Name-free combinators for concurrency

L.G. Meredith[1]
and Michael Stay[2]

[1] RChain Cooperative
greg@rchain.coop
[2] Pyrofex Corp.
stay@pyrofex.net

Abstract. Yoshida demonstrated how to eliminate the bound names coming from the input prefix in the asynchronous π-calculus, but her combinators still depend on the new operator to bind names. We modify Yoshida's combinators by replacing new and replication with reflective operators to provide the first combinator calculus with no bound names into which the asynchronous π-calculus has a faithful embedding.

1 Introduction

Many term calculi, like λ-calculus or π-calculus, involve binders for names, and the mathematics of bound variable names is subtle. Schönfinkel introduced the SKI combinator calculus in 1924 to clarify the role of quantified variables in intuitionistic logic by eliminating them [**?**]; Curry developed Schönfinkel's ideas much further. The difficulties are not merely theoretical, but represent a real practical challenge in the design of programming languages. In fact binding is one of the key features of the PoPLMark Challenge [**?**]. Certainly, the recent work by Jamie Gabbay and Andrew Pitts [**?**] and others [**?**] on nominal set theory has put the study of bound names and substitution on a much nicer foundation that can be shown to extend to practical implementations. However, it introduces an intriguing conundrum.

Specifically, the work of Gabbay and Pitts relies on a version of set theory (Fraenkel-Mostowski set theory, which we abbreivate to FM set theory) that admits an infinite supply of "atoms". This raises the question of where these atoms come from, which has both theoretical and practical implications. On the practical side, infinite sets of "atomic" entities, i.e. entities with no internal structure, are not realizable on modern computers. Modern computers fundamentally rely on sets of elements with effective internal structure to provide the kind of compression necessary to produce or compute with infinite sets. The natural numbers is a prime example. Because of their very regular internal structure the entire set can be represented by a single recursive equation. Potentially then, the natural numbers or some other effectively representable set could provide the source of atoms used in an FM-set theoretic account of binding in practical implementations. However, this raises a new question.

In order to represent and compute these effectively representable sets a notion of computation must already be in place. If that notion of computation relies on a notion of binding, then not a lot progress has been made! As in [**?**] we argue that this circularity, instead of being an obstacle to overcome, might be a clue to an alternative approach to binding phenomena; and one that ties together two important computational phenomena that have not normally been considered as related. More explicitly, we extend the argument made by Meredith and Radestock that reflection suffices to provide the "atoms" used in the π-calculus as channels to produce the first name-free set of combinators that enjoys a full and faithful interpretation of the calculus.

While the focus of the paper is largely on the technical results it is useful to consider the larger context motivating them. Reflection and meta-programming features more generally are part of a growing number of mainstream languages. Java, C#, even the Haskell and OCaml communities have seen growing interest in these features with efforts like template Haskell and MetaOCaml, respectively. In large measure this has to do the fact that programming at industrial scale requires the leverage of computer programs to write computer programs. Thus, meta-programming features are simply a practical necessity. On the

other hand, reflection and meta-programming have not received a theoretical account that fits well with strong typing. This is one of the reasons why language designs based on typed λ-calculi have been so slow to adopt reflection as a feature by comparison to other language designs.

In this setting, the idea that a single feature already enjoying widespread adoption could account for such a subtle phenomenon as the kind of binding found in the π-calculus is both intriguing and worth exploring, even if FM set theory provides a satisfying account of computation with nominal phenomena in other respects. However, it is precisely the foundational theoretical questions raised by the FM set theoretic approach that motivates the investigation in the first place: what better place to look for the source of "atoms" than in the reification of theory of computation requiring them?

To be clear, we are not focusing on ordinary abstraction, as that is well solved by abstraction elimination from the λ-calculus to $\mathcal{SKI}$. Instead we are focused on binders for fresh names. For example, the π-calculus ([**?**]) has two binders for names: the new operator, which introduces a new name into scope, and the input prefix, which introduces a name for labeling locations for substitution. Yoshida [**?**] describes an elimination algorithm that gets rid of input prefixes which corresponds in many respects to the elimination of lambda abstraction; but her combinators still fundamentally depend on the new operator. In complementary work, Meredith and Radestock [**?**] introduce reflective operators into a higher-order π-calculus and implment new and replication in terms of reflection. Here, we present a fusion of those ideas: a name-free concurrent combinator calculus into which Yoshida's combinators have a faithful embedding.

Role of compositionality in theories of concurrency It is also interesting how compositionality plays out in this setting. As we emphasize throughout the paper, the π-calculus is not a closed theory, but one depending on a theory of names. In other words, a fully specified theory of processes involves a composition of theories, namely [1] an application of the π-calculus with some theory of names. In this sense we derive the ρ-calculus from the fixed point of applying the theory of processes to itself (regarded as a theory of names). Symbolically,

$$\mathcal{R} = \Pi[@\mathcal{R}]$$

where Π produces a theory of processes from whatever it is given as a theory of names and $@\mathcal{R}$ is the quoted forms of said processes, regarding them as names.

This is exactly the kind of design level thinking that compositionality should promote, and it is actually surprising that this particular solution for a name-free version of a concurrent calculus wasn't found sooner. In point of fact, the initial implementations of ρ-calculus in OCaml, Haskell, and Scala all used exactly this fixed point formulation to define the syntax for the ρ-calculus. The fact that the code type-checks and that the types are inhabited provides some assurance of the soundness of the construction.

The combinator versions take compositionality a step further by removing even more of the syntax, effectively arriving at a variant of an applicative algebra over a small handfull of combinators to account for all binding and mobility phenomena. Further still, the soundness of the semantics makes essential use of compositionality because we are effectively composing Yoshida's original semantics with the reflective account of name construction and deconstruction; thus illustrating that compositionality is not just for design-level thinking, but provides powerful compression of proofs.

