). Vol. 925. Springer Berlin Heidelberg, Berlin, Heidelberg, 97–136. doi:10.1007/3-540-59451-5_4

Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in Action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming - ICFP '13*. ACM Press, Boston, Massachusetts, USA, 145. doi:10.1145/2500365.2500590

Oleg Kiselyov, Chung-chieh Shan, Daniel P. Friedman, and Amr Sabry. 2005. Backtracking, Interleaving, and Terminating Monad Transformers: (Functional Pearl). In *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming (ICFP '05)*. ACM, New York, NY, USA, 192–203. doi:10.1145/1086365.1086390

Edward Kmett. 2015. Hyperfunctions: Hyperfunctions. https://hackage.haskell.org/package/hyperfunctions

Edward Kmett. 2025. Machines. https://hackage.haskell.org/package/machines

Sava Krstić, John Launchbury, and Duško Pavlović. 2001a. Categories of Processes Enriched in Final Coalgebras. In *Foundations of Software Science and Computation Structures (Lecture Notes in Computer Science)*, Furio Honsell and Marino Miculan (Eds.). Springer, Berlin, Heidelberg, 303–317. doi:10.1007/3-540-45315-6_20

Sava Krstić, John Launchbury, and Duško Pavlović. 2001b. Hyperfunctions. In *FICS 2001 Workshop on Fixed Points in Computer Science*. Firenze, Italy. http://wwwusers.di.uniroma1.it/~labella/partecipation.html

John Launchbury, Sava Krstić, and Timothy E. Sauerwein. 2000. *Zip Fusion with Hyperfunctions*. Technical Report. Oregon Graduate Institute. https://launchbury.blog/wp-content/uploads/2019/01/zip-fusion-with-hyperfunctions.pdf

John Launchbury, Sava Krstić, and Timothy E. Sauerwein. 2013. Coroutining Folds with Hyperfunctions. *Electronic Proceedings in Theoretical Computer Science* 129 (Sept. 2013), 121–135. doi:10.4204/EPTCS.129.9 arXiv:1309.5135

Robin Milner, G. Goos, J. Hartmanis, W. Brauer, P. Brich Hansen, D. Gries, C. Moler, G. Seegmüller, J. Stoer, and N. Wirth (Eds.). 1980. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science, Vol. 92. Springer, Berlin, Heidelberg. doi:10.1007/3-540-10235-3

Rasmus Ejlers Møgelberg and Niccolò Veltri. 2019. Bisimulation as Path Type for Guarded Recursive Types. *Proceedings of the ACM on Programming Languages* 3, POPL, Article 4 (Jan. 2019), 4:1–4:29 pages. doi:10.1145/3290317

Faron Moller. 1990a. The Importance of the Left Merge Operator in Process Algebras. In *Automata, Languages and Programming*, Michael S. Paterson (Ed.), Vol. 443. Springer, Berlin, Heidelberg, 752–764. doi:10.1007/BFb0032072

F. Moller. 1990b. The Nonexistence of Finite Axiomatisations for CCS Congruences. In *[1990] Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science*. 142–153. doi:10.1109/LICS.1990.113741

Peter D. Mosses. 2010. Programming Language Description Languages. In *Formal Methods: State of the Art and New Directions*, Paul Boca, Jonathan P. Bowen, and Jawed Siddiqi (Eds.). Springer, London, 249–273. doi:10.1007/978-1-84882-736-3_8

Ulf Norell. 2009. Dependently Typed Programming in Agda. In *AFP 2008*, Pieter Koopman, Rinus Plasmeijer, and Doaitse Swierstra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 230–266. doi:10.1007/978-3-642-04652-0_5

Ruben P. Pieters and Tom Schrijvers. 2019. Faster Coroutine Pipelines: A Reconstruction. In *Practical Aspects of Declarative Languages (Lecture Notes in Computer Science)*, José Júlio Alferes and Moa Johansson (Eds.). Springer International Publishing, Cham, 133–149. doi:10.1007/978-3-030-05998-9_9

