Give or take

Non-Determinism, Linear Logic and Session Types

Pepijn Kokke
CDT Pervasive Parallelism
pepijn.kokke@ed.ac.uk

January 9, 2017

Abstract

The aim of this project is to extend the session-typed π-calculus CP (Wadler, 2012) with *global* non-determinism—i.e. without using an explicit program construct for non-determinism—while preserving the strong guarantees of termination and deadlock freedom that are offered by the correspondence with classical linear logic.

Our extended version of CP will allow us to model more complex concurrent processes, such as an online ticket vendor which can sell some finite number of tickets, with customers racing to obtain a ticket. We will be able to model further complications, such as a ticket vendor for which all tickets are distinct—e.g. by being labeled with distinct serial numbers—and vendors who maintain a list detailing which customer bought which ticket. All of these are open problems for logic-inspired session-typed systems which guarantee termination.

1 Introduction

Curry-Howard correspondences are a powerful tool in the design of programming languages. Once such a correspondence between a programming language and a logical system is established, we can leverage any property of the logic system as a property of the programming language—and vice versa. One such property is termination. In logic, it is crucial that cut elimination—which corresponds to *computation*—terminates. We can leverage this property to state that all our programs terminate.

In practice, strong properties such as termination tend to be both a blessing and a curse. We know that our programs will never enter an infinite loop—a blessing, as such bugs are arguably difficult to debug. However, it also restricts one of the programmers most powerful tools—general recursion. There are many restricted forms of recursion which are known to terminate. Examples are primitive recursive functions, structural recursion (Burstall, 1969), Walther recursion (Walther, 1994), and sized types (Lee et al., 2001). Whether programming in a total function language is feasible depends largely on whether the form of recursion is sufficiently natural to programmers. However, if we do not have such a restricted form of recursion, and a proof that it terminates, then we are nowhere.

The π-calculus (Milner et al., 1992) is a process calculus—a formal system which models concurrent computation. It is arguably as foundational as the λ-calculus, in the sense that it is equally bare bones: it does not come equipped with any notions of data types or values, but consists solely of communication channels and processes. A Curry-Howard correspondence between a variant of the π-calculus and linear logic has only recently been discovered (Caires and Pfenning, 2010; Wadler, 2012). In these two systems, there is a tight correspondence between the typing rules and the inference rules of intuitionistic and classical linear logic, respectively, and between computation (or communication) and cut elimination. It is the latter, once again, which turns out to be both a curse and a blessing.

One property which cut elimination tends to have is determinism—if we apply cut elimination to the same proof, we always get the same result. However, in the context of concurrent computation this is *not* what we want. We may want to model non-deterministic computations—e.g. races. One example of such a race, which we will use throughout this text, is a pâtisserie. Imagine a pâtisserie which only has *one* cake left in store. Now imagine two customers, leaving their houses to go any buy a cake at exactly the same time. Who will end up with a cake, and who will get a plate of disappointment? This depends entirely on outside conditions—who lives closer, who gets struck in traffic, etc—and, importantly, it does not depend on either customer or the pâtisserie.

In this text we will propose an extension of CP (Wadler, 2012) which is capable of modeling such interactions without losing the strong properties which the correspondence with classical linear logic gives us. A consequence of this is that we will have to make cut elimination non-deterministic. In what follows, we will describe how we will extend classical linear logic, based on the work on bounded linear logic (BLL; Girard et al., 1992), to allow for some margin of non-determinism.

2 Background

Before we delve into our proposed solution, it would be wise to briefly discuss related work. In the following sections, we will briefly touch on the syntax and semantics of the π-calculus, the session-typing paradigm, which has led to a correspondence between a typed π-calculus and linear logic, current solutions to non-determinism in such π-calculi, and bounded linear logic.

2.1 The π-calculus

The π-calculus is a process calculus, introduced by Milner et al. (1992). It models processes communicating on named channels. The terms of the π-calculus are defined as follows, over an infinite set of names which we denote by lowercase letters.

$$
\begin{array}{lll}
S, P, Q, R & ::= & \\
& \nu x.P & \text{create new channel x, then run P} \\
& x[y].P & \text{send y on channel x, then run P} \\
& x(y).P & \text{receive y on channel x, then run P} \\
& P \mid Q & \text{run P and Q in parallel} \\
& 0 & \text{halt}
\end{array}
$$

Formally, the reduction of π-calculus terms is defined using a structural congruence, which itself is defined as an extension over renaming.

$$
\begin{gathered}
P \mid 0 \equiv P \quad P \mid Q \equiv Q \mid P \quad (P \mid Q) \mid R \equiv P \mid (Q \mid R) \\
(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P \quad (\nu x)0 \equiv 0 \\
(\nu x)(P \mid Q) \equiv (\nu x)P \mid Q \quad (x \text{ is not free in } Q)
\end{gathered}
$$

These equivalences state that parallel composition is commutative and associative, and has the inactive process as its unit; that the order of channel creation does not matter; that creating a channel for the inactive process is pointless; and that it is harmless to extend the scope of a channel to include Q if the channel is not used in Q. Reduction is then defined with the following rule

$$
x[z].P \mid x(y).Q \longrightarrow P \mid Q\{z/y\} \quad \text{(communication)}
$$

Together with three other rules which state that we can perform communication under parallel composition and name restriction, and that we can rewrite by structural congruence.

The terms discussed so far would be sufficient for a discussion of the π-calculus. However, Wadler (2012) uses some non-standard constructs in his formulation of CP. With the next section in mind, we discuss these extensions as well.