Outline of the paper To be fully self-contained this paper would need to present four different calculi and two different encodings: the original π-calculus, Yoshida's combinator calculus, and the encoding from the term calculus to the combinator; the ρ-calculus, and the encoding from the π-calculus to the ρ-calculus and the new reflective combinator calculus, and the encoding from the term calculus to the reflective combinator calculus. Such a manifest provides all the technical inventory to illustrate how the two encoding techniques, prefix elimination and new elimination, combine and how the encoding from the π-calculus can be constructed by composing the encoding into the ρ-calculus with the encoding into the reflective combinator calculus. We have provided the complete manifest, but have pushed the

[1] pun gratefully accepted

presentation of the ρ-calculus and the π-calculus and the encoding of the latter into the former in an appendix at the end as the π-calculus is quite well known at this point, and the ρ-calculus has been part of the concurrency literature for over a decade. This organization allows us to focus on the newer results of a name-free combinator calculus for mobile concurrency.

Related work As mentioned previously, there is a long history of interest in combinatorial presentations of computational models, with Yoshida's work representing a seminal development for concurrent computation. We have also recently become aware of [**?**]. They achieve a similar goal, but without reflection, using a technique pioneered by Quine, who was also a pioneer of reflective techniques. [**?**] [**?**]

2 A reflective higher-order concurrent combinator calculus

2.1 Yoshida's original combinator calculus

This is the briefest account of Yoshida's original calculus while her paper, which is nearly 20 years old, is still an outstanding piece of research and well worth the effort to read in conjunction with these results.

ATOM
$$P ::= 0 \mid \mathsf{m}(a,b) \mid \mathsf{d}(a,b,c) \mid \mathsf{k}(a) \mid \mathsf{fw}(a,b) \mid \mathsf{b_r}(a,b) \mid \mathsf{b_l}(a,b) \mid \mathsf{s}(a,b,c)$$
PROCESS
$$\mid (\mathsf{new}\ a)P \mid P|P \mid *P$$

We write $\mathsf{c};\mathsf{c}';\ldots;$ to denote these agents. $\mathsf{m}(a,b)$ (message) carries a name b to name a, d (duplicator) distributes a message to two locations, fw (forwarder) forwards a message (thus linking two locations), k (killer) kills a message, while $\mathsf{b_r}$ (right binder), $\mathsf{b_l}$ (left binder) and s (synchroniser) generate new links. In particular $\mathsf{b_r}$ and $\mathsf{b_l}$ represent two different ways of binding names – in $\mathsf{b_r}$ one uses the received name for output, while in $\mathsf{b_l}$ one uses it for input. In contrast, s is used for pure synchronisation without value passing, which is indeed necessary in interaction scenarios.

As in the π-calculus, the new operator is a binding operator for names, so Yoshida's calculus also has a notion of free and bound names.

$$\mathsf{FN}(0) := \emptyset \qquad \mathsf{FN}(\mathsf{k}(a)) := \{a\} \qquad \mathsf{FN}(\mathsf{m}(a,b)) = \mathsf{FN}(f(a,b)) = \mathsf{FN}(\mathsf{b_r}(a,b)) = \mathsf{FN}(\mathsf{b_l}(a,b)) := \{a,b\}$$

$$\mathsf{FN}(\mathsf{d}(a,b,c)) = \mathsf{FN}(\mathsf{s}(a,b,c)) := \{a,b,c\} \qquad \mathsf{FN}((\mathsf{new}\ a)P) := \mathsf{FN}(P) \setminus \{a\}$$

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

The bound names of a process, $\mathsf{BN}(P)$, are those names occurring in P that are not free. For example, in $(\mathsf{new}\ b)\mathsf{m}(a,b)$, the name a is free, while b is bound.

In the following definition, $\vec{x}$ indicates a list of names, $u :\vec{x}$ indicates the concatenation of u onto the vector, and abuse set notation $u \in\vec{x}$ to assert or require that u occurs in $\vec{x}$.

Definition 1. *Two processes, P, Q, are alpha-equivalent if $P = Q\{\vec{y}\ /\ \vec{x}\}$ for some $\vec{x}\in \mathsf{BN}(Q)$, $\vec{y}\in \mathsf{BN}(P)$, where $Q\{\vec{y}\ /\ \vec{x}\}$ denotes the capture-avoiding substitution of $\vec{y}$ for $\vec{x}$ in Q.*

Definition 2. *The* structural congruence $\equiv$ *between processes [?] is the least congruence containing alpha-equivalence and satisfying the commutative monoid laws (associativity, commutativity and* 0 *as identity) for parallel composition $|$.*

Rewrite rules

$$\begin{array}{rr} \mathsf{d}(a,b,c)|\mathsf{m}(a,x) \to \mathsf{m}(b,x)|\mathsf{m}(c,x) & \mathsf{b_r}(a,b)|\mathsf{m}(a,x) \to \mathsf{fw}(b,x) \\ \mathsf{k}(a)|\mathsf{m}(a,x) \to 0 & \mathsf{b_l}(a,b)|\mathsf{m}(a,x) \to \mathsf{fw}(x,b) \\ \mathsf{fw}(a,b)|\mathsf{m}(a,x) \to \mathsf{m}(b,x) & \mathsf{s}(a,b,c)|\mathsf{m}(a,x) \to \mathsf{fw}(b,c) \end{array}$$

$$*P \to P|*P$$

$$\frac{P \to P'}{P|Q \to P'|Q} \qquad \frac{P \equiv P' \qquad P' \to Q' \qquad Q' \equiv Q}{P \to Q}$$