John C. Reynolds. 1993. The Discoveries of Continuations. *LISP and Symbolic Computation* 6, 3 (Nov. 1993), 233–247. doi:10.1007/BF01019459

Olin Shivers and Matthew Might. 2006. Continuations and Transducer Composition. *ACM SIGPLAN Notices* 41, 6 (June 2006), 295–307. doi:10.1145/1133255.1134016

Vilhelm Sjöberg. 2015. Why Must Inductive Types Be Strictly Positive? https://vilhelms.github.io/posts/why-must-inductive-types-be-strictly-positive/

Leon P Smith. 2009. Lloyd Allison's Corecursive Queues: Why Continuations Matter. *The Monad.Reader* 14, 14 (July 2009), 28. https://meldingmonads.files.wordpress.com/2009/06/corecqueues.pdf

Michael Snoyman. 2011. Conduit. https://hackage.haskell.org/package/conduit-0.0.0.1

Michael Spivey. 2017. Faster Coroutine Pipelines. *Proceedings of the ACM on Programming Languages* 1, ICFP (Aug. 2017), 5:1–5:23. doi:10.1145/3110249

Eneia Todoran. 2000. Metric Semantics for Synchronous and Asynchronous Communication: A Continuation-based Approach. *Electronic Notes in Theoretical Computer Science* 28 (Jan. 2000), 101–127. doi:10.1016/S1571-0661(05)80632-2

Niccolò Veltri and Andrea Vezzosi. 2020. Formalizing $\pi$-Calculus in Guarded Cubical Agda. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2020)*. Association for Computing Machinery, New York, NY, USA, 270–283. doi:10.1145/3372885.3373814

Niccolò Veltri and Andrea Vezzosi. 2023. Formalizing CCS and $\pi$-Calculus in Guarded Cubical Agda. *Journal of Logical and Algebraic Methods in Programming* 131 (Feb. 2023), 100846. doi:10.1016/j.jlamp.2022.100846

Philip Wadler. 1995. Monads for Functional Programming. In *Advanced Functional Programming (Lecture Notes in Computer Science)*, Johan Jeuring and Erik Meijer (Eds.). Springer, Berlin, Heidelberg, 24–52. doi:10.1007/3-540-59451-5_2

## A Hofmann's Rou Type and Algorithms

Previously (Section 2.5) we did not have the space to fully explain Hofmann's Rou type and algorithm, so we will do so here. First, Hofmann's original algorithm, here adapted to rose trees:

```
bfeH :: Tree a → [a]                                  data Tree a = a & [Tree a]
bfeH t = extract (br t Over)
  where
    br :: Tree a → Rou [a] [a] → Rou [a] [a]
    br (x & xs) fw = Next (λbw → x : unfold fw (bw ∘ flip (foldr br) xs))

extract :: Monoid a ⇒                                 unfold :: Rou a a →
             Rou a a → a                                          (Rou a a → a) → a
extract (Next k) = k extract                          unfold (Next k) h = k h
extract Over      = ϵ                                 unfold Over      h = h Over
```

It's a little difficult to see what's going on with this algorithm, although we can see that there are two functions for "unwrapping" a Rou value, which each handle the Over case differently. The *extract* function terminates recursion when it encounters Over, whereas the *unfold* function continues with the continuation it was supplied. Unfortunately, the *br* function is somewhat impenetrable.

