Operational semantics in logical form

Lucius Gregory Meredith[1]
and Mike Stay[2]

[1] Managing Partner, RTech Unchained 9336 California Ave SW, Seattle, WA 98103, USA
lgreg.meredith@gmail.com
[2] CTO, Pyrofex
metaweta@gmail.com

Abstract. We describe an algorithm that takes as input a model of computation specified as a rewrite system and generates as output a spatial-behavioral type system for the model.

1 Introduction

Programmers who work on large scale deployments, like Facebook, Google, or Microsoft services, know that it is just not cost effective to write code at scale in an untyped language. Why waste human time and effort on writing tests (and debugging them) and worrying about coverage for cases that a compiler can completely and exhaustively automate? That is why the big three (Facebook, Google, and Microsoft) [12] [5] [19] all spent tens of millions of dollars each developing typed versions of javascript. It's cheaper than continuing to throw good money after bad on commercial development at industrial scale in an untyped language. All told, the cost to the industry of this one issue alone is in the hundreds of millions of dollars.

But, as we move deeper and deeper into the era of distributed concurrent computing, the sorts of errors that mainstream programming languages catch with their type systems are only the tip of the iceberg. The Java type system cannot catch liveness errors. The Rust type system cannot catch security errors. However, the programming language semantics community has known for 30 years about techniques for catching a much wider range of safety and liveness errors. Session types, spatial and behavioral types are well explored areas bearing considerable fruit on just these sorts of issues. And, just like with javascript, it is taking the community ages to get these techniques into the hands of front line developers.

What if it didn't have to be this way? What if the type system for spatial and behavioral types could be algorithmically generated from the specification of the programming language? That's what this paper is about: an algorithm that takes as input a description of a model of computation as a rewrite system and *generates* as output a spatial-behavioral type system for the model.

The formalization of the rewrite system uses Fiore's higher-order abstract syntax to specify the syntax with binders [14] [15]. This is augmented with a theory of graphs specifying the source and targets of the rewrite rules. This yields what amounts to a Cartesian closed category (CCC) [23], call it $\mathcal{C}$. To $\mathcal{C}$ we apply the following recipe. First we hit is with Yoneda [6], resulting in a category of presheaves [26]. We then hit that with

a construction described below turning the homs into complete Heyting algebras [21]. Finally, we take the Grothendieck construction [26]. The result is that types are pairs $(\mathbf{U}, \mathbf{X})$ consisting of a sort, $\mathbf{X}$, from the original lambda theory and a filter on it, $\mathbf{U}$.

The resulting category, denoted by $\mathsf{NT}(\mathcal{C})$, enjoys a rich logical structure (in fact, it is a topos [18]) and thus provides a stable means for combining collections of witnesses of various properties in various ways. Specifically the homs are complete Heyting algebras and hence enjoy that particular menu of lattice operations. Additionally, there are products and sums, which can be made to interact coherently with the lattice operations. Likewise, we have two different forms of implication.

Further, because the rewrites have been explicitly encoded we get typed versions of redex constructors, as well as typed versions of reduction, and modal operators ala Hennessy-Milner logics [34], all of this mechanism delivers a powerful behavioral typing apparatus. Additionally, we find that there are typed versions of the term constructors. These play in the role of spatial types.

1.1 Summary of contributions and outline of paper

Summary of contributions In principle, everything here could be derived by a careful analysis of Williams and Stay's native types [45]. However, the native types paper rushes to introduce advanced features, like dependent types, while missing certain key points. For example, the derived meta theory introduces new terms that are not covered by the reduction relation of the calculus. Specifically, the product and sum terms introduced have no corresponding reductions and thus the type judgments presented in the paper are effectively useless for real programs because they will never reduce. Moreover, it is actually important to pay attention to the fact that there are two forms of conjunctions, disjunctions, and implication, mirroring the sort of structure typically found in semantics for linear logic [3].

Fortunately, it is possible to expand the reduction relation in a natural way to include the reduction of these terms. Additionally, one of the main goals of this work has been to provide an algorithm parametric in the kind of collections used to gather witnesses. Here we illustrate that the use of Yoneda and sub can be modified to provide a much wider range of collections, and yielding logical connectives corresponding to the usual operations on those collections.

Finally, we set up a discussion regarding the proper role and shape of the derived meta theory. In native types this meta theory is decidedly sequential. This means that the logic is at odds with computational models like the π-calculus [34] and the rho-calculus [29] that are *concurrent*. What is needed is an analog of the topos that is fundamentally concurrent, a rho-pos, if you will. This is because concurrency is more fundamental than sequentiality.

Outline of paper First we provide an intuitive motivating example. Then we review Fiore's higher order abstract syntax [14] [15] as a mechanism for providing a categorical model for syntax with binders. To this we add a theory of graphs to capture the operational semantics. Then we review the native types construction [45] in more detail.

The review out of the way we present the algorithm. Then illustrate it by deriving a type system for the core of rholang [30]. We prove a substitutability theorem [25]. We consider some examples types. Finally we conclude with some future directions for research.

2 Monoidal logic: a motivating example

In this section we develop a (not so) toy example that motivated your author's search for an algorithm of this generality. We present a logic for reasoning about monoids.

2.1 Freely generated monoids

$$\underset{\text{Monoid-Expr}}{M[G] ::= \mathsf{e} \mid g \mid M[G]{*}M[G]}$$

where $g \in G$. The syntax is only slightly unusual. It's almost standard BNF except that the grammar is parametric in a set G of generators. Readers familiar with parametric types found in languages like OCaml, F#, Scala, Haskell, Java, etc should be very comfortable with this notation. The grammar freely generates a collection of expressions over G involving $*$ and e, which we write $\mathcal{L}(M[G])$. We must whack this collection down by the familiar relations.

$$\begin{aligned} e{*}m = m = m{*}e \\ (m_1{*}m_2){*}m_3 \quad = \quad m_1{*}(m_2{*}m_3) \end{aligned}$$

In other words, formally the monoid is $\mathcal{L}(M[G])/=$.

2.2 Monoid logic: syntax and semantics

$$\begin{aligned} \phi, \psi ::= \mathsf{true} \mid \neg\phi \mid \phi\&\psi \\ \mid \mathbf{e} \mid \mathbf{g} \mid \phi{*}\psi \end{aligned}$$

	collection operations	
$[\![\mathsf{true}]\!]$	$=$	$\mathcal{L}(M[G])$
$[\![\neg\phi]\!]$	$=$	$\mathcal{L}(M[G])\backslash[\![\phi]\!]$
$[\![\phi\&\psi]\!]$	$=$	$[\![\phi]\!] \cap [\![\psi]\!]$
	spatial operations	
$[\![\mathbf{e}]\!]$	$=$	$\{m \in \mathcal{L}(M[G]) : m = \mathsf{e}\}$
$[\![\mathbf{g}]\!]$	$=$	$\{m \in \mathcal{L}(M[G]) : m = g\}$
$[\![\phi{*}\psi]\!]$	$=$	$\{m \in \mathcal{L}(M[G]) : \exists m_1, m_2 . m = m_1{*}m_2, m_1 \in [\![\phi]\!], m_2 \in [\![\psi]\!]\}$

The logic neatly divides into operations on collections of witnesses (the usual boolean connectives, interpreted as operations on sets), and spatial operations, i.e. logical operations derived from the syntax of monoid *expressions*. It is completely algorithmically generated. Any theory presented via generators and relations like this (and hence any Lawvere theory, and hence a certain class of monads) has a corresponding logic.