Translating the π-calculus into Yoshida's combinators We assume the following annotations ($+$ stands for the output and $-$ stands for the input), which denote how each name is used in the rules of interaction:

$$\mathsf{m}(a^+, v^\pm); \mathsf{d}(a^-, b^+, c^+); \mathsf{k}(a^-); \mathsf{fw}(a^-, b^+); \mathsf{b_l}(a^- b^+); \mathsf{b_r}(a^- b^-); \mathsf{s}(a^- b^- c^+)$$

Note the annotated polarities are preserved by reduction, e.g.

$$\mathsf{d}(a^-, b^+, c^+) | \mathsf{m}(a^+, v) \to \mathsf{m}(b^+, v) | \mathsf{m}(b^+, v)$$

It is worth pointing out that we use a slightly different syntax for input-guarded processes. Where most readers familiar with $x?(y)P$ we write $\mathsf{for}(y \leftarrow x)P$, not only to be more instep with modern programming languages, but also because it generalizes more naturally to join constructions, such as $\mathsf{for}(y_1 \leftarrow x_1; \ldots; y_n \leftarrow x_n)P$.

$$[\![a!(b)]\!] = \mathsf{m}(a, b) \qquad [\![\mathsf{for}(x \leftarrow a)P]\!] = \mathsf{for}^*(x \leftarrow a)[\![P]\!] \qquad [\![P|Q]\!] = [\![P]\!]|[\![Q]\!] \qquad [\![(\mathsf{new}\ x)P]\!] = (\mathsf{new}\ x)[\![P]\!]$$

$$[\![*P]\!] = *[\![P]\!] \qquad\qquad [\![0]\!] = 0$$

where

$$\begin{array}{lll}
(I) & \mathsf{for}^*(x \leftarrow a)(P|Q) & = (\mathsf{new}\ c_1 c_2)(\mathsf{d}(a, c_1, c_2)|\mathsf{for}^*(x \leftarrow c_1)P|\mathsf{for}^*(x \leftarrow c_2)Q) \\
(II) & \mathsf{for}^*(x \leftarrow a)(\mathsf{new}\ c')P & = (\mathsf{new}\ c)\mathsf{for}^*(x \leftarrow a)P\{c/c'\} \\
(III) & \mathsf{for}^*(x \leftarrow a)0 & = \mathsf{k}(a) \\
(IV) & \mathsf{for}^*(x \leftarrow a)*P & = (\mathsf{new}\ c)(\mathsf{fw}(a, c)|*\mathsf{for}^*(x \leftarrow c)(P|\mathsf{m}(c, x))) \\
(V) & \mathsf{for}^*(x \leftarrow a)c(v^+, \boldsymbol{w}) & = (\mathsf{new}\ c)(\mathsf{s}(a, c, v)|c(c^+, \boldsymbol{w})) \quad x \notin \{v\boldsymbol{w}\} \\
(VI) & \mathsf{for}^*(x \leftarrow a)c(v^-, \boldsymbol{w}) & = (\mathsf{new}\ c)(\mathsf{s}(a, v, c)|c(c^-, \boldsymbol{w})) \quad x \notin \{v\boldsymbol{w}\} \\
(VII) & \mathsf{for}^*(x \leftarrow a)\mathsf{m}(v, x) & = \mathsf{fw}(a, v) \quad x \neq v \\
(VIII) & \mathsf{for}^*(x \leftarrow a)\mathsf{fw}(x, v) & = \mathsf{b_l}(a, v) \quad x \neq v \\
(IX) & \mathsf{for}^*(x \leftarrow a)\mathsf{fw}(v, x) & = \mathsf{b_r}(a, v) \quad x \neq v \\
(X) & \mathsf{for}^*(x \leftarrow a)c(\boldsymbol{v}_1 x^+ \boldsymbol{v}_2) & = (\mathsf{new}\ c)\mathsf{for}^*(x \leftarrow a)(\mathsf{fw}(c, x)|c(\boldsymbol{v}_1 c \boldsymbol{v}_2)) \quad x \notin \boldsymbol{v}_1 \\
(XI) & \mathsf{for}^*(x \leftarrow a)c(x^- \boldsymbol{v}) & = (\mathsf{new}\ c)\mathsf{for}^*(x \leftarrow a)(\mathsf{fw}(x, c)|c(c\boldsymbol{v})) \\
(XII) & \mathsf{for}^*(x \leftarrow a)\mathsf{b_r}(v, x^-) & = (\mathsf{new}\ c_1 c_2 c_3)\mathsf{for}^*(x \leftarrow a)(\mathsf{d}(v, c_1, c_2)|\mathsf{s}(c_1, x, c_3)|\mathsf{b_r}(c_2, c_3)) \quad x \notin \{v\} \\
(XIII) & \mathsf{for}^*(x \leftarrow a)\mathsf{s}(v, x^-, w) & = (\mathsf{new}\ c_1 c_2)\mathsf{for}^*(x \leftarrow a)(\mathsf{s}(v, c_1, c_2)|\mathsf{m}(c_1, x)|\mathsf{b_l}(c_2, w)) \quad x \notin \{v\}
\end{array}$$

2.2 Reflective higher-order (RHO) combinator calculus

The π-calculus is not a closed theory, but rather a theory dependent upon some theory of names. Taking an operational view, one may think of the π-calculus as a procedure that when handed a theory of names provides a theory of processes that communicate over those names. This openness of the theory has been exploited in π-calculus implementations like the execution engine in Microsoft's Biztalk [**?**], where an ancillary binding language provides a means of specifying a 'theory' of names: *e.g.*, names may be TCP/IP ports, or URLs, or object references, *etc.* But foundationally, one might ask if there is a closed theory of processes, *i.e.* one in which the theory of names arises from and is wholly determined by the theory of processes. Meredith and Radestock have shown that this is not only possible, but results in a calculus that enjoys both the features of concurrency and meta-programming [**?**]. The key idea is to provide the ability to quote processes, effectively reifying them as names, and to unquote them, effectively reflecting names back as processes.

