Keywords
higher category theory, concurrency, message-passing, types, Curry-Howard

ABSTRACT
We present an approach to modeling computational calculi using higher category theory. While the paper focuses on applications to the mobile process calculi—and more specifically, the π-calculus—because they provide unique challenges for categorical models, the approach extends smoothly to a variety of other computational calculi, including important milestones such as the lazy λ-calculus. One of the key contributions is a method of restricting rewrites to specific contexts inspired by catalysis in chemical reactions.

Submission to arXiv

Higher category models of mobile process calculi

Mike Stay
Google
stay@google.com

L.G. Meredith
Biosimilarity, LLC
lgreg.meredith@biosimilarity.com

1. INTRODUCTION

One of the major distinctions in programming language semantics has been the division between denotational and operational semantics. In the former computations are interpreted as mathematical objects which—more often than not—completely unfold the computational dynamics, and are thus infinitary in form. In the latter computations are interpreted in terms of rules operating on finite syntactic structure. Historically, categorical semantics for programming languages, even variations such as games semantics which capture much more of the intensional structure of computations, are distinctly denotational in flavor. Meanwhile, operational semantics continues to dominate in the presentation of calculi underlying programming languages used in practice.

Motivated, in part, by the desire to make a closer connection between theory and practice, the approach taken here investigates a direct categorical interpretation of the operational presentation of the π-calculus. It's not only due to lack of space that we focus on a minimal presentation of the calculus. One of the goals has been to provide a modular semantics to address a range of different calculi and modeling options. For example, a significant bifurcation occurs in the treatment of names with Milner's original calculus hiding all internal structure of names, while the ρ-calculus variant provides a reflective version in which names are the codes or processes. The semantics presented here can be seen as a framework capable of providing a categorical interpretation of both variants. Of particular interest to theoreticians and implementers, the semantics shines light on a key difference between the categorical and computational machinery it interprets. The latter is intrinsically lazy in the sense that all contexts where rewrites can apply must be explicitly spelled out (cf the context rules), while the former is intrinsically eager; in fact, one of the contributions of the paper is the delineation of an explicit control mechanism to prevent unwanted rewrites that would otherwise create an insurmountable divergence between the two formalisms.

1.0.1 *Organization of the rest of the paper*

In the remainder of the paper we present the core fragment of the calculus we model followed by a manifest of the categorical equipment needed to faithfully model it. Then we give the semantics function an sketch a proof that the interpretation is fully abstract.

2. THE CALCULUS

Some examples of process expressions.

2.1 Our running process calculus

2.1.1 *Syntax*

$P ::= 0$	stopped process
$\mid x?(y_1, \ldots, y_N) \Rightarrow P$	input
$\mid x!(y_1, \ldots, y_N)$	output
$\mid P \mid Q$	parallel

2.1.2 *Free and bound names*

$$
\begin{aligned}
&\mathcal{FN}(0) \coloneqq \emptyset \\
&\mathcal{FN}(x?(y_1, \ldots, y_N) \Rightarrow P) \coloneqq \\
&\quad \{x\} \cup (\mathcal{FN}(P) \setminus \{y_1, \ldots y_N\}) \\
&\mathcal{FN}(x!(y_1, \ldots, y_N)) \coloneqq \{x, y_1, \ldots, y_N\} \\
&\mathcal{FN}(P \mid Q) \coloneqq \mathcal{FN}(P) \cup \mathcal{FN}(Q)
\end{aligned}
$$

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 $\mathcal{N}(P)$.

2.1.3 Structural congruence

The *structural congruence* of processes, noted $\equiv$, is the least congruence containing α-equivalence, $\equiv_\alpha$, making $(P, |, 0)$ into commutative monoids.

2.1.4 Operational Semantics

$$\frac{|\vec{y}| = |\vec{z}|}{x?(\vec{y}) \Rightarrow P \mid x!(\vec{z}) \rightarrow P\{@\vec{z}/\vec{y}\}} \quad (\textsc{Comm})$$

In addition, we have the following context rules:

$$\frac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q} \quad (\textsc{Par})$$

$$\frac{P \equiv P' \qquad P' \rightarrow Q' \qquad Q' \equiv Q}{P \rightarrow Q} \quad (\textsc{Equiv})$$

2.1.5 Bisimulation

Definition 2.1.1. *An* observation relation, $\downarrow_{\mathcal{N}}$, *over a set of names,* $\mathcal{N}$, *is the smallest relation satisfying the rules below.*

$$\frac{x \in \mathcal{N}}{x!(\vec{y}) \downarrow_{\mathcal{N}} x} \quad (\textsc{Out-barb})$$

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

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

Notice that $x?(y) \Rightarrow P$ has no barb. Indeed, in rho-calculus as well as other asynchronous calculi, an observer has no direct means to detect if a sent message has been received or not.

Definition 2.1.2. *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* $\mathcal{S}_{\mathcal{N}}$.