Yet, it is is very expressive. Consider the following formula.

$$prime := \neg\mathbf{e} \;\&\; \neg(\neg\mathbf{e} * \neg\mathbf{e})$$

As the reader may verify, this picks out exactly the primes of any monoid generated from a set G of generators. In fact, an extension of this algorithm was used to *generate* namespace logic. The extension adds a third section to the logic that provides an algorithmic treatment of the Hennessy-Milner style modal connectives corresponding to the rewrite rules of the rho-calculus. Monoidal logic is a proper fragement of namespace logic and the formula for primes defined above can also be used to pick out *single-threaded* processes in the rho-calculus, illustrating an unexpected connection between primality and sequentiality.

This analysis was the beginning of a search for the proper expression of the algorithm that captures not only model-checking semantics like the one above, but a type and judgment system. The search lead to a category theoretic treatment. Originally, it seemed that the approach might be captured by distributive laws amongst monads. However, some recent no-go theorems for distributive laws showed that the approach could not work for the parameterization contemplated.

In this connection, it is worth describing the intuitions that have lead to the algorithm in its current form and scope. The reality is that when taken together Curry-Howard and realizability force a particular shape to the semantics of logical formulae. They must be collections of witnesses, where the witnesses are computations evincing the properties they bear witness to. Traditionally, logicians and computer scientists alike have chosen sets as the natural form of collection. However, that is not the only choice. In fact, it is not the only natural choice.

This perspective becomes clear when we think about the other role that logic plays. Logic is also the basis of query. Currently, Github is only searchable in terms meta-data. Imagine querying it using types: given a type, a program that type checks is a match, otherwise a miss. The developers of Hoogle [37] tried this for searching Haskell's Cabal package system. The problem with Hoogle is that the type system is too coarse. Both

$$\mathsf{sort} : \mathsf{list\ int} \to \mathsf{list\ int}$$

and

$$\mathsf{shuffle} : \mathsf{list\ int} \to \mathsf{list\ int}$$

bear the same type signature, but they do opposite things. Spatial behavioral types, however, yield a considerably more expressive query language.

Note, however, that Github repos are not *sets*. They undergo update, including insertion, deletion, and modification. Being able to parameterize the algorithm to take as input the kind of collections used to collect witnesses, and thereby generate the corresponding logical connectives is vital to be able to consider query applications like the one contemplated here.

Christian Williams and Mike Stay devised the core of the formalization of the algorithm considered here. However, there are some key differences.

3 Specifying a model of computation: lambda theories and higher order abstract syntax

We follow Fiore and Hur [14] closely.

Definition 1. *A second order signature Σ enjoys the following data $\Sigma = (T, O, |-|)$ with*

- *T a set of types*
- *O a set of operators*
- *$|-| : (T^* \times T^* \to T)$ and arity function.*

We write $o : (\overrightarrow{\sigma_1})\tau_1 \times \ldots \times (\overrightarrow{\sigma_n})\tau_n \to \tau$ when $|o| = ((\overrightarrow{\sigma_1})\tau_1, \ldots, (\overrightarrow{\sigma_n})\tau_n, \tau)$

3.1 Typing contexts

We will consider terms in typing contexts. Typing contexts have two zones, each respectively typing variables and metavariables. Variable typings are types. Metavariable typings are parameterised types: a metavariable of type $[\sigma_1, \ldots, \sigma_n]\tau$, when parameterised by terms of type $\sigma_1, \ldots, \sigma_n$ yield a term of type τ. Thus, we use the following representation for typing contexts $m_1 : [\overrightarrow{\sigma_1}]\tau_1, \ldots, m_j : [\overrightarrow{\sigma_j}]\tau_j \rhd x_1 : \sigma'_1, \ldots, x_k : \sigma'_k$.

3.2 Terms

Signatures give rise to terms. These are built up by means of operators from both variables and metavariables, and hence referred to as second-order. Terms are considered up to α-equivalence by stipulating that for every operator o in the term $o(\ldots, (\overrightarrow{x}_i)t_i, \ldots)$ the $\overrightarrow{x}_i$ are bound in t_i.

3.3 Judgments and inference rules

Judgments Judgments have a two part context, one for metavariables (Θ), and the other for variables (Γ), and a single type inference for the consequence. Symbolically, we write $\Theta \rhd \Gamma \vdash t : \tau$, pronouncing it "$\Theta$ and Γ think that t is of type τ."

Inference rules Because we have inference rules both for lambda theories and also the meta theory, there are rather a bit more of them, hence we allow a bit of flexibility in the format. When space allows, we adopt a horizontal rather than vertical syntax. Thus for an inference rule with judgments $J_1 \ \ldots \ J_n$ as the hypotheses, and J as the consequent, we write $J_1 \ \ldots \ J_n \Vdash J$. When there are no hypotheses, we write $\Vdash J$. When horizontal space is at a premium, then we opt for the more traditional format

$$\textsc{Rule}\quad \frac{J_1 \ \ldots \ J_n}{J}$$

Typed term formation The reader familiar with the ordinary simply typed λ-calculus will see the family resemblance. The key difference is that we have metavariables which enriches contexts, abstraction, and application.

$$\textsc{Variable}\quad \frac{}{\Theta \rhd x : \tau, \Gamma \vdash x : \tau}$$

$$\textsc{Meta-variable}\quad \frac{m : [\tau_1, \ldots, \tau_n]\tau, \Theta \rhd \Gamma \vdash t_i : \tau_i}{m : [\tau_1, \ldots, \tau_n]\tau, \Theta \rhd \Gamma \vdash m[t_1, \ldots, t_n] : \tau}$$

$$\textsc{Abstraction}\quad \frac{\Theta \rhd \overrightarrow{x}_1 : \overrightarrow{t}_1, \ldots, \overrightarrow{x}_n : \overrightarrow{t}_n, \Gamma \vdash t_i : \tau_i \quad o : (\overrightarrow{\sigma_1})\tau_1 \times \ldots \times (\overrightarrow{\sigma_n})\tau_n \to \tau}{\Theta \rhd \Gamma \vdash o((\overrightarrow{x}_1)t_i, \ldots, (\overrightarrow{x}_n)t_n) : \tau}$$

$$\textsc{Application}\quad \frac{\Theta \rhd \overrightarrow{x}_1 : \overrightarrow{t}_1, \ldots, \overrightarrow{x}_n : \overrightarrow{t}_n, \Gamma \vdash t : \tau \quad \Theta \rhd \Gamma \vdash t_i : \tau_i}{\Theta \rhd \Gamma \vdash t\{t_i/x_i\} : \tau}$$

$$\textsc{Meta-application}\quad \frac{m_1 : [\overrightarrow{\sigma}_1]\tau_1, \ldots, m_n : [\overrightarrow{\sigma}_n]\tau_n \rhd \Gamma \vdash t : \tau \quad \Theta \rhd \overrightarrow{x}_1 : \overrightarrow{\sigma}_1, \ldots, \overrightarrow{x}_n : \overrightarrow{\sigma}_n, \Gamma \vdash t_i : \tau_i}{\Theta \rhd \Gamma \vdash t\{\!\{(\overrightarrow{x}_i)t_i/m_i\}\!\} : \tau}$$

Substitution

$$x_j\{t_i/x_i\} = t_j$$

$$m[\ldots, s, \ldots]\{t_i/x_i\} = m[\ldots, s\{t_i/x_i\}, \ldots]$$

$$(o(\ldots, (y_1, \ldots, y_k)s, \ldots))\{t_i/x_i\} = o(\ldots, (z_1, \ldots, z_k)((s\{z_j/y_j\})\{t_i/x_i\}), \ldots)$$