One way we can understand this implementation is by redefining it in terms of combinators similar to those we used to implement subtraction and *zip* (Sections 2.3 and 2.4). The implementation of breadth-first traversal itself will follow the levels-based "zippy" style of Jones and Gibbons [1993].

```
bfez :: Tree a → [a]                                  lz [ ] ys = ys
bfez t = concat (walk t)                              lz xs [ ] = xs
  where                                               lz (x : xs) (y : ys) = (x ++ y) : lz xs ys
    walk :: Tree a → [[a]]
    walk (x & xs) =
      [x] : foldr lz [ ] (map walk xs)
```

This algorithm performs a breadth-first enumeration by building a list of levels in the tree; sibling sub-trees are combined with the *lz* ("long zip") function. Though declarative, this function is not linear, which the Rou version can help fix.

In the Rou version, the Rou [*a*] [*a*] type plays the role of the list-of-lists. To replicate $bfe_z$ above fully, then, we will need to implement the analogues of the functions *concat*, *lz*, and the list constructors cons and nil. The nil constructor is easy: it corresponds to Over. The other three functions actually already have analogues on hyperfunctions, from Section 2.3: *concat* corresponds to *run*, *lz* to ⊙, and cons to ◂. We can adapt these from hyperfunctions to Rou as follows: *run* goes to *extract* above, and composition/zipping and cons are defined as follows:

```
(•) :: Rou a a → Rou a a → Rou a a                   (◂) :: (a → b) → Rou a b → Rou a b
Over   • g = g                                        f ◂ h = Next (λk → f (k h))
Next f • g = Next (λh1 →
  f (λh2 → unfold g (λh3 → h1 (h2 • h3))))
```

Finally, with all of these operators defined, we can define a version of *bfe* that mirrors the structure of $bfe_z$, but executes in linear time, using the Rou type.

```
bfer :: Tree a → [a]                                  walk :: Tree a → Rou [a] [a]
bfer t = extract (walk t)                             walk (x & xs) =
  where                                                 (x:) ◂ foldr (•) Over (map walk xs)
```

## B Pipes and Coroutines

Here we will provide the full implementation of Pipes from Sections 5.1 and 5.2.

We use the following basic hyperfunctions definitions from elsewhere in the paper:

```
newtype a ↬ b = Hyp {ι :: (b ↬ a) → b}

type Producer o a = (o → a) ↬ a            type Consumer i a = a ↬ (i → a)
```

Also, both types we define here will have monad instances that follow the pattern of the Cont monad, given below:

```
newtype Cont r a = Cont {runCont :: (a → r) → r}

instance Monad (Cont r) where
  return x = Cont (λk → k x)
  xs ≫= k = Cont (λk′ → runCont xs (λx → runCont (k x) k′))
```

We also use the MonadCont typeclass [Gill 2022], which we will not include here.

### B.1 Pipes

This is a reimplementation of Spivey's continuation-based Pipes, now explicitly using the hyperfunction type.

The Pipe type itself:

```
newtype Pipe r i o m a = MkPipe                        type Result r i o =
  ((a → Result (m r) i o) → Result (m r) i o)           Producer i r → Consumer o r → r

unPipe :: Pipe r i o m a → (a → Result (m r) i o) → Result (m r) i o
unPipe (MkPipe k) = k
```

This type has a Monad instance that is the same as Cont above. It also has a MonadTrans [Jones 1995] instance:

```
instance MonadTrans (Pipe r i o) where
  lift :: ∀m a.Monad m ⇒ m a → Pipe r i o m a
  lift m = MkPipe (λk p c → m ≫= λm′ → k m′ p c)
```

The interface:

```
yield :: o → Pipe r i o m ()
yield o = MkPipe (λk p c → ι c (Hyp (k () p)) o)

await :: Pipe r i o m i
await = MkPipe (λk p c → ι p (Hyp (λp′ x → k x p′ c)))

halt :: m r → Pipe r i o m x
halt x = MkPipe (λ_ _ _ → x)

merge :: Pipe r i x m ⊥ → Pipe r x o m ⊥ → Pipe r i o m a
merge ix xo = MkPipe (λ_ p c → unPipe xo absurd (Hyp (unPipe ix absurd p)) c)

runPipe :: Pipe r () () m () → m r
runPipe xs = unPipe xs (const ι) (rep (λk → k ())) (rep const)
```

