The rho calculus at 20

L.G. Meredith[1]

CEO, F1R3FLY.io 9336 California Ave SW, Seattle, WA 98103, USA,
ceo@f1r3fly.io

Abstract. We discuss developments and advances in the rho calculus 20 years after its inception. Specifically, we discuss efficient implementation techniques for modern computers. We discuss key variants of the the calculus that make it suitable for a broad range of applications. We discuss the use of rho-style reflection in other formalisms, such as set theory, and graph theory. Finally, we discuss what the implications of the rho calculus in the context of rapidly accelerating developments in AI.

1 Introduction

This December will find my submission of the first paper about the rho calculus to be 20 years in the rear view mirror. A lot has happened since then. Apart from production implementations of scalable decentralized asset management systems based on the calculus, a lot of work has been done on the calculus itself. Indeed, just this May (2024) i discovered a conservative extension of the calculus that fills a hole in the account of reflection that has long bothered me. And, last April (2023) i found a fuzzy version of the rho calculus that is just waiting for someone to take further. In 2018 i discovered a variant of the calculus (which i dubbed the *space calculus*) that makes it a computational analog (in a certain precise sense) of solutions of the Einstein field equations.

Thankfully, i am not the only one who has found the rho calculus interesting enough to think about. Stay and Williams studied a variant they called the π-rho calculus as an example of a system that could be typed in their native types construction. And Stian Lybech showed that rho calculus is more expressive than the the π-calculus. The list goes on.

The aim of this note is to collect, and summarize in one place, developments over the last 20 years in the study of the rho calculus. i suspect that there are more researchers than i know about who are interested in the questions raised by the calculus. Hopefully, this note can serve that community, and possibly spark wider interest due to the discoveries to date.

1.1 Summary of contributions and outline of paper

More specifically, this note will develop the following.

- the modern, programming-language-friendly syntax for the calculus;
- a handful of useful syntactic sugar coatings that make the calculus a suitable semantic framework for a realistic protocol-oriented programming language, including, but not limited to:

    - adding first class values;
    - adding unforgeable names (the π-rho calculus);
    - adding syntax for joins and sequences and choice;
- multi-level agency mechanisms in rho:
    - annihilation: a recursive reduction relation;
    - a procedurally reflective rho calculus;
    - the space calculus;
- the fuzzy variant of rho;
- probabilistic and quantum variations of rho;
- revisiting the encodings of π into rho and vice versa.

But, we will also touch briefly on the techniques used to represent reflective computation in the rho calculus and demonstrate that they are not bound to this specific setting. In particular, they can be applied to set theory and graph theory, yielding variations of these theories that have very useful features.

Additionally, we will look at relating the rho calculus to physical intuition. There has been a great deal of work on using mobile process calculi to describe network protocols and concurrent algorithms. It is not too much of an abstraction to see how these techniques apply to physical processes such as chemical reactions, signaling regimes within and between cells. But the rho calculus can also be used to model ordinary classical physics problems. We illustrate some of these.

Finally, we will conclude with some thoughts about the foundations of mathematics and its relationship to agency. As a preview, i believe that set theory (and even to some extent category theory) are theories of *data structures*. Implicitly, the community has believed that (only) mathematicians have agency and their formalisms merely carry information content between these agents. But the advances in artificial intelligence (AI) tell a different story. To some extent AI has brought their formalisms to life and is on the verge of giving them agency. As a result, i want to argue that a proper foundation of mathematics needs to include a notion of agency and that the ingredients of a foundational account of agency are present in the mobile process calculi, generally, and a pragmatically foundational account is present in the rho calculus, specifically.

2 Concurrent process calculi and spatial logics

2.1 Computation as interaction

Before diving into a narrative of the history and motivations underlying the development of the *process calculi* it is useful to understand the profound shift in thinking they represent. For the longest time (at least 400 years) people have thought about computation, abstractly, as *functions applied to data*. Whether you review how maths and computing are taught in universities and other academic institutions – these curriculae embody much of the history of their development – or you review the actual history from Newton's calculus to today, you will find this perspective either implicitly or explicitly organizing both formalisms and intuition. The process calculi, on the other hand, take a very different perspective. The core intuition they embody is that *computation arises from interaction.*

From this perspective there isn't this strange division between program and data that makes a hash of agency. Every computation is understood as a pattern of interaction amongst autonomous agents.

What's crucially important is that this perspective can be made precise and exact. It yields a coherent collection of conceptual and computational tools that have at least as broad a range of applicability as Newton's calculus. In this sense, the process calculi constitute nothing less than a conceptual revolution. They change everything in the accepted ontological landscape, from notions of time and space to notions of complexity. But, the revolution is hiding in plain sight and to bring it into focus requires understanding something of the history and development and motivations that gave rise to the paradigm of computation as interaction.

2.2 A brief history of process calculi

In the last thirty years the process calculi have matured into a remarkably powerful analytic tool for reasoning about concurrent and distributed systems, such as Internet communication, security, or application protocols ([1], [17], [18]), but also chemical reactions, and biological protocols such as cell signaling regimes ([31], [30].). Process-calculus-based algebraic specification of computational processes began with Milner's Calculus for Communicating Systems (CCS) [23] and Hoare's Communicating Sequential Processes (CSP) [16]. Being largely influenced by the hardware models of the time, these formalisms were based on a fixed communication topology, where computational processes are considered to be something like components soldered to a motherboard.

But as radio and cellular technologies, and protocols like TCP/IP became popularized, this influenced the evolution of these formalisms. In particular, the trend toward physical mobility of computational devices gave rise to the mobile process calculi, called so because they envisioned a network of computational processes connected by a dynamic communication topology that evolves as the processes compute. In a sense the computational processes contemplated by these formalisms are mobile in much the same way as participants at a conference who randomly bump into one another and trade mobile numbers and email addresses, or molecules that randomly bump into one another and exchange electrons. It was this innovation of mobility that greatly expanded the scope of applicability of these formalisms.

Perhaps the most paradigmatic version of these calculi is Milner, Parrow, and Walker's π-calculus [27], [28], which Milner refined with [24] and [26]. In the π-calculus the dominant metaphor is very much processes as mobile agents (e.g., conference participants, or molecules) bumping into one another exchanging channels over which they can communicate. Yet, another equally compelling metaphor is that processes are mobile agents swimming in various nested compartments or membranes. These membranes are called *ambients* because they both surround and are surrounded by other ambients, but also because they *move.* That is, they can enter or exit one another are. In this sense the membranes themselves are the computational agents. These intuitions are formalized in Cardelli and Gordon's ambient calculus [9], and considerably refined by Cardelli's brane calculi [8].

Note that it is common practice in the mobile process calculi literature to refer to channels as *names* and in this note we will use these terms interchangeably. In the more abstract setting of that includes calculi like the ambient calculus where names are really tags associated with mobile membranes this more generic moniker makes sense. In both the π and ambient calculi names are the means of synchronization and rendezvous for mobile processes.

It is notable that to this day there is no fully abstract encoding of the ambient calculus into the π-calculus, suggesting that the ambient calculus is somehow more expressive than the π-calculus. That said, expressivity is not always a benefit. It is quite difficult to map the ambient style of computation to trustless distributed Internet-based protocols. The expressivity requires a great deal more coordination which, in turn, requires a great deal more trust. Meanwhile, the π-calculus has a ready mapping onto a wide range of Internet protocols, which i exploited vigorously in building Microsoft's BizTalk Process Orchestration. And the rho calculus has an even better mapping to Internet protocols, which i exploited even more vigorously in building RChain and F1R3FLY.io.

2.3 Hennessy-Milner and spatial logics, and session types

Companion to these calculi are logics and type systems which enable practitioners not only to reason about the correctness of individual agents, or specific ensembles of agents, but also about entire classes of agents, identified as the witnesses of logical formulae. These began with the Hennessy-Milner logics which build on Kripke's interpretation of modal logic [26]. More recently, these logics have been given new superpowers in the form of being able to detect structural properties. Notably, we find Cardelli and Caires's groundbreaking spatial logic [4] [6] [7]. The interested reader might also look at *session types* [11].

This topic is worthy of a much richer and deeper discussion. However, we relegate that to a separate paper in which we unify them all with a single algorithm, the OSLF algorithm. This takes as input a model of computation formatted as a graph structured lambda theory and outputs a spatial-behavioral type system for that model of computation. The type system enjoys the Hennessy-Milner property, namely that two computations inhabit the same bisimulation equivalence class iff they satisfy exactly the same set of types. We leave the details to the paper describing this algorithm.

2.4 Compositionality and bisimulation

Among the many reasons for the continued success of the mobile process calculi's approach to organizing and analyzing computation are two central points. First, the process algebras provide a compositional approach to the specification, analysis and execution of concurrent and distributed systems. Owing to Milner's original insights into computation as interaction [25], the process calculi are so organized that the behavior —the semantics— of a system may be composed from the behavior of its components. This means that specifications can be constructed in terms of components —without a global view of the system— and assembled into increasingly complete descriptions.

The second central point is that process algebras have a potent proof principle, yielding a wide range of effective and novel proof techniques [34] [32] [33]. In particular, *bisimulation* encapsulates an effective notion of process equivalence that has been used in applications as far-ranging as algorithmic games semantics [3] and the construction of model-checkers [5]. The essential notion can be stated in an intuitively recursive formulation: a *bisimulation* between two processes P and Q is an equivalence relation E relating P and Q such that: whatever action of P can be observed, taking it to a new state P', can be observed of Q, taking it to a new state Q', such that P' is related to Q' by E and vice versa. P and Q are *bisimilar* if there is some bisimulation relating them. Part of what makes this notion so robust and widely applicable is that it is parameterized in the actions observable of processes P and Q, thus providing a framework for a broad range of equivalences and up-to techniques [34] all governed by the same core principle [33].

3 The reflective higher order calculus

It is in the context of the evolution of the mobile process calculi that Meredith and Radestock's *r*eflective *h*igher-*o*rder, or rho calculus arises [1] [21]. While it is possible to give a faithful encoding of the ambient calculus into the rho calculus, the rho calculus can be viewed as the natural successor to the π-calculus. One way to look at the relationship between the two is that the π-calculus is not actually a single calculus. Instead, it is a recipe or *function* for constructing a calculus of agents that communicate over *some* collection of channels. That is, it is parametric in a *theory* of channels[2]. This intuition can be made very precise and corresponds to a particular strategy of implementation called two-level type decomposition. But, if the π-calculus is a function taking a theory of channels to a theory of agents communicating over those channels, then the rho calculus is the *least fixed point* of that function. The rho calculus says we need look no farther than the *communicating agent themselves* for our theory of channels.

More specifically, the calculus envisions a new kind of distinction, one between the *code* of a process, and a *running instance* of a process. It will not have escaped the reader's attention that this is a commonplace notion in the practice of building concurrent and distributed systems. Indeed, the development of the rho calculus stole a page from the previous evolution of process algebras. The step introducing mobility was the theory coming into better alignment with practice. Computing devices became mobile, to keep up the formalisms had to follow suit and devise a notion of mobile processes. Likewise,

[1] The π-calculus gets its name because it was the first Greek letter available after the λ in Church's calculus by that name, and just happens to be the Greek letter corresponding to the English 'p' for process. Likewise, the rho calculus gets its name from its delivery of higher-order features through the use of reflection, and it just happens that the acronym spells out *rho*, the English spelling of the Greek letter that comes after π.

[2] Note that here we are using the term 'theory' in a technical sense, such as Peano arithmetic, or a Lawvere theory. We don't mean theoretical in the usual sense of that word, but rather a collection of entities (such as the Natural Numbers) that can be generated from a small set of rules.

the practice of building the software that comprises concurrent and distributed systems makes a distinction between code and running instances of the code. We argue, however, that providing an account of this distinction in our formalism is much more fundamental than just keeping up with practice. We argue that this reason this is such a ubiquitous practice is because it is a fundamental aspect of the underlying order and organization of computation.

3.1 Coding as a fundamental aspect of computation

Indeed, Gödel's incompleteness results fundamentally depend on a notion of *coding*. To achieve the self-reference needed to construct a logical statement of the form "This sentence is not provable." Gödel encoded logical formulae into the natural numbers and then used logical statements about the natural numbers to construct self-referential statements like that one. In his construction, the coding trick appears to be exogenous to logic and arithmetic. Indeed a careful study of these formalisms shows that it is, in fact, *intrinsic*. That is, both the natural numbers, and logic naturally reflect on themselves, if you look more carefully at those mathematical objects. More to the point, however, this intrinsic, loopy self-reference that arises from the notion of code is also at the heart of Turing's solution to the Entscheidungsproblem and Cantor's separation of the countably infinite from the uncountably infinite.