Meta-substitution

$$x\{\!\{(\overrightarrow{x}_i)t_i/m_i\}\!\} = x$$

$$m_k[s_1,\ldots,s_m]\{\!\{(\overrightarrow{x}_i)t_i/m_i\}\!\} = t_k\{(s_j\{\!\{(\overrightarrow{x}_i)t_i/m_i\}\!\})/x_{i,j}\}$$

$$(o(\ldots,(\overrightarrow{x})s,\ldots))\{\!\{(\overrightarrow{x}_i)t_i/m_i\}\!\} = o(\ldots,(\overrightarrow{x})s\{\!\{(\overrightarrow{x}_i)t_i/m_i\}\!\},\ldots)$$

3.4 Equations

$$\textsc{Axiom-Eq}\ \frac{}{f : X \to TY, g : X \to TY, E \vdash f = g : X \to TY}$$

$$\textsc{Reflexivity-Eq}\ \frac{f : X \to TY}{E \vdash f = f : X \to TY} \qquad \textsc{Symmetry-Eq}\ \frac{E \vdash f = g : X \to TY}{E \vdash g = f : X \to TY}$$

$$\textsc{Transivity-Eq}\ \frac{E \vdash f = g : X \to TY \quad E \vdash g = h : X \to TY}{E \vdash f = h : X \to TY}$$

$$\textsc{Composition-Eq}\ \frac{E \vdash f_1 = g_1 : X \to TY \quad E \vdash f_2 = g_2 : Y \to TZ}{E \vdash f_1; f_2 = g_1; g_2 : X \to TZ}$$

$$\textsc{Parameterization-Eq}\ \frac{E \vdash f = g : X \to TY}{E \vdash f\langle P\rangle = g\langle P\rangle : X \to TY} \qquad \textsc{Local-Character-Eq}\ \frac{E \vdash f\ e_i = g\ e_i : X_i \to TY}{E \vdash f = g : X \to TY}$$

Where given $h : X \to TY$

$$h\langle P\rangle := X \otimes P \xrightarrow{h\otimes id} TY \otimes P \xrightarrow{strength_T} T(Y \otimes P)$$

$$\{e_i : X_i \to X\}\ \textit{are jointly epi}$$

4 A brief review of native types

Consider the following recipe. Given a category, Th (we use this name to be suggestive that our category is a theory) we apply Yoneda to it, resulting in a category of presheaves. To this we apply a construction sub to the representables, converting them to complete Heyting algebras. To this we apply the Grothendieck construction. In symbols

$$\mathsf{NT} := \int \mathsf{sub}\ \mathsf{Y}$$

This is the core construction of native types.

It treats models of computation as presented by some rewrite system with binders, and thus begins with a meta-theory, CCC, the 2-category of Cartesian closed categories. It treats NT as an endofunctor on CCC, deriving a new meta theory $\mathsf{NT}(\mathsf{C})$ for each category, $\mathcal{C}$, in $|\mathsf{CCC}|$, the objects of CCC. The language presented below is a syntax for the derived meta-theory.

4.1 The topos

First, let's review how subobject classifiers work in a topos.

Given a category $\mathsf{Th} = 1 \xrightarrow{f} M$, where 1 is terminal, Th has five morphisms: $\{1, M, f, !, f!\}$, where $!\colon M \to 1$ and $f!\colon M \xrightarrow{!} 1 \xrightarrow{f} M$.

The subobject classifier: Ω. To understand the subobject classifier Ω in the topos $\hat{\mathsf{Th}} = \mathrm{Set}^{\mathsf{Th}^{\mathrm{op}}}$, let's consider the behavior on objects and morphisms.

Behavior on objects. $\Omega(1)$ is the set of sieves on 1, the set of subsets of morphisms into 1 that are closed under precomposition with morphisms in Th. $\{1, !\} = \top$ $\{\} = \bot$ We can't have 1 without $! = 1!$. We can't have $!$ without $1 = !f$.

$\Omega(M)$ is the set of sieves on M, the set of subsets of morphisms into M that are closed under precomposition with morphisms in Th. $\{M, f, f!\} = \top$ $\{f, f!\}$ $\{\} = \bot$ We can't have M without $f = Mf$ and $f! = Mf!$. We can't have f without $f!$. We can't have $f!$ without $f = f!f$.

Behavior on morphisms. $\Omega(f)(S) =$ union of morphisms $g \in \{1, !\}$ such that $fg \in S$

$$\begin{aligned}\Omega(f)\colon \quad \Omega(M) &\to \Omega(1)\\ \{M, f, f!\} &\mapsto \{1, !\}\\ \{f, f!\} &\mapsto \{1, !\}\\ \{\} &\mapsto \{\}\end{aligned}$$

$\Omega(!)(S) =$ union of morphisms $g \in \{M, f, f!\}$ such that $!g \in S$

$$\begin{aligned}\Omega(!)\colon \quad \Omega(1) &\to \Omega(M)\\ \{1, !\} &\mapsto \{M, f, f!\}\\ \{\} &\mapsto \{\}\end{aligned}$$

$\Omega(f!)(S) =$ union of morphisms $g \in \{M, f, f!\}$ such that $f!g \in S$

$$\begin{aligned}\Omega(M)\colon \quad \Omega(M) &\to \Omega(M)\\ \{M, f, f!\} &\mapsto \{M, f, f!\}\\ \{f, f!\} &\mapsto \{M, f, f!\}\\ \{\} &\mapsto \{\}\end{aligned}$$

Also, obviously, Ω of an identity morphism is the identity function on the set S.

Logical operations

Negation. In this section, $!$ is redefined temporarily: $\neg\colon \Omega \to \Omega$ is the character of $\bot\colon 1 \to \Omega$, where $\bot$ is the character of $!\colon 0 \to 1$.

Remark 1. The object 0 in the topos $\begin{array}{l} 0 : \mathsf{Th}^{\mathsf{op}} \to \mathrm{Set} \\ \qquad c \mapsto \emptyset \end{array}$

Remark 2. The object 1 in the topos $\begin{array}{l} 1 : \mathsf{Th}^{\mathsf{op}} \to \mathrm{Set} \\ \qquad c \mapsto 1 \end{array}$

Remark 3. The natural transformation $!\colon 0 \to 1$ The natural transformation $!\colon 0 \to 1$ assigns to each object the trivial function $!\colon \emptyset \to 1$.

Remark 4. The character $\bot\colon 1 \to \Omega$ of $!$ The character $\bot\colon 1 \to \Omega$ of $!$ is a natural transformation that assigns to every object X the function that maps $\bullet \in 1$ to $\bot \in \Omega(X)$. It has to send 1 to $\bot$ because if it didn't, the pullback would be 1, not 0 as given.

Remark 5. The character $\neg\colon \Omega \to \Omega$ of $\bot$ The character $\neg\colon \Omega \to \Omega$ of $\bot$ is a natural transformation assigning to each object the function that takes $\bot$ to $\top$ and everything else to $\bot$. If anything other than $\bot$ mapped to $\top$, the pullback would be bigger than 1.