Once the monad instance is implemented on this type (and the monad transformer, and the relevant mtl classes) following Cont above, it is possible to write programs like the following:

```
lhs :: Pipe () () Int IO ⊥
lhs = do
  liftIO (printf "entered lhs\n")
  yield 1
  liftIO (printf "yielded 1\n")
  yield 2
  liftIO (printf "yielded 2\n")
  halt (pure ())
```

```
rhs :: Pipe () Int () IO ⊥
rhs = do
  liftIO (printf "entered rhs\n")
  x ← await
  liftIO (printf "received %d\n" x)
  y ← await
  liftIO (printf "received %d\n" y)
  halt (pure ())
```

```
>>> runPipe (merge lhs rhs)
entered rhs
entered lhs
received 1
yielded 1
received 2
```

## B.2 Coroutine

This is the implementation of the coroutine from Section 5.2.

$$\textbf{type}\ \text{Channel}\ r\ i\ o = (o \to r) \looparrowright (i \to r) \qquad \textbf{type}\ \text{Suspension}\ r\ i\ o = \text{Channel}\ r\ o\ i \to r$$

$$\textbf{newtype}\ \text{Co}\ r\ i\ o\ m\ a = \text{Co}\ \{ route :: (a \to \text{Suspension}\ (m\ r)\ i\ o) \to \text{Suspension}\ (m\ r)\ i\ o \}$$

Again, this type is a Monad and MonadTrans. We can also implement MonadCont, because this type has the same structure as Cont.

The following is the interface:

```
merge :: Co r i x m ⊥ → (x → Co r x o m ⊥) → Co r i o m a
merge xs ys = Co (λ_ h → route xs absurd (Hyp (λh′ x → route (ys x) absurd (h ∘ h′))))

yield :: o → Co r i o m i
yield x = Co (λk h → ι h (Hyp (flip k)) x)

await :: MonadCont m ⇒ Co r i o m r → m (Either r (o, i → Co r i o m r))
await c = callCC $ λk →
  Left <$> route c (λx _ → return x)
                  (Hyp (λh o → k (Right (o, λi → Co (λ_ s → ι h s i)))))

await′ :: MonadCont m ⇒ Co ⊥ i o m ⊥ → m (o, i → Co ⊥ i o m ⊥)
await′ c = either absurd id <$> await c

runCo :: Co r i i m i → m r
runCo c = route c idS id
```

And here is an example program:

```
gen :: MonadIO m ⇒ Co ⊥ Int Int m ⊥
gen = forever $ do
  liftIO (putStrLn "Entered gen")
  x ← yield 1
  liftIO (putStrLn ("yielded 1, received: " ++ show x))
```

```
    x ← yield 2
    liftIO (putStrLn ("yielded 2, received: " ++ show x))
    x ← yield 3
    liftIO (putStrLn ("yielded 3, received: " ++ show x))

prog :: Co ⊥ Int Int (ContT () IO) ⊥ → ContT () IO ()
prog gen = do
    liftIO (putStrLn "Entered prog")
    (x, kgen) ← await′ gen
    liftIO (putStrLn ("received: " ++ show x))
    (x, kgen) ← await′ (kgen 10)
    liftIO (putStrLn ("sent 10, received: " ++ show x))
    (x, _) ← await′ (kgen 20)
    liftIO (putStrLn ("sent 20, received: " ++ show x))
```

```
>>> runContT (prog gen) return
Entered prog
Entered gen
received: 1
yielded 1, received: 10
sent 10, received: 2
yielded 2, received: 20
sent 20, received: 3
```