Moreover, the scope of this phenomenon is not restricted to some of the most profound results in mathematics and logic over the last couple of centuries. It's also at the heart of the origin of life. Some four billion years ago the Earth's chemistry stumbled on a way to code for the production of proteins involved in certain self-replicating chemical reactions. Arguments about the validity of the Weissman barrier to one side [40], it is a matter of experimentally established fact that the codons at the heart of the genetic apparatus provide the mechanism for both the self-replication of the chemical processes underlying life, but also the transmission of inherited information about fitness to environment. As such, we believe that phenomenon of coding is a much more profound phenomenon than has been appreciated and deserves to be given a first-class representation in our computational formalisms.

Once reflection is introduced into the mobile process calculus setting, higher-order capabilities and recursion are child's play. Perhaps more surprisingly, so is the mysterious new operator of the π-calculus tasked with generating fresh channels. But, we are getting ahead of ourselves. Suffice it to say that using reflection as a first-class representation of the trick of coding – a trick originally discovered by life and then independently rediscovered by Cantor, Gödel, and Turing – is a real banger.

Intriguingly, not only does introducing coding explicitly into the formalism solve a number of practical problems associated with the π-calculus, it simultaneously solves a number of problems associated with corresponding logics and type systems, as well. The interested reader should see namespace logic [20].

4 An idiosyncratic view of related work

Here we direct the reader to a small sample of the considerable work on the mobile process calculi. This sample is far from representative of the overall body of work, but instead constitutes a snapshot of what your author felt was salient at some point during the last twenty years. Indeed, the reader is encouraged to investigate the literature on her own to get an independent perspective on this rich field of research.

- telecommunications, networking, security and application level protocols [1] [2] [17] [18];
- programming language semantics and design [17] [15] [14] [36];
- webservices [17] [18] [19];
- blockchain [22]
- and biological systems [8] [10] [31] [30].

5 The syntax and semantics of the core rho calculus

We now summarize a technical presentation of the core calculus. The typical presentation of such a calculus generalizes a generators and relations presentation of an algebra [29]. The grammar below, describing term constructors in the language of process expressions (which we abbreviate to processes in the sequel), freely generates the set of processes. We denote this set Proc, and the set of channels (aka names) over which these processes communicate by $@\mathsf{Proc}$. The set Proc (and by extension $@\mathsf{Proc}$) is then quotiented by a relation known as structural congruence, denoted $\equiv$, and it is over this set that the notion of computation is expressed.

In particular, computation is expressed as a small handful of rewrite rules which should be viewed as defining a state transition relation. Thus, process expressions capture *states* and the form of the expression determines how one process (state) *might* evolve to another (state). The word 'might' is doing some heavy lifting in that previous sentence, as the calculus is *not* deterministic, in fact it is not even confluent. As such, the calculus can be used to represent all manner of common human (and natural) resource distribution protocols, such as first come first served, which sits at the heart of everything from airline reservation systems to concert ticket sales. By comparison, Church's λ-calulus (the core of programming languages such as Haskell or Scala) does not – without considerable modification – natively support the expression of such protocols.

5.1 The rho calculus in less than a page

PROCESS

$$P, Q ::= \mathsf{0} \mid \mathsf{for}(y \leftarrow x)P \mid x!(Q) \mid *x \mid P|Q$$

NAME

$$x, y ::= @P$$

EQUIV

$$P|\mathsf{0} \equiv P \quad P|Q \equiv Q|P \quad P|(Q|R) \equiv (P|Q)\mathsf{R}$$

ALPHA

$$\mathsf{for}(y \leftarrow x)P \equiv \mathsf{for}(z \leftarrow x)(P\{z/y\}) \;\mathsf{if}\, z \notin \mathsf{FN}(P)$$

$$\frac{\text{COMM}}{\mathsf{for}(y \leftarrow x)P \mid x!(\overrightarrow{Q}) \rightarrow P\{@Q/y\}}$$

$$\text{PAR}\quad \frac{P \rightarrow P'}{P|Q \rightarrow P'|Q} \qquad \text{EQUIV}\quad \frac{P \equiv P' \qquad P' \rightarrow Q' \qquad Q' \equiv Q}{P \rightarrow Q}$$

This presentation is essentially that of [21] recast with a programmer-friendly syntax that extends smoothly to fork-join patterns commonly found in human decision-making processes. The reader should note that, even with a programmer-friendly syntax, this calculus is considerably smaller than a corresponding presentation of the π-calculus. Indeed is as concise and compact as the λ-calculus, yet considerably more expressive. What follows below is an exegesis of the technical details hidden in this more compact presentation. Notably, we provide definitions of free and bound names, and other housekeeping matters such as name equality. The reader more familiar with these notions should feel free to skim this section, however the section on bisimulation is of some importance and worthy of more focused attention.

5.2 The fine print

Notational interlude when it is clear that some expression t is a sequence (such as a list or a vector), and a is an object that might be meaningfully and safely prefixed to that sequence then we write $a : t$ for the sequence with a prefixed (aka "consed") to t. We write $t(i)$ for the ith element of t. In the sequel we use $\overrightarrow{x}$ (resp. $\overrightarrow{P}$) denotes a vector of *names* (resp. *processes*) of length $|\overrightarrow{x}|$ (resp. $|\overrightarrow{P}|$).

Process grammar

$$\overset{\text{PROCESS}}{P, Q ::= \mathsf{0} \mid \mathsf{for}(y \leftarrow x)P \mid x!(Q) \mid *x \mid P|Q} \qquad \overset{\text{NAME}}{x, y ::= @P}$$

As mentioned above we use Proc (resp. $@\mathsf{Proc}$) to denote the language of processes (resp. names) freely generated by this grammar. But how are we to think about expressions in this language, intuitively?

- $\mathsf{0}$ is the ground of this tiny little language. It represents the *stopped* process, or the process that does nothing; in some real sense this is the quintessential neutral element in this computational dynamics, as we shall see;
- $\mathsf{for}(y \leftarrow x)P$ is the process that is waiting at channel x for some data or message that it will bind to y and then proceed to do P; if $\mathsf{0}$ is neutral, then for-comprehension is receptive, it waits for action at x before it can do anything;
- $x!(Q)$ is the process that is outputting the datum or message Q (which is also a process) along the channel, x; if $\mathsf{0}$ is neutral, and $\mathsf{for}(y \leftarrow x)P$ is reception, then $x!(Q)$ is active; it represents actively sending Q floating down the channel x to be picked up by some $\mathsf{for}(y \leftarrow x)P$;

- $*x$ is the process that takes the code represented by x and begins running it;
- $P|Q$ is the process that is the parallel or concurrent composition of the process P with the process Q; it introduces the principle that individual computations can actually be collectives of autonomous, yet coordinating computation.

As foreshadowed by the syntactic category of expressions of the form $*x$, a channel or name is merely the code of some process, $@P$. In this sense we may think of these two operators $*$ and $@$ as dual to one another. The latter delivers the code of some process, while the former delivers a running process from its code.

Next, we quotient this set by the smallest equivalence relation containing α-equivalence that makes $(\mathsf{Proc}, |, 0)$ into a commutative monoid. In order to define this relation we require a definition of free and bound names to define α-equivalence.

Definition 1. Free and bound names *The calculation of the free names of a process, P, denoted $\mathsf{FN}(P)$ is given recursively by*

$$\mathsf{FN}(0) = \emptyset \qquad \mathsf{FN}(\mathsf{for}(y \leftarrow x)(P)) = \{x\} \cup \mathsf{FN}(P) \setminus \{y\} \qquad \mathsf{FN}(x!(P)) = \{x\} \cup \mathsf{FN}(P)$$

$$\mathsf{FN}(P|Q) = \mathsf{FN}(P) \cup \mathsf{FN}(Q) \qquad \mathsf{FN}(x) = \{x\}$$

An occurrence of x in a process P is bound if it is not free. The set of names occurring in a process (bound or free) is denoted by $\mathsf{N}(P)$.

5.3 Substitution

We use the notation $\{\overrightarrow{y}/\overrightarrow{x}\}$ to denote partial maps, $s : @\mathsf{Proc} \to @\mathsf{Proc}$. A map, s lifts, uniquely, to a map on process terms, $\widehat{s} : \mathsf{Proc} \to \mathsf{Proc}$. Historically, it is convention to use σ to range over lifted subsitutions, $\widehat{s}$, to write the application of a substitution, σ to a process, P, using postfix notation, with the substitution on the right, $P\sigma$, and the application of a substitution, s, to a name, x, using standard function application notation, $s(x)$. In this instance we choose not to swim against the tides of history. Thus,

Definition 2. *given $x = @P'$, $u = @Q'$, $s = \{u/x\}$ we define the lifting of s to $\widehat{s}$ (written below as σ) recursively by the following equations.*

$$0\sigma := 0$$

$$(P|Q)\sigma := P\sigma|Q\sigma$$

$$(\mathsf{for}(y \leftarrow v)P)\sigma := \mathsf{for}(z \leftarrow \sigma(v))((P\widehat{\{z/y\}})\sigma)$$

$$(x!(Q))\sigma := \sigma(x)!(Q\sigma)$$

$$(*y)\sigma := \begin{cases} Q' & y \equiv_{\mathsf{N}} x \\ *y & \textit{otherwise} \end{cases}$$

where

$$\{\widehat{@Q/@P}\}(x) = \{@Q/@P\}(x) = \begin{cases} @Q & x \equiv_{\mathsf{N}} @P \\ x & \textit{otherwise} \end{cases}$$

and z is chosen distinct from $@P$, $@Q$, the free names in Q, and all the names in R. Our α-equivalence will be built in the standard way from this substitution.

Definition 3. *Then two processes, P, Q, are alpha-equivalent if $P = Q\{\overrightarrow{y}/\overrightarrow{x}\}$ for some $\overrightarrow{x} \in \mathsf{BN}(Q)$, $\overrightarrow{y} \in \mathsf{BN}(P)$, where $Q\{\overrightarrow{y}/\overrightarrow{x}\}$ denotes the capture-avoiding substitution of $\overrightarrow{y}$ for $\overrightarrow{x}$ in Q.*

Definition 4. *The* structural congruence $\equiv$ *between processes [35] is the least congruence containing alpha-equivalence and satisfying the commutative monoid laws (associativity, commutativity and* $\mathsf{0}$ *as identity) for parallel composition* $|$.

Definition 5. *The* name equivalence $\equiv_{\mathsf{N}}$ *is the least congruence satisfying these equations*

$$\textsc{Quote-drop} \quad @{*}x \equiv_{\mathsf{N}} x \qquad\qquad \textsc{Struct-equiv} \quad \frac{P \equiv Q}{@P \equiv_{\mathsf{N}} @Q}$$

The astute reader will have noticed that the mutual recursion of names and processes imposes a mutual recursion on alpha-equivalence and structural equivalence via name-equivalence. Fortunately, all of this works out pleasantly and we may calculate in the natural way, free of concern. The reader interested in the details is referred to the original paper on the rho calculus [21].

Remark 1. One particularly useful consequence of these definitions is that $\forall P.@P \notin \mathsf{FN}(P)$. It gives us a succinct way to construct a name that is distinct from all the names in P and hence fresh in the context of P. For those readers familiar with the work of Pitts and Gabbay, this consequence allows the system to completely obviate the need for a fresh operator, and likewise provides a canonical approach to the semantics of freshness.

Equipped with the structural features of the term language we can present the dynamics of the calculus.

5.4 Operational semantics

Finally, we introduce the computational dynamics. What marks these algebras as distinct from other more traditionally studied algebraic structures, e.g. vector spaces or polynomial rings, is the manner in which dynamics is captured. In traditional structures, dynamics is typically expressed through morphisms between such structures, as in linear maps between vector spaces or morphisms between rings. In algebras associated with the semantics of computation, the dynamics is expressed as part of the algebraic structure itself, through a reduction reduction relation typically denoted by $\rightarrow$. Below, we give a recursive presentation of this relation for the calculus used in the encoding.

$$\text{COMM}\quad \frac{x_t \equiv_{\mathsf{N}} x_s}{\mathsf{for}(y \leftarrow x_t)P \mid x_s!(\overrightarrow{Q}) \rightarrow P\{@Q/y\}} \qquad \text{PAR}\quad \frac{P \rightarrow P'}{P|Q \rightarrow P'|Q}$$

$$\text{EQUIV}\quad \frac{P \equiv P' \qquad P' \rightarrow Q' \qquad Q' \equiv Q}{P \rightarrow Q}$$

We write $P \rightarrow$ if $\exists Q$ such that $P \rightarrow Q$ and $P \not\rightarrow$, otherwise.