There are three main groups of extensions. The first of these is the link construct, $x \leftrightarrow y$, which forwards messages from x to y, or vice versa, depending on the direction of information flow. The second is binary choice. Such a choice involves two processes—one offering a choice, and one making it.

$x[\text{inl}].P$	choose left on x, then run P
$x[\text{inr}].P$	choose right on x, then run P
case x $\{P; Q\}$	receive choice on X, then run P or Q

Lastly, Wadler (2012) uses empty versions of communication and choice. These will be used as the interpretations of the unit types of linear logic. Empty communication corresponds to sending a ping and halting, or receiving a ping.

$x[].0$	send ping on x, then halt
$x().P$	wait for ping on x, then run P

Empty choice will be a nullary choice. Since there are no options in a nullary choice, we will have no construct to send one. The empty case construct, which offers a nullary choice, will consequently have to wait forever.

case x $\{\ ;\}$	empty choice

2.2 Session Types and Linear Logic

Linear logic was discovered by Girard, in 1987, as a generalization of classical and intuitionistic logic, and from the beginning it held great promise for computer science. Limiting the unbounded reuse of propositions, it is a logic of resources and usages. Therefore, it could—and has since—be used to describe programming languages which regulate access to memory. Furthermore, it has always held the promise of being a logic of communication. Both of these are made apparent in the language which Girard (1987) uses to describe his logic. When he speaks about the exponentials, he speaks about storage and registers. Yet he names one of the operators par ($\parr$), and talks about programs asking questions and giving answers.

Honda (1993), inspired by linear logic, developed session types. While his session-typed functional programming language is linear, and it uses two of linear logic's connectives, there is no direct correspondence. In fact, it is not until 2010 that Caires and Pfenning establish a tight correspondence between the π-calculus and intuitionistic linear logic.

Two years later, Wadler (2012) establishes a similar correspondence between the π-calculus and classical linear logic. He calls the resulting system CP. In the same paper, he ties this work in with the branch of research on session-typed functional languages, spawned by Honda (1993), using a variant of the language developed by Gay and Vasconcelos (2009), which he calls GV.

In this text we will focus on CP, as we believe that a process calculus and a classical type system are the right language to describe communication. In Figure 1, we show the typing rules for the rudimentary fragment of CP—that is, the fragment without unbounded communication or polymorphism. This leaves us with the four binary connectives—tensor, par, plus and with—and their units—one, bottom, zero and top, respectively.

As described above, CP has a tight correspondence with linear logic. One consequence of this is that communication is identified with cut elimination. The problem with this tight correspondence is that communication is deterministic.

2.3 Linearity and Non-Determinism

There are several existing approaches which add non-determinism to linear typed process calculi. However, all of these approaches are centered around a shared idea: local choice. Let us take the work by Atkey et al. (2016) as an example. They add two new typing rules.

$$
\frac{P \vdash \Gamma \qquad Q \vdash \Gamma}{P + Q \vdash \Gamma} \qquad \frac{}{\text{fail} \vdash \Gamma}
$$

The first rule is local choice: it joins two processes P and Q of the same type, and when run will non-deterministically

$$\text{Type } A, B \coloneqq A \mid A^{\perp} \mid 1 \mid A \otimes B \mid \perp \mid A \parr B \mid 0 \mid A \oplus B \mid \top \mid A \,\&\, B$$

$$\frac{}{x{\leftrightarrow}y \vdash x\colon A, y\colon A^{\perp}}\ \textsc{Ax} \qquad \frac{P \vdash \Gamma, x\colon A \qquad Q \vdash \Delta, x\colon A^{\perp}}{\nu x.(P \mid Q) \vdash \Gamma, \Delta}\ \textsc{Cut}$$

$$\frac{P \vdash \Gamma, y\colon A \qquad Q \vdash \Delta, x\colon B}{x[y].(P \mid Q) \vdash \Gamma, \Delta, x\colon A \otimes B}\ \otimes \qquad \frac{P \vdash \Gamma, y\colon A, xB}{x(y).P \vdash \Gamma, x\colon A \parr B}\ \parr$$

$$\frac{P \vdash \Gamma, x\colon A}{x[\text{inl}].P \vdash \Gamma, x\colon A \oplus B}\ \oplus_1 \qquad \frac{P \vdash \Gamma, x\colon B}{x[\text{inr}].P \vdash \Gamma, x\colon A \oplus B}\ \oplus_2 \qquad \frac{P \vdash \Gamma, x\colon A \qquad Q \vdash \Delta, x\colon B}{\text{case } x\ \{P; Q\} \vdash \Gamma, \Delta}\ \&$$

$$\frac{}{x[].0 \vdash x\colon 1}\ 1 \qquad \frac{P \vdash \Gamma}{x().P \vdash \Gamma, x\colon \perp}\ \perp \qquad \text{(no rule for 0)} \qquad \frac{}{\text{case } x\ \{;\} \vdash x\colon \top}\ \top$$

Figure 1: Typing rules for rudimentary CP.

choose to run one of them.

$$P + Q \longrightarrow P$$
$$P + Q \longrightarrow Q$$

The second rule models failure—an optional unit for local choice. From the perspective of logic, the inclusion of failure has obvious negative consequences. Indeed, Atkey et al. (2016) show that in the presence of local choice and failure, we get proofs of $\top \multimap 0$ and $A \,\&\, B \multimap A \oplus B$.