## B.3 Reference-Based Coroutine

The coroutine implementation we use in Section 5.3 is designed to be used with an IORef. Here are the new implementations (where ResumeChannel is the name of the module for Co above).

```
await :: (MonadCont m, MonadIO m) ⇒ IORef (Co r i o m r) → m (Either r (o, i → m ()))
await c =
    liftIO (readIORef c) ≫=
    fmap (fmap (fmap (fmap (liftIO ∘ writeIORef c)))) ∘
    ResumeChannel.await

send :: (MonadCont m, MonadIO m) ⇒ IORef (i → Co r i o m r) → i → m (Either r o)
send c i = do
    cval ← liftIO (readIORef c)
    ResumeChannel.await (cval i) ≫= λcase
        Right (o, k) → do liftIO (writeIORef c k)
            return (Right o)
        Left r → return (Left r)

await′ :: (MonadCont m, MonadIO m) ⇒ IORef (Co ⊥ i o m ⊥) → m (o, i → m ())
await′ = fmap (either absurd id) ∘ await

send′ :: (MonadCont m, MonadIO m) ⇒ IORef (i → Co ⊥ i o m ⊥) → i → m o
send′ c = fmap (either absurd id) ∘ send c
```

## C Proofs

In this appendix, we will provide the proof of the main theorem of the paper: Theorem 3.12. This proof follows from two main lemmas, Lemmas 3.10 and 3.11. We will also prove Lemma 3.9.

All of these proofs can also be found in our Agda formalisation. In fact, the structure of this section follows the structure of the Agda proofs closely, and we will make reference to the formalisation throughout. As such, this section can be used as a guide to the formalisation.

**Structure.** The main theorem that we aim to prove is Theorem 3.12.

$$\forall p, q.\ p \sim q \iff [\![p]\!]_{\text{Communicator}} \equiv [\![q]\!]_{\text{Communicator}}$$

The statement of this theorem can be found in the mechanisation in the file `CCS/Homomorphism.agda`. This relies on the combination of three other facts: Lemmas 3.10 and 3.11 and Theorem 3.7. This combination is described in Section 3.4, here we will prove the first of these two lemmas. The third component, Theorem 3.7, has already been proven by Veltri and Vezzosi [2023], so here we take it as a given, and give it as a postulate in the mechanisation. We also specialise the variable names to $\mathbb{N}$ in this proof: we need this to pick fresh variables given a finite AST.

*Remark C.1 (The Proc Type).* Veltri and Vezzosi [2023] have already formalised much of the Proc type, so we do not repeat their work here. However, we do include a basic replication of their type in our formalisation (found in the file `CCS/Proc.agda`). While Veltri and Vezzosi employ sophisticated techniques to prove well-foundedness for their model of CCS, unfortunately those techniques are not sufficient in our case when the Communicator type is involved.

In this section, and in our mechanisation, the Proc type is represented by nested finite sets. This type has three constructors: one for the empty process ($\emptyset$ : Proc $n$), one for the singleton process ($\wr\_\int$ : Act $n \times$ Proc $n \to$ Proc $n$), and one for the union of two processes ($\cup$ : Proc $n \to$ Proc $n \to$ Proc $n$). The usual quotients (i.e. the laws of a commutative, idempotent monoid) apply.

In this appendix, we will first give a short proof of Lemma 3.9, and then we will describe the structure of the proofs of Lemmas 3.10 and 3.11, before giving the proofs in more detail.

Proof of Lemma 3.9. $\forall (p : \text{Proc}\ n).\ [\![[\![p]\!]\!\uparrow_{\text{Communicator}\ n\ (\text{Proc}\ n)}]\!]\!\downarrow_{\text{Proc}\ n} \equiv p$

In our formalisation, this proof is given as *proc⟶com⟶proc*, in `CCS/Homomorphism.agda`. The proof itself proceeds by pattern-matching on $p$:

- $p := \emptyset$ Then identity here holds definitionally, since $[\![\emptyset]\!]\!\uparrow\ \equiv \mathbb{0}$, and $[\![\mathbb{0}]\!]\!\downarrow\ \equiv \iota\ \mathbb{0}\ \mathbb{1}\ \mathfrak{q} \equiv \mathbb{0}$.
- $p := \wr a, q \int$ This similarly applies largely definitionally, however we apply the proof recursively to $q$, which is allowed since it is guarded under $a$.
- $p := q \cup r$ Again, most of this applies definitionally. We do have to call the proof recursively, on $q$ and $r$, but this does not need to be guarded because recall that the set is *finite*.