5.5 Dynamic quote: an example

Anticipating something of what's to come, let $z = @P$, $u = @Q$, and $x = @y!(*z)$. Now consider applying the substitution, $\widehat{\{u/z\}}$, to the following pair of processes, $w!(y!(*z))$ and $w!(*x) = w!(*@y!(*z))$.

$$\begin{aligned} w!(y!(*z))\widehat{\{u/z\}} &= w!(y!(Q)) \\ w!(*x)\widehat{\{u/z\}} &= w!(*x) \end{aligned}$$

The body of the quoted process, $@y!(*z)$, is impervious to substitution, thus we get radically different answers. In fact, by examining the first process in an input context, e.g. $\mathsf{for}(z \leftarrow x)w!(y!(*z))$, we see that the process under the output operator may be shaped by prefixed inputs binding a name inside it. In this sense, the combination of input prefix binding and output operators will be seen as a way to dynamically construct processes before reifying them as names.

6 Replication

As mentioned before, it is known that replication (and hence recursion) can be implemented in a higher-order process algebra [35]. As our first example of calculation with the machinery thus far presented we give the construction explicitly in the rho calculus.

$$\begin{aligned} D_x &:= \mathsf{for}(y \leftarrow x)(x!(y)|*y) \\ !_x P &:= x!(D_x|P)|D_x \end{aligned}$$

$$
\begin{array}{ll}
!_x P & \\
= & x!((\mathsf{for}(y \leftarrow x)(x!(y)|{*y}))|P)|\mathsf{for}(y \leftarrow x)(x!(y)|{*y}) \\
\rightarrow & (x!(y)|{*y})\{@(\mathsf{for}(y \leftarrow x)({*y}|x!(y)))|P/y\} \\
= & x!(@(\mathsf{for}(y \leftarrow x)(x!(y)|{*y}))|P)|(\mathsf{for}(y \leftarrow x)(x!(y)|{*y}))|P \\
\rightarrow & \ldots \\
\rightarrow^* & P|P|\ldots
\end{array}
$$

Of course, this encoding, as an implementation, runs away, unfolding $!P$ eagerly. A lazier and more implementable replication operator, restricted to input-guarded processes, may be obtained as follows.

$$!\mathsf{for}(v \leftarrow u)P := x!(\mathsf{for}(v \leftarrow u)(D(x)|P))|D(x)$$

Remark 2. Note that the lazier definition still does not deal with summation or mixed summation (i.e. sums over input and output). The reader is invited to construct definitions of replication that deal with these features.

Further, the definitions are parameterized in a name, x. Can you, gentle reader, make a definition that eliminates this parameter and guarantees no accidental interaction between the replication machinery and the process being replicated – i.e. no accidental sharing of names used by the process to get its work done and the name(s) used by the replication to effect copying. This latter revision of the definition of replication is crucial to obtaining the expected identity $!!P \sim !P$.

Remark 3. The reader familiar with the lambda calculus will have noticed the similarity between D and the paradoxical combinator.

[Ed. note: the existence of this seems to suggest we have to be more restrictive on the set of processes and names we admit if we are to support no-cloning.]

Bisimulation The computational dynamics gives rise to another kind of equivalence, the equivalence of computational behavior. As previously mentioned this is typically captured *via* some form of bisimulation.

The notion we use in this paper is derived from weak barbed bisimulation [26].

Definition 6. *An* observation relation, $\downarrow_{\mathcal{N}}$, *over a set of names,* $\mathcal{N}$, *is the smallest relation satisfying the rules below.*

$$\frac{y \in \mathcal{N},\ x \equiv_{\mathsf{N}} y}{x!(v) \downarrow_{\mathcal{N}} x} \qquad (\textsc{Out-barb})$$

$$\frac{P \downarrow_{\mathcal{N}} x \text{ or } Q \downarrow_{\mathcal{N}} x}{P|Q \downarrow_{\mathcal{N}} x} \qquad (\textsc{Par-barb})$$

We write $P \Downarrow_{\mathcal{N}} x$ *if there is* Q *such that* $P \Rightarrow Q$ *and* $Q \downarrow_{\mathcal{N}} x$.

Definition 7. *An* $\mathcal{N}$-barbed bisimulation *over a set of names,* $\mathcal{N}$*, is a symmetric binary relation* $\mathcal{S}_{\mathcal{N}}$ *between agents such that* $P \; \mathcal{S}_{\mathcal{N}} Q$ *implies:*

1. *If* $P \rightarrow P'$ *then* $Q \Rightarrow Q'$ *and* $P' \; \mathcal{S}_{\mathcal{N}} Q'$*.*
2. *If* $P \downarrow_{\mathcal{N}} x$*, then* $Q \Downarrow_{\mathcal{N}} x$*.*

P is $\mathcal{N}$*-barbed bisimilar to Q, written* $P \dot{\approx}_{\mathcal{N}} Q$*, if* $P \; \mathcal{S}_{\mathcal{N}} Q$ *for some* $\mathcal{N}$*-barbed bisimulation* $\mathcal{S}_{\mathcal{N}}$*.*

Contexts One of the principle advantages of computational calculi from the λ-calculus to the π-calculus is a well-defined notion of context, contextual-equivalence and a correlation between contextual-equivalence and notions of bisimulation. The notion of context allows the decomposition of a process into (sub-)process and its syntactic environment, its context. Thus, a context may be thought of as a process with a "hole" (written $\Box$) in it. The application of a context K to a process P, written $K[P]$, is tantamount to filling the hole in K with P. In this paper we do not need the full weight of this theory, but do make use of the notion of context in the proof the main theorem.

$$\overset{\text{CONTEXT}}{K ::= \Box \mid \mathsf{for}(\overrightarrow{y} \leftarrow x)K \mid x!(\overrightarrow{P}, K, \overrightarrow{Q}) \mid K|P}$$

Definition 8 (contextual application). *Given a context K, and process P, we define the* contextual application*,* $K[P] := K\{P/\Box\}$*. That is, the contextual application of K to P is the substitution of P for* $\Box$ *in K.*

Remark 4. Note that we can extend the definition of free and bound names to contexts.

7 Implementing rho calculus on modern computers

7.1 Processes and names

The algebra of process states (recapitulated below for convenience) is directly and succinctly represented by the Scala code in the listing below.

$$\overset{\text{PROCESS}}{P, Q ::= 0 \mid \mathsf{for}(y \leftarrow x)P \mid x!(Q) \mid *x \mid P|Q} \qquad \overset{\text{NAME}}{x, y ::= @P}$$

The fact that this compiles and the types are inhabited can be seen as a mechanical proof that – despite the strange mutual recursion between names and processes – the algebra is not ill-founded.

```
trait ProcessStates {
   type Name

   trait ProcessState
   case class Input(
      channel : Name, variable : Name, cont : ProcessState
   ) extends ProcessState
   case class Output(
      channel : Name, payload : ProcessState
   ) extends ProcessState
   case class Composition(
      left : ProcessState, right : ProcessState
   ) extends ProcessState
   case class Deref( name : Name ) extends ProcessState
}

trait Nominals {
   type Process

   trait Nominal
   case class Quote( proc : Process ) extends Nominal
}

object RhoStates extends ProcessStates with Nominals {
   type Process = ProcessState
   type Name = Nominal
   case object Zero extends Process
}
```

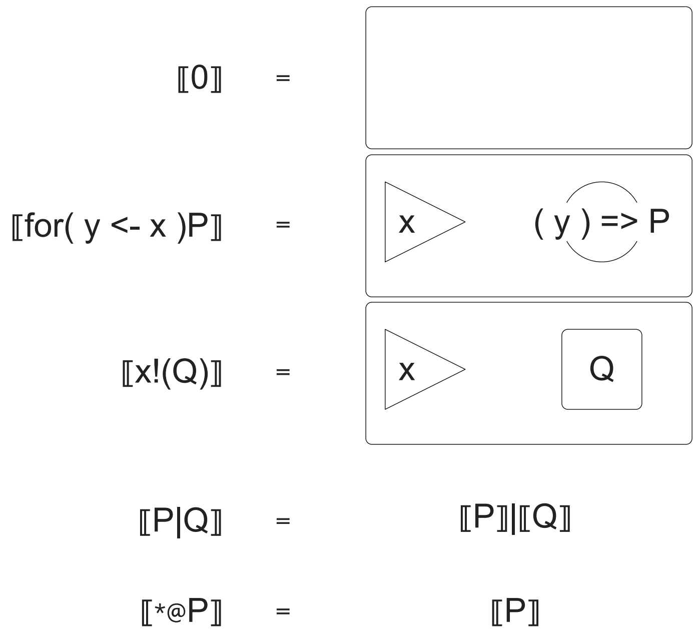
Remark 5. Note that this implementation makes it possible to easily incorporate other kinds of nominals, such as strings, or deBruijn indices, or URIs. The code simply makes a wrapper case class that extends the Nominal trait. This correspond to the fact that because rho is Turing complete we can create process encodings of all of the those data types and simply quote them to get those types of names. This illustrates one of the key uses of the notion of *namespace* in the rho calculus.

Remark 6. A namespace is simple a collection of names. This can be given extensionally, such as listing out the names that are members of a set. Or, it can be given intensionally, by providing a rule for recognizing when a name is included in a namespace or not. Namespaces are incredibly powerful, and this account is the first we know about that tames what is otherwise a typically thorny and error-prone account found in mainstream languages.

Next, we illustrate how to turn these process states into execution on a modern computer.

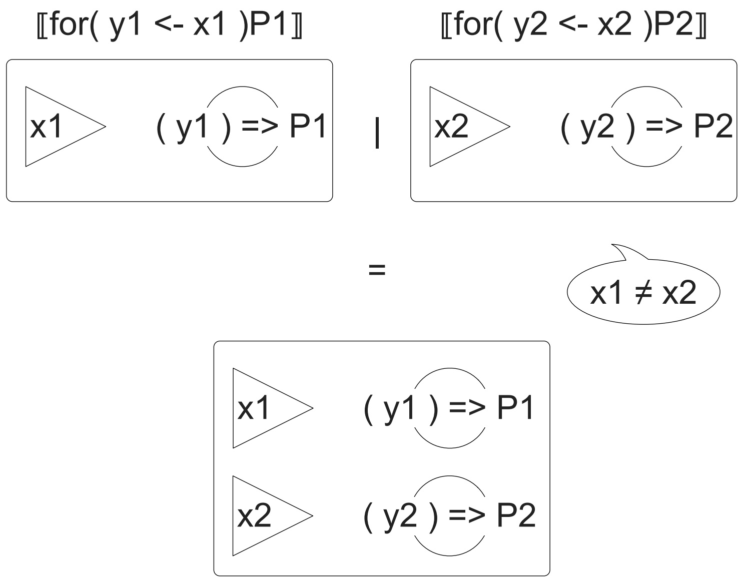
7.2 RSpace: a new kind of key-value store

It turns out that a variant of the Linda tuple space where input is not blocking is the critical innovation to an efficient implementation. Instead of blocking we turn input from a key into the storage of a continuation at the key. As the astute reader has already guessed, (the hashes of) channels become keys.

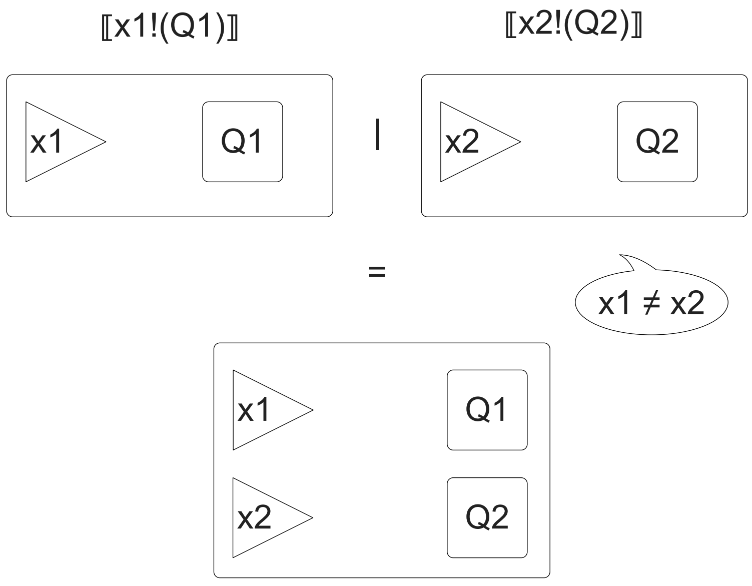


Thus, the diagram above indicates how to turn a given process state into an RSpace instance. The 4th equation depends on an operation on RSpace instances corresponding to parallel composition of process states.