Caires (2014); Caires and Pérez (2017) use a similar approach. The difference in their work is that they embed local choice and failure in a monad, represented by the exponentials $\&A$ and $\oplus A$.

$$\frac{P \vdash \Gamma, x\colon A}{x[\text{some}].P \vdash \Gamma, x\colon \&A} \qquad \frac{P \vdash \&\Gamma, x\colon A}{x(\text{some}).P \vdash \&\Gamma, x\colon \oplus A}$$

$$\frac{P \vdash \&\Gamma \qquad Q \vdash \&\Gamma}{P + Q \vdash \&\Gamma} \qquad \frac{}{x[\text{none}].0 \vdash x\colon \&A}$$

The first two rules setup a pair of exponentials, and the latter two correspond to local choice and failure.

Why do we want to avoid local choice? Let us go back to our example. Recall that we would like to model a pâtisserie which has one remaining cake, and two customers racing in order to purchase it. We *can* encode this example using local choice. We write the program for the pâtisseur as the local choice between

$$x[\text{cake}].y[\text{disappointment}].0$$
$$+$$
$$y[\text{cake}].x[\text{disappointment}].0$$

where x and y are the communication channels with the two customers.

However, this does not model our race. This models the scenario where our pâtiseur decides which customer gets to purchase the cake on a whim, and then waits for that customer regardless of whether the other customer has already arrived at the store. This is not what we intended—and worse, it is *rude*.

As we emphasized earlier, the responsibility for choosing who gets the cake should not lie with the pâtiseur, nor should it lie with either customer.

2.4 Bounded Linear Logic

Bounded linear logic (BLL) was introduced by Girard et al. in 1992, with the intention of demonstrating the feasibility of type systems which guarantee a certain complexity. Specifically, it is a restricted version of intuitionistic linear logic, in which all programs have polynomial time complexity. It achieves this by requiring that the number of memory accesses is bounded by a polynomial. This is done by replacing the usual rules for the exponentials with the following bounded variants (which are presented here *clasically*, in order to make their subsequent integration into CP easier to follow).

$$\frac{\vdash ?_m\Gamma, A}{\vdash ?_{mn}\Gamma, !_n A}\ \textsc{Store} \qquad \frac{\vdash \Gamma}{\vdash \Gamma, ?_0 A}\ \textsc{Weaken}$$

$$\frac{\vdash \Gamma, ?_n A, ?_m A}{\vdash \Gamma, ?_{n+m} A}\ \textsc{Contract} \qquad \frac{\vdash \Gamma, A}{\vdash \Gamma, ?_1 A}\ \textsc{Derelict}$$

It is clear that Girard et al. (1992) use the exponentials to model memory. In fact, Girard (1987) already describes the unbounded exponential $!A$ in such terms:

> An answer of type $!A$ corresponds to the idea of writing an answer of type A in some stable register so that it can be used *ad nauseam*.

In BLL, the only difference is that when we write a value to a register, we specify how many times that value will be read, and the type system ensures that a stored value will be read exactly as many times as we intended.

In the context of session types, $!A$ and $?A$ are traditionally used to model servers and clients, though their semantics remain unchanged. This results in servers acting like stable registers—sending out the same value over and over again.

3 Methodology

In this section, we will discuss which changes we will make to CP to be able to model non-deterministic computation. In addition, we will sketch process semantics for these constructs.

3.1 Give and take

Our aim is to add some construct for global non-determinism to CP. In the π-calculus, the non-determinism comes from

two things: channel sharing and the structural congruence relation. This allows you to write a process such as e.g.

$$(x[u].x[v].P \mid x(y).Q \mid x(z).R)$$

This process is non-deterministic, because it is not determined a priori which value sent along x will be received by which process.

In CP, however, channels are strictly binary—all communication is introduced by the Cut-rule, which creates a channel with two endpoints, and hands each endpoint to *one* process (see Figure 1). Furthermore, CP defines its reduction semantics on *typed* processes, and its structural congruence is weaker than that of the π-calculus. As a consequence, computation in CP is completely deterministic. There, we plan to extend CP with types which capture safe name sharing. We do this by extending CP with a pair of bounded exponentials, modeled on those of bounded linear logic. These are $?_n A$ (give) and $!_n A$ (take), where $n \in \mathbb{N}$. We refer to the resulting system as $\mathrm{CP}_{\mathrm{ND}}$.

Our interpretation of $?_n A$ remains close to BLL—it represents a process which communicates as A, n times. We depart from BLL, however, with our interpretation of $!_n A$. Instead of interpreting $!_n A$ as a stable register, which gives out the same value n times, we take $!_n A$ to model a *pool* of n independent processes, each of which communicates as A in its own way. It is this *pooling* of processes which allows for name sharing, and therefore non-determinism.

$$\frac{P \vdash \Gamma, x\colon ?_m A, x'\colon ?_n A}{P\{x/x'\} \vdash \Gamma, x\colon ?_{m+n} A}\ \textsc{Contract}$$

$$\frac{P \vdash ?_n \Gamma, y\colon A}{x(y).P \vdash ?_n \Gamma, x\colon !_1 A}\ !_1 \qquad \frac{P \vdash \Gamma, y\colon A}{x[y].P \vdash \Gamma, x\colon ?_1 A}\ ?$$

$$\frac{P \vdash \Gamma, x\colon !_m A \qquad Q \vdash \Delta, x\colon !_n A}{P \mid Q \vdash \Gamma, \Delta, x\colon !_{m+n} A}\ \textsc{Pool}$$

The trick to non-determinism is in our proposed cut elimination procedure. When faced with a cut on a channel of type $?_n A^{\perp} / !_n A$, the cut elimination procedure will pick an arbitrary process from the pool of processes $!_n A$ for each successive communication of the process $?_n A^{\perp}$. Thus, we rewrite such a cut into n smaller cuts.

