New, on reflection

L.G. MEREDITH, ALLEN L. BROWN, JR., AND DAVID RICHTER

Abstract. Many of the features one might consider for a practical programming language based on mobile process algebras [**?**][**?**] including implementation strategies for recursion; sending and receiving complex data; macros; and durability across transactions can all be related to *serialization*, in the sense of producing an external representation of a process. Towards a theory which accounts for these phenomena, internally, we study a construction for adding reflection to name-passing calculi.

We illustrate the construction by applying it to the pure asynchronous π-calculus, to arrive at a variant we dub the ρ-calculus. The calculus is Turing-complete and yet is free of any explicit construct for recursion or replication. Moreover, it endures no explicit operator for the construction and binding of fresh names. Instead, the reflective mechanism it introduces, allowing for turning processes into names and names into processes, suffices to recapitulate all of the functionality normally found in non-reflective process algebras. We also consider some examples that suggest it is a promising start to treating the phenomena mentioned afore.

1. Introduction

Many of the features one might consider for a practical programming language based on mobile process algebras [**?**][**?**] including implementation strategies for recursion; sending and receiving complex data; macros; and durability across transactions can all be related to *serialization*, in the sense of producing an external representation of a process. For example, one of the key points of transaction semantics in systems like `Biztalk` [**?**] is *durability*. A running process restarts from the last known transacted outcome in the event of failure and restart. Such durability requires (upon entering a transacted context) the ability to serialize a representation of the state of the process and commit it to durable store, like a database management system. It also requires being able to deserialize that representation into an executing process on restart.

Another example comes in the form of macros. A sophisticated macro capability, such as one sees in implementations of the language `Scheme` [**?**], works by being able to turn program source into data, operate on the data, and turn the modified data back into program source. What unifies these two examples, in the mobile process algebra setting, is that program source may be identified with program state in the sense that there is no hidden virtual machine state that also needs to be captured. Thus, having a solution to one means, effectively, having a solution for the other.

These two features are also distinguished from each other by a certain characteristic. In the case of recovery, program state may very well be inspected by the recovering program to make decisions regarding what branch to execute. That is, application level logic makes use of reified process state. Even in the

Key words and phrases. concurrency, distribution, message-passing, process calculus, service-oriented architecture, quality of service, service level agreement.

most sophisticated a macro systems, on the other hand, the program logic is insulated from the macro-expansion process. But, it is still the case, in the sophisticated macro system, that the programming language in which macros are expressed is the language in which program source is expressed. In this sense, macros are 'meta-programs', or programs that calculate programs, while the durability feature of systems like `Biztalk` make some application programs also be meta-programs.

By way of introducing a nomenclature for discussing the results of the paper, we observe that– although the usage is a bit stretched– in common parlance, a structured way of turning program into data and data into program has been dubbed *reflection*. In point of fact, there are several forms of reflection that may be distinguished from one another: structural, procedural and lexical.

Structural reflection is best exemplified in a language like `Java` [**?**]. Program elements, e.g. classes, may be turned into program manipulable data structures. But, it is not the case that the state of a virtual machine executing a `Java` program is manipulable by a program running in that virtual machine. This is to be contrasted with procedurally reflective languages like `3-Lisp` [**?**] and `Rosette` [**?**] where the state of the executing machinery is also made available as program-manipulable data.

In lexical reflection, every element of the *syntax* of the language has a representation as a manipulable program data structure. This is not the case in languages like `Java`. But, it is found in `Rosette` and `3-Lisp`. Note that while this is not generally recognized under this term in the literature, the phenomenon is real. Among other things, this feature forms the basis of an architecture for parser generation.

The approach to name-passing calculi that we study here effectively allows a treatment of all three types of reflection within the theory.

In a similar vein, it is really beyond human capability to do significant calculation with the π-calculus as it stands. Without the ability to send and receive complex data, calculation quickly becomes incomprehensible. One needs a higher level abstraction, for example, to carry out any sort of ordinary arithmetic. Providing a higher level language for data manipulation begs the question: how does the data language relate to the control language? Why isn't there only one language? Adding reflection to name-passing calculi suggests it may be possible to achieve a unification by making the program constructors of the language *be* the data constructors. Of course, it doesn't specify which process constructors represent the data abstractions.