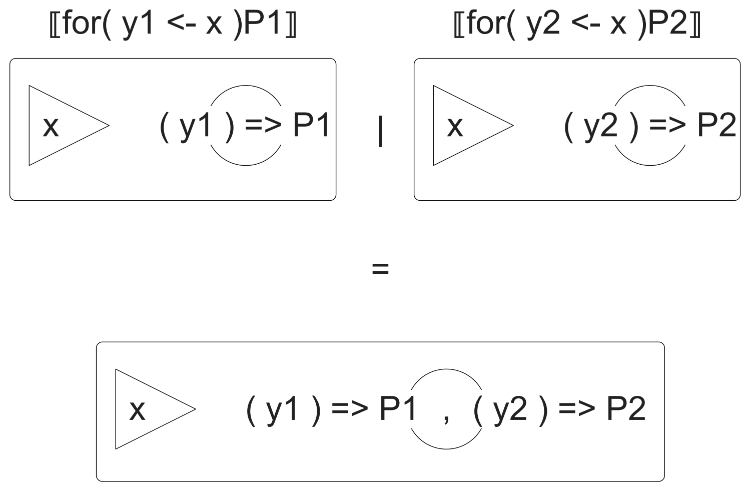
Parallel composition of RSpace*s* If we are combining two RSpaces that only have a single key-value pair, each; and the value is data and not continuation; and the keys are not equal, then we simply combing them into a single RSpace containing both key-value pairs.



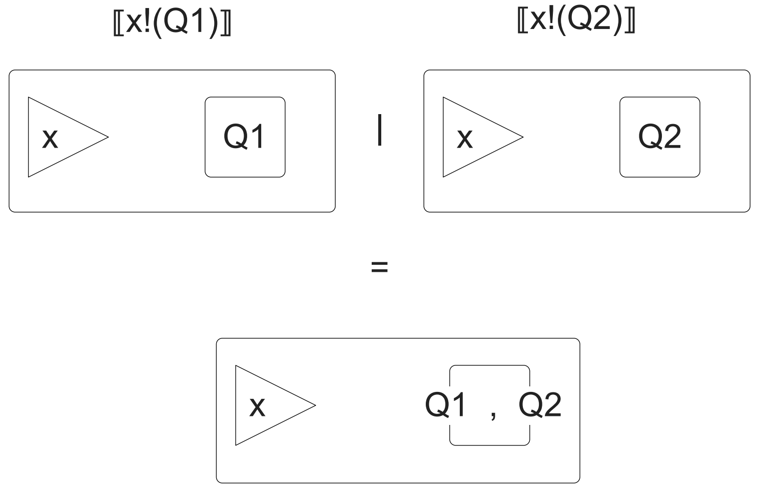
More generally, if none of the keys of the respective spaces overlaps, then we simply combine them into a single map.



Similarly, if we are combining two RSpaces that only have a single key-value pair, each; and the keys map to continuations; and the keys are not equal, then we simply combing them into a single RSpace containing both key-value pairs. Again, this generalizes to the case when the the key sets are disjoint.



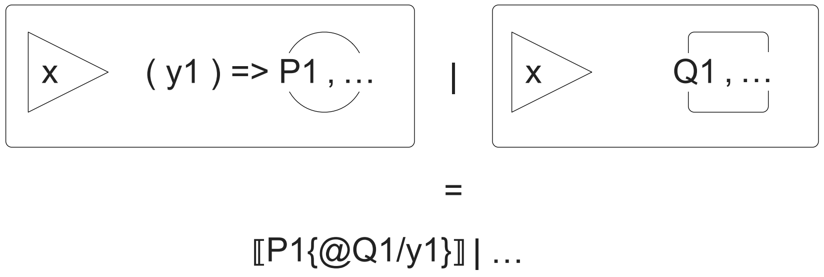
If we are combining two RSpaces that only have a single key-value pair, each; and the keys point to values; and the keys are equal, then we convert that into a single RSpace containing where the key maps to a multiset of the values.



Dually, for continuations, when we are combining two RSpaces that only have a single key-value pair, each; and the keys point to continuations; and the keys are equal, then we convert that into a single RSpace containing where the key maps to a multiset of the continuations.

These two cases obviously generalize to key sets that are not disjoint – but overlapping keys have the same polarity (data versus continuation) in each space.

What remains is when we have opposite polarity at the same key.



In this case, one simply zips over the overlapping keys applying the continuations to the data. [3]

Note that this is completely sequential execution. The only non-determinism is the fact that the sets are unordered. Can we do better? Can we utilize additional physical threads available on modern hardware to scale execution?

7.3 Compiling rho to RSpace

The following code relies on an implementation of RSpace that supports a polymorphic method p that takes a channel as its first argument and either a value or a continuation as its second argument. When it puts a value at a key that maps to a continuation, it removes the continuation and then applies the continuation to the value. Dually, when it puts a continuation at a key that maps to a value it removes the value and applies the continuation to it. Otherwise, it accumulates values or continuations in multisets mapped to by keys. Readers interested in more detail can consult [13].

```
trait RhoRuntime {
   def execute( p : RPorQ )( r : RSpace ) : Unit = {
     p match {
       case Zero => { }
       case Input( x, y, p ) => { r.put( x, ( y ) => p ) }
       case Output( x, q ) => { r.put( x, Q ) }
       case Par( p, q ) => {
         val t1 = new Thread() {
           override def run() = { execute( p )( r ) } };
         val t2 = new Thread() {
           override def run() = { execute( q )( r ) }
         }; t1.run(); t2.run()
       }
       case Deref( Ref( p ) ) => { execute( p )( r ) } }
} }
```

[3] We leave it to the reader to work out the details regarding what happens when the two sets are not the same size.

Ordering rho terms Another aspect of efficient execution of rho involves ordering terms. It turns out it is possible to give a total order to rho terms such that there is a unique representative for each of the equivalence classes of $\equiv$. Thus, if $\mathsf{N}(P)$ denotes the unique representative of $[P]_{\equiv}$, then when hashing a channel, say $@P$, we calculate actually hash $@\mathsf{N}(P)$. Again, for readers interested in more detail, please see [13].

8 Useful syntactic sugar

8.1 First class values

While it is standard in calculi such as the λ-calculus to define a variety of common values such as the natural numbers and booleans in terms of Church-numeral style encodings, it is equally common to simply embed values directly into the calculus. Not being higher-order, this presents some challenges for the π-calculus, but for the rho-calculus, everything works out very nicely if we treat values, e.g. the naturals, the booleans, the reals, etc as processes. This choice means we can meaninfully write expressions like $x!(\mathsf{5})$ or $u!(\mathsf{true})$, and in the context $\mathsf{for}(y \leftarrow x)P[*y]|x!(\mathsf{5})$ the value $\mathsf{5}$ will be substituted into P. Indeed, since operations like addition, multiplication, etc. can also be defined in terms of processes, it is meaningful to write expressions like $\mathsf{5}|+|\mathsf{1}$, and be confident that this expression will reduce to a process representing $\mathsf{6}$. Thus, we can also use standard mathematical expressions, such as $\mathsf{5}+\mathsf{1}$, as processes, and know that these will evaluate to their expected values. Further, when combined with for-comprehensions, we can write algebraic expressions, such as $\mathsf{for}(y \leftarrow x)\mathsf{5}{+}{*y}$, and in contexts like $(\mathsf{for}(y \leftarrow x)\mathsf{5}{+}{*y})|x!(1)$ this will evaluate as expected, producing the process (aka value) $\mathsf{6}$.

VALUES

$$\mathsf{Proc} ::= \cdots \mid \mathsf{Bool} \mid \mathsf{Long} \mid \mathsf{String} \mid \mathsf{Uri} \mid \mathsf{Collection} \mid \ldots$$

COLLECTION

$$\mathsf{Collection} ::= \mathtt{[}[\mathsf{Proc}]\mathtt{]} \mid \mathtt{(}\mathsf{Proc}\mathtt{,}\ [\mathsf{Proc}]\mathtt{)} \mid \mathtt{Set(}[\mathsf{Proc}]\mathtt{)} \mid \{[\mathsf{Proc}\mathtt{:}\mathsf{Proc}]\}$$

8.2 Pattern matching

Remember that rho allows processes to be sent: $x!(Q)$. Since we can send values, it will be useful to be able to pattern match on values. The following pattern-matching syntax captures very commonly used idioms.

Patterns

PATTERN
ProcPattern ::= _ | Var | @ProcPattern | VarRefKind Var | LogicalPattern | ValuePattern

LOGICAL-PATTERN
LogicalPattern ::= ProcPattern $\vee$ ProcPattern | ProcPattern $\wedge$ ProcPattern | ~ProcPattern

VALUE-PATTERN
ValuePattern ::= Ground | Collection | `Nil` | SimpleType

VAR-REF-KIND
VarRefKind ::= = | =*

Match targets Match targets are the syntactic contexts where patterns may occur and thus generate substitutions to be applied in the sequel.

MATCH-TARGETS
Proc ::= ... | `contract` Name ([Name] NameRemainder) = { Proc }
| `for` ([Receipt]){ Proc } | `select` { [Branch] }
| `match` Proc { [Case] } | `let` Decl Decls `in` { Proc }

Contracts Contracts provide for recursive definitions of processes. Here is an example of a contract that computes a (partial) fold over a list.

```
contract loop(
   @accumulatedValue, @lst,
   combinatorAndCondition, return
) = {
    match lst {
      [head ...tail] => {
        new result in {
          combinatorAndCondition!(
             head, accumulatedValue, *result
          ) |
          for (@done, @newValue <- result) {
            if (done){ return!(done, newValue) }
            else {
              loop!(
                 newValue, tail,
                 *combinatorAndCondition, *return
              )
            }
          }
        }
      }
      _ => { return!(false, accumulatedValue) }
    }
  }
```

Comprehensions Comprehensions become considerably enriched. They are not only the place where data can be matched, but also the place where we introduce more subtle control flow. As such, we deal with those together in the section on joins.

Select Select represents guarded summation.

Match Match allows dispatch on value shape without incurring the cost of synchronization. We find an example use of match in the fold computation above.

```
match lst {
  [head ...tail] => {
    new result in {
      combinatorAndCondition!(
         head, accumulatedValue, *result
      ) |
      for (@done, @newValue <- result) {
        if (done){ return!(done, newValue) }
        else {
          loop!(
             newValue, tail,
             *combinatorAndCondition, *return
          )
        }
      }
    }
  }
  _ => { return!(false, accumulatedValue) }
}
```

Let Let allows binding without incurring the cost of synchronization.

8.3 Joins

The basic template for joins is:

```
for(
  y11 <- x11 & ... & ym1 <- xm1 ; // received in any order,
     ... ; // but all received before the next row
  y1n <- x1n & ... & ymn <- xmn
){ ... }
```

We extend this template to include not only linear consumption of output ($y \leftarrow x$); but also reads ($y \lll x$), where the data is not consumed, but merely copied; and subscriptions ($y \Leftarrow x$) where the continuation is not consumed, but a copy forked off.

Additionally, we add syntax for synchronous interaction. We distinguish $y \leftarrow x$?! which is paired with (); and $y \leftarrow x$!?(...) the equivalent of an remote procedure call (rpc).

JOINS

$$\mathsf{Proc2} ::= \mathsf{Proc3} \mid \ldots \mid \texttt{for}\ (\mathsf{[Receipt]})\{\ \mathsf{Proc}\ \}$$

$$\mathsf{Receipt} ::= \mathsf{[LinearBind]} \mid \mathsf{[RepeatedBind]} \mid \mathsf{[PeekBind]}$$

$$\mathsf{[Receipt]} ::= \mathsf{Receipt} \mid \mathsf{Receipt}\ ;\ \mathsf{[Receipt]}$$

$$\mathsf{LinearBind} ::= \mathsf{[Name]}\ \mathsf{NameRemainder} \leftarrow \mathsf{NameSource}$$

$$\mathsf{[LinearBind]} ::= \mathsf{LinearBind} \mid \mathsf{LinearBind}\ \&\ \mathsf{[LinearBind]}$$

$$\mathsf{NameSource} ::= \mathsf{Name} \mid \mathsf{Name}\texttt{?!} \mid \mathsf{Name}\texttt{!?(}\mathsf{[Proc]}\texttt{)}$$

$$\mathsf{RepeatedBind} ::= \mathsf{[Name]}\ \mathsf{NameRemainder} \Leftarrow \mathsf{Name}$$

$$\mathsf{[RepeatedBind]} ::= \mathsf{RepeatedBind} \mid \mathsf{RepeatedBind}\&\mathsf{[RepeatedBind]}$$

$$\mathsf{PeekBind} ::= \mathsf{[Name]}\ \mathsf{NameRemainder} \lll \mathsf{Name}$$

$$\mathsf{[PeekBind]} ::= \mathsf{PeekBind} \mid \mathsf{PeekBind}\ \&\ \mathsf{[PeekBind]}$$

8.4 Unguessable versus unforgeable names

Since all the names of the rho calculus are generated from the codes of processes we know all of them up front. Security on channels, therefore, amounts to unguessability. There are an infinite number of names and we have to arrange our protocols to make it very, very hard to guess which channels are in use at any given time. On the one hand, we could delegate that to some black box which delivers us the next unguessable name when we ask for it. In this sense, it is therefore useful to reintroduce the π-calculus' new operator as a standin for that black box.