An interesting consequence of this is that if the endsequent does not contain any channels of type $?_n A$ or $!_n A$, then we should be able to eliminate *all* usage of the rules for non-determinism. This is good—after running a non-deterministic program, we should expect to end up in some deterministic end state.

4 Relation to bounded linear logic

The system $\mathrm{CP}_{\mathrm{ND}}$ differs in some fundamental ways from BLL. First and foremost, this is obvious in our chosen model: BLL models programs with restricted memory access, whereas $\mathrm{CP}_{\mathrm{ND}}$ models non-deterministic communicating processes. Directly related to this is the fact that $\mathrm{CP}_{\mathrm{ND}}$ is classical, whereas BLL is intuitionistic.

However, there are some more subtle difference as well. In some sense, $!_1$ and Pool are a generalization of Store. We can show this by deriving Store for $n \geq 1$.

$$\dfrac{\dfrac{\begin{array}{c}\text{IH}\\ \vdots\\ \vdash ?_{mn}\Gamma, !_{mn}A\end{array} \qquad \begin{array}{c}\rho\\ \vdots\\ \vdash ?_m\Gamma, !_m A\end{array}}{\vdash ?_{mn}\Gamma, ?_m\Gamma, !_{m(n+1)}A}\ \textsc{Pool}}{\vdash ?_{m(n+1)}\Gamma, !_{m(n+1)}A}\ \textsc{Contract}$$

We emulate a stable register, which can be accessed n times to obtain the value ρ, by pooling together n copies of the process ρ. The obvious implication of this emulation is that we actually *store* n copies, instead of letting a single copy be accessed n times.

One the other hand, we cannot derive Store for $n = 0$. We also do not have weakening. If we wish $\mathrm{CP}_{\mathrm{ND}}$ to be an extension of BLL, we should add two more rules.

$$\frac{P \vdash ?_n \Gamma, y\colon A}{0 \vdash ?_0 \Gamma, x\colon !_0 A}\ !_0 \qquad \frac{P \vdash \Gamma}{P \vdash \Gamma, x\colon ?_0 A}\ \textsc{Weaken}$$

These rules model *irrelevance*. This becomes more obvious once we look at the processes associated with $!_0$: it checks the type of a process only to throw it out.

We believe that these rules will play no role in the cut elimination of the *relevant* portion of $\mathrm{CP}_{\mathrm{ND}}$, and therefore choose to leave them out, restricting the exponentials to the case where $n \geq 1$ However, they may become interesting in more expressive versions of CP, such as one which adds dependent types (see McBride, 2016)

4.1 A pâtisserie and its customers

We can use the new connectives to implement non-deterministic programs. We will discuss an example: a pâtisserie, and its customers. To keep things simple, we will model a pâtisserie which has exactly *one* cake to sell, and we will model the scenario where there are *two* customers racing to buy that cake. However, the rules presented in subsection 3.1 can model such interactions with an arbitrary number of cakes and an arbitrary number of customers.

In Figure 2, we show two programs involved in our example. The first process represents a customer. Customers want a channel over which they either can buy a cake—trading a coin for a cake—or which tells them when cake is unavailable. This is represented using the type $(\textsc{Coin} \otimes \textsc{Cake}^{\perp}) \,\&\, \perp$. The second process is the pâtisserie itself. It offers up a channel of type $?_2(\textsc{Coin}^{\perp} \parr \textsc{Cake}) \oplus 1$—a channel over which two customers can try to purchase a cake simultaneously.

To run the program, we pool two customers together, and have them communicate with the store through cut elimination. Note that the responsibility of choosing which customer gets the cake does not lie with the pâtisserie—instead, it is decided by cut elimination. The cut on $x\colon ?_n((\textsc{Coin}^{\perp} \parr \textsc{Cake}) \oplus 1)$ is replaced by two smaller cuts on the channels x and x'—the two channels that were merged using Contract (see Figure 2). Both these channels have the same type, and therefore it is up to the cut elimination procedure to decide which of these channels is cut together with which client channel.

It may strike the reader as odd to see the pâtisserie represented using $?_n A$ instead of $!_n A$. In the literature on linear logic inspired session types, the type $!A$, and not $?A$, is traditionally taken to represent servers (Caires and Pfenning, 2010; Wadler, 2012). We argue that this is no mistake on our part. In both of these works, the term corresponding to the