1 $\quad \begin{array}{l} \neg_1\colon \Omega(1) \to \Omega(1) \\ \qquad \top \mapsto \bot \\ \qquad \bot \mapsto \top \end{array}$

M $\quad \begin{array}{l} \neg_M\colon \Omega(M) \to \Omega(M) \\ \qquad \top \mapsto \bot \\ \qquad \{f, f!\} \mapsto \bot \\ \qquad \bot \mapsto \top \end{array}$

Conjunction. TBD

Disjunction. TBD

4.2 Once again, but with monoids

Here we take Th to be the finite product category generated by

- one generating sort M
- $n + 2$ term constructors:
  - n generators $g_i : 1 \to M$
  - identity $e : 1 \to M$
  - multiplication $* : M \times M \to M$
- equations for assoc, unit laws

Interpretation of the logic of monoids In section 2 we introduced a logic of monoids naively generated from sets of monoid expressions. Here we show how to interpret the formulae of that logic via the native type theory construction, in terms of a recursive specification of the function $[\![\cdot]\!] : \mathcal{L}(M[G]) \to \int \mathsf{sub}\ \mathsf{Y}$

collection operations

$$
\begin{array}{rcl}
[\![\mathsf{true}]\!] & = & \langle M, \mathsf{t} \rangle \\
[\![\neg\phi]\!] & = & \langle M, \pi_2([\![\mathsf{true}]\!]) \backslash \pi_2([\![\phi]\!]) \rangle \\
[\![\phi \& \psi]\!] & = & [\![\phi]\!] \cap [\![\psi]\!]
\end{array}
$$

spatial operations

$$
\begin{array}{rcl}
[\![\mathbf{e}]\!] & = & \{m \in \mathcal{L}(M[G]) : m = \mathsf{e}\} \\
[\![\mathbf{g}]\!] & = & \{m \in \mathcal{L}(M[G]) : m = g\} \\
[\![\phi * \psi]\!] & = & \{m \in \mathcal{L}(M[G]) : \exists m_1, m_2 . m = m_1 * m_2, m_1 \in [\![\phi]\!], m_2 \in [\![\psi]\!]\}
\end{array}
$$

where $\mathsf{t} : \mathsf{Th}^{\mathsf{op}} \to \mathrm{Set}$ is the constant functor that returns $\mathsf{Th}(1, M)$

5 The difference between types and formulae

In this paper we differentiate between types and formulae. In particular, what the paper on the native type theory construction refers to by the work type is not the same was what is typically in the programming language semantics community. Formulae are model checked. That is, when someone writes $t \models \phi$, we take them to mean $[\![t]\!] \in [\![\phi]\!]$. Native type theory is effectively still squarely in the model checking camp.

Types come with a notion of type checking that provides an algorithmic correspondence between the proof that a give term inhabits a type with the structure of the term.

6 The algorithm

6.1 Notation

P, Q, R range over terms, while p, q, r range over term variables, e ranging over terms and variables. $\mathbf{U}, \mathbf{V}, \mathbf{W}$ range over filters $\{\mathbf{Q}\}$ characteristic filter of process Q. $\mathbf{X}, \mathbf{Y}, \mathbf{Z}$ range over sorts. $\mathbf{P}$ is the sort of process terms. $\mathbf{R}$ is the sort of reduction terms (redexes).

Type assertions are of the form $e : \mathbf{U} : \mathbf{X}$ where $\mathbf{X}$ may be considered a *kind* and $\mathbf{U}$ is a type interpreted as a *filter* on the kind. Contexts are sequences of type assertions. Judgments are written in more or less standard form $\Gamma \vdash P : \mathbf{U} : \mathbf{X}$.

Note that the input is a CCC presented as a λ-theory, and hence also enjoys a judgment style presentation. Much of the machinery below is simply lifting those judgments to the $\mathsf{NT}(\mathsf{CCC})$ level. Likewise, it is possible to project back down. We adopt the following notation to denote the projection: $\Gamma_\lambda \vdash_\lambda e : \mathbf{X}$.

In general, we assume no "heating" rules in the rewrite systems. That is, without loss of generality, every redex must be a term built from a binary term constructor, with one of the terms playing in the role of program and one playing in the role of environment. For example, in λ-calculus application is the binary term construction, the term in function position is the program, and the term in argument position is environment. In the rho-calculus parallel composition is the binary term constructor and one of the subterms

is program and the other is environment (it does not matter which). Without loss of generality, we write $P \odot Q$ for that binary term.

6.2 Generic types

Boundaries

$$\textsc{Axiom}\quad \Vdash p : \mathbf{U} : \mathbf{X} \vdash p : \mathbf{U} : \mathbf{X}$$

$$\textsc{Character}\quad \Gamma \vdash Q : \mathbf{U} : \mathbf{X} \Vdash \Gamma \vdash Q : \{\mathbf{Q}\} : \mathbf{X} \qquad \textsc{Top}\quad \Gamma \vdash P : \mathbf{U} : \mathbf{X} \Vdash \Gamma \vdash P : \top : \mathbf{X}$$

$$\textsc{Composition}\quad \Gamma \vdash P : \mathbf{U} : \mathbf{X} \quad p : \mathbf{U} : \mathbf{X}, \Delta \vdash Q : \mathbf{V} : \mathbf{Y} \Vdash \Gamma, \Delta \vdash Q\{P/p\} : \mathbf{V} : \mathbf{Y}$$

Disjunctions

$$\textsc{Union}\quad \Gamma \vdash P : \mathbf{U} : \mathbf{X} \Vdash \Gamma \vdash P : \mathbf{U} \vee \mathbf{V} : \mathbf{X}$$

$$\textsc{InL}\quad \Gamma \vdash P : \mathbf{U} : \mathbf{X} \Vdash \Gamma \vdash \mathsf{in}_L P : \mathbf{U} + \mathbf{V} : \mathbf{X} + \mathbf{Y}\ \text{(V:Y)}$$

$$\textsc{InR}\quad \Gamma \vdash Q : \mathbf{V} : \mathbf{Y} \Vdash \Gamma \vdash \mathsf{in}_R Q : \mathbf{U} + \mathbf{V} : \mathbf{X} + \mathbf{Y}\ \text{(U:X)}$$

$$\textsc{Match-Case}\quad \frac{p : \mathbf{U} : \mathbf{X}, \Gamma \vdash P : \mathbf{W} : \mathbf{Z} \quad q : \mathbf{V} : \mathbf{Y}, \Delta \vdash Q : \mathbf{W} : \mathbf{Z}}{r : \mathbf{U} + \mathbf{V} : \mathbf{X} + \mathbf{Y}, \Gamma, \Delta \vdash \mathsf{match}\ r\ \mathsf{case}\ \mathsf{in}_L\ p \Rightarrow P; \mathsf{case}\ \mathsf{in}_R\ q \Rightarrow Q : \mathbf{W} : \mathbf{Z}}$$

Conjunctions

$$\textsc{Intersection}\quad \Gamma \vdash P : \mathbf{U} : \mathbf{X} \quad \Gamma \vdash P : \mathbf{V} : \mathbf{X} \Vdash \Gamma \vdash P : \mathbf{U} \wedge \mathbf{V} : \mathbf{X}$$

$$\textsc{Product}\quad \Gamma \vdash P : \mathbf{U} : \mathbf{X} \quad \Delta \vdash Q : \mathbf{V} : \mathbf{Y} \Vdash \Gamma, \Delta \vdash P \times Q : \mathbf{U} \times \mathbf{V} : \mathbf{X} \times \mathbf{Y}$$

$$\textsc{Proj1}\quad p : \mathbf{U} : \mathbf{X}, \Gamma \vdash P : \mathbf{W} : \mathbf{Z} \Vdash r : \mathbf{U} \times \mathbf{V} : \mathbf{X} \times \mathbf{Y}, \Gamma \vdash \mathsf{let}\ \langle p, \_ \rangle\ =\ r\ \mathsf{in}\ P : \mathbf{W} : \mathbf{Z}$$

$$\textsc{Proj2}\quad q : \mathbf{V} : \mathbf{Y}, \Gamma \vdash Q : \mathbf{W} : \mathbf{Z} \Vdash r : \mathbf{U} \times \mathbf{V} : \mathbf{X} \times \mathbf{Y}, \Gamma \vdash \mathsf{let}\ \langle \_, q \rangle\ =\ r\ \mathsf{in}\ Q : \mathbf{W} : \mathbf{Z}$$