However, in many settings, such as in the RChain and F1R3FLY.io implementations execution is arranged so that names are not merely unguessable, but *unforgeable*, meaning that the execution environment guarantees not to allow outside agents to generate certain names. Thus, even if some malefactor guesses a particular name, it cannot eavesdrop on the channel associated with it because it can't execute code that uses that name because it didn't have the right to generate it. It could only do so had its code been in a scope where the generated name was sent to it by a (presumably) willing party.

Thus, the new operator not only provides useful abstraction over a black boxed unguessability algorithm, but can also be used to support unforgeability. Hence, we reintroduce to rho as a conservative extension.

$$\begin{array}{c}\text{UNFORGEABLE}\\ P ::= \ldots \mid \texttt{new}\ [\mathsf{NameDecl}]\ \texttt{in}\ P\end{array}$$

$$\mathsf{NameDecl} ::= \mathsf{Var} \mid \mathsf{Var}(\mathsf{Uri})$$

$$[\mathsf{NameDecl}] ::= \mathsf{NameDecl} \mid \mathsf{NameDecl}, [\mathsf{NameDecl}]$$

8.5 Summation

Mixed summation The presentation given so far is often referred to as the monadic, asynchronous version of the rho-calculus. There is a natural polyadic version, in which send and receive result in an exchange of a tuple of processes. Further there is a natural extension in which we recognize that certain processes can be made to behave like values and can therefore be replaced by more familiar representations of those values, and on top of this extension it becomes natural and compelling to add pattern-matching to send and receive.

All of these extensions, which we will explore in some detail in the subsequent sections, really only constitute syntactic sugar. Values, pattern-matching, let expressions, and more can be desugared back down to the original calculus. However, summation, aka non-deterministic choice, where a process is in a superposition of states that get "collapsed" through interaction with and environment, affords versions of the calculus that significantly extend the expressive power of the calculus.

Specifically, mixed summation, or non-deterministic choice over both guarded input (for-comprehension) and output is not merely a conservative extension of the calculus. Although there is an encoding of the calculus with mixed summation to the asynchronous calculus, it is not par-preserving. That is, if $[\![-]\!]_{async} : \mathsf{MixSumProc} \rightarrow \mathsf{Proc}$ is a mapping from the rho-calculus with mixed summation to the asynchronous polyadic rho-calulus, then it cannot be the case that

$$[\![P|Q]\!]_{async} = [\![P]\!]_{async}|[\![Q]\!]_{async}$$

for any such encoding. For good measure we throw in synchronous communication, but it is the mixed summation that constitutes the real jump in expressive power. These features are useful when dealing with probabilistic exection for both stochastic and quantum regimes. When relating our model of concurrent computation to these other notions it is important to track the points where there are significant increases in expressive power of our target language.

Because Milner's presentation of the polyadic π-calculus with mixed summation is so parsimonious we use it as a template for a similar version of the rho-calculus.

SUMMATION
$$M, N ::= \mathsf{0} \mid x.A \mid M + N$$

AGENT
$$A ::= (\overrightarrow{x})P \mid [\overrightarrow{P}]Q$$

PROCESS
$$P, Q ::= M \mid P|Q \mid *x$$

NAME
$$x ::= @P$$

In this presentation we adopt the syntactic conventions

$$\mathsf{for}(\overrightarrow{y} \leftarrow x)P := x.(\overrightarrow{y})P \qquad x!(\overrightarrow{Q});P := x.[\overrightarrow{Q}]P$$

The structural equivalence is modified thusly.

Definition 9. *The* structural congruence $\equiv$ *between processes is the least congruence containing alpha-equivalence and satisfying the commutative monoid laws (associativity, commutativity and* $\mathsf{0}$ *as identity) for parallel composition* $|$ *and summation* $+$*.*

The COMM rule is modified to incorporate non-deterministic choice.

COMM
$$\frac{x_t \equiv_{\mathsf{N}} x_s, \quad |\overrightarrow{y}| = |\overrightarrow{Q}|}{(R_1 + \mathsf{for}(\overrightarrow{y} \leftarrow x_t)P) \mid (x_s!(\overrightarrow{Q}).P' + R_2) \rightarrow P\{\overrightarrow{@Q}/\overrightarrow{y}\}|P'}$$

And contexts are likewise extended in the obvious manner.

SUMMATION-CONTEXT
$$K_M ::= \square \mid x.K_A \mid K_M + M$$

AGENT-CONTEXT
$$K_A ::= (\overrightarrow{x})K_P \mid [\overrightarrow{P}, K_P, \overrightarrow{P'}]Q \mid [\overrightarrow{P}]K_P$$

PROCESS-CONTEXT
$$K_P ::= K_M \mid P|K_P$$

Remark 7. Below we detail some of the useful syntactic sugar that makes reasoning and programming with the rho calculus considerably more convenient. The reader will be able to verify that that all the notational conventions developed below still make sense for the calculus extended with mixed summation.

9 Multilevel Agency

The rho calculus provides a number of important perspectives on multilevel agency. In this section we focus on three of them that can be reduced to calculi that can be coded up and executed on modern computer hardware.

9.1 Annihilation

Another important variation has to do with the rewrite rules. There is a nested recursion version of the COMM-rule that aligns with intuitions about the recursive nature of behavior in compositionally defined agents. While for your author the maths make it clearer than the English it is worth setting out some of the motivations for this variation of the dynamics of the rho calculus.

The reductionist view of science is that phenomena like medicinal cures rest on the phenomena of biology. That is, we find explanation of the mechanism underlying the medical procedure in the biological phenomena the procedure interacts with in some fashion. Meanwhile, the phenomena of biology rests, in turn, on the phenomena of chemistry; and chemistry rests on (nuclear) physics. By way of an example, in the reductionist narrative the interaction of two blood cells in the veins of the body of a person is ultimately explained in terms of the interactions of various chemical compounds, which are in turn explained in terms of the interaction of various atoms and their subatomic components, such as electrons, etc.

While this is accepted philosophy and pedagogy, it is quite difficult to reduce this account to actual computations, except in very limited or schematic and abstract fashion. For example, getting differential equations to work effectively across such scales (from cells to quarks) is simply not practical. The reductionist picture is comforting and to some extent supported by evidence, but not reducible to practice except in extremely limited cases.

The variation of the rho calculus dynamics presented here was developed to explore the power of compositionality to capture this kind of tower of reduction. Essentially, we develop of notion of process interaction that in turn relies on the interaction of "lower level" processes, and so on all the way down to a real bottom. The notion gives a precise definition of what it means to be "lower level" and how that relates to computational dynamics. In some sense, this is the crudest of pictures of multilevel agency, and yet given the efficacy of the process calculi to faithfully represent chemical, biochemical, and biological phenomena, there is some hope that it might not just be a pleasant abstraction.

First, we define what it means for two processes to annihilate each other.

Definition 10. *Annihilation: Processes P and Q are said to annihilate one another, written $P \perp Q$, just when $\forall R.P|Q \rightarrow^* R \Rightarrow R \rightarrow^* \mathsf{0}$.*

Thus, when $P \perp Q$, all rewrites out of $P|Q$ eventually lead to $\mathsf{0}$. Evidently, $P \perp Q \iff Q \perp P$, and $\mathsf{0} \perp \mathsf{0}$. Naturally, we can extend annihilation to names: $x \perp y \iff \dot{x} \perp \dot{y}$.

Annihilation affords a new version of the COMM-rule:

COMM
$$\frac{x_t \perp x_s, \quad |\overrightarrow{y}| = |\overrightarrow{Q}|}{(R_1 + \mathsf{for}(\overrightarrow{y} \leftarrow x_t)P) \mid (x_s!(\overrightarrow{Q}).P' + R_2) \to P\{\overrightarrow{@Q}/\overrightarrow{y}\}|P'}$$

All annihilation-based reduction happens in terms of reductions that happen at a lower degree of quotation, and grounds out in the fact that $\mathsf{0} \perp \mathsf{0}$ and thus $@\mathsf{0} \perp @\mathsf{0}$.

Example 1. For example, let $P_1 := \mathsf{for}(@\mathsf{0} \leftarrow @\mathsf{0})\mathsf{0}|@\mathsf{0}!(\mathsf{0})$. Then $P_1 \to \mathsf{0}$ because $\mathsf{0} \perp \mathsf{0}$. Suppose now that we set $x_0^-, x_0^+ := @\mathsf{0}$. Then, we can write P_1 as $\mathsf{for}(x_0^- \leftarrow x_0^-)\mathsf{0}|x_0^+!(\mathsf{0})$. Now, set

$$x_1^- := @(\mathsf{for}(x_0^- \leftarrow x_0^-)\mathsf{0}) \qquad\qquad x_1^+ := @(x_0^+!(\mathsf{0}))$$

Then define $P_2 := \mathsf{for}(x_1^- \leftarrow x_1^-)\mathsf{0}|x_1^+!(\mathsf{0})$. Then $P_2 \to \mathsf{0}$ because $x_1^- \perp x_1^+$, and hence $\mathsf{for}(x_1^- \leftarrow x_1^-)\mathsf{0} \perp x_1^+!(\mathsf{0})$.

More generally, set

$$x_i^- := @(\mathsf{for}(x_{i-1}^- \leftarrow x_{i-1}^-)\mathsf{0}) \qquad\qquad x_i^+ := @x_{i-1}^+!(\mathsf{0})$$

$$P_i := \mathsf{for}(x_{i-1}^- \leftarrow @x_{i-1}^-)\mathsf{0}|@x_{i-1}^+!(\mathsf{0})$$

Then $P_i \to \mathsf{0}$ and hence $x_i^- \perp x_i^+$.

This allows for a measure of reduction complexity.

9.2 Procedural reflection

The perspective on multilevel agency embodied in this conservative extension to the calculus is focused on a process being able to "see into the future of another process". That is, for a given process P to probe how another process, say Q, might evolve. This ability to "imagine" the behavior of another becomes introspection when a process turns this imaginative capacity onto itself. This introspection creates a kind of reflective tower, ala $\mathsf{3-Lisp}$ [37] or Brown [38]. Different floors or levels in this tower correspond to different levels of introspection. That is, level n is one level of introspection deeper than level $n-1$. It is in this sense that we view procedural reflection as providing a kind of multilevel agency.

Note that the fact that the π-calculus can be encoded into the rho calculus means that rho is Turing complete, or a model of universal computation. The reason we say that the procedurally reflective rho calculus is a conservative extension is because it doesn't add any fundamentally new expressive power in the way that adding an oracle that answers certain halting questions would. As such, we can write a meta-circular interpreter for the rho calculus in the rho calculus. Having done so, we could make reflection on the evolution of a process available to the process undergoing evolution. In some sense, this is the rho calculus equivalent of functional programming phenomena like call/cc, or more generally, delimited continuations [12].

To achieve this ability for a process, say P to look into its future we introduce an additional syntactic category, $x?P$. Then, if P evolves in a single step to P', then $x?P$ evolves to $x!(P')$. That is, a next step in P's evolution is made available at x. Rendering this idea in symbols, is simply adding the new syntactic category, extending the definition of free names in the obvious way, and one additional rewrite rule.

$$\overset{\text{PROCESS}}{P, Q ::= \mathsf{0} \mid \mathsf{for}(y \leftarrow x)P \mid x!(Q) \mid *x \mid x?P \mid P|Q} \qquad \overset{\text{NAME}}{x, y ::= @P}$$

$$\overset{\text{EQUIV}}{P|\mathsf{0} \equiv P \quad P|Q \equiv Q|P \quad P|(Q|R) \equiv (P|Q)\mathsf{R}}$$

$$\overset{\text{ALPHA}}{\mathsf{for}(y \leftarrow x)P \equiv \mathsf{for}(z \leftarrow x)(P\{z/y\} \ \mathsf{if} z \notin \mathsf{FN}(P)}$$

$$\overset{\text{COMM}}{\mathsf{for}(y \leftarrow x)P \mid x!(\overrightarrow{Q}) \rightarrow P\{@Q/y\}}$$

$$\overset{\text{PAR}}{\frac{P \rightarrow P'}{P|Q \rightarrow P'|Q}} \qquad \overset{\text{EQUIV}}{\frac{P \equiv P' \qquad P' \rightarrow Q' \qquad Q' \equiv Q}{P \rightarrow Q}}$$

$$\overset{\text{REFL}}{\frac{P \rightarrow P'}{x?P \rightarrow x!(P')}}$$

What makes this version of procedural reflection distinct and interesting is that it fits well within a concurrency setting. It is common folklore that call/cc-like continutations and concurrency are not the best bedfellows. Essentially, call/cc provides the ability to revert *global* state. But, in some very real sense there is no global state, or the state shared amongst a collection of agents is partitioned in such a way that not every agent has access and there is no shared way to roll back the clock for all agents. Once one agent has fired a missile, there is no time machine that rolls everyone's state back and puts the missile back in the silo. This fact about the partitioning of state is closely related to the famous arrow of time problem in physics, but we won't delve into it here [39]. Instead, we note that the form procedural reflection takes in the rho calculus is perfectly amenable to multi-agent settings in which state is partitioned and time cannot be globally rolled back.