□

Proofs of Lemmas 3.10 and 3.11.

$$\forall (p : \text{P}\ n).\ [\![[\![p]\!]_{\text{Communicator}}]\!]\!\downarrow\ \equiv [\![p]\!]_{\text{Proc}} \qquad \forall (p : \text{P}\ n).\ [\![[\![p]\!]_{\text{Proc}}]\!]\!\uparrow\ \equiv [\![p]\!]_{\text{Communicator}}$$

Both of these proofs proceed quite similarly. Their proofs in our mechanisation can be found in the file `CCS/Homomorphism.agda`; the details of Lemma 3.10 are in `CCS/Homomorphism/CtoP.agda`; and the details of Lemma 3.11 are in `CCS/Homomorphism/PtoC.agda`. Since their structure is quite similar, we will proceed by focusing on Lemma 3.10; Lemma 3.11 is effectively the same proof mirrored.

As explained in Section 3.4, we will first rewrite the CCS term in question into the common form $\nu_s\ [] \cdot (p \mathbin{\|\!\lfloor} \mathbb{0})$. Then, from here we proceed by applying Lemma 3.15.

We have already explained our high-level argument for well-foundedness; specifically for this proof, each of the sub-lemmas Lemmas 3.15, C.2 and C.3 inductively recurse on one of their arguments. In other words, each of these proofs is guaranteed to only take finite time before transferring the proof burden to the next lemma in the chain. The Lemmas C.2 and C.3 then may refer back to Lemma 3.15: this call is *not* inductive or guaranteed to terminate, but it is *guarded* in each case. □

Proof of Lemma 3.15. $\forall n, p, q.\ [\![ [\![ \nu_s n.(p \mathbin{\|\!\lfloor} q) ]\!]_{\mathsf{Communicator}} ]\!]\!\downarrow\ \equiv [\![ \nu_s n.(p \mathbin{\|\!\lfloor} q) ]\!]_{\mathsf{Proc}}$

This proof proceeds by induction on $p$. Since $p$ is a finite, inductive term, recursion on subterms of $p$ does not need justification via guardedness. In our mechanisation, this proof can be found in the file `CCS/Homomorphism/CtoP.agda`, with the name $\lfloor$*-hom*$\downarrow$.

The proof proceeds by induction on $p$. The first few cases are simple, consisting basically of rewriting and then recursion on subterms.

$p \coloneqq q \oplus r$ Both $\nu$ and $\mathbin{\|\!\lfloor}$ distribute over $\oplus$, so we distribute and proceed recursively.

$p \coloneqq q \parallel r$ We rewrite $p \parallel q \equiv (p \mathbin{\|\!\lfloor} q) \oplus (q \mathbin{\|\!\lfloor} p)$, and then proceed with the corresponding other cases.

$p \coloneqq q \mathbin{\|\!\lfloor} r$ We apply the pseudo-associativity of $\mathbin{\|\!\lfloor}$, rewriting $[\![ [\![ \nu_s n.((p \mathbin{\|\!\lfloor} q) \mathbin{\|\!\lfloor} r) ]\!] ]\!]\!\downarrow$ to $[\![ [\![ \nu_s n.(p \mathbin{\|\!\lfloor} (q \parallel r)) ]\!] ]\!]\!\downarrow$, and proceeding on $p$.

$p \coloneqq {!\,q}$ We rewrite using ${!\,p} \equiv p \mathbin{\|\!\lfloor} {!\,p}$, and recurse on $p$.