Surprisingly, it also has consequences for practical issues of space management and the implementation of recursive features, like $!P$. For example, if the $(\nu\ x)-$ operator is interpreted as an operation involving memory allocation then there is a problem satisfying the structural equivalence

$$(\nu\ x)(\nu\ x)P \equiv (\nu\ x)P$$

The left hand side may cause a memory fault while the right hand side does not. If, on the other hand, the operator is not interpreted as an operation involving memory allocation, then what is the interpretation? In point of fact, it says that the ν-binder must be *lazily* related to memory allocation. Considering these implementation strategies is at level of detail too rich for name-passing process algebras presented to date.

Lucian Wischik was motivated by this problem (and others) with the implementation of the $(\nu\ x)-$ operator to develop the *global π-calculus* [**?**]. We were very much surprised to see how similar our behavioral rules were to his when we developed this calculus. Of course, there is a difference. The difference is that we say exactly which set of names are the global names and provide a rationale for picking this set: the names are the reified processes.

Another point that bears mentioning is the treatment of recursive constructs, like $!P$. In most practical implementations this is achieved by making program source (at some level of representation) multiply instantiable. The mobile process algebras provide only a specification of this behavior (in the form of the

structural congruence rules for $!P$, or in some presentations, in the dynamics). They steadfastly remain silent on implementation strategies, and rightly so.

One thing is clear, the approach leads to a theory enjoying sufficiently rich detail to describe implementation strategies for this specification. In some sense, the situation is analogous to the difference between the λ-calculus and the π-calculus. The latter is a theory with a richer, or more intensional structure, allowing a finer grain of distinctions to be made. Likewise, the calculus at which we arrive is a calculus that allows a more intensional characterization of the features the name-passing calculi take for granted.

In this paper we illustrate the construction for a pure asynchronous calculus. But, we stress that this construction works for name-passing calculi, in general.

2. The calculus

To motivate the technical presentation we wish to make an observation about the asynchronous π-calculus: it is a theory parametric in a theory of *names*. Standard presentations assume a supply of names and a theory of equality on them. This point is made abundantly clear in such presentations by the grammar describing terms of the language: there is no production for names. Names are taken to be terminals in the grammar. [1]

2.0.1. *Quote.* We will adopt a different tac by closing the theory processes, making processes serve as their own theory of names. So, the first point of departure from a standard presentation of the asynchronous π-calculus is a production for names in the grammar. A name will be a *quoted* process, $\ulcorner P \urcorner$.

2.0.2. *Lift.* Note that name construction is not process construction. So, if one wants to be able to generate a name from a given process, there must be an operator in the term language that creates a name from a process. This is the motivation for the production $x\langle\!|P|\!\rangle$, dubbed here the *lift* operator.

In terms of the nomenclature above, that this mechanism allows (for one half of) both structural reflection should be clear: program elements are being turned into data. As we shall see, though, it is also possible to saturate processes with the lift operator. Intuitively, this means that every continuation is reified before evaluation. This mechanism forms the basis of procedural reflection.

2.0.3. *Dropping ν.* The lift operator turns out to play a role analogous to $(\nu\ x)P$. It is certainly a mechanism by which processes may manufacture fresh occurrences of names. As such, it turns out not to be necessary to provide the ν-binder. We will illustrate the implementation of ν in the sequel.

Remark 2.0.1. On the basis of responses to talks about this material, at this point we issue a warning to the reader: it will turn out that quoted processes are opaque to substitutions. But, this operator is not. This will provide a key difference between $x\langle\!|P|\!\rangle$ and outputing a quoted process.

2.0.4. *Eval.* Of course, if a name is a quoted process, it would be very handy to have a way of *evaluating* such an entity. Thus, one of term constructors for processes takes a name, say x, and extracts the process from it, $\urcorner x \ulcorner$. This operator, nicknamed *drop* to parallel lift, is the other half of the reflective mechanism; it allows data to be transformed back into program.

[1] `Biztalk` takes advantage of this fact in its binding language. A binding says what kind of terminal a name is, e.g. a object reference, or an `MSMQ` queue.