Naturally, all of the syntactic sugar described above works with this variant of rho. As a result we can provide a very compact implementation of exploring the state space of a process.

```
contract checkStateSpace( p, c, t, a ) = {
  // check that we still have a positive number of attempts left
  match a with {
    case 0 => ( p, t.set( @c, { @p : "fails" } ) ) // nope
    case _ => { // yep
      // if the state already matches the condition
      match check( p, c ) with {
        // we are done
        case true => (p, t.set( @c, { @p : "succeeds" } ) )
        // otherwise...
        case false => {
          // check to see if p has a successor state
          match p with {
            // nope
            case Nil => ( p, t.set( @c, { @p : "fails" } ) )
            // yep, so check to see if we have seen p before
            case _ => {
              match t.getOrElse( @c, { @p : p } ) with {
                // we have encountered a failure,
                // so we attempt to get a different branch
                case { @p : "fail" } => {
                  checkStateSpace!( p', c, t, a-1 )
                }
                // we have encountered a loop
                // in the state space graph
                case { @p : p' } => {
                  checkStateSpace!( p', c, t, a-1 )
                }
                // we are descending a branch
                case { pcode : pproc } => {
                  new n in {
                    // let pproc run for 1 step
                    n?( pproc )
                    // catch the next state
                    | for( @p' <- n ) {
                      checkStateSpace!(
                         p', c,
                         t.set( @c, { @p : p' } ), a
                      )
                    }
```

```
                        }
                      }

                    }
                  }
                }
              }
            }
          }
        }
      }
```

9.3 Programmable contexts

Context-parameterised COMM rules By way of introduction we first note a natural variation of the COMM rule.

$$\text{COMM-K} \quad \mathsf{for}(y \leftarrow x)P \mid x!(\overrightarrow{Q}) \rightarrow P\{@K[Q]/y\}$$

The intuition behind this variation is that often P and Q are at different levels of some protocol stack. For example, P may be an application written to use TCP as a communication protocol. Meanwhile, Q may be an application write to use HTTP as a communication protocl. This is still a common occurrence in building systems out of 3rd party Internet applications. It is possible to write adapters that enable P to communicate with Q. We can encapsulate this adapter in the context K.

Likewise, we can abstract error correction protocols into a context K by having K be comprised of the encoder and decoder processes. More generally, K represents a form of pipe-fitting or impedance matching between processes that would otherwise not be able to effectively communicate.

Process grammar

PROCESS

$$P, Q ::= \mathsf{0} \mid \mathsf{U}(x) \mid \mathsf{for}(y \leftarrow x)P \mid x!(Q) \mid P|Q \mid *x \mid \mathsf{COMM}(K)$$

NAME

$$x, y ::= @\langle K, P\rangle$$

CONTEXT

$$K ::= \square \mid \mathsf{for}(y \leftarrow x)K \mid x!(K) \mid P|K$$

Definition 11. Free and bound names *The calculation of the free names of a process, P, denoted $\mathsf{FN}(P)$ is given recursively by*

$$\mathsf{FN}(0) = \emptyset \qquad \mathsf{FN}(\mathsf{U}(x)) = \{x\} \qquad \mathsf{FN}(\mathsf{for}(y \leftarrow x)P) = \{x\} \cup \mathsf{FN}(P) \setminus \{y\}$$

$$\mathsf{FN}(x!(P)) = \{x\} \cup \mathsf{FN}(P) \qquad \mathsf{FN}(P|Q) = \mathsf{FN}(P) \cup \mathsf{FN}(Q) \qquad \mathsf{FN}(*x) = \{x\}$$

$$\mathsf{FN}(\mathsf{COMM}(K)) = \mathsf{FN}(K) \qquad \mathsf{FN}(\Box) = \emptyset \qquad \mathsf{FN}(\mathsf{for}(y \leftarrow x)K) = \{x\} \cup \mathsf{FN}(K) \setminus \{y\}$$

$$\mathsf{FN}(x!(K)) = \{x\} \cup \mathsf{FN}(K) \qquad \mathsf{FN}(P|K) = \mathsf{FN}(P) \cup \mathsf{FN}(K)$$

An occurrence of x *in a process* P *is bound if it is not free. The set of names occurring in a process (bound or free) is denoted by* $\mathsf{N}(P)$.

9.4 Operational semantics

Finally, we introduce the computational dynamics. What marks these algebras as distinct from other more traditionally studied algebraic structures, e.g. vector spaces or polynomial rings, is the manner in which dynamics is captured. In traditional structures, dynamics is typically expressed through morphisms between such structures, as in linear maps between vector spaces or morphisms between rings. In algebras associated with the semantics of computation, the dynamics is expressed as part of the algebraic structure itself, through a reduction reduction relation typically denoted by $\rightarrow$. Below, we give a recursive presentation of this relation for the calculus used in the encoding.

Catalyze

$$\mathsf{U}(x) \mid *@\langle K, Q\rangle \rightarrow \mathsf{COMM}(K) \mid x!(Q)$$

Comm

$$\frac{x_t \equiv_{\mathsf{N}} x_s}{\mathsf{COMM}(K) \mid \mathsf{for}(y \leftarrow x_t)P \mid x_s!(Q) \rightarrow P\{@\langle K, Q\rangle/y\}}$$

Par

$$\frac{P \rightarrow P'}{P|Q \rightarrow P'|Q}$$

Equiv

$$\frac{P \equiv P' \qquad P' \rightarrow Q' \qquad Q' \equiv Q}{P \rightarrow Q}$$

We write $P \rightarrow$ if $\exists Q$ such that $P \rightarrow Q$ and $P \not\rightarrow$, otherwise.

9.5 Movement in space: an example

In the following example let $x = @\langle\Box, 0\rangle$ and $y = @\langle K, Q\rangle$ for some K and Q.

$$\begin{array}{l} \quad \mathsf{for}(y \leftarrow x)*y|x!(P)|\mathsf{COMM}(K_1) \\ \rightarrow \\ \quad *y\{@\langle K_1, P\rangle/y\} \\ = \\ \quad K_1[P] \end{array}$$

Now, if K_1 is also of the form $\mathsf{for}(y' \leftarrow x')*y'|x'!(\Box)|\mathsf{COMM}(K_2)|R$, then P will move to the location $K_2[P]|R$. That is, $K_1[P] \rightarrow K_2[P]|R$. And if K_2 is likewise of the form $\mathsf{for}(y'' \leftarrow x'')*y''|x''!(\Box)|\mathsf{COMM}(K_3)|R'$, then P will move to the location $K_3[P]|R|R'$.

Thus, we have a means to describe movement of a process from location to location.

10 Fuzzy rho

10.1 Manifest of goods

We need an action on process states, together with a binary operation taking a pair of names to an element of that action.

$$(- * -) : A \times \mathsf{Proc} \rightarrow \mathsf{Proc}$$

$$(- \circ -) : @\mathsf{Proc} \times @\mathsf{Proc} \rightarrow A$$

Equipped with these we can define a new COMM rule

$$\frac{\text{COMM}}{\mathsf{for}(y \leftarrow x)P \mid u!(Q) \rightarrow P\{@((u \circ x) * Q)/y\}}$$

The intuition is that communication isn't based on exact match. Instead, the strength of the match, as measured by $u \circ x$, determines how much of the data is transmitted, which is calculated by $(u \circ x) * Q$.

To tighten these intuitions we ask for a "measure" on Proc. We use an order-theoretic gadget, namely a quantale, to compare process states. That is, we ask for $\mu : \mathsf{Proc} -> \mathsf{K}$ where K is a quantale. Finally, we require the action to carry the order information. That is $A = \mathsf{K}$. Then we need that

$$a1 \leq a2 \Rightarrow \mu(a1 * P) \leq \mu(a2 * P)$$

Under these conditions we can calculate

$$\mathsf{for}(y \leftarrow x)P \mid u_1!(\overrightarrow{Q}) \rightarrow P\{@(u_1 \circ x) * Q/y\}$$

$$\mathsf{for}(y \leftarrow x)P \mid u_2!(\overrightarrow{Q}) \rightarrow P\{@(u_2 \circ x) * Q/y\}$$

Which ensures that if the match is worse, less data is transmitted.

10.2 A model

First, let's adopt a notation that when $x = @P$, then $\#x = P$. Now, we set $A = @\mathsf{Proc}$. This choice affords the following definitions.

$$x_1 \circ x_2 = @(\#x_1 | \#x_2)$$

$$u * P = P\{(u \circ x)/x | x \in \mathsf{FN}(P)\}$$

$$\mu(P) = @P$$

which the reader can check satisfies the conditions.

11 Stochastic and quantum rho

If we follow the route followed by the development of the stochastic π-calculus, then the core idea is that we can associate to each channel (type) a rate. The rate determines the relative frequency of interaction and thus, just as with standard practice in chemistry, we can use a form of the Gillespie algorithm to turn process expressions decorated with channel rates into simulations.

Somewhat surprisingly, this technique extends to quantum simulations. We use complex numbers for rates and there is a corresponding quantum version of Gillespie that turns process expressions decorated with complex valued channel rates into simulations. Under this interpretation the COMM rule becomes a version of the Born rule.

However, there are different approaches to consider which we will return to after providing the details of the more standard approaches.

11.1 Stochastic rho

TBD

11.2 Quantum rho

TBD

11.3 Stochasticity and annihilation

TBD

11.4 Quantum mechanics directly encoded

TBD

12 Translating other calculi into rho

12.1 The π-calculus

TBD

12.2 The λ-calculus

TBD

12.3 The ambient calculus

TBD

13 Applied reflection: Reflective set theory and reflective graphs

In this section we illustrate, albeit briefly, that reflection has a much wider utility than just computation. We describe reflective versions of set theory and graph theory and illustrate via some core examples how reflection accounts for an interesting range of phenomena.

13.1 Reflective set theory

This presentation will be just a sketch. A more rigorous formal account is most likely built using Awodey's algebraic set theory.

RED-SET
$$\mathsf{S}[A] ::= \{(A \mid \mathsf{S}[A])^*\}$$

BLACK-SET
$$\mathsf{S}[A] ::= \{(A \mid \mathsf{S}[A])^*\}$$

RSET-PLAYER
$$\mathsf{RSet}_{player} = \mathsf{S}[\mathsf{S}[\mathsf{RSet}_{player}]]$$

RSET-OPPONENT
$$\mathsf{RSet}_{opponent} = \mathsf{S}[\mathsf{S}[\mathsf{RSet}_{opponent}]]$$

RSET
$$\mathsf{RSet} = \mathsf{RSet}_{player} \oplus \mathsf{RSet}_{opponent}$$

The intuition is that we have two distinct *copies* of set theory, which we will denote for purposes of discussion, red set theory and black set theory. Both of these theories are considered parametric in a theory of atoms. Thus, these theories are instances of a Fraenkl-Mostowski set theory (FM set theory). But, now we make the atoms of the red set theory be black sets, and the atoms of the black set theory be red sets.

Again, computer science provides a unique perspective on these age old notions. The idea of copies of a theory comes about when we reify meta theory as theory, something implicit in Awodey's work. Computer science also provides a very practical means for realizing it: we have different constructors (red braces versus black braces); and different accessors (a red element-of operation versus a black element of operation). Further, computer science is quite comfortable with these kinds of mutually recursive structures. The mutual recursion bottoms out in the red empty set and the black empty set.

Now, the presentation above is only suggestive because Kleene star would only generate sets with finite cardinality. This is why Awodey's algebraic set theory is the better formalism to adapt to formalize these intuitions. But, from the perspective of computer science this presentation very much matches how it might be coded up.

What motivates such a strange construction? It turns out that FM set theory has been successfully applied to reasoning about nominal phenomena, such as binders in the π-calculus. But, once again, one is left with a dissatisfying theory of atoms without structure. This theory explains where atoms come from and how they can have exactly the same structure as sets. In other words, it reconciles FM set theory with ZF set theory.