3. CATEGORICAL MACHINERY

We take our models in cartesian closed 2-categories; the 2-category Cat of categories, functors, and natural transformations is the principal example. We denote the terminal object by 1; the internal hom by a lollipop $\multimap$; and duplication and deletion of an object X by $\Delta_X : X \rightarrow X \times X$ and $\delta_X : X \rightarrow 1$, respectively.

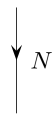
4. THE INTERPRETATION

Given the abstract syntax of a term calculus like that in section 2.1.1, we introduce an object in our 2-category for each parameter of the calculus. The π-calculus is parametric in a set of names and a set of processes, so we have objects N and P. We introduce 1-morphisms for each term constructor, 2-morphisms for each reduction relation, and equations for structural equivalence.

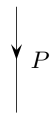
We also add 1-morphisms from P to P to mark contexts in which reductions may occur. In the π-calculus, all reductions occur at the topmost level. There are some benefits to constructing the top context out of the existing binary morphism $| \colon P \times P \rightarrow P$ and a unary morphism $COMM \colon 1 \rightarrow P$. [[Rewrite that stuff in more detail.]]

The theory of the π-calculus is the free cartesian closed 2-category on

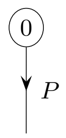
- an objects N for names,



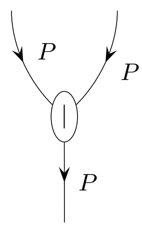
- an object P for processes,



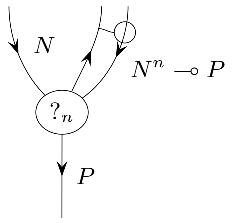
- a 1-morphism $0 \colon 1 \rightarrow P$,



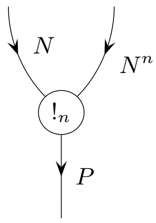
- a 1-morphism $| \colon P \times P \rightarrow P$,



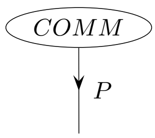
- a 1-morphism $?_n \colon N \times (N^n \multimap P) \rightarrow P$ for each natural number $n \geq 0$,



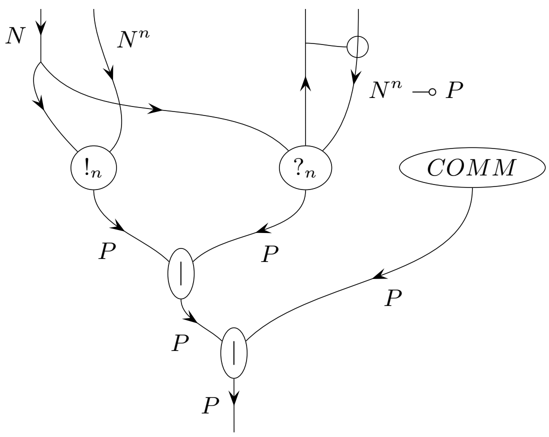
- a 1-morphism $!_n \colon N \times N^n \to P$ for each natural number $n \geq 0$,



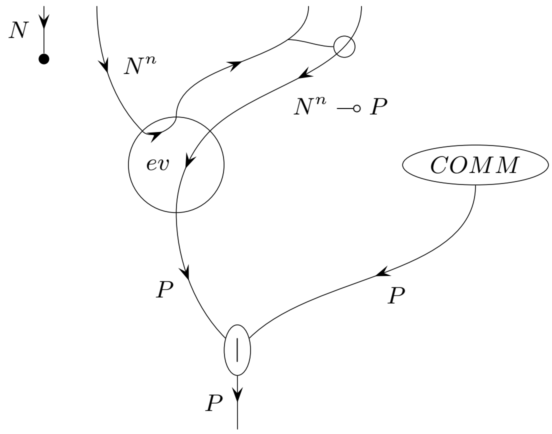
- a 1-morphism $COMM \colon 1 \to P$,



- a 2-morphism $comm_n$ encoding the COMM rule for each natural number $n \geq 0$, and



$comm_n \Downarrow$



- equations making $(P, |, 0)$ into a commutative monoid

4.1 Semantics

$$
\begin{aligned}
&[\![0]\!] \coloneqq \ldots \\
&[\![x?(y_1, \ldots, y_N) \Rightarrow P]\!] \coloneqq \ldots \\
&[\![x!(y_1, \ldots, y_N)]\!] \coloneqq \ldots \\
&[\![P \mid Q]\!] \coloneqq \ldots
\end{aligned}
$$

5. CONCLUSIONS AND FUTURE WORK

To model lazy lambda calculus, we use two unary contexts T and U, mark the topmost context with T, and use two rules:

- $T(App(l, r)) \Rightarrow U(App(T(l), r))$ for moving T to the leftmost subterm, and
- $U(App(T(Lam(x, M)), N) \Rightarrow T(M\{N/x\})$ for $\beta-$reduction.

TBD

Acknowledgments.. TBD

6. REFERENCES