2.0.5. *Dropping replication.* It is well-known that replication is not required in a higher-order process algebra [**?**]. While our algebra is *not* higher-order in the traditional sense (there are not formal process variables of a different type from names) it has all the features of a higher-order process algebra. Thus, it turns out that there is no need for a term for recursion. To illustrate this we present an implementation of $!P$ in this calculus. Intuitively, this will amount to receiving a quoted form of a process, evaluating it, while making the quoted form available again.

2.0.6. *Name equality.* In standard presentations of the asynchronous π-calculus, the statement of the communications rule critically depends on the equality that comes with the theory of names. The situation is the same here. For output and input processes to communicate it is important to know how to determine whether the subject guards are the same. But, another interesting point of departure between the calculi is reflected in the fact that in the former, the theory of equality of names is opaque. It must be give a priori, and this determination is deemed independent of the theory of processes. In the latter, the equality between names is not opaque, but given explicitly, and is determined by the structure of processes. More about this point later, though.

2.0.7. *Notation.* We distinguish between syntactic categories and elements thereof. Thus, **P** denotes a syntactic category, while P is an element of that category.

ρ-calculus

$$
\begin{array}{rcl}
\mathbf{P} & ::= & 0 \\
 & | & \mathtt{x[x]} \\
 & | & \mathtt{x(x)\ .\ P} \\
 & | & \mathtt{x}\langle\!|\mathbf{P}|\!\rangle \\
 & | & \mathbf{P}\ \mathtt{|}\ \mathbf{P} \\
 & | & \ulcorner\mathbf{x}\urcorner \\
\mathbf{x} & ::= & \ulcorner\mathbf{P}\urcorner
\end{array}
$$

2.1. **Free and bound names.** The syntax has been carefully chosen so that a bindging occurrence of a name is sandwiched between round braces. Thus, the calculation of the free names of a process, P, denoted $\mathcal{FN}(P)$ is given recursively by

$$
\begin{array}{rcl}
\mathcal{FN}(0) & = & \emptyset \\
\mathcal{FN}(x\mathtt{[}y\mathtt{]}) & = & \{x, y\} \\
\mathcal{FN}(x\mathtt{(}y\mathtt{)\ .\ }P) & = & \{x\} \cup (\mathcal{FN}(P) \setminus \{y\}) \\
\mathcal{FN}(x\langle\!|P|\!\rangle) & = & \{x\} \cup \mathcal{FN}(P) \\
\mathcal{FN}(P\ \mathtt{|}\ Q) & = & \mathcal{FN}(P) \cup \mathcal{FN}(Q) \\
\mathcal{FN}(\ulcorner x\urcorner) & = & \{x\}
\end{array}
$$

2.2. **Structural congruence.** The *structural congruence* of processes, noted $\equiv$, is the least congruence, containing $\equiv_\alpha$ that satisfies the following laws:

$$\begin{array}{rcl} \mathbf{P} \mid 0 & \equiv \mathbf{P} \equiv & 0 \mid \mathbf{P} \\ \mathbf{P}_0 \mid \mathbf{P}_1 & \equiv & \mathbf{P}_1 \mid \mathbf{P}_0 \\ (\mathbf{P}_0 \mid \mathbf{P}_1) \mid \mathbf{P}_2 & \equiv & \mathbf{P}_0 \mid (\mathbf{P}_1 \mid \mathbf{P}_2) \end{array}$$

2.3. **Name equivalence.** we must first equip our names with a theory of equality. We take this to be the smallest equivalence relation generated by the following rules.

$$\frac{}{\ulcorner\urcorner x^{\ulcorner\urcorner} \equiv_N x} \qquad \text{(Quote-drop)}$$

$$\frac{P \equiv Q}{\ulcorner P \urcorner \equiv_N \ulcorner Q \urcorner} \qquad \text{(Struct-equiv)}$$

2.4. **Substitution.** As is standard we use $P\{\vec{x_1}/\vec{x_0}\}$ to denote capture-avoiding substitution of the names $\vec{x_1}$ for the names $\vec{x_0}$. But, some care must be taken regarding substitutions and quoted processes, i.e. $\ulcorner P \urcorner$.