13.2 Reflective graphs

13.3 Well-formedness

This is an algebraic theory, and thus it is syntactic in nature. It provides a language of graphs that are built out of logical sentences. In more detail, the theory admits three kinds of sentences:

- recognizing an admissible vertex: $\mathsf{G}[X,V];\Gamma \vdash v$;
- recognizing an admissible variable: $\mathsf{G}[X,V];\Gamma \vdash x$;
- and, recognizing a well formed graph: $\mathsf{G}[X,V];\Gamma \vdash g$

Variables are used to capture references to vertices which are in turn used to form edges between vertices. As such, judging the wellformedness of a graph depends on the use of references. Hence, a judgement makes use of a dependency list, Γ which is just a sequence of variables. That is,

$$\Gamma ::= () \mid x, \Gamma$$

Notationally, we overload the comma to indicate concatenation of sequences. Thus, given $\Gamma_1 = x_{11}, \ldots, x_{1m}$ and $\Gamma_2 = x_{21}, \ldots, x_{2n}$, then $\Gamma_1, \Gamma_2 = x_{11}, \ldots, x_{1m}, x_{21}, \ldots, x_{2n}$

A graph expression is given by the grammar

$$g, h ::= 0 \mid v|g \mid x|g \mid g \otimes h \mid \mathsf{let}\ x = v\ \mathsf{in}\ g \mid \langle \mathsf{let}\ x_1 = v_1\ \mathsf{in}\ g, \mathsf{let}\ x_2 = v_2\ \mathsf{in}\ h\rangle$$

Pronunciation The graph constructors are pronounced as follows.

– 0 – "the empty graph", or just "empty";
– $v|g$ – "g with the vertex v adjoined", or "adjoin v to g";
– $g_1 \otimes g_2$ – "the graph formed by juxtaposing g_1 and g_2", or "juxtapose g_1 and g_2", or just "g_1 and g_2";
– $\mathsf{let}\ x = v\ \mathsf{in}\ g$ – "let x stand for v in g"; or "let x be v in g";
– $\langle \mathsf{let}\ x_1 = v_1\ \mathsf{in}\ g_1, \mathsf{let}\ x_2 = v_2\ \mathsf{in}\ g_2 \rangle$ – "the graph formed by connecting g_1 to g_2 with and edge from x_1 to x_2"; or, "connect g_1 to g_2 with and edge from x_1 to x_2".

We say x is fresh in g if x does not occur in g.

The judgment $\mathsf{G}[X, V]; \Gamma \vdash g$ is pronounced "$\mathsf{G}[X, V]$ thinks that g is well-formed, given dependencies, Γ ." Similarly, $\mathsf{G}[X, V]; \Gamma \vdash v$ is pronounced "$\mathsf{G}[X, V]$ thinks that v is an admissible vertex, given dependencies, Γ;" and $\mathsf{G}[X, V]; \Gamma \vdash x$ is pronounced "$\mathsf{G}[X, V]$ thinks that x is an admissible variable, given dependencies, Γ."

A rule of the form

$$\frac{H_1, \ldots, H_n}{C} R$$

is pronounced "R concludes that C given H_1, …, H_n".

The rules for judging when a vertex or a variable are admissible or graph is well formed are as follows

$$\frac{}{\mathsf{G}[X, V]; () \vdash 0} Foundation$$

$$\frac{v \in V}{\mathsf{G}[X, V]; () \vdash v} Verticity \quad \frac{x \in X}{\mathsf{G}[X, V]; \emptyset \vdash x} Variation$$

$$\frac{\mathsf{G}[X, V]; \Gamma \vdash g \quad \mathsf{G}[X, V]; () \vdash v}{\mathsf{G}[X, V]; \Gamma \vdash v|g} Participation \quad \frac{\mathsf{G}[X, V]; \Gamma \vdash g \quad \mathsf{G}[X, V]; () \vdash x}{\mathsf{G}[X, V]; \Gamma, x \vdash x|g} Dependence$$

$$\frac{\mathsf{G}[X, V]; \Gamma_1 \vdash g_1 \ \mathsf{G}[X, V]; \Gamma_2 \vdash g_2}{\mathsf{G}[X, V]; \Gamma_1, \Gamma_2 \vdash g_1 \otimes g_2} Juxtaposition[\Gamma_1 \cap \Gamma_2 = \emptyset]$$

$$\frac{\mathsf{G}[X, V]; \Gamma, x \vdash v|g}{\mathsf{G}[X, V]; \Gamma \vdash \mathsf{let}\ x = v\ \mathsf{in}\ g} Nomination[x\ fresh\ in\ g]$$

$$\frac{\mathsf{G}[X, V]; \Gamma_1 \vdash \mathsf{let}\ x_1 = v_1\ \mathsf{in}\ g_1 \quad \mathsf{G}[X, V]; \Gamma_2 \vdash \mathsf{let}\ x_2 = v_2\ \mathsf{in}\ g_2}{\mathsf{G}[X, V]; \Gamma_1, \Gamma_2 \vdash \langle \mathsf{let}\ x_1 = v_1\ \mathsf{in}\ g_1, \mathsf{let}\ x_2 = v_2\ \mathsf{in}\ g_2 \rangle} Connection[\Gamma_1 \cap \Gamma_2 = \emptyset]$$

Where's the reflection? As with all the other formalism put forward in this paper, this account of graphs begins with a two-level type decomposition. Specifically, it acknowledges that as a type, the type of graphs is dependent on the type of vertices and edges. But, once this dependency is acknowledged, we can readily form and solve fixed point equations. Such as

$$\mathsf{R}_V = \mathsf{G}[X, 1 + \mathsf{R}_V]$$

In fact, it is also possible to do something similar for edges.

13.4 Reflection as integration

It is common to think of the derivative of a polynomial functor as punching a hole in the data type, that is, for a given functor, T, the functor $\partial\mathsf{T}$ is the type of 1-holed contexts of T. By the same token, we can think of reflection as a kind of reflection in the sense that it is plugging holes in a functor. Specifically, if $\mathsf{T}[X]$ is parametric in X, then $\mathsf{R}(\mathsf{T}) = \mathsf{T}[\mathsf{R}(\mathsf{T})]$ represents the elimination of that dependency.

Unfortunately, a duality between derivation and reflection supporting an analog of the fundamental theorem of calculus is not so easily obtained. In particular, it is not generally the case that we have $\mathsf{R}(\partial\mathsf{T}) = \mathsf{T}$. It would be of interest to find conditions under which this does hold.

14 From formalism to physical intuition and back

14.1 Indexed namespaces

Let's begin the discussion by identifying an indexed set of names.

$$\mathsf{Z}(0) = @0$$

$$\mathsf{Z}(n) = @(\mathsf{for}(y \leftarrow \mathsf{Z}(n-1))0)$$

$$\mathsf{Z} = \{\mathsf{Z}(n) \mid n \in \mathbb{N}\}$$

While process states are perfectly suited for using channels as locations at which to rendezvous for data exchange, correlating that to more common notions of locations, such as points in a metric space, is not immediately obvious. The namespace Z, because of its natural indexed structure, provides an intuitive bridge between the two ways of thinking about locations. Indeed, given a map like this from the naturals to names, it should be relatively straightforward for the reader to extend this idea to the rationals, and indeed to the computable reals.

14.2 Processes as physical forces

Next, we consider a collection of processes that we will think of as components in a “force”, where by force we mean something like a physical force, such as the electromagnetic force or the strong or weak forces.

$$\mathbb{F}(n) = \mathsf{for}(y \leftarrow \mathsf{Z}(n))\{\mathsf{Z}(n+1)!(*y) \mid \mathbb{F}(n)\}$$

$$\mathbb{F} = \Pi_{n \in \mathbb{N}} \mathbb{F}$$

Now, model the action of this force on an object, we represent the object as a process as well, say Q, and locate it somewhere in Z, i.e. $\mathsf{Z}(i)!(Q)$. Then the action of $\mathbb{F}$ on Q, is just the parallel composition, $\mathbb{F} \mid \mathsf{Z}(i)!(Q)$. Note that because $\mathsf{Z}(i)!(Q)$ cannot reduce, it tends to stay at rest. However, when acted on by an outside force, namely $\mathbb{F}$, we notice that Q is relocated first to $\mathsf{Z}(i+1)$ and then $\mathsf{Z}(i+2)$ and tends to stay on this linear trajectory unless there are other forces at play.

It is also worth noting that there is a natural notion of *rate* of travel. Specifically, COMM events provide a local notion of time. We can see that $\mathbb{F}$ moves objects located in Z along at a rate of one unit of distance (as measure by the index) per COMM event.

$$\mathbb{F} \mid \mathsf{Z}(i)!(Q) \rightarrow \mathbb{F} \mid \mathsf{Z}(i+1)!(Q)$$

It is quite straightforward to write down forces that are non-linear. For example,

$$\mathbb{G}(n) = \mathsf{for}(y \leftarrow \mathsf{Z}(n))\{\mathsf{Z}(n^2)!(*y) \mid \mathbb{G}(n)\}$$

$$\mathbb{G} = \Pi_{n \in \mathbb{N}} \mathbb{G}$$

The reader is invited to write down, following this method, an analog of Newtonian gravity. Of course, to do this requires we associate to objects a notion of mass. One of the most natural ways to do this is to measure the computational complexity of a process and use (a function of) this number as a stand in for mass.

15 Conclusions and future work

One way to understand the rho-calculus is a study in the *seams* of the π-calculus. There are several holes in Milner’s calculus:

– *the zero process*, $\mathsf{0}$, is an input to the theory; adding other elements here corresponds to supplying “builtin” processes, and as such constitutes a way to add seamlessly add values to the language;
– *names* are an input to the theory, which we discussed at length;

– *the source of non-determinism* is an input to the theory.

The results in the stochastic and quantum regimes can really be seen as an initial exploration of the a general theory of this third dependency.

Stepping back to an even wider perspective we need to understand the rho calculus not as a single model of computation. Instead, it is a way of thinking about computation. It says that *computation arises from interaction*. This is a remarkable shift from the point of view normally espoused in theoretical computer science. Both Turing and Church's models saw computation as *functions acting on data*.

Indeed, there is a similar kind of divergence of viewpoints in the physical sciences. Chemistry arises from interactions of molecules. With the exception of gravitation, standard model physics arises from interaction of particles. Yet, all these disciplines are using the computational toolkit initially developed by Newton, which again envisages computation as functions acting on data.

Even more modern views of computation, such as are embodied in category theory, are still strongly oriented towards this functional view. A topos, for example, is really just a Cartesian closed category (CCC) with a subobject classifer; but as a CCC it is a model of the λ-calculus, which is decidedly functional in its perspective.

We hope that this review of the vast range of computational phenomenon successfully, indeed elegantly, accounted for by the interactive paradigm will inspire the reader who has made it this far to launch her own investigation into the matter. The rho calculus provides not a single model of computation, but a conceptual, and mathematically precise toolkit for explore the interactive paradigm.

But the development doesn't stop here. In much the same way that the rho calculus and its variants provide a framework for the investigation of individual protocols, algorithms, and processes seen as patterns of interaction, the OSLF algorithm provides a framework for the investigation of populations of protocols, algorithms, and processes seen as patterns of interactions. Rather than having to craft – by hand – a new type system for each variant of the rho calculus (or indeed each variant model of computation) that is discovered, OSLF allows one to *generate* a type system automatically. The formulae or types of these generated type systems correspond to populations of processes that are bisimilar to each other. This gives an entirely different vantage point on the relationships of computation, logic, and interaction.

In particular, this cuts to the very foundations of mathematics. From one perspective we can view set theory as a *data* structure designed to support protocols (namely proofs) between agents (namely mathematicians). That the foundations of mathematics need only formalize the data structures supporting proofs might have made sense as a language for mathematicians practicing before the advent of high performance networked computing infrastructure, but in the age of $\mathsf{ChapGPT}$ we have to acknowledge that computer programs – which are themselves just mathematical objects – have enough agency not only to do proofs and come up with counter examples, but to make conjectures!

We submit that a language of mathematics that supports modern practice should embrace the language of agency and take it to its bosom. That is, we believe that the mobile process calculi not only provide a much more subtle language for the data structures used to conduct proofs, but also a language to reason about agents that used those data structures, be they mathematicians, computer programs or collectives comprised of both.

Acknowledgments. The author wishes to thank Mike Stay for his long time collaboration and Lylli and Max and Naomi and Gabi Meredith for putting up with my absences while developing this theory.