The same technique can be applied to Yoshida's combinators. We remove new names and replication, introduce quoting/unquoting operators. Notice that this effectively allows processes in the second argument of a send, because the name in the second argument is 'merely' a quoted process. This affords an opportunity to introduce an extra rewrite governing the interaction between sending and unquoting.

ATOM
$P ::= 0 \mid \mathsf{m}(a, @P) \mid \mathsf{d}(a,b,c) \mid \mathsf{k}(a) \mid \mathsf{fw}(a,b) \mid \mathsf{b_r}(a,b) \mid \mathsf{b_l}(a,b) \mid \mathsf{s}(a,b,c)$

PROCESS
$\mid *a \mid P|P$

NOMINAL
$a ::= @P$

Rewrite rules

$$\begin{aligned} \mathsf{d}(a,b,c)|\mathsf{m}(a,@P) &\to \mathsf{m}(b,@P)|\mathsf{m}(c,@P) \\ \mathsf{k}(a)|\mathsf{m}(a,@P) &\to 0 \\ \mathsf{fw}(a,b)|\mathsf{m}(a,@P) &\to \mathsf{m}(b,@P) \\ \mathsf{b_r}(a,b)|\mathsf{m}(a,@P) &\to \mathsf{fw}(b,@P) \end{aligned} \qquad \begin{aligned} \mathsf{b_l}(a,b)|\mathsf{m}(a,@P) &\to \mathsf{fw}(@P,b) \\ \mathsf{s}(a,b,c)|\mathsf{m}(a,@P) &\to \mathsf{fw}(b,c) \\ *(a)|\mathsf{m}(a,@P) &\to P \end{aligned}$$

$$\frac{P \to P'}{P|Q \to P'|Q} \qquad \frac{P \equiv P' \qquad P' \to Q' \qquad Q' \equiv Q}{P \to Q}$$

Definition 3. *The* structural congruence $\equiv$ *between processes [?] is the least congruence satisfying the commutative monoid laws (associativity, commutativity and* 0 *as identity) for parallel composition* $\mid$ *and* $*(@(P)) \equiv P$.

Note that alpha equivalence is no longer part of structural congruence. While there is a faithful embedding of Yoshida's combinators into RHO combinators (see below), RHO combinators can see the internal structure of names and distinguish them.

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

Definition 4 (Replication). $D(x,v,w) := (\mathsf{d}(x,v,w)|\mathsf{fw}(v,x)|*(w))$

$$\begin{aligned} *_{(x,v,w)}P &= \mathsf{m}(x, @(D(x,v,w) \mid P)) \mid D(x,v,w) \\ &= \mathsf{d}(x,v,w) \mid \mathsf{fw}(v,x) \mid *(w) \mid \mathsf{m}(x, @(D(x,v,w) \mid P)) \\ &\to \mathsf{m}(v, @(D(x,v,w) \mid P)) \mid \mathsf{m}(w, @(D(x,v,w) \mid P)) \mid \mathsf{fw}(v,x) \mid *(w) \mid \mathsf{m}(x, @(D(x,v,w) \mid P)) \\ &\to \mathsf{m}(x, @(D(x,v,w) \mid P)) \mid \mathsf{m}(w, @(D(x,v,w) \mid P)) \mid *(w) \\ &\to \mathsf{m}(x, @(D(x,v,w) \mid P)) \mid D(x,v,w) \mid P \\ &= *_{(x,v,w)}P \mid P \end{aligned}$$

Of course, this encoding, as an implementation, runs away, unfolding $*P$ eagerly. It is possible to obtain a lazier replication operator restricted to the embedding of input-guarded π processes. The reader familiar with the λ-calculus will have noticed the similarity between D and the "paradoxical" or "fixed point" combinator Y.

Implementing new names with reflection Here we provide an encoding of Yoshida's combinator calculus into the RHO combinator calculus. Since all names are global in the RHO combinator calculus, we encounter a small complication in the treatment of free names at the outset. There are several ways to handle this. One is to insist that the translation be handed a closed program, one in which all names are bound either by input or by restriction, but this feels inelegant. Another is to provide an environment, $r : \mathcal{N}_{\text{Yoshida}} \to @\mathsf{Proc}$, for mapping the free names in a Yoshida process into names in the RHO combinator calculus. Maintaining the updates to the environment, however, obscures the simplicity of the translation. We adopt a third alternative.

To hammer home the point that Yoshida's combinator calculus is parameterized in a theory of names, we instantiate her calculus with the names of the RHO combinator calculus. This is no different than instantiating her calculus using the natural numbers, or the set of URLs as the set of names. Just as there is no connection between the structure of these kinds of names and the structure of processes in

the π-calculus, there is no connection between the processes quoted in the names used by the theory and the processes generated by the theory, and we exploit this fact.

Let @Proc be set of names in the RHO combinator calculus, Proc be the set of terms of the RHO combinator calculus, and $\mathsf{Proc}_{\text{Yoshida}}$ be the set of terms of her combinator calculus built using @Proc *as the names*. The translation will be given in terms of a function

$$[\![-]\!]_2(-,-) : \mathsf{Proc}_{\text{Yoshida}} \times @\mathsf{Proc} \times @\mathsf{Proc} \to \mathsf{Proc}.$$

The guiding intuition is that we construct alongside the process a distributed memory allocator, the process' access to which is mediated through the second argument to the function, called p below. The first argument, called n below, determines the shape of the memory for the given allocator.