Now, let $Proc$ denote the set of processes and $\ulcorner Proc \urcorner$ denote the set of names. Then, substitutions are defined as partial maps, $s : \ulcorner Proc \urcorner \to \ulcorner Proc \urcorner$. These lift, uniquely, to maps $\urcorner s \ulcorner : Proc \to Proc$ by the following equations.

$$\begin{array}{rcl} (0)^{\urcorner}\{\ulcorner Q \urcorner/\ulcorner P \urcorner\}^{\ulcorner} & = & 0 \\ (R \mid S)^{\urcorner}\{\ulcorner Q \urcorner/\ulcorner P \urcorner\}^{\ulcorner} & = & (R)^{\urcorner}\{\ulcorner Q \urcorner/\ulcorner P \urcorner\}^{\ulcorner} \mid (S)^{\urcorner}\{\ulcorner Q \urcorner/\ulcorner P \urcorner\}^{\ulcorner} \\ (x(y)\ .\ R)^{\urcorner}\{\ulcorner Q \urcorner/\ulcorner P \urcorner\}^{\ulcorner} & = & (x)\{\ulcorner Q \urcorner/\ulcorner P \urcorner\}(z)\ .\ ((R^{\urcorner}\{z/y\}^{\ulcorner})^{\urcorner}\{\ulcorner Q \urcorner/\ulcorner P \urcorner\}^{\ulcorner}) \\ (x[y])^{\urcorner}\{\ulcorner Q \urcorner/\ulcorner P \urcorner\}^{\ulcorner} & = & (x)\{\ulcorner Q \urcorner/\ulcorner P \urcorner\}[(y)\{\ulcorner Q \urcorner/\ulcorner P \urcorner\}] \\ (x\langle\!|R|\!\rangle)^{\urcorner}\{\ulcorner Q \urcorner/\ulcorner P \urcorner\}^{\ulcorner} & = & (x)\{\ulcorner Q \urcorner/\ulcorner P \urcorner\}\langle\!|R^{\urcorner}\{\ulcorner Q \urcorner/\ulcorner P \urcorner\}^{\ulcorner}|\!\rangle \\ (\urcorner x \ulcorner)^{\urcorner}\{\ulcorner Q \urcorner/\ulcorner P \urcorner\}^{\ulcorner} & = & \begin{cases} Q & x \equiv_N \ulcorner P \urcorner \\ \urcorner x \ulcorner & otherwise \end{cases} \end{array}$$

where

$$(x)\{\ulcorner Q \urcorner/\ulcorner P \urcorner\} = \begin{cases} \ulcorner Q \urcorner & x \equiv_N \ulcorner P \urcorner \\ x & otherwise \end{cases}$$

and z is chosen distinct from $\ulcorner P \urcorner$, $\ulcorner Q \urcorner$, the free names in Q, and all the names in R.

Anticipating something of what's to come, these definitions should make it clear that there is a significant difference between $x\langle\!|P|\!\rangle$ and $x[\ulcorner P \urcorner]$. Specifically, consider the following pair of processes:

$$\begin{array}{rcl} w\langle\!|y[z]|\!\rangle^{\urcorner}\{u/z\}^{\ulcorner} & = & w\langle\!|y[u]|\!\rangle \\ w[\ulcorner y[z] \urcorner]^{\urcorner}\{u/z\}^{\ulcorner} & = & w[\ulcorner y[z] \urcorner] \end{array}$$

Thus, the lift operator will be seen as a way to dynamically construct processes before reifying them as names.

Remark 2.4.1. Note that the substitution for alpha-equivalence, which – by abuse of notation – we write $\mathbf{P}\{\mathbf{x}_0/\mathbf{x}_1\}$, will use name equivalence in its tests, but will not perform the drop. That is, its definition is identical to the one above, except for the following case.

$$(\ulcorner x \urcorner)\{Q/P\} = \begin{cases} \ulcorner Q \urcorner & x \equiv_N \ulcorner P \urcorner \\ \urcorner x \urcorner & otherwise \end{cases}$$

Remark 2.4.2. The astute reader will have noticed that substitution is mutually recursive with $\equiv_N$ and $\equiv$. In particular, to carry out a substitution may require that one check the equivalence of two names. This may require that one check whether the processes they quote are structurally equivalent. This may require that one check that they are alpha-convertible. This, in turn, may require that one carry out a substitution!