Implications

IMPLICATION1
$$\Gamma \vdash Q : \mathbf{V} : \mathbf{X} \Vdash \Gamma \vdash Q : \mathbf{U} \Rightarrow \mathbf{V} : \mathbf{X} \text{ (U:X)}$$

IMPLICATION2
$$\Gamma \vdash P : \mathbf{V} \Rightarrow \bot : \mathbf{X} \Vdash \Gamma \vdash P : \mathbf{U} \Rightarrow \mathbf{V} : \mathbf{X} \text{ (U:X)}$$

SEPARATION
$$\Gamma \vdash P_1 : \mathbf{U}_1 : \mathbf{X} \quad \Gamma \vdash P_2 : \mathbf{U}_2 : \mathbf{X} \Vdash \Gamma \vdash P_2 : \mathbf{U}_1 \Rightarrow \bot_{\mathbf{X}} : \mathbf{X} \ (\mathbf{U}_1 \wedge \mathbf{U}_2 = \bot_{\mathbf{X}})$$

ABSTRACTION
$$p : \mathbf{U} : \mathbf{X}, \Gamma \vdash Q : \mathbf{V} : \mathbf{Y} \Vdash \Gamma \vdash \lambda p.Q : \mathbf{U} \to \mathbf{V} : \mathbf{X} \to \mathbf{Y}$$

APPLICATION
$$q : \mathbf{W} : \mathbf{Z}, \Gamma \vdash Q : \mathbf{V} : \mathbf{Y} \quad \Delta \vdash P : \mathbf{U} : \mathbf{X} \Vdash \Gamma, r : \mathbf{U} \to \mathbf{W} : \mathbf{X} \to \mathbf{Z}, \Delta \vdash Q\{r(P)/q\} : \mathbf{V} : \mathbf{Y}$$

Internalized operations

PAIR
$$\Gamma \vdash P : \mathbf{U} : \mathbf{P} \quad \Delta \vdash Q : \mathbf{V} : \mathbf{P} \Vdash \Gamma, \Delta \vdash \langle P, Q \rangle : \langle \mathbf{U}, \mathbf{V} \rangle : \mathbf{P}$$

FST
$$\Vdash \vdash \pi_1 : \{\pi_1\} : \mathbf{P}$$

SND
$$\Vdash \vdash \pi_2 : \{\pi_2\} : \mathbf{P}$$

TAGL
$$\Gamma \vdash P : \mathbf{U} : \mathbf{P} \Vdash \Gamma \vdash \mathsf{tag}_L\ P : \mathbf{U} \oplus \mathbf{V} : \mathbf{P} \text{ (V:P)}$$

TAGR
$$\Gamma \vdash Q : \mathbf{V} : \mathbf{P} \Vdash \Gamma \vdash \mathsf{tag}_R\ Q : \mathbf{U} \oplus \mathbf{V} : \mathbf{P} \text{ (U:P)}$$

SUM
$$\Gamma \vdash P : \mathbf{U} : \mathbf{P} \quad \Delta \vdash Q : \mathbf{V} : \mathbf{P} \Vdash \Gamma, \Delta \vdash [P, Q] : [\mathbf{U}, \mathbf{V}] : \mathbf{P}$$

$$\mathsf{int}^+ : P + P \to P \qquad \mathsf{int}^- : ((P + P) \to P) \to P$$

$$\mathsf{tag}_i P := \mathsf{int}^+(\mathsf{in}_i P)$$

$$[P, Q] := \mathsf{int}^-(\lambda r.\mathsf{match}\ r\ \mathsf{case}\ \mathsf{in}_L\ p \Rightarrow P \odot p; \mathsf{case}\ \mathsf{in}_R\ q \Rightarrow Q \odot q)$$

Internalized redex constuctors

MatchL-Redex
$$\frac{\Gamma_1 \vdash P_1 : \mathbf{U}_1 : \mathbf{P} \quad \Gamma_2 \vdash P_2 : \mathbf{U}_2 : \mathbf{P} \quad \Gamma_1 \vdash P_3 : \mathbf{U}_3 : \mathbf{P}}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash \mathsf{match}_L(P_1, P_2, P_3) : \mathbf{match}_L(P_1, P_2, P_3) : \mathbf{R}}$$

MatchR-Redex
$$\frac{\Gamma_1 \vdash P_1 : \mathbf{U}_1 : \mathbf{P} \quad \Gamma_2 \vdash P_2 : \mathbf{U}_2 : \mathbf{P} \quad \Gamma_1 \vdash P_3 : \mathbf{U}_3 : \mathbf{P}}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash \mathsf{match}_R(P_1, P_2, P_3) : \mathbf{match}_R(P_1, P_2, P_3) : \mathbf{R}}$$

Proj1-Redex
$$\Gamma_1 \vdash P_1 : \mathbf{U}_1 : \mathbf{P} \quad \Gamma_2 \vdash P_2 : \mathbf{U}_2 : \mathbf{P} \Vdash \Gamma_1, \Gamma_2 \vdash \mathsf{proj}_1(P_1, P_2) : \mathbf{proj}_1(P_1, P_2) : \mathbf{R}$$

Proj2-Redex
$$\Gamma_1 \vdash P_1 : \mathbf{U}_1 : \mathbf{P} \quad \Gamma_2 \vdash P_2 : \mathbf{U}_2 : \mathbf{P} \Vdash \Gamma_1, \Gamma_2 \vdash \mathsf{proj}_2(P_1, P_2) : \mathbf{proj}_2(P_1, P_2) : \mathbf{R}$$

Equations from NT(CCC)

MatchL-Eqn
$$\frac{\Gamma \vdash R : \mathbf{U} : \mathbf{X} \quad p : \mathbf{U} : \mathbf{X}, \Delta_1 \vdash P : \mathbf{W} : \mathbf{Z} \quad q : \mathbf{V} : \mathbf{Y}, \Delta_2 \vdash Q : \mathbf{W} : \mathbf{Z}}{\Gamma, \Delta_1, \Delta_2 \vdash (\mathsf{match}\ \mathsf{in}_L\ R\ \mathsf{case}\ \mathsf{in}_L\ p \Rightarrow P; \mathsf{case}\ \mathsf{in}_R\ q \Rightarrow Q) = P\{R/p\} : \mathbf{W} : \mathbf{Z}}$$

MatchR-Eqn
$$\frac{\Gamma \vdash R : \mathbf{V} : \mathbf{Y} \quad p : \mathbf{U} : \mathbf{X}, \Delta_1 \vdash P : \mathbf{W} : \mathbf{Z} \quad q : \mathbf{V} : \mathbf{Y}, \Delta_2 \vdash Q : \mathbf{W} : \mathbf{Z}}{\Gamma, \Delta_1, \Delta_2 \vdash (\mathsf{match}\ \mathsf{in}_R\ R\ \mathsf{case}\ \mathsf{in}_L\ p \Rightarrow P; \mathsf{case}\ \mathsf{in}_R\ q \Rightarrow Q) = Q\{R/q\} : \mathbf{W} : \mathbf{Z}}$$

Proj1-Eqn
$$\frac{p : \mathbf{U} : \mathbf{X}, \Gamma \vdash P : \mathbf{W} : \mathbf{Z} \quad \Delta_1 \vdash Q_1 : \mathbf{U} : \mathbf{X} \quad \Delta_2 \vdash Q_2 : \mathbf{V} : \mathbf{Y}}{\Gamma, \Delta_1, \Delta_2 \vdash (\mathsf{let}\ \langle p, \_ \rangle\ =\ Q_1 \times Q_2\ \mathsf{in}\ P) = P\{Q_1/p\} : \mathbf{W} : \mathbf{Z}}$$

