Operational semantics in logical form

L.G. Meredith[1]

Managing Partner, RTech Unchained 9336 California Ave SW, Seattle, WA 98103, USA,
lgreg.meredith@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) 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 in 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.

1.1 Summary of contributions and outline of paper

Summary of contributions TBD

Outline of paper TBD

2 Monoidal logic: a motivating example

TBD

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

TBD

4 A brief review of native types

TBD

5 The algorithm

5.1 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 $\Gamma \vdash P : \mathbf{U} : \mathbf{X}$. Because there are rather a bit more of them, inference rules are written in a somewhat non-standard form. 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

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

Consider the following recipe. Given a category, C, 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, C, in $|\mathsf{CCC}|$. The language presented below is a syntax for the derived meta-theory.

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 $P|Q$ for that binary term.

5.2 Generic types

Boundaries

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

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

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

Disjunctions

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

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

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

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

Conjunctions

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

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

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

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

Implications

Implication1
$$\Gamma \vdash Q : \mathbf{V} : \mathbf{X} \Vdash \Gamma \vdash Q : \mathbf{U} \Rightarrow \mathbf{V} : \mathbf{X}$$

Implication2
$$\Gamma \vdash P : \mathbf{V} \Rightarrow \bot : \mathbf{X} \Vdash \Gamma \vdash P : \mathbf{U} \Rightarrow \mathbf{V} : \mathbf{X}$$

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

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

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

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

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

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

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

Equations from NT(CCC)

Reductions from NT(CCC)

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

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

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

Modal operators

5.3 Lifted types

Lifted term constuctors

Lifted redex constuctors

Lifted equations from the λ-theory

6 Type system for rholang

6.1 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) [19] and Hoare's Communicating Sequential Processes (CSP) [12], and continue through the development of the so-called mobile process calculi, e.g. Milner, Parrow and Walker's π-calculus [22], [23], Cardelli and Caires's spatial logic [4] [6] [7], or Meredith and Radestock's reflective calculi [17] [16]. 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] [13] [14];
- programming language semantics and design [13] [11] [10] [30];
- webservices [13] [14] [15];
- blockchain [18]
- and biological systems [8] [9] [25] [24].

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 [20], 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 [28] [26] [27]. 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 [28] all governed by the same core principle [27].

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.

6.2 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 [17] 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

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

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

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

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

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 $\mathsf{N}(P)$.

6.3 Substitution

We use Proc for the set of processes, @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 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.*

$$0\sigma := 0$$

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

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

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

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

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 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 [29] 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 appendix **??**.

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.

6.4 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.

Definition 6. COMM*-events: when* $P \rightarrow P'$ *we can record the information justifying this reduction step with an expression of the form* $\mathsf{COMM}_{(P,P')}(x_t, x_s, \sigma)$*, leaving off the subscript* $_{(P,P')}$ *when the context makes the source* (P) *and target* (P') *states clear. Likewise, if* $c = \mathsf{COMM}_{(P,P')}(x_t, x_s, \sigma)$*, we write* $\mathsf{src}(c) = P$*,* $\mathsf{trgt}(c) = P'$*,* $\mathsf{src}_{\mathsf{N}}(c) = x_s$*, and* $\mathsf{trgt}_{\mathsf{N}}(c) = x_t$ *and* $P \xrightarrow{c} P'$*.*

We use $\alpha, \beta, ...$ to range over reduction *paths*. That is, if

$$P_1 \overset{\mathsf{COMM}(x_{t_1}, x_{s_1}, \sigma_1)}{\rightarrow} P_2 \overset{\mathsf{COMM}(x_{t_2}, x_{s_2}, \sigma_2)}{\rightarrow} \cdots \overset{\mathsf{COMM}(x_{t_{n-1}}, x_{s_{n-1}}, \sigma_{n-1})}{\rightarrow} P_n,$$

then we may write

$$\alpha = \mathsf{COMM}(x_{t_1}, x_{s_1}, \sigma_1) : \mathsf{COMM}(x_{t_2}, x_{s_2}, \sigma_2) : \ldots : \mathsf{COMM}(x_{t_{n-1}}, x_{s_{n-1}}, \sigma_{n-1}).$$

Definition 7. *The set of paths between* P *and* Q*, written* $\mathsf{paths}(P, Q)$ *is defined by* $\mathsf{paths}(P, Q) := \{c : \alpha \ : P \xrightarrow{c} P', \alpha \in \mathsf{paths}(P', Q)\}$ *and* ϵ *denotes the empty path.*

6.5 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.

7 Replication

As mentioned before, it is known that replication (and hence recursion) can be implemented in a higher-order process algebra [29]. As our first example of calculation with the machinery thus far presented we give the construction explicitly in the ρ-calculus.

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

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

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 [21].

Definition 8. *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 9. *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.

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

Definition 10 (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.

8 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 [16].

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. Vincent Danos and Cosimo Laneve. Core formal molecular biology. In *ESOP*, volume 2618 of *Lecture Notes in Computer Science*, pages 302–318. Springer, 2003.
10. 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.
11. 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.
12. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
13. 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.
14. Cosimo Laneve and Gianluigi Zavattaro. Foundations of web transactions. In *FoSSaCS*, volume 3441 of *Lecture Notes in Computer Science*, pages 282–298. Springer, 2005.
15. Greg Meredith. Documents as processes: A unification of the entire web service stack. In *WISE*, pages 17–20. IEEE Computer Society, 2003.
16. L. Gregory Meredith and Matthias Radestock. Namespace logic: A logic for a reflective higher-order calculus. In *TGC* [17], pages 353–369.
17. L. Gregory Meredith and Matthias Radestock. A reflective higher-order calculus. *Electr. Notes Theor. Comput. Sci.*, 141(5):49–67, 2005.
18. Lucius Meredith, Jan 2017.
19. Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.
20. Robin Milner. Elements of interaction - turing award lecture. *Commun. ACM*, 36(1):78–89, 1993.
21. Robin Milner. The polyadic π-calculus: A tutorial. *Logic and Algebra of Specification*, Springer-Verlag, 1993.
22. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992.
23. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, II. *Inf. Comput.*, 100(1):41–77, 1992.
24. 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.
25. 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.
26. Davide Sangiorgi. Beyond bisimulation: The "up-to" techniques. In *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 161–171. Springer, 2005.
27. Davide Sangiorgi. On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst.*, 31(4):15:1–15:41, 2009.
28. 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.
29. Davide Sangiorgi and David Walker. *The Pi-Calculus - a theory of mobile processes*. Cambridge University Press, 2001.

30. 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.