$p \coloneqq \mathbb{0}$ This holds definitionally.

The last two cases are more interesting.

$p \coloneqq \nu m \cdot q$ For this identity, we first take a fresh name, which we will call $f$. Since $f \notin q$ (because $f$ is fresh), we have $\nu f \cdot q \equiv q$. Furthermore, we can rename $m$ to $f$ in $p$, and the identity $\nu f \cdot [m/f]p \equiv \nu s \cdot p$ holds. Finally, we also have the identity $\nu n \cdot p \mathbin{\|\!\lfloor} \nu n \cdot q \equiv \nu n \cdot (\nu n \cdot p \mathbin{\|\!\lfloor} \nu n \cdot q)$. Combined, all of this lets us rewrite $[\![ [\![ \nu_s n.((\nu m \cdot p) \mathbin{\|\!\lfloor} q) ]\!] ]\!]\!\downarrow$ to $[\![ [\![ \nu_s (f : n).([m/f]p \mathbin{\|\!\lfloor} q) ]\!] ]\!]\!\downarrow$. Then, we can proceed by recursing on $[m/f]p$. Though $[m/f]p$ is not strictly a subterm of $\nu m \cdot p$, for the purposes of well-foundedness it suffices, because the size of the $[m/f]p$ is the same as $p$.

$p \coloneqq a \cdot q$ Recall that $(a \cdot p) \mathbin{\|\!\lfloor} q \equiv \mathit{sync}_{io}\ p\ q \oplus \mathit{step}_l\ p\ q$ holds on both Communicator and Proc; we can apply this identity and proceed on both sides of $\oplus$ with Lemmas C.2 and C.3.

□

Lemma C.2. $\forall n, a, p, q.\ [\![ \nu_s n.\ \mathit{step}_l\ [\![ a \cdot p ]\!]\ [\![ q ]\!] ]\!]\!\downarrow\ \equiv \nu_s n.\ \mathit{step}_l\ [\![ a \cdot p ]\!]\ [\![ q ]\!]$

Proof. This proof can be found in our mechanisation in the file `CCS/Homomorphism/CtoP.agda`, named *step-hom*$\downarrow$. There are two cases for this proof: if $a$ is contained in the list of names $n$, then both sides evaluate to $\mathbb{0}$, and the proof is done. Otherwise, the term reduces to $a \cdot [\![ \nu_s n.\ [\![ p ]\!] \parallel [\![ q ]\!] ]\!]\!\downarrow$. From here, we can rewrite $p \parallel q \equiv (p \mathbin{\|\!\lfloor} q) \oplus (q \mathbin{\|\!\lfloor} p)$, distribute over $\oplus$, and recurse by applying Lemma 3.15. This recursion is allowed, since we are guarded under $a$. □

Lemma C.3. $\forall n, a, p, q.\ [\![ \nu_s n.\ \mathit{sync}_{io}\ [\![ a \cdot p ]\!]\ [\![ q ]\!] ]\!]\!\downarrow\ \equiv \nu_s n.\ \mathit{sync}_{io}\ [\![ a \cdot p ]\!]\ [\![ q ]\!]$

Proof. This proof can be found in our mechanisation in the file `CCS/Homomorphism/CtoP.agda`, named *sync-hom*$\downarrow$. Similarly to Lemma C.2, if $a$ is present in $n$, then both sides evaluate to $\mathbb{0}$. Also, if $a \coloneqq \tau$ or $a \coloneqq \overline{n}$ then both sides also evaluate to $\mathbb{0}$. Finally, we pattern-match on $q$, following a similar pattern of distribution and rewriting to Lemma 3.15. When $q \coloneqq b \cdot r$, where $b$ is a communication match with $a$, (i.e. $a \coloneqq \underline{n}$ and $b \coloneqq \overline{n}$), then we emit a $\tau$, and proceed guarded under that action calling back to Lemma 3.15. □