This does not present a problem, however, because each time around the circuit the names are getting strictly simpler. Ultimately, this recursion bottoms out in being able to check that 0 is the same as or structurally equivalent to 0. In other words, the mutual recursion is well-founded.

Having observed the mutual recursion, however, it is now convenient to add one more structural equivalence law.

$$\mathbf{P} \equiv \mathbf{P}\{\mathbf{x}_0/\mathbf{x}_1\},\ \textit{if}\ \mathbf{x}_0 \equiv_N \mathbf{x}_1$$

2.5. **Operational Semantics.** The reduction rules for ρ-calculus are

$$\frac{}{\mathbf{x}_0[\mathbf{x}_2] \mid \mathbf{x}_0(\mathbf{x}_1)\ .\ \mathbf{P} \to \mathbf{P}\{\mathbf{x}_2/\mathbf{x}_1\}} \quad \text{(Comm)}$$

$$\frac{}{\mathbf{x}\langle\!|\mathbf{P}|\!\rangle \to \mathbf{x}[\ulcorner \mathbf{P} \urcorner]} \quad \text{(Lift)}$$

In addition, we have the following context rules:

$$\frac{\mathbf{P}_0 \to \mathbf{P}_1}{\mathbf{P}_0 \mid \mathbf{P}_2 \to \mathbf{P}_1 \mid \mathbf{P}_2} \quad \text{(Par)}$$

$$\frac{\mathbf{P}_2 \equiv \mathbf{P}_0 \qquad \mathbf{P}_0 \to \mathbf{P}_1 \qquad \mathbf{P}_1 \equiv \mathbf{P}_3}{\mathbf{P}_2 \to \mathbf{P}_3} \quad \text{(Equiv)}$$

We write $\Rightarrow$ for $\to^*$.

3. Replication

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 ρ-calculus.

$$\begin{aligned} D(x) &\triangleq x(y)\ .\ (x[y] \mid \urcorner y \ulcorner) \\ !P(x) &\triangleq x\langle\!|D(x) \mid P|\!\rangle \mid D(x) \end{aligned}$$

$$
\begin{array}{rc}
& !P(x) \\
\triangleq & x\langle\!(x(y) \,.\, (x[y] \mid \ulcorner y \urcorner)) \mid P \rangle\!\rangle \mid x(y) \,.\, (x[y] \mid \ulcorner y \urcorner) \\
\to & x[\ulcorner (x(y) \,.\, (x[y] \mid \ulcorner y \urcorner)) \mid P \urcorner] \mid x(y) \,.\, (x[y] \mid \ulcorner y \urcorner) \\
\to & (x[y] \mid \ulcorner y \urcorner) \urcorner \{ \ulcorner (x(y) \,.\, (\ulcorner y \urcorner \mid x[y])) \mid P \urcorner / y \} \ulcorner \\
\triangleq & x[\ulcorner (x(y) \,.\, (x[y] \mid \ulcorner y \urcorner)) \mid P \urcorner] \mid (x(y) \,.\, (x[y] \mid \ulcorner y \urcorner)) \mid P \\
\to & \ldots \\
\to & P \mid P \mid \ldots
\end{array}
$$

Of course, this implementation runs away. The standard approach to a more implementable replication operator is to restrict it to input-guarded processes. E.g.,

$$!u(v) \,.\, P \triangleq u(v) \,.\, (x\langle\!(u(v) \,.\, D(x)) \mid P \rangle\!\rangle \mid D(x))$$

4. Bisimulation

Having taken the notion of restriction out of the language, we carefully place it back into the notion of observation, and hence into the notion of program equality, i.e. bisimulation. That is, we parameterize the notion of barbed bisimulation by a set of names over which we are allowed to set the barbs.

The motivation will become clearer when we treat the encoding of the asynchronous π-calculus.

Definition 4.0.1. An *observation relation*, $\downarrow_{\mathcal{N}}$, over a set of names, $\mathcal{N}$, is the smallest relation satisfying the rules below.

$$\frac{y \in \mathcal{N},\ x \equiv_N y}{x[v] \downarrow_{\mathcal{N}} x} \qquad \text{(Out-barb)}$$

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

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

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

Definition 4.0.2. 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 $P \to P'$ 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 barbed bisimulation $\mathcal{S}$.