References

1. Martín Abadi and Bruno Blanchet. Analyzing security protocols with secrecy types and logic programs. In *POPL*, pages 33–44. ACM, 2002.
2. Martín Abadi and Bruno Blanchet. Secrecy types for asymmetric communication. *Theor. Comput. Sci.*, 298(3):387–415, 2003.
3. Samson Abramsky. Algorithmic game semantics and static analysis. In *SAS*, volume 3672 of *Lecture Notes in Computer Science*, page 1. Springer, 2005.
4. Luís Caires. Behavioral and spatial observations in a logic for the pi-calculus. In *FoSSaCS*, pages 72–89, 2004.
5. Luis Caires. Spatial logic model checker, Nov 2004.
6. Luís Caires and Luca Cardelli. A spatial logic for concurrency (part I). *Inf. Comput.*, 186(2):194–235, 2003.
7. Luís Caires and Luca Cardelli. A spatial logic for concurrency - II. *Theor. Comput. Sci.*, 322(3):517–565, 2004.
8. Luca Cardelli. Brane calculi. In *CMSB*, volume 3082 of *Lecture Notes in Computer Science*, pages 257–278. Springer, 2004.
9. Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theor. Comput. Sci.*, 240(1):177–213, 2000.
10. Vincent Danos and Cosimo Laneve. Core formal molecular biology. In *ESOP*, volume 2618 of *Lecture Notes in Computer Science*, pages 302–318. Springer, 2003.
11. Mariangiola Dezani-Ciancaglini and Ugo de'Liguoro. Sessions and session types: An overview. In Cosimo Laneve and Jianwen Su, editors, *Web Services and Formal Methods, 6th International Workshop, WS-FM 2009, Bologna, Italy, September 4-5, 2009, Revised Selected Papers*, volume 6194 of *Lecture Notes in Computer Science*, pages 1–28. Springer, 2009.
12. R. Kent Dybvig, Simon L. Peyton Jones, and Amr Sabry. A monadic framework for delimited continuations. *J. Funct. Program.*, 17(6):687–730, 2007.
13. F1R3FLY.io developers. F1r3fly.io implementation, 2024. [Online; accessed 24-July-2024].
14. Cédric Fournet, Fabrice Le Fessant, Luc Maranget, and Alan Schmitt. Jocaml: A language for concurrent distributed and mobile programming. In *Advanced Functional Programming*, volume 2638 of *Lecture Notes in Computer Science*, pages 129–158. Springer, 2002.
15. Cédric Fournet, Georges Gontheir, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR 1996*, volume 1119 of *Lecture Notes in Computer Science*, pages 406–421. Springer-Verlag, 1996.
16. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.
17. Allen L. Brown Jr., Cosimo Laneve, and L. Gregory Meredith. Piduce: A process calculus with native XML datatypes. In *EPEW/WS-FM*, volume 3670 of *Lecture Notes in Computer Science*, pages 18–34. Springer, 2005.
18. Cosimo Laneve and Gianluigi Zavattaro. Foundations of web transactions. In *FoSSaCS*, volume 3441 of *Lecture Notes in Computer Science*, pages 282–298. Springer, 2005.
19. Greg Meredith. Documents as processes: A unification of the entire web service stack. In *WISE*, pages 17–20. IEEE Computer Society, 2003.
20. L. Gregory Meredith and Matthias Radestock. Namespace logic: A logic for a reflective higher-order calculus. In *TGC* [21], pages 353–369.
21. L. Gregory Meredith and Matthias Radestock. A reflective higher-order calculus. *Electr. Notes Theor. Comput. Sci.*, 141(5):49–67, 2005.
22. Lucius Meredith, Jan 2017.
23. Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.
24. Robin Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, 1992.
25. Robin Milner. Elements of interaction - turing award lecture. *Commun. ACM*, 36(1):78–89, 1993.
26. Robin Milner. The polyadic π-calculus: A tutorial. *Logic and Algebra of Specification*, Springer-Verlag, 1993.
27. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992.
28. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, II. *Inf. Comput.*, 100(1):41–77, 1992.
29. nLab authors. generators and relations. `https://ncatlab.org/nlab/show/generators+and+relations`, July 2024. Revision 7.

30. Corrado Priami, Aviv Regev, Ehud Shapiro, and William Silverman. Application of a stochastic name-passing calculus to representation and simulation of molecular processes. *Inf. Process. Lett.*, 80(1):25–31, 2001.

31. Aviv Regev, William Silverman, and Ehud Shapiro. Representation and simulation of biochemical processes using the pi-calculus process algebra. In *Pacific Symposium on Biocomputing*, pages 459–470, 2001.

32. Davide Sangiorgi. Beyond bisimulation: The "up-to" techniques. In *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 161–171. Springer, 2005.

33. Davide Sangiorgi. On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst.*, 31(4):15:1–15:41, 2009.

34. Davide Sangiorgi and Robin Milner. The problem of "weak bisimulation up to". In *CONCUR*, volume 630 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 1992.

35. Davide Sangiorgi and David Walker. *The Pi-Calculus - a theory of mobile processes.* Cambridge University Press, 2001.

36. Peter Sewell, Pawel T. Wojciechowski, and Asis Unyapoth. Nomadic pict: Programming languages, communication infrastructure overlays, and semantics for mobile computation. *ACM Trans. Program. Lang. Syst.*, 32(4):12:1–12:63, 2010.

37. Brian Cantwell Smith. Reflection and semantics in lisp. In Ken Kennedy, Mary S. Van Deusen, and Larry Landweber, editors, *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages, Salt Lake City, Utah, USA, January 1984*, pages 23–35. ACM Press, 1984.

38. Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: A nonreflective description of the reflective tower. *LISP Symb. Comput.*, 1(1):11–37, 1988.

39. Wikipedia contributors. Arrow of time — Wikipedia, the free encyclopedia, 2024. [Online; accessed 24-July-2024].

40. N. William and Parker. The germ-plasm: a theory of heredity (1893), by august weismann. 2018.

16 Appendix: rholang specification

16.1 Rholang grammar

PROCESS

Proc ::= Proc1 | Proc|Proc1

CONDITIONAL

Proc1 ::= Proc2 | `if` (Proc) Proc2 | `if` (Proc) Proc2 `else` Proc1
| `new` [NameDecl] `in` Proc1 | Name `!?`([Proc]) SynchSendCont

IO

Proc2 ::= Proc3 | `contract` Name ([Name] NameRemainder) = { Proc }
| `for` ([Receipt]){ Proc } | `select` { [Branch] } | `match` Proc4 { [Case] }
| Bundle { Proc } | `let` Decl Decls `in` { Proc }

OUTPUT

Proc3 ::= Proc4 | Name Send ([Proc])

DISJUNCTION

Proc4 ::= Proc5 | Proc4 `or` Proc5

CONJUNCTION

Proc5 ::= Proc6 | Proc5 `and` Proc6

MATCHING

Proc6 ::= Proc7 | Proc7 `matches` Proc7 | Proc6 `==` Proc7 | Proc6 `!=` Proc7

COMPARISON

Proc7 ::= Proc8 | Proc7 < Proc8 | Proc7 <= Proc8 | Proc7 > Proc8
| Proc7 >= Proc8

ADDITION

Proc8 ::= Proc9 | Proc8 + Proc9 | Proc8 - Proc9 | Proc8 ++ Proc9
| Proc8 -- Proc9

MULTIPLICATION

Proc9 ::= Proc10 | Proc9 * Proc10 | Proc9 / Proc10 | Proc9 % Proc10
| Proc9 %% Proc10

NEGATION

Proc10 ::= Proc11 | `not` Proc10 | -Proc10

NESTING

Proc11 ::= Proc12 | Proc11 . Var([Proc]) | (Proc4)

DEREFERENCE

Proc12 ::= Proc13 | *Name

MATCH-DISJUCTION

Proc13 ::= Proc14 | VarRefKind Var | Proc13 \/ Proc14

MATCH-CONJUNCTION

Proc14 ::= Proc15 | Proc14 /\ Proc15

MATCH-NEGATION

Proc15 ::= Proc16 | ~Proc15

???

$$\mathsf{Proc16} ::= \{\mathsf{Proc}\} \quad | \ \mathsf{Ground} \quad | \ \mathsf{Collection} \quad | \ \mathsf{ProcVar} \quad | \ \mathtt{Nil} \quad | \ \mathsf{SimpleType}$$

???

$$[\mathsf{Proc}] ::= \mathbf{eps} \quad | \ \mathsf{Proc} \quad | \ \mathsf{Proc}\ \mathtt{,}\ [\mathsf{Proc}]$$

$$\mathsf{Decl} ::= [\mathsf{Name}]\ \mathsf{NameRemainder} \leftarrow [\mathsf{Proc}]$$

$$\mathsf{LinearDecl} ::= \mathsf{Decl}$$

$$[\mathsf{LinearDecl}] ::= \mathsf{LinearDecl} \quad | \ \mathsf{LinearDecl}\,;[\mathsf{LinearDecl}]$$

$$\mathsf{ConcDecl} ::= \mathsf{Decl}$$

$$[\mathsf{ConcDecl}] ::= \mathsf{ConcDecl} \quad | \ \mathsf{ConcDecl}\ \mathtt{\&}\ [\mathsf{ConcDecl}]$$

$$\mathsf{Decls} ::= \mathbf{eps} \quad | \ \mathtt{;}\ [\mathsf{LinearDecl}] \quad | \ \mathtt{\&}\ [\mathsf{ConcDecl}]$$

$$\mathsf{SynchSendCont} ::= \mathtt{.} \quad | \ ;\mathsf{Proc1}$$

$$\mathsf{ProcVar} ::= \_ \quad | \ \mathsf{Var}$$

$$\mathsf{Name} ::= \_ \quad | \ \mathsf{Var} \quad | \ \mathtt{@}\mathsf{Proc12}$$

$$[\mathsf{Name}] ::= \mathbf{eps} \quad | \ \mathsf{Name} \quad | \ \mathsf{Name}\,,[\mathsf{Name}]$$

Bundle ::= `bundle+` | `bundle-` | `bundle0` | `bundle`

Receipt ::= ReceiptLinearImpl | ReceiptRepeatedImpl | ReceiptPeekImpl

[Receipt] ::= Receipt | Receipt `;` [Receipt]

ReceiptLinearImpl ::= [LinearBind]

LinearBind ::= [Name]NameRemainder ← NameSource

[LinearBind] ::= LinearBind | LinearBind&[LinearBind]

NameSource ::= Name | Name`?!` | Name`!?(`[Proc]`)`

ReceiptRepeatedImpl ::= [RepeatedBind]

RepeatedBind ::= [Name]NameRemainder<=Name

[RepeatedBind] ::= RepeatedBind | RepeatedBind&[RepeatedBind]

ReceiptPeekImpl ::= [PeekBind]

PeekBind ::= [Name]NameRemainder<<-Name

[PeekBind] ::= PeekBind | PeekBind&[PeekBind]

Send ::= `!` | `!!`

Branch ::= ReceiptLinearImpl=>Proc3

[Branch] ::= Branch | Branch[Branch]

Case ::= Proc13=>Proc3

[Case] ::= Case | Case[Case]

NameDecl ::= Var | Var`(`UriLiteral`)`

[NameDecl] ::= NameDecl | NameDecl`,`[NameDecl]

BoolLiteral ::= `true` | `false`

Ground ::= BoolLiteral | LongLiteral | StringLiteral | UriLiteral

$$\mathsf{Collection} ::= \mathtt{[}[\mathsf{Proc}]\mathsf{ProcRemainder}\mathtt{]} \quad \mid \mathsf{Tuple} \quad \mid \mathtt{Set(}[\mathsf{Proc}]\mathsf{ProcRemainder}\mathtt{)}$$
$$\mid \{[\mathsf{KeyValuePair}]\mathsf{ProcRemainder}\}$$

$$\mathsf{KeyValuePair} ::= \mathsf{Proc}\,\mathtt{:}\,\mathsf{Proc}$$

$$[\mathsf{KeyValuePair}] ::= \mathbf{eps} \quad \mid \mathsf{KeyValuePair} \quad \mid \mathsf{KeyValuePair}\,\mathtt{,}\,[\mathsf{KeyValuePair}]$$

$$\mathsf{Tuple} ::= \mathtt{(}\mathsf{Proc}\,\mathtt{,)} \quad \mid \mathtt{(}\mathsf{Proc}\,\mathtt{,}\,[\mathsf{Proc}]\mathtt{)}$$

$$\mathsf{ProcRemainder} ::= \ldots\mathsf{ProcVar} \quad \mid \mathbf{eps}$$

$$\mathsf{NameRemainder} ::= \ldots\mathtt{@}\mathsf{ProcVar} \quad \mid \mathbf{eps}$$

$$\mathsf{VarRefKind} ::= \mathtt{=} \quad \mid \mathtt{=*}$$

$$\mathsf{SimpleType} ::= \mathtt{Bool} \quad \mid \mathtt{Int} \quad \mid \mathtt{String} \quad \mid \mathtt{Uri} \quad \mid \mathtt{ByteArray}$$

16.2 Rholang operational semantics

TBD