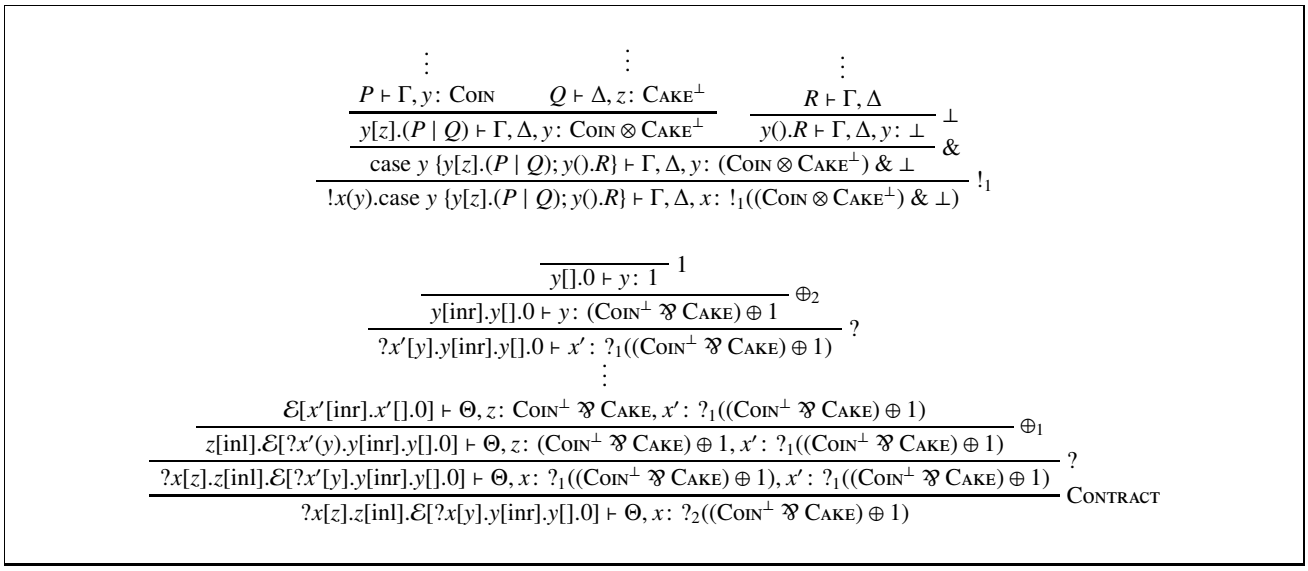

Figure 2: A pâtisserie and its customers.

server type $!A$ is an input—a *blocking* read. This is perfectly in line with our conception of a web server, which waits for clients to send it a request. However, in a pâtisserie, it is the customer who waits in line until the pâtissier has time for them.[1]

4.2 Take tensors, give pars

The bounded modalities give ($?_nA$) and take ($!_nA$) have a strong connection to par ($\parr$) and tensor ($\otimes$)—they are vectors of pars and tensors.[2] We can demonstrate this showing that we can fold a series of pars into a give.

$$\dfrac{\dfrac{\dfrac{}{\vdash A^\perp, A}\,\text{Ax}}{\vdash A^\perp, ?_1A}\,? \qquad \begin{matrix}\text{IH}\\ \vdots\\ \vdash A^\perp \otimes \ldots \otimes A^\perp, ?_nA\end{matrix}}{\dfrac{\vdash A^\perp \otimes A^\perp \otimes \ldots \otimes A^\perp, ?_1A, ?_nA}{\vdash A^\perp \otimes A^\perp \otimes \ldots \otimes A^\perp, ?_{n+1}A}\,\textsc{Contract}}\,\otimes$$

And vice versa—we can unfold a give into a series of pars.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash A^\perp, A}\,\text{Ax}}{\vdash A^\perp, ?_1A}\,? \qquad \begin{matrix}\text{IH}\\ \vdots\\ \vdash !_nA^\perp, ?_1A \parr \ldots \parr ?_1A\end{matrix}}{\vdash !_1A^\perp, ?_1A}\,!_1}{\dfrac{\vdash !_{n+1}A^\perp, ?_1A, ?_1A \parr \ldots \parr ?_1A}{\vdash !_{n+1}A^\perp, ?_1A \parr ?_1A \parr \ldots \parr ?_1A}\,\parr}\,\textsc{Pool}$$

This second proof feels a little unsatisfactory, due to the presence of the $?_1A$ on the unfolded values. We will discuss this in subsection 4.3. For now, observe that we can lift all values in the first proof into $?_1$, and obtain the following equivalence.

$$?_nA \multimap ?_1A \parr \ldots \parr ?_1A.$$

We can give similar proofs for take and tensor. Thus, we also have

$$!_nA \multimap !_1A \otimes \ldots \otimes !_1A.$$

And lastly, if we included empty vector, using $!_0$ and Weaken, we could extend this correspondence to

$$?_0A \multimap \perp \quad \text{and} \quad !_0A \multimap 1.$$

4.3 Conflating $?_1A$ and $!_1A$

In the previous section, we have demonstrated the strong connection between give and take, and tensor and par. However, the final equivalences were unsatisfying, as we could only unfold a value $?_nA$ to n values of type $?_1A$. It would be much more pleasing if we had

$$?_nA \multimap A \parr \ldots \parr A \quad \text{and} \quad !_nA \multimap A \otimes \ldots \otimes A.$$

And we already have one direction of these proofs! Why not the other? The reason lies with the $!_1$ rule: it requires every value in its context to be under a $?_n$ exponential. Why is this the case? In unbounded linear logic, this is necessary to preserve linearity—otherwise, programs usable *ad nauseam* could depend on linear resources. In bounded linear logic, it makes it possible to succinctly express the Store rule, and separates reads from writes. However, in CP_{ND}, we believe there is no good reason to keep this separation.

Given our description of $?_nA$ and $!_nA$ as vectors of pars and tensors, one would expect that we have $?_1A \multimap !_1A$. For unit vectors, it should not matter whether we build them with tensors or pars, as we will never have need for either! Therefore, we propose an alternative version of $!_1$ which simply drops this restriction.

$$\dfrac{P \vdash \Gamma, y\colon A}{x(y).P \vdash \Gamma, x\colon A}\,!_1$$

When we add this rule, we conflate $?_1A$ and $!_1A$. We could already derive $!_1A \multimap ?_1A$, but now we also have the converse.

$$\dfrac{\dfrac{}{x{\leftrightarrow}y \vdash y\colon A^\perp, x\colon A}\,\text{Ax}}{z(x).w(y).x{\leftrightarrow}y \vdash w\colon !_1A^\perp, z\colon !_1A}\,!_1(2)$$

This conflation leaves $?_nA$ and $!_nA$, for $n > 1$, untouched. We believe it to be a benign conflation, and will use it in the remainder of this text whenever we mention $!_1$ or CP_{ND}.