Since Yoshida's calculus is parametric in a set of names, we can choose @Proc for that set. Given a process P in Yoshida's calculus, we pick names n and p in the RHO calculus such that $n \neq p$ and both are distinct from the free names of P. Then we define

$$[\![P]\!] = [\![P]\!]_2(n,p),$$

Name allocation will make heavy use of the following two name constructors

$$\begin{aligned} x^l &:= @\mathsf{b_l}(x, \mathsf{m}(x, *x)) \\ x^r &:= @\mathsf{b_r}(x, \mathsf{m}(x, *x)) \end{aligned}$$

Note that by construction, $@P$ cannot occur as a name in P and hence any name derived from a process that is built using $@P$ cannot occur in P. Thus, the effect of the superscripts l and r on a name x is to construct a name that is guaranteed to be fresh with respect to the free names of the process being interpreted. More generally, mentioning a name, say n, in the constructor of a another name, say n', guarantees distinction between n and n'; likewise, mentioning a process, say P, in the constructor of a name n guarantees that n is fresh in P. The particular choices of combinators used in the name constructors are irrelevant for the purposes of freshness. We make heavy use of this fact in our interpretation of prefix elimination.

The interpretation function $[\![-]\!]_2(n,p)$ is straightforward for all but replication and new.

$$\begin{aligned} [\![0]\!]_2(n,p) &= 0 \\ [\![c(\overrightarrow{a})]\!]_2(n,p) &= c(\overrightarrow{a}), \text{ where } c \text{ is any combinator but } m \\ [\![\mathsf{m}(a, @P)]\!]_2(n,p) &= \mathsf{m}(a, @[\![P]\!]_2(n,p)) \\ [\![P \mid Q]\!]_2(n,p) &= [\![P]\!]_2(n^l, p^l) \mid [\![Q]\!]_2(n^r, p^r) \end{aligned}$$

These latter two forms require extra care. We define them in terms of a prefix form and then use a version of Yoshida's prefix elimination to remove the prefix.

$$\begin{aligned} [\![*P]\!]_2(n,p) &= \mathsf{m}(x, @[\![P]\!]_3(n^r, p^r)) | D(x,v,w) | \mathsf{m}(n^r, *n^l) | \mathsf{m}(n^r, *n^l) \\ &\quad \text{where } x = @([\![P]\!]_2(n,p))^{ll}, v = @([\![P]\!]_2(n,p))^{lr}, \text{ and } w = @([\![P]\!]_2(n,p))^{rr} \\ [\![(\mathsf{new}\ x)P]\!]_2(n,p) &= [\![\mathsf{for}(x \leftarrow p?)[\![P]\!]_2(n^l, p^l) | \mathsf{m}(p,n)]\!]_4(n,p) \end{aligned}$$

As expected, the interpretation of replication makes use of the D operator defined above. Note that name allocation is intertwined with prefix and new elimination.

$$[\![P]\!]_3(n,p) := [\![\mathsf{for}(n' \leftarrow n?)\mathsf{for}(p' \leftarrow p?)([\![P]\!]_2(n',p') | (D(x) | n!(n'^l) | p!(p'^l)))]\!]_4.$$

To handle prefix elimination we must import most of Yoshida's algorithm. The key difference is that we must allocate names that guarantee freshness relative to the free names of the processes being translated. In those rules below with a "where" clause, the specific choice of combinators in the names is not so important as mentioning those names and processes with respect to which the name mush be fresh. For example, in rule I, for prefix to a parallel composition, we must ensure that v and w are fresh with respect to the names in P and Q, and distinct from each other, and then we must update both the name allocator for each parallel component and the channels on which fresh names are communicate (so that there is no interference between the two components) in the recursive call.

Because the input to $[\![-]\!]_4$ is already in combinator form, we do not have to import rules 2 – 4 of her algorithm. Like her, we assume the following annotations (+ stands for the output and − stands for the input), which denote how each name is used in the rules of interaction:

$$\mathsf{m}(a^+, v^\pm); \mathsf{d}(a^-, b^+, c^+); \mathsf{k}(a^-); \mathsf{fw}(a^-, b^+); \mathsf{b_l}(a^- b^+); \mathsf{b_r}(a^- b^-); \mathsf{s}(a^- b^- c^+)$$

As above the annotated polarities are preserved by reduction, e.g.

$$\mathsf{d}(a^-, b^+, c^+)|\mathsf{m}(a^+, v) \to \mathsf{m}(b^+, v)|\mathsf{m}(b^+, v)$$

Similarly, for economy of expression, we emulate Yoshida's use of c to represent any combinator matching the arity specification.