Proj2-Eqn
$$\frac{p : \mathbf{V} : \mathbf{Y}, \Gamma \vdash P : \mathbf{W} : \mathbf{Z} \quad \Delta_1 \vdash Q_1 : \mathbf{U} : \mathbf{X} \quad \Delta_2 \vdash Q_2 : \mathbf{V} : \mathbf{Y}}{\Gamma, \Delta_1, \Delta_2 \vdash (\mathsf{let}\ \langle \_, p \rangle\ =\ Q_1 \times Q_2\ \mathsf{in}\ P) = P\{Q_2/p\} : \mathbf{W} : \mathbf{Z}}$$

Liftings from the λ-theory

Lifted-Singleton
$$\Gamma_\lambda \vdash_\lambda P : \mathbf{X} \Vdash \Gamma \vdash P : \{\mathbf{P}\} : \mathbf{X}$$

Swap
$$\Gamma \vdash P : \mathbf{U} : \mathbf{X} \quad \Gamma_\lambda \vdash_\lambda P = Q : \mathbf{X} \Vdash \Gamma \vdash Q : \mathbf{U} : \mathbf{X}$$

Lifted-Eqn
$$\Gamma \vdash P : \mathbf{U} : \mathbf{X} \quad \Gamma \vdash Q : \mathbf{U} : \mathbf{X} \quad \Gamma_\lambda \vdash_\lambda P = Q : \mathbf{X} \Vdash \Gamma \vdash P = Q : \mathbf{U} : \mathbf{X}$$

Reductions from NT(CCC)

MatchL-Src
$$\frac{\Gamma \vdash \mathsf{match}_L(P_1,P_2,P_3) : \mathbf{match}_L(\mathbf{U}_1,\mathbf{U}_2,\mathbf{U}_3) : \mathbf{R}}{\Gamma \vdash \mathsf{src}(\mathsf{match}_L(P_1,P_2,P_3)) = [P_2,P_3] \ \odot\ \mathsf{tag}_L\ P_1 : [\mathbf{U}_2,\mathbf{U}_3] \ \odot\ \mathbf{U}_1 \oplus \mathbf{V} : \mathbf{P}}\ \text{(V:P)}$$

MatchL-Trgt
$$\frac{\Gamma \vdash \mathsf{match}_L(P_1,P_2,P_3) : \mathbf{match}_L(\mathbf{U}_1,\mathbf{U}_2,\mathbf{U}_3) : \mathbf{R}}{\Gamma \vdash \mathsf{trgt}(\mathsf{match}_L(P_1,P_2,P_3)) = P_2 \ \odot\ P_1 : \mathbf{U}_2 \ \odot\ \mathbf{U}_1 : \mathbf{P}}$$

MatchR-Src
$$\frac{\Gamma \vdash \mathsf{match}_R(P_1,P_2,P_3) : \mathbf{match}_R(\mathbf{U}_1,\mathbf{U}_2,\mathbf{U}_3) : \mathbf{R}}{\Gamma \vdash \mathsf{src}(\mathsf{match}_R(P_1,P_2,P_3)) = [P_2,P_3] \ \odot\ \mathsf{tag}_R\ P_1 : [\mathbf{U}_2,\mathbf{U}_3] \ \odot\ \mathbf{U}_1 \oplus \mathbf{V} : \mathbf{P}}\ \text{(V:P)}$$

MatchR-Trgt
$$\frac{\Gamma \vdash \mathsf{match}_R(P_1,P_2,P_3) : \mathbf{match}_R(\mathbf{U}_1,\mathbf{U}_2,\mathbf{U}_3) : \mathbf{R}}{\Gamma \vdash \mathsf{trgt}(\mathsf{match}_R(P_1,P_2,P_3)) = P_3 \ \odot\ P_1 : \mathbf{U}_3 \ \odot\ \mathbf{U}_1 : \mathbf{P}}$$

Proj1-Src
$$\frac{\Gamma \vdash \mathsf{Proj}_1(P_1,P_2) : \mathbf{proj}_1(\mathbf{U}_1,\mathbf{U}_2) : \mathbf{R}}{\Gamma \vdash \mathsf{src}(\mathsf{proj}_1(P_1,P_2)) = \pi_1 \ \odot\ \langle P_1,P_2\rangle : \{\pi_1\} \ \odot\ \langle \mathbf{U}_1,\mathbf{U}_2\rangle : \mathbf{P}}$$

Proj1-Trgt
$$\frac{\Gamma \vdash \mathsf{proj}_1(P_1,P_2) : \mathbf{proj}_1(\mathbf{U}_1,\mathbf{U}_2) : \mathbf{R}}{\Gamma \vdash \mathsf{trgt}(\mathsf{proj}_1(P_1,P_2)) = P_1 : \mathbf{U}_1 : \mathbf{P}}$$

Proj2-Src
$$\frac{\Gamma \vdash \mathsf{Proj}_2(P_1,P_2) : \mathbf{proj}_2(\mathbf{U}_1,\mathbf{U}_2) : \mathbf{R}}{\Gamma \vdash \mathsf{src}(\mathsf{proj}_2(P_1,P_2)) = \pi_2 \ \odot\ \langle P_1,P_2\rangle : \{\pi_2\} \ \odot\ \langle \mathbf{U}_1,\mathbf{U}_2\rangle : \mathbf{P}}$$

Proj1-Trgt
$$\frac{\Gamma \vdash \mathsf{proj}_2(P_1,P_2) : \mathbf{proj}_2(\mathbf{U}_1,\mathbf{U}_2) : \mathbf{R}}{\Gamma \vdash \mathsf{trgt}(\mathsf{proj}_2(P_1,P_2)) = P_2 : \mathbf{U}_2 : \mathbf{P}}$$

Modal operators

Step-Future
$\Gamma \vdash R : \mathbf{E} : \mathbf{R} \Vdash \Gamma \vdash \mathsf{src}(R) : \Diamond\mathsf{trgt}(\mathbf{E}) : \mathbf{P}$

Step-History
$\Gamma \vdash R : \mathbf{E} : \mathbf{R} \Vdash \Gamma \vdash \mathsf{trgt}(R) : \blacklozenge\mathsf{src}(\mathbf{E}) : \mathbf{P}$

7 Type system for rholang

7.1 λ-calculus : ML :: rho-calculus : rholang

In the last thirty years the process calculi have matured into a remarkably powerful analytic tool for reasoning about concurrent and distributed systems. Process-calculus-based algebraical specification of processes began with Milner's Calculus for Communicating

Systems (CCS) [32] and Hoare's Communicating Sequential Processes (CSP) [20], and continue through the development of the so-called mobile process calculi, e.g. Milner, Parrow and Walker's π-calculus [35], [36], Cardelli and Caires's spatial logic [7] [9] [10], or Meredith and Radestock's reflective calculi [29] [28]. The process-calculus-based algebraical specification of processes has expanded its scope of applicability to include the specification, analysis, simulation and execution of processes in domains such as:

- telecommunications, networking, security and application level protocols [1] [2] [22] [24];
- programming language semantics and design [22] [17] [16] [44];
- webservices [22] [24] [27];
- blockchain [31]
- and biological systems [11] [13] [39] [38].

Among the many reasons for the continued success of this approach 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 [33], 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 [42] [40] [41]. In particular, *bisimulation* encapsulates an effective notion of process equivalence that has been used in applications as far-ranging as algorithmic games semantics [4] and the construction of model-checkers [8]. 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 [42] all governed by the same core principle [41].

And yet, there is no mainstream language built from a calculus like the rho-calculus in the same way that SML is built from the λ-calculus. Until rholang.