[1] This does not take away the feeling that perhaps $?_nA$ and $!_nA$ are a misnomer, and they should be referred to as $\parr_nA$ and $\otimes_nA$ instead—see subsection 4.2.

[2] The exponentials $?A$ and $!A$ have similar relations to par and tensor, except that they represent unbounded streams of pars and tensors, instead of bounded vectors, and that objects in a $!A$ stream are identical.

4.4 Conflating $?_n A$ and $!_n A$

Give and take impose a strict structure on the kinds of non-deterministic which we can model. Specifically, we always need one server process which offers to communicate n times, and n independent client processes There is a good reason for this—it ensures that after cut elimination, we are left with a single, central process—the server process. However, this may not always be what we desire. Imagine a matchmaking service. This could be a dating site, an online market place—any process whose purpose it is to link *two* pools of independent processes. In such a scenario, it is entirely reasonable to write programs which end up as several independent processes. For instance,

$$(x[u].P \mid x[v].Q \mid x(y).R \mid x(z).S)$$

In the above program, the names in P, Q, R and S are disjoint, aside from x. This means that whichever way we choose to communicate along x, we will end up with two completely independent processes. Logically, allowing such processes corresponds to adding the Mix-rule.

$$\frac{P \vdash \Gamma \qquad Q \vdash \Delta}{P \mid Q \vdash \Gamma, \Delta}\ \textsc{Mix}$$

If we add Mix to our type system, we get proofs of $\bot \multimap 1$ and $A \otimes B \multimap A \parr B$. Vice versa, if we have a proof of $\bot \multimap 1$, we can derive Mix. This phenomenon was recently investigated in the context of CP by Atkey et al. (2016). Unsurprisingly, due to the relation between $?_n/!_n$ and $\parr/\otimes$, Mix also gives us $!_n A \multimap ?_n A$.

$$\dfrac{\dfrac{\dfrac{}{\vdash A^\perp, A}\ \textsc{Ax}}{\vdash ?_1 A^\perp, ?_1 A}\ ?(2) \qquad \begin{array}{c}\textsc{IH}\\ \vdots\\ \vdash ?_n A^\perp, ?_n A\end{array}}{\dfrac{\vdash ?_1 A^\perp, ?_1 A, ?_n A^\perp, ?_n A}{\vdash ?_{n+1} A^\perp, ?_{n+1} A}\ \textsc{Cont}(2)}\ \textsc{Mix}$$

This relation holds the other way around as well—if we have a proof of $?_2 1$, we can derive Mix.

From Atkey et al. (2016), we also know that if we add MultiCut, we get $1 \multimap \bot$ and $A \parr B \multimap A \otimes B$ (and vice versa).

$$\frac{\vdash \Gamma, A_1^\perp, \ldots, A_n^\perp \qquad \vdash \Delta, A_1, \ldots, A_n}{\vdash \Gamma, \Delta}\ \textsc{MultiCut}$$

Adding MultiCut also gives us $?_n A \multimap !_n A$, see Figure 3, resulting in a proof of $!_n A \multimap\!\!\circ\ ?_n A$. The other way around, however, this relation does not hold.

We formulate a system in which we conflate $?_n A$ and $!_n A$, following Atkey et al. (2016) in calling this connective $\star_n A$—though, since they provide no pronunciation other than "an access point of type A", I will pronounce it as "nod". We add a rule for $\star_1$ elimination, a contraction rule and the Mix rule.

$$\frac{P \vdash \Gamma, y\colon A}{x \star y.P \vdash \Gamma, x\colon \star_1 A}\ \star \qquad \frac{P \vdash \Gamma, y\colon \star_m A, z\colon \star_n A}{P\{x/y, z\} \vdash \Gamma, x\colon \star_{m+n} A}\ \textsc{Contr.}$$

There is no reason to add Pool, as it is derivable from Contract and Mix.

$$\dfrac{\dfrac{\vdash \Gamma, \star_m A \qquad \vdash \Delta, \star_n A}{\vdash \Gamma, \Delta, \star_m A, \star_n A}\ \textsc{Mix}}{\vdash \Gamma, \Delta, \star_{m+n} A}\ \textsc{Contract}$$

We will refer to this system as $\mathrm{CP}^\star_{\mathrm{ND}}$.

In $\mathrm{CP}^\star_{\mathrm{ND}}$, we can derive a restricted version of MultiCut. By applying Cut to a "vector" of As, we can simulate communication on multiple channels, as long as all of these channels have the same session type.

$$\dfrac{\dfrac{\dfrac{\vdash \Gamma, A^\perp, \ldots, A^\perp}{\vdash \Gamma, \star_1 A^\perp, \ldots, \star_1 A^\perp}\ \star}{\vdash \Gamma, \star_n A^\perp}\ \textsc{Contr.} \qquad \dfrac{\dfrac{\vdash \Delta, A, \ldots, A}{\vdash \Delta, \star_1 A, \ldots, \star_1 A}\ \star}{\vdash \Delta, \star_n A}\ \textsc{Contr.}}{\vdash \Gamma, \Delta}\ \textsc{Cut}$$

It is possible that cut elimination for $\mathrm{CP}^\star_{\mathrm{ND}}$ terminates. The uses of MultiCut which lead to deadlocks invariably communicate on two channels of dual types, as in

$$\dfrac{\dfrac{}{x \leftrightarrow y \vdash x\colon A^\perp, y\colon A}\ \textsc{Ax} \qquad \dfrac{}{y \leftrightarrow x \vdash y\colon A^\perp, x\colon A}\ \textsc{Ax}}{\nu x.\nu y.(x \leftrightarrow y \mid y \leftrightarrow x) \vdash}\ \textsc{MultiCut}$$