$$
\begin{array}{lll}
(I) & [\![\mathsf{for}(x \leftarrow p?)P|Q]\!]_4(n,q) & := (\mathsf{d}(p,v,w)|[\![\mathsf{for}(x \leftarrow v?)P]\!]_4(n_1,q_1)|[\![\mathsf{for}(x \leftarrow w?)Q]\!]_4(n_2,q_2) \\
 & & \quad \text{where } v = @(\mathsf{m}(q, @\mathsf{b_l}(q,n)|P|Q)),\ w = @(\mathsf{m}(q, @\mathsf{b_r}(q,n)|P|Q)), \\
 & & \quad n_1 = @(\mathsf{b_l}(v,w)|\mathsf{m}(q, @\mathsf{m}(v, *w))),\ n_2 = @(\mathsf{b_r}(v,w)|\mathsf{m}(q, @\mathsf{m}(v, *w))) \\
 & & \quad q_1 = @(\mathsf{b_l}(n_1,n_2)|\mathsf{m}(q, @\mathsf{m}(v, *w))),\ q_2 = @(\mathsf{b_r}(n_1,n_2)|\mathsf{m}(q, @\mathsf{m}(v, *w))) \\
(V) & [\![\mathsf{for}(x \leftarrow p?)c(v^+, w)]\!]_4(n,q) & := \mathsf{s}(p,a,v)|c(a^+, \overrightarrow{w}), \text{ where } a = @(\mathsf{m}(q,n)|c(v^+, \overrightarrow{w})) \text{ and } x \notin v : \overrightarrow{w} \\
(VI) & [\![\mathsf{for}(x \leftarrow p?)c(v^-, \overrightarrow{w})]\!]_4(n,q) & := \mathsf{s}(p,v,a)|c(a^-, \overrightarrow{w}), \text{ where } a = @(\mathsf{m}(q,n)|c(v^-, w)) \text{ and } x \notin v : \overrightarrow{w} \\
(VII) & [\![\mathsf{for}(x \leftarrow p?)\mathsf{m}(v,x)]\!]_4(n,q) & := \mathsf{fw}(p,v) \\
(VIII) & [\![\mathsf{for}(x \leftarrow p?)\mathsf{fw}(x,v)]\!]_4(n,q) & := \mathsf{b_l}(p,v) \\
(IX) & [\![\mathsf{for}(x \leftarrow p?)\mathsf{fw}(v,x)]\!]_4(n,q) & := \mathsf{b_r}(p,v) \\
(X) & [\![\mathsf{for}(x \leftarrow p?)c(\overrightarrow{v}, x^+, \overrightarrow{w})]\!]_4(n,q) & := [\![\mathsf{for}(x \leftarrow p?)\mathsf{fw}(a,x)|c(\overrightarrow{v}, a^+, \overrightarrow{w})]\!]_4(n',q) \\
 & & \quad \text{where } a = @(\mathsf{m}(q,n)|c(\overrightarrow{v}, x^+, \overrightarrow{w})) \text{ and } n' = @(\mathsf{m}(a, @\mathsf{m}(q, *n))) \\
(XI) & [\![\mathsf{for}(x \leftarrow p?)c(x^-, \overrightarrow{v})]\!]_4(n,q) & := [\![\mathsf{for}(x \leftarrow p?)\mathsf{fw}(x,a)|c(a^-, \overrightarrow{v})]\!]_4(n',q) \\
 & & \quad \text{where } a = @(\mathsf{m}(q,n)|c(x^-, v)) \text{ and } n' = @(\mathsf{m}(a, @\mathsf{m}(q, *n))) \\
(XII) & [\![\mathsf{for}(x \leftarrow p?)\mathsf{b_r}(v,x^-)]\!]_4(n,q) & := [\![\mathsf{for}(x \leftarrow p?)(\mathsf{d}(v,w_1,w_2)|\mathsf{s}(w_1,x,w_3)|\mathsf{b_r}(w_2,w_3))]\!]_4(n',q) \\
 & & \quad \text{where } w_1 = @(\mathsf{b_l}(q,n)|\mathsf{m}(q, @\mathsf{b_r}(v,x^-))),\ w_2 = @(\mathsf{b_r}(q,n)|\mathsf{m}(q, @\mathsf{b_r}(v,x^-))), \\
 & & \quad w_3 = @(\mathsf{b_l}(p,v)|\mathsf{m}(w_1,w_2), \text{ and } n' = @(\mathsf{s}(w_1,w_2,w_3)) \\
(XIII) & [\![\mathsf{for}(x \leftarrow p?)\mathsf{s}(v,x^-,w)]\!]_4(n,q) & := [\![\mathsf{for}(x \leftarrow p?)(\mathsf{s}(v,w_1,w_2)|\mathsf{m}(w_1,x)|\mathsf{b_l}(w_2,w))]\!]_4(n',q) \\
 & & \quad \text{where } w_1 = @(\mathsf{b_l}(q,n)|\mathsf{s}(v,x^-,w)),\ w_2 = @(\mathsf{b_r}(q,n)|\mathsf{s}(v,x^-,w)), \\
 & & \quad \text{and } n' = @(\mathsf{m}(w_1,w_2))
\end{array}
$$

Note that all new-binding is now interpreted, as in Wischik's global π-calculus, as an input [**?**].

It is also noteworthy that the translation is dependent on how the parallel compositions in a process are associated. Different associations will result in different bindings for new names. This will not result in different behavior, however: while the RHO combinators can encode different behaviors depending on the choice of name, Yoshida's combinators cannot and the embedding is insensitive to the choice.

Faithfulness of the translation

Definition 5. *An* observation relation, $\downarrow_{\mathcal{N}}$, *over a set of names,* $\mathcal{N}$, *is the smallest relation satisfying the rules below.*

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

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

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

This definition is parametric in the the argument accepted in the second position in the message combinator, $\mathsf{m}(x-)$, *i.e.* the payload of the message: in the RHO combinators the payload is a process, while in Yoshida's the payload is a name. Likewise, because the definition of barbed bisimulation given below is dependent on the definition of the observation relation, the definition is really a template for the notion of bisimulation that must be instantiated to the kind of payload accepted by the message combinator.

Definition 6. *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 $\mathcal{S}_{\mathcal{N}}$.

Theorem 1. $P \dot{\approx}_{\pi} Q \iff [\![P]\!] \dot{\approx}_{FN(P)} [\![Q]\!]$.

Proof (Proof sketch). The forward direction $\Rightarrow$ is immediate from the definition of the translation. The reverse direction is only interesting in the case of ! and new. The replication case follows immediately from the calculation following definition **??**. In the new case, transitions on new-bound names will be in one-to-one correspondence with names provided by the name parameters of the translation function. By construction, these are not observable by the observation relation.