7.2 The syntax and semantics of the notation system

We now summarize a technical presentation of the calculus that embodies our theory of dynamics. The typical presentation of such a calculus follows the style of giving generators and relations on them. The grammar, below, describing term constructors, freely generates the set of processes, Proc. This set is then quotiented by a relation known as structural congruence and it is over this set that the notion of dynamics is expressed. This

presentation is essentially that of [29] with the addition of polyadicity and summation. For readability we have relegated some of the technical subtleties to an appendix.

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.

Process grammar

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

Note that $\overrightarrow{x}$ (resp. $\overrightarrow{P}$) denotes a vector of names (resp. processes) of length $|\overrightarrow{x}|$ (resp. $|\overrightarrow{P}|$). We adopt the following useful abbreviations.

$$\Pi\,\overrightarrow{P} := \Pi_{i=1}^{|\overrightarrow{P}|} P_i := P_1|\ldots|P_{|\overrightarrow{P}|}$$

Definition 2. Free and bound names *The calculation of the free names of a process, P, denoted $\mathsf{FN}(P)$ is given recursively by*

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

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

where $\{\overrightarrow{x}\} := \{x_1, \ldots, x_{|\overrightarrow{x}|}\}$ *and* $\mathsf{FN}(\overrightarrow{P}) := \bigcup \mathsf{FN}(P_i)$.

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 $\mathsf{N}(P)$.

7.3 Substitution

We use Proc for the set of processes, $@\mathsf{Proc}$ for the set of names, and $\{\overrightarrow{y}/\overrightarrow{x}\}$ 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, 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 3. *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.*

$$0\sigma := 0$$

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

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

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

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

where

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

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 4. *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 5. *The* structural congruence $\equiv$ *between processes [43] is the least congruence containing alpha-equivalence and satisfying the commutative monoid laws (associativity, commutativity and* $\mathsf{0}$ *as identity) for parallel composition* $|$.

Definition 6. *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 appendix **??**.

Remark 6. 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.

7.4 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.

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

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

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

7.5 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 $w!(*x) = w!(*@y!(*z))$.

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

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.

8 Replication

As mentioned before, it is known that replication (and hence recursion) can be implemented in a higher-order process algebra [43]. As our first example of calculation with the machinery thus far presented we give the construction explicitly in the ρ-calculus.

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

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

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 7. 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 8. 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 [34].

Definition 7. *An* observation relation, $\downarrow_{\mathcal{N}}$, *over a set of names,* $\mathcal{N}$, *is the smallest relation satisfying the rules below.*

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

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

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

Definition 8. *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 $P \to P'$ 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.

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

Definition 9 (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 9. Note that we can extend the definition of free and bound names to contexts.

8.1 Lifted types

In this section, for clarity, we illustrate the procedure for lifting the rules of a specific rewrite system by way of example. Specifically, we illustrate the procedure lifting the rho-calculus theory to the $\mathsf{NT}(\mathsf{CCC})$ level.

Lifted term constuctors Note that because of the lifted-singleton rule, it is not necessary to give a typing for the $\mathsf{0}$ term.

For-Comprehension
$$y : @\mathbf{W} : \mathbf{N}, \Gamma \vdash P : \mathbf{U} : \mathbf{P} \quad \Delta \vdash x : \mathbf{V} : \mathbf{N} \Vdash \Gamma, \Delta \vdash \mathsf{for}(y \leftarrow x)P : \mathbf{for}(@\mathbf{W} \leftarrow \mathbf{V})\mathbf{U} : \mathbf{P}$$

Output
$$\Gamma \vdash x : \mathbf{U} : \mathbf{N} \quad \Delta \vdash Q : \mathbf{V} : \mathbf{P} \Vdash \Gamma, \Delta \vdash x!(Q) : \mathbf{U}!(\mathbf{V}) : \mathbf{P}$$

Parallel
$$\Gamma \vdash P : \mathbf{U} : \mathbf{P} \quad \Delta \vdash Q : \mathbf{V} : \mathbf{P} \Vdash \Gamma, \Delta \vdash P|Q : \mathbf{U}|\mathbf{V} : \mathbf{P}$$

Deref
$$\Gamma \vdash x : \mathbf{U} : \mathbf{N} \Vdash \Gamma \vdash *x : *\mathbf{U} : \mathbf{P}$$

Quote
$$\Gamma \vdash P : \mathbf{U} : \mathbf{P} \Vdash \Gamma \vdash @x : @\mathbf{U} : \mathbf{P}$$

Equations

ParMonoidId
$$\frac{\Gamma \vdash P : \mathbf{U} : \mathbf{P}}{\Gamma \vdash P|0 = P : \mathbf{U} : \mathbf{P}}$$

ParMonoidAssoc
$$\frac{\Gamma \vdash (P|Q)|R : \mathbf{U} : \mathbf{P}}{\Gamma \vdash (P|Q)|R = P|(Q|R) : \mathbf{U} : \mathbf{P}}$$

ParMonoidComm
$$\frac{\Gamma \vdash P|Q : \mathbf{U} : \mathbf{P}}{\Gamma \vdash P|Q = Q|P : \mathbf{U} : \mathbf{P}}$$

Lifted redex constuctors

Comm
$$\frac{\Gamma_1 \vdash x : \mathbf{V} : \mathbf{N} \quad y : @\mathbf{V} : \mathbf{N}, \Gamma_2 \vdash P : \mathbf{W} : \mathbf{P} \quad \Gamma_3 \vdash Q : \mathbf{V} : \mathbf{P}}{\Gamma_1, \Gamma_2, \Gamma_3 \vdash \mathsf{comm}(x, y, P, Q) : \mathbf{comm}(\mathbf{U}, \mathbf{V}, \mathbf{W}) : \mathbf{R}}$$

Eval
$$\Gamma \vdash P : \mathbf{U} : \mathbf{P} \Vdash \Gamma \vdash \mathsf{eval}(P) : \mathbf{eval}(\mathbf{U}) : \mathbf{R}$$

ParL
$$\Gamma \vdash R : \mathbf{E} : \mathbf{R} \quad \Delta \vdash P : \mathbf{U} : \mathbf{P} \Vdash \Gamma, \Delta \vdash \mathsf{par}_L(R, P) : \mathbf{par}_L(\mathbf{E}, \mathbf{U}) : \mathbf{R}$$

ParR
$$\Gamma \vdash R : \mathbf{E} : \mathbf{R} \quad \Delta \vdash P : \mathbf{U} : \mathbf{P} \Vdash \Gamma, \Delta \vdash \mathsf{par}_R(P, R) : \mathbf{par}_R(\mathbf{U}, \mathbf{E}) : \mathbf{R}$$

Reductions from the λ-theory

Comm-Src
$$\frac{\Gamma \vdash \mathsf{comm}(x, y, P, Q) : \mathbf{comm}(\mathbf{U}, \mathbf{V}, \mathbf{W}) : \mathbf{R}}{\Gamma \vdash \mathsf{src}(\mathsf{comm}(x, y, P, Q)) = \mathsf{for}(y \leftarrow x)P|x!(Q) : \mathbf{for}(@\mathbf{V} \leftarrow \mathbf{U})\mathbf{W}|\mathbf{U}!(\mathbf{V}) : \mathbf{P}}$$

Comm-Trgt
$$\frac{\Gamma \vdash \mathsf{comm}(x, y, P, Q) : \mathbf{comm}(\mathbf{U}, \mathbf{V}, \mathbf{W}) : \mathbf{R}}{\Gamma \vdash \mathsf{trgt}(\mathsf{comm}(x, y, P, Q)) = P\{@Q/y\} : \mathbf{W} : \mathbf{P}}$$

Eval-Src
$$\Gamma \vdash \mathsf{eval}(P) : \mathbf{eval}(\mathbf{U}) : \mathbf{R} \Vdash \Gamma \vdash \mathsf{src}(\mathsf{eval}(P)) = *@P : *@\mathbf{U} : \mathbf{P}$$