5. Interpreting π-calculus

Here we provide an encoding of the pure asynchronous π-calculus into the ρ-calculus.

Since all names are global in the ρ-calculus, there is a small complication in the treatment of free names. 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). Another is to provide an environment for the free names, i.e. an injective map, $r : \mathcal{N}_\pi \to \ulcorner Proc \urcorner$, mapping the free names into names in the ρ-calculus. Maintaining the updates to the environment obscures the simplicity of the translation. We adopt a third alternative.

Observe, again, that π-calculus is parameterized in a theory of names. So, we build a π-calculus in which the names are the names of ρ-calculus.

5.1. π-calculus.

More formally,

π-calculus

$$\begin{array}{rcl} \mathbf{P} & ::= & 0 \\ & | & \mathtt{x[x]} \\ & | & \mathtt{x(x)\ .\ P} \\ & | & (\nu\ \mathtt{x})\mathbf{P} \\ & | & \mathbf{P}\ \mathtt{|}\ \mathbf{P} \\ & | & \mathtt{!P} \\ \mathbf{x} & ::= & x \in \ulcorner Proc \urcorner \end{array}$$

5.2. Structural congruence.

Definition 5.2.1. 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:

$$\begin{array}{rcl} (\nu\ x)0 & \equiv & 0, \\ (\nu\ x)(\nu\ x)P & \equiv & (\nu\ x)P, \\ (\nu\ x)(\nu\ y)P & \equiv & (\nu\ x)(\nu\ y)P, \\ P\ \mathtt{|}\ (\nu\ x)Q & \equiv & (\nu\ x)P\ \mathtt{|}\ Q,\ \textit{if}\ x \notin \mathcal{FN}(P) \end{array}$$

(2) the recursion law:

$$\mathtt{!}P \equiv P\ \mathtt{|}\ \mathtt{!}P$$

(3) the name equivalence law:

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

5.3. Operational semantics.

The operational semantics is standard.

$$\frac{}{\mathbf{x}_0\mathtt{[}\mathbf{x}_2\mathtt{]}\ \mathtt{|}\ \mathbf{x}_0\mathtt{(}\mathbf{x}_1\mathtt{)}\ \mathtt{.}\ \mathbf{P} \to \mathbf{P}\{\mathbf{x}_2/\mathbf{x}_1\}} \quad \text{(Comm)}$$

In addition, we have the following context rules:

$$\frac{\mathbf{P}_0 \to \mathbf{P}_1}{\mathbf{P}_0\ \mathtt{|}\ \mathbf{P}_2 \to \mathbf{P}_1\ \mathtt{|}\ \mathbf{P}_2} \quad \text{(Par)}$$

$$\frac{\mathbf{P}_0 \to \mathbf{P}_1}{(\nu\ \mathbf{x})\mathbf{P}_0 \to (\nu\ \mathbf{x})\mathbf{P}_1} \quad \text{(New)}$$

$$\frac{\mathbf{P}_2 \equiv \mathbf{P}_0 \qquad \mathbf{P}_0 \to \mathbf{P}_1 \qquad \mathbf{P}_1 \equiv \mathbf{P}_3}{\mathbf{P}_2 \to \mathbf{P}_3} \quad \text{(Equiv)}$$

We write $\Rightarrow$ for $\to^*$.

5.4. **The translation.** Given a process, P, we pick n and p such that $n \neq p$ and distinct from the free names of P. For example, $n = \ulcorner \Pi_{m \in \mathcal{FN}(P)} m\texttt{[}\ulcorner 0 \urcorner\texttt{]} \urcorner$ and $p = \ulcorner \Pi_{m \in \mathcal{FN}(P)} m\texttt{(}\ulcorner 0 \urcorner\texttt{)}\ .\ 0 \urcorner$. Then