Remark 1. In light of this theorem, it is worth pointing out that this version of the RHO combinators has no rule for *introducing* terms of the form $*x$. The b_r and b_l combinators introduce new names from processes, but the do not introduce new reflection terms. Yet, this calculus suffices to faithfully represent the Yoshida combinators. This is because the translation function is carefully introducing just those terms, via the $D(x, v, w)$ operator, guided by the use of replication in the source to the translation.

3 Conclusion and future work

We have shown how to construct a concurrent higher-order combinator calculus that uses reflection to avoid the necessity for new and bound names. Yoshida's combinators, and therefore the asynchronous π-calculus, have a faithful embedding into the calculus. While the results are interesting in their own right we developed them to serve a larger purpose. In a forthcoming paper we demonstrate an algorithm taking a graph-enriched Lawvere theory (representing a formal specification of a term calculus) and a more vanilla Lawvere theory (representing a specification of a notion of collection, such as set, or bag, or list, etc) and produce a type system for the term calculus enjoying soundness and completeness. Nominal phenomena do not neatly fit inside the expressive power of graph-enriched Lawvere theories, thus potentially limiting the scope of the applicability of this algorithm. However, with the reflective techniques we can extend this algorithm to cover many languages and term calculi with binding.

At a more foundational level it turns out that the reflective techniques can be applied to set theory itself deriving a form of FM set theory in which the "atoms" of one set theory are the sets of another copy of set theory. Roughly speaking, we may imagine that there are "red" sets and "black" sets. The "term constructors" of red set theory (the curly braces of set notation) can use either red sets or black

sets, which the element-of predicate ($\in$) red set theory can only inspect red sets and treats black sets as "atomic". The symmetric situation holds for black set theory: the element-of predicate cannot see inside red sets. This framework all the building blocks necessary to realize a version of FM set theory from a more traditional set theory with reflection and provides a bridge between Gabbay and Pitts' account of nominal phenomena and the reflection-based approach.

4 Appendix: Translating the π-calculus into the ρ-calculus

4.1 ρ-calculus

It is striking to compare the ρ-calculus with the π-calculus as the former is a vastly simpler theory and yet has enjoys more features. Specifically, the specification of the ρ-calculus's structural equivalence and reduction rules are notably simpler, with one small technical caveat: name equivalence depends on structural equivalence which depends upon α-equivalence which depends on name equivalence. Meredith and Radestock show that this cycle terminates innocently due to the design of the grammar. This technical complexity seems a small price to pay for a much simpler calculus that enjoys both higher order communication as well reflection and meta-programming features.

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

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

Definition 7. Free and bound names *The calculation of the free names of a process, P, denoted* $\mathsf{FN}(P)$ *is given recursively by*

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

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

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

Definition 8. *The* structural congruence $\equiv$ *between processes [?] is the least congruence containing alpha-equivalence and satisfying the commutative monoid laws (associativity, commutativity and* 0 *as identity) for parallel composition $|$.*

Definition 9. *The* name equivalence $\equiv_N$ *is the least congruence satisfying these equations*

QUOTE-DROP
$$@*x \equiv_N x$$

STRUCT-EQUIV
$$\frac{P \equiv Q}{@P \equiv_N @Q}$$

4.2 Operational semantics

COMM
$$\frac{x_{trgt} \equiv_N x_{src}}{\mathsf{for}(y \leftarrow x_{trgt})P|x_{src}!(Q) \to P\{@Q/y\}}$$

PAR
$$\frac{P \to P'}{P|Q \to P'|Q}$$

EQUIV
$$\frac{P \equiv P' \qquad P' \to Q' \qquad Q' \equiv Q}{P \to Q}$$

4.3 π-calculus

In this presentation of the π-calculus we update the syntax for input-guarded processes to reflect the widespread adoption of comprehension notation in languages ranging from Scala to Python for use in reactive programming. Here we go with the Scala notation writing $\mathsf{for}(y \leftarrow x)P$ where Milner might have written $x?(y)P$. Admittedly, it's somewhat more verbose, but conveys to a younger generation of programmers more familiar with reactive programming the intended semantics of the expression. Similarly, since we keep the output expression $x!(y)$ we avoid collision with the traditional notation for reflection ($!P$) by writing $*P$ instead, which is at least somewhat reminiscent of the Kleene star and the familiar from regular expressions.

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

Note that there is no production for x's and y's in the grammar. This reflects the fact that the π-calculus is *parametric* in the collection channel names. That collection merely has to be countably infinite and have an effective equality. As such it is perfectly reasonable to choose a collection names, namely the names of the ρ-calculus. We can make this choice without loss of generality because we can always choose some other countably infinite set with an effective equality, say $\mathcal{N}$ and then require an invertible map, $\mathsf{code} : \mathcal{N} \to @\mathsf{Proc}$.