Lemma C.4. $p \mathbin{||\!\!\_} \mathbb{0} \equiv p$

Proof. This holds on Communicator simply:

$$\begin{aligned}
&\iota\ (p \mathbin{||\!\!\_} \mathbb{0})\ q\ m \equiv \\
&\iota\ p\ (\mathbb{0} \mathbin{||} q)\ m \equiv \quad \{\text{Lemma C.6}\} \\
&\iota\ p\ q\ m
\end{aligned}$$

On Proc:

$$\begin{aligned}
&p \mathbin{||\!\!\_} \mathbb{0} &&\equiv \\
&\mathit{sync}_{io}\ p\ \mathbb{0} \oplus \mathit{step}_l\ p\ \mathbb{0} &&\equiv \\
&\mathit{step}_l\ p\ \mathbb{0} &&\equiv \\
&p
\end{aligned}$$

□

Lemma C.5. $\forall p.\ !\,p \equiv p \mathbin{||\!\!\_}\ !\,p$

Proof. We prove this identity on Proc

$$\begin{aligned}
&!\ p && \equiv \quad \{\text{Eq.(31)}\} \\
&\mathit{step}_l\ (p \oplus \mathit{sync}_{io}\ p\ p)\ (!\ p) && \equiv \\
&\mathit{step}_l\ p\ (!\ p) \oplus \mathit{step}_l\ (\mathit{sync}_{io}\ p\ p)\ (!\ p) && \equiv \\
&\mathit{step}_l\ p\ (!\ p) \oplus \mathit{sync}_{io}\ p\ (\mathit{step}_l\ p\ (!\ p)) && \equiv \\
&\mathit{step}_l\ p\ (!\ p) \oplus \mathit{sync}_{io}\ p\ (\mathit{step}_l\ p\ (!\ p)) \oplus \mathbb{0} && \equiv \\
&\mathit{step}_l\ p\ (!\ p) \oplus \mathit{sync}_{io}\ p\ (\mathit{step}_l\ p\ (!\ p)) \oplus \mathit{sync}_{io}\ p\ (\mathit{sync}_{io}\ p\ (\mathit{step}_l\ p\ (!\ p))) && \equiv \\
&\mathit{step}_l\ p\ (!\ p) \oplus \mathit{sync}_{io}\ p\ (\mathit{step}_l\ p\ (!\ p)) \oplus \mathit{sync}_{io}\ p\ (\mathit{step}_l\ (\mathit{sync}_{io}\ p\ p)\ (!\ p)) && \equiv \\
&\mathit{step}_l\ p\ (!\ p) \oplus \mathit{sync}_{io}\ p\ (\mathit{step}_l\ p\ (!\ p) \oplus \mathit{step}_l\ (\mathit{sync}_{io}\ p\ p)\ (!\ p)) && \equiv \quad \{\text{Lemma C.7}\} \\
&\mathit{step}_l\ p\ (!\ p) \oplus \mathit{sync}_{io}\ p\ (!\ p) && \equiv \\
&\mathit{sync}_{io}\ p\ (!\ p) \oplus \mathit{step}_l\ p\ (!\ p) && \equiv \\
&p \mathbin{||\!\!\_}\ !\ p
\end{aligned}$$

□

Lemma C.6. $\mathbb{0} \mathbin{||} p \equiv p$

Proof. Since $\mathbb{0} \mathbin{||\!\!\_} p \equiv \mathbb{0}$ holds by definition on both Communicator and Proc, this follows from Lemma C.4. □

Lemma C.7. $\forall a, p, q, r.\ \mathit{sync}_{io}\ (a \cdot p)\ (q \oplus r) \equiv \mathit{sync}_{io}\ (a \cdot p)\ q \oplus \mathit{sync}_{io}\ (a \cdot p)\ r$

Proof. This again applies on Proc because bind distributes over $+\!\!+$. □