Why is $\mathrm{CP}^\star_{\mathrm{ND}}$ interesting? It models an even more permissive form of non-determinism than $\mathrm{CP}_{\mathrm{ND}}$+Mix. In the latter, we can model a program in which two pools of processes are non-deterministically linked together. However, we cannot model a program in which two pools of sequentially communicating processes are linked together. For instance,

$$(x[u_1].x[u_2].P \mid x[v].Q \mid x(w).R \mid x(z_1).x(z_2).S)$$

We *can* model such a process in $\mathrm{CP}^\star_{\mathrm{ND}}$. Why would we want to? It may not unreasonable—in the case of a pâtisserie—to want to model a customer buying *two or more* pastries. This would require us to have, on the one side, a process for the store which communicates with n customers sequentially. On the other side, we would have a pool of processes which may communicate, several times, with the store. We could model this by having the store offer a transaction which can be used some bounded number of times, e.g.

$$?_n(\textsc{Coin} \otimes \textsc{Cake}^\perp) \parr (\forall m \leq n.?_m(\textsc{Coin} \otimes \textsc{Cake}^\perp)),$$

but a simpler solution is to just require the customer to return to the pool after the first transaction, and wait for another turn.

4.5 Relation to recursion

There is still an obvious open question. In many of the proofs throughout this proposal, we've left open branches, labeled IH for induction hypothesis. However, as it stands we are unable to express these induction proofs *within* the languages $\mathrm{CP}_{\mathrm{ND}}$ or $\mathrm{CP}^\star_{\mathrm{ND}}$. To do so, we would have to add some sort of induction principle. For instance,

$$\frac{\vdash ?_1 \Gamma, !_1 \Delta \qquad \vdash \forall n.!_n \Gamma^\perp, ?_n \Delta^\perp, ?_{n+1} \Gamma, !_{n+1} \Delta}{\vdash \forall n.?_n \Gamma^\perp, !_n \Delta}\ \textsc{Ind}$$

$$\frac{\vdash \forall n.?_n \Gamma, !_n \Delta}{\vdash ?_m \Gamma, !_m \Delta}\ \forall$$

This would add a restricted form of induction and polymorphism over resource variables to our language.

Lindley and Morris (2016) describe an extension of CP, called μCP, which adds connectives for structural recursion. It would be interesting to see what the interactions are between these recursion principles and induction on resource variables.

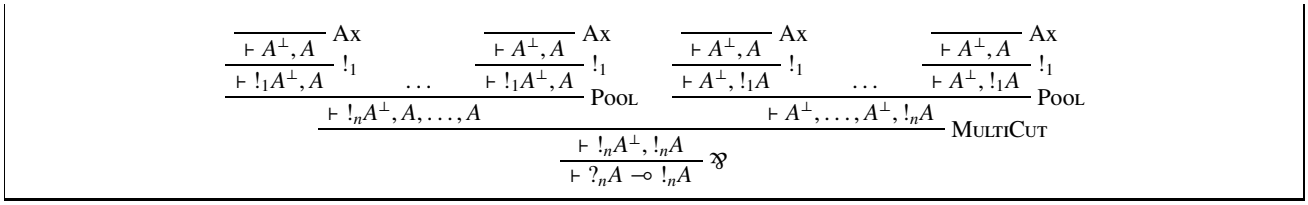

Figure 3: Conflation of $?_n A$ and $!_n A$ in the presence of MultiCut.

Using the above induction principle, we can unfold a $?_n A/!_n A$ into a stream or pars or tensors, i.e.

$$?_n A \otimes (\mu X.A \parr X) \multimap (\mu X.A \parr X),$$
$$!_n A \otimes (\mu X.A \otimes X) \multimap (\mu X.A \otimes X),$$

or fold the first n values from a stream into a $?_n A/!_n A$,

$$(\nu X.A \parr X) \multimap (?_n A \parr \nu X.A \parr X),$$
$$(\nu X.A \otimes X) \multimap (!_n A \parr \nu X.A \otimes X).$$

Equivalently, we can unfold a $?_n A/!_n A$ into a list,

$$?_n A \multimap (\mu X.(A \parr X) \oplus 1),$$
$$!_n A \multimap (\mu X.(A \otimes X) \oplus 1),$$

though, in order to unfold a list into a $!_n A$, we would need to add existential quantification over resource variables.

We could these constructions in order to, for example, implement a ticket server which sells n tickets, labeled 1 to n, by recursion.

It is not possible to encode μ-recursion and ν-corecursion in terms of the induction principle, as usage of the induction principle is always bounded by a resource variable n. The other way around is unsure, though unlikely, as there is no obvious way to model the increasing resource index in the induction in a simply-typed setting.

4.6 Formalization

We plan to formalize the systems $\mathrm{CP}_{\mathrm{ND}}$ and $\mathrm{CP}^{\star}_{\mathrm{ND}}$ using Agda (Norell, 2009), and implement their cut elimination procedures. This will give us a strong guarantee that the described cut elimination procedures are correct and terminating. For this, we will use the techniques for separating types and usages used in McBride (2016).

5 Evaluation

We can evaluate the success of this project by the success of $\mathrm{CP}_{\mathrm{ND}}$ and $\mathrm{CP}^{\star}_{\mathrm{ND}}$, and the new kinds of non-deterministic programs which these systems will be able to model. Through its tight correspondence with bounded linear logic, it is likely that $\mathrm{CP}_{\mathrm{ND}}$ will have a well-behaved cut elimination procedure. This is less true for $\mathrm{CP}^{\star}_{\mathrm{ND}}$, as their semantics closely resemble those of access points, which are known to permit deadlocks and encodings of general recursion. The formalization will play an important role in the evaluation of these properties, as it will be able to give strong guarantees about the correctness and termination properties of our systems.