4.4 Structural congruence

Definition 10. *The* structural congruence, $\equiv$, *between processes is the least congruence closed with respect to alpha-renaming, satisfying the abelian monoid laws for parallel (associativity, commutativity and* 0 *as identity), and the following axioms:*

1. *the scope laws:*

$$\begin{aligned}(\mathsf{new}\ x)0 &\equiv 0,\\ (\mathsf{new}\ x)(\mathsf{new}\ x)P &\equiv (\mathsf{new}\ x)P,\\ (\mathsf{new}\ x)(\mathsf{new}\ y)P &\equiv (\mathsf{new}\ y)(\mathsf{new}\ x)P,\\ P|(\mathsf{new}\ x)Q &\equiv (\mathsf{new}\ x)P|Q,\ \textit{if}\ x \notin \mathsf{FN}(P)\end{aligned}$$

2. *the recursion law:*

$$*P \equiv P|*P$$

3. *the name equivalence law:*

$$P \equiv P\{x/y\},\ \textit{if}\ x \equiv_N y$$

4.5 Operational semantics

COMM
$$\mathsf{for}(y \leftarrow x)P|x!(v) \to P\{v/y\}$$

PAR
$$\frac{P \to P'}{P|Q \to P'|Q}$$

NEW
$$\frac{P \to P'}{(\mathsf{new}\ x)P \to (\mathsf{new}\ x)P'}$$

EQUIV
$$\frac{P \equiv P' \qquad P' \to Q' \qquad Q' \equiv Q}{P \to Q}$$

Here we stick with tradition and write $\Rightarrow$ for $\to^*$, and rely on context to distinguish when $\to$ means reduction in the π-calculus and when it means reduction in the ρ-calculus. The set of π-calculus processes will be denoted by Proc_π.

4.6 The translation

The translation will be given by a function, $[\![-]\!](-,-) : \mathsf{Proc}_\pi \times @\mathsf{Proc} \times @\mathsf{Proc} \to \mathsf{Proc}$. The guiding intuition is that we construct alongside the process a distributed memory allocator, the process' access to which is mediated through the second argument to the function. The first argument determines the shape of the memory for the given allocator.

Given a process, P, we pick n and p such that $n \neq p$ and distinct from the free names of P. For example, $n = @\Pi_{m \in \mathsf{FN}(P)} m!(@0)$ and $p = @\Pi_{m \in \mathsf{FN}(P)} \mathsf{for}(@0 \leftarrow m)0$. Then

$$[\![P]\!] = [\![P]\!]_{2nd}(n,p)$$

where

$$\begin{aligned}
[\![0]\!]_{2nd}(n,p) &= 0 \\
[\![x!(@Q)]\!]_{2nd}(n,p) &= x!(Q) \\
[\![\mathsf{for}(y \leftarrow x)P]\!]_{2nd}(n,p) &= \mathsf{for}(y \leftarrow x)[\![P]\!]_{2nd}(n,p) \\
[\![P|Q]\!]_{2nd}(n,p) &= [\![P]\!]_{2nd}(n^l,p^l)|[\![Q]\!]_{2nd}(n^r,p^r) \\
[\![*P]\!]_{2nd}(n,p) &= x!([\![P]\!]_{3rd}(n^r,p^r))|D(x)|n^r!n^l|p^r!p^l \\
[\![(\mathsf{new}\ x)P]\!]_{2nd}(n,p) &= \mathsf{for}(x \leftarrow p)[\![P]\!]_{2nd}(n^l,p^l)|p!(n)
\end{aligned}$$

and

$$\begin{aligned}
x^l &\triangleq @x!(x) \\
x^r &\triangleq @\mathsf{for}(x \leftarrow x?)\ 0 \\
[\![P]\!]_{3rd}(n'',p'') &\triangleq \mathsf{for}(n \leftarrow n''?)\mathsf{for}(p \leftarrow p''?)([\![P]\!]_{2nd}(n,p)|(D(x)|n''!(n^l)|p''!(p^l)))
\end{aligned}$$

Remark 2. Note that all new-binding is now interpreted, as in Wischik's global π-calculus, as an input guard [**?**].

Remark 3. It is also noteworthy that the translation is dependent on how the parallel compositions in a process are associated. Different associations will result in different bindings for new-ed names. This will not result in different behavior, however, as the bindings will be consistent throughout the translation of the process.

Theorem 2 (Correctness). $P \dot{\approx}_\pi Q \iff [\![P]\!] \dot{\approx}_{r(FN(P))} [\![Q]\!]$.

Proof sketch: An easy structural induction.

One key point in the proof is that there are contexts in the ρ-calculus that will distinguish the translations. But, these are contexts that can see the fresh names, n, and the communication channel, p, for the 'memory allocator'. These contexts do not correspond to any observation that can be made in the π-calculus and so we exclude them in the ρ-calculus side of our translation by our choice of N for the bisimulation. This is one of the technical motivations behind our introduction of a less standard bisimulation.

Example 1. In a similar vein consider, for an appropriately chosen p and n we have

$$[\![(\mathsf{new}\ v)(\mathsf{new}\ v)u!(v)]\!] = \mathsf{for}(v \leftarrow p)(\mathsf{for}(v \leftarrow @p!(p))(u!(v)|@p!(p))!(@n!(n)))|p!(n)$$

and

$$[\![(\mathsf{new}\ v)u!(v)]\!] = \mathsf{for}(v \leftarrow p)(u!(v))|p!(n)$$

Both programs will ultimately result in an output of a single fresh name on the channel u. But, the former program will consume more resources. Two names will be allocated; two memory requests will be fulfilled. The ρ-calculus can see this, while the π-calculus cannot. In particular, the π-calculus requires that $(\mathsf{new}\ x)(\mathsf{new}\ x)P \equiv (\mathsf{new}x)P$.

Implementations of the π-calculus, however, having the property that $(\mathsf{new}\ x)P$ involves the allocation of memory for the structure representing the channel x come to grips with the implications this requirement has regarding memory management. If memory is allocated upon encountering the new-scope, there are situations where the left-hand side of the equation above will fail while the right-hand will succeed. Remaining faithful to the equation above requires that such implementations are *lazy* in their interpretation of $(\mathsf{new}\ x)P$, only allocating the memory for the fresh channel at the first moment when that channel is used.

Having a detailed account of the structure of names elucidates this issue at the theoretical level and may make way to offer guidance to implementations.