Eval-Trgt
$$\Gamma \vdash \mathsf{eval}(P) : \mathbf{eval}(\mathbf{U}) : \mathbf{R} \Vdash \Gamma \vdash \mathsf{trgt}(\mathsf{eval}(P)) = P : \mathbf{U} : \mathbf{P}$$

Par-Src
$$\frac{\Gamma \vdash \mathsf{par}(R, P) : \mathbf{par}(\mathbf{E}, \mathbf{U}) : \mathbf{R}}{\Gamma \vdash \mathsf{src}(\mathsf{par}(R, P)) = \mathsf{src}(R) : \mathbf{par}(\mathbf{src}(\mathbf{E}), \mathbf{U}) : \mathbf{R}}$$

Par-Trgt
$$\frac{\Gamma \vdash \mathsf{par}(R, P) : \mathbf{par}(\mathbf{E}, \mathbf{U}) : \mathbf{R}}{\Gamma \vdash \mathsf{trgt}(\mathsf{par}(R, P)) = \mathsf{par}(\mathsf{trgt}(R), P) : \mathbf{par}(\mathbf{trgt}(\mathbf{E}), \mathbf{U}) : \mathbf{R}}$$

9 Main theorem: substitutability

The Liskov substitution principle [25] is vitally important for a type systems, especially if we want our type system to provide guidance for automated refactoring and other kinds of support we might expect of a type-directed interactive design environment, such as IntelliJ or Eclipse. In Milner's polyadic π-calculus tutorial [34] he demonstrates exactly the sort of theorem for his logic that we need to formalize Liskov's principle.

$$P \dot{\approx} Q \iff \forall \phi.(P \vDash \phi \iff Q \vDash \phi)$$

As Caires points out, spatial connectives can see structural properties that Milner's logic cannot. In particular, bisimulation cannot distinguish parallel composition from the summ of all interleavings. Fortunately, by including proved versions of the structural equalities we recover Milner's theorem.

Theorem 1 (substitutability).

$$P \dot{\approx} Q \iff \forall(\mathbf{U}, \mathbf{X}).(P : (\mathbf{U}, \mathbf{X}) \iff Q : (\mathbf{U}, \mathbf{X}))$$

10 Some examples of types

10.1 A compile time firewall

TBD

10.2 Race detection

TBD

10.3 Secrecy

TBD

11 Conclusions and future work

We have presented an algorithm for generating a spatial-behavioral type system for a model of computation.

One of the most important discoveries of process calculi – as models of computation – is that they can abstractly describe non-deterministic behavior and leave to implementation the *source of that non-determinism*. Thus, for the rho-calculus non-determism can be actual races derived from message arrival order non-determinism over physically implemented protocols, or it can be from a 1-norm probability distribution, or as recently discovered, it can be from a 2-norm probability distribution.

These last two cases are important as they extend the models to simulating and potentially programming physical substrates including chemical and biological computation as well as quantum computation. In the same way that Hennessey-Milner style logics have been extended to the stochastic and quantum settings, we believe that this algorithm can be extended to cover these cases.

Acknowledgments. The author wishes to thank Christian Williams for his work on categorifying the algorithm originally, and naively developed by the author in [28]. Further, the author wishes to acknowledge the longstanding and fruitful collaboration with Mike Stay whose friendship and engagement have been invaluable.

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. Computational interpretations of linear logic. *Theor. Comput. Sci.*, 111(1&2):3–57, 1993.
4. Samson Abramsky. Algorithmic game semantics and static analysis. In *SAS*, volume 3672 of *Lecture Notes in Computer Science*, page 1. Springer, 2005.
5. Lars Bak. How dart learned from past object-oriented systems (keynote). In *SPLASH (Companion Volume)*, page 3. ACM, 2015.
6. Michael Barr and Charles Wells. *Category theory for computing science*. Prentice Hall International Series in Computer Science. Prentice Hall, 1990.
7. Luís Caires. Behavioral and spatial observations in a logic for the pi-calculus. In *FoSSaCS*, pages 72–89, 2004.
8. Luis Caires. Spatial logic model checker, Nov 2004.
9. Luís Caires and Luca Cardelli. A spatial logic for concurrency (part I). *Inf. Comput.*, 186(2):194–235, 2003.
10. Luís Caires and Luca Cardelli. A spatial logic for concurrency - II. *Theor. Comput. Sci.*, 322(3):517–565, 2004.
11. Luca Cardelli. Brane calculi. In *CMSB*, volume 3082 of *Lecture Notes in Computer Science*, pages 257–278. Springer, 2004.
12. Avik Chauduri. A static type checker for javascript, Aug 2017.
13. Vincent Danos and Cosimo Laneve. Core formal molecular biology. In *ESOP*, volume 2618 of *Lecture Notes in Computer Science*, pages 302–318. Springer, 2003.
14. Marcelo P. Fiore and Chung-Kil Hur. Second-order equational logic (extended abstract). In *CSL*, volume 6247 of *Lecture Notes in Computer Science*, pages 320–335. Springer, 2010.
15. Marcelo P. Fiore and Ola Mahmoud. Second-order algebraic theories - (extended abstract). In *MFCS*, volume 6281 of *Lecture Notes in Computer Science*, pages 368–380. Springer, 2010.
16. 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.
17. 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.
18. Robert Goldblatt. *Topoi - the categorial analysis of logic, Second rev. Edition*, volume 98 of *Studies in logic and the foundations of mathematics*. North-Holland, 1984.
19. Shu Yu Guo. Ecma-262, Jul 2022.
20. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
21. P.T. Johnstone. *Stone Spaces*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 1982.
22. 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.
23. J. Lambek and P.J. Scott. *Introduction to Higher-Order Categorical Logic*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 1988.
24. Cosimo Laneve and Gianluigi Zavattaro. Foundations of web transactions. In *FoSSaCS*, volume 3441 of *Lecture Notes in Computer Science*, pages 282–298. Springer, 2005.
25. Barbara Liskov. Keynote address - data abstraction and hierarchy. In *OOPSLA Addendum*, pages 17–34. ACM, 1987.
26. S. MacLane and I. Moerdijk. *Sheaves in Geometry and Logic: A First Introduction to Topos Theory*. Universitext. Springer New York, 2012.
27. Greg Meredith. Documents as processes: A unification of the entire web service stack. In *WISE*, pages 17–20. IEEE Computer Society, 2003.
28. L. Gregory Meredith and Matthias Radestock. Namespace logic: A logic for a reflective higher-order calculus. In *TGC* [29], pages 353–369.
29. L. Gregory Meredith and Matthias Radestock. A reflective higher-order calculus. *Electr. Notes Theor. Comput. Sci.*, 141(5):49–67, 2005.

30. Lucius Meredith, Jan 2017.
31. Lucius Meredith, Jan 2017.
32. Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.
33. Robin Milner. Elements of interaction - turing award lecture. *Commun. ACM*, 36(1):78–89, 1993.
34. Robin Milner. The polyadic π-calculus: A tutorial. *Logic and Algebra of Specification*, Springer-Verlag, 1993.
35. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992.
36. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, II. *Inf. Comput.*, 100(1):41–77, 1992.
37. Neil Mitchell. Welcome to hoogle, May 2011.
38. 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.
39. 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.
40. Davide Sangiorgi. Beyond bisimulation: The "up-to" techniques. In *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 161–171. Springer, 2005.
41. Davide Sangiorgi. On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst.*, 31(4):15:1–15:41, 2009.
42. 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.
43. Davide Sangiorgi and David Walker. *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press, 2001.
44. 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.
45. Christian Williams and Michael Stay. Native type theory. *CoRR*, abs/2102.04672, 2021.