It would be a bonus if we could find a way to unify the induction principle and the principles for structural recursion, but this is unlikely unless we move to a vastly more powerful system, e.g. dependent types.

6 Research outputs

By the end of this project, I hope to have produced the following outputs. For $\mathrm{CP}_{\mathrm{ND}}$ and $\mathrm{CP}^{\star}_{\mathrm{ND}}$, I hope to have:

- formulated the system;
- given a cut elimination procedure;
- given a proof of termination;
- given a proof of deadlock freedom;
- formalized the system; and
- characterized the non-determinism.

In addition, I hope to have characterized the relation between the two systems, both logically and with respect to non-determinism and concurrency.

7 Workplan

The outputs given in the previous section can be assigned to three distinct categories: theory, evaluation, implementation. The first four—all of which belong to theory—are all dependent on the previous outputs. In addition, due to the similarity of the systems $\mathrm{CP}_{\mathrm{ND}}$ and $\mathrm{CP}^{\star}_{\mathrm{ND}}$, the theoretical outputs for each system will be similar. For this reason, I will work on $\mathrm{CP}_{\mathrm{ND}}$ and $\mathrm{CP}^{\star}_{\mathrm{ND}}$ in parallel, at least until the formulation of a cut elimination procedure, which is when I will have learned if $\mathrm{CP}^{\star}_{\mathrm{ND}}$ is a sane logical system. Throughout the process, I will make an effort to formalize my results in Agda.

After we are satisfied that they are sane type systems, I will attempt to characterize the non-determinism inherent in each system—with respect to each other, local choice, and the π-calculus in general.

Finally, I will write throughout, but will focus my effort more strongly on writing in the last four months. For a more detailed overview, see Figure 4.

References

Atkey, R., Lindley, S., and Morris, J. G. (2016). Conflation confers concurrency. In Lindley, S., McBride, C., Trinder, P., and Sannella, D., editors, *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*, Lecture Notes in Computer Science.

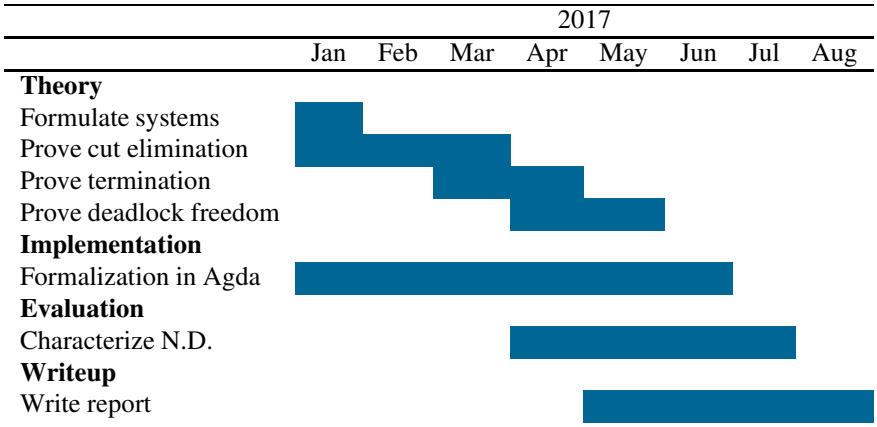

Figure 4: My workplan.

Burstall, R. M. (1969). Proving properties of programs by structural induction. *The Computer Journal*, 12(1):41–48.

Caires, L. (2014). Types and logic, concurrency and nondeterminism. In Abadi, M., Gardner, P., Gordon, A., and Mardare, R., editors, *Essays for the Luca Cardelli Fest*. Microsoft Research.

Caires, L. and Pérez, J. A. (2017). Linearity, control effects, and behavioral types. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. Springer.

Caires, L. and Pfenning, F. (2010). *Session Types as Intuitionistic Linear Propositions*, pages 222–236. Springer Berlin Heidelberg, Berlin, Heidelberg.

Gay, S. J. and Vasconcelos, V. T. (2009). Linear type theory for asynchronous session types. *Journal of Functional Programming*, 20(01):19.

Girard, J.-Y. (1987). Linear logic. *Theoretical Computer Science*, 50(1):1–101.

Girard, J.-Y., Scedrov, A., and Scott, P. J. (1992). Bounded linear logic: A modular approach to polynomial-time computability. *Theor. Comput. Sci.*, 97(1):1–66.

Honda, K. (1993). Types for dyadic interaction. In *CONCUR'93*, pages 509–523. Springer Nature.

Lee, C. S., Jones, N. D., and Ben-Amram, A. M. (2001). The size-change principle for program termination. In *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '01*. Association for Computing Machinery (ACM).

Lindley, S. and Morris, J. G. (2016). Talking bananas: Structural recursion for session types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 434–447, New York, NY, USA. ACM.

McBride, C. (2016). I got plenty o' nuttin'. In *A List of Successes That Can Change the World*, pages 207–233. Springer Nature.

Milner, R., Parrow, J., and Walker, D. (1992). A calculus of mobile processes, i. *Information and Computation*, 100(1):1–40.

Norell, U. (2009). Dependently typed programming in agda. In *Advanced Functional Programming*, pages 230–266. Springer Nature.

Wadler, P. (2012). Propositions as sessions. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, pages 273–286, New York, NY, USA. ACM.

Walther, C. (1994). On proving the termination of algorithms by machine. *Artificial Intelligence*, 71(1):101–157.