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

where

$$\begin{aligned}
[\![0]\!]_{2nd}(n,p) &= 0 \\
[\![x\texttt{[}y\texttt{]}]\!]_{2nd}(n,p) &= x\texttt{[}y\texttt{]} \\
[\![x\texttt{(}y\texttt{)}\ .\ P]\!]_{2nd}(n,p) &= x\texttt{(}y\texttt{)}\ .\ [\![P]\!]_{2nd}(n,p) \\
[\![P \mid Q]\!]_{2nd}(n,p) &= [\![P]\!]_{2nd}(n^l,p^l) \mid [\![Q]\!]_{2nd}(n^r,p^r) \\
[\![!P]\!]_{2nd}(n,p) &= x\langle\!\langle [\![P]\!]_{3rd}(n^r,p^r) \rangle\!\rangle \mid D(x) \mid n^r\texttt{[}n^l\texttt{]} \mid p^r\texttt{[}p^l\texttt{]} \\
[\![(\nu\ x)P]\!]_{2nd}(n,p) &= p\texttt{(}x\texttt{)}\ .\ [\![P]\!]_{2nd}(n^l,p^l) \mid p\texttt{[}n\texttt{]}
\end{aligned}$$

where

$$\begin{aligned}
x^l &\triangleq \ulcorner x\texttt{[}x\texttt{]} \urcorner \\
x^r &\triangleq \ulcorner x\texttt{(}x\texttt{)}\ .\ 0 \urcorner \\
[\![P]\!]_{3rd}(n'',p'') &\triangleq n''\texttt{(}n\texttt{)}\ .\ p''\texttt{(}p\texttt{)}\ .\ ([\![P]\!]_{2nd}(n,p) \mid (D(x) \mid n''\texttt{[}n^l\texttt{]} \mid p''\texttt{[}p^l\texttt{]}))
\end{aligned}$$

Remark 5.4.1. Note that all ν-binding is now explicit, as in Wischik's global π-calculus. That is, the treatment of $(\nu\ x)P$ is as an input guard.

Remark 5.4.2. 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 ν-ed names. This will not result in different behavior, however, as the bindings will be consistent throughout the translation of the process.

Theorem 5.4.3 (Correctness)**.** $P \dot{\approx}_\pi Q \iff [\![P]\!] \dot{\approx}_{r(\mathcal{FN}(P))} [\![Q]\!]$.

Proof sketch: An easy structural induction.

It is important to note 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'.

Example 5.4.4. In a similar vein consider, for an appropriately chosen p and n we have

$$[\![(\nu\ v)(\nu\ v)u\texttt{[}v\texttt{]}]\!] = p\texttt{(}v\texttt{)}\ .\ ((\ulcorner p\texttt{[}p\texttt{]} \urcorner\texttt{(}v\texttt{)}\ .\ u\texttt{[}v\texttt{]}) \mid (\ulcorner p\texttt{[}p\texttt{]} \urcorner\texttt{[}\ulcorner n\texttt{[}n\texttt{]} \urcorner\texttt{]})) \mid p\texttt{[}n\texttt{]}$$

and

$$[\![(\nu\ v)u[v]]\!] = p(v)\ .\ u[v]\ |\ p[n]$$

Both programs will result in a fresh name bound to v. 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. This is entirely appropriate of a theory aimed at explaining some of the details of practical *implementation*.

5.5. **Higher-order π-calculus.** As noted above, the lift and drop operators of the ρ-calculus effectively give it features of a higher-order calculus without adding process variables. The translation of the higher-order π-calculus is quite similar to the translation for π-calculus.

Of course, the higher-order π-calculus has application and one may wonder how this is accomplished. This is where the susceptibility of lift to substitutions comes in handy. For example, to translate the parallel composition of a process that sends an abstraction, $(v)P$, to a process that receives it and applies it to the values, v we calculate

$$[\![x[(v)P]\ |\ (x(Y)\ .\ Y\langle v\rangle)]\!](z) = (z(v)\ .\ x(\!|[\![P]\!](z')|\!)\ |\ (x(y)\ .\ \ulcorner y\urcorner\ |\ z[[\![v]\!](z'')])$$

where the translation is parameterized in a channel, z, for sending values, and z' and z'' are constructed from z in some manner analogous to what is done with n and p above.

6. Examples

6.1. **Durability.** Every process in the ρ-calculus is equivalent to a *lift-saturated* one, the translation defined on input by

$$[\![x(y)\ .\ P]\!](z) \quad = \quad x(y)\ .\ (z(\!|[\![P]\!](z)|\!)\ |\ z(w)\ .\ \ulcorner w\urcorner)$$

and identity or recursion everywhere else.

Lift saturation is very much like a continuation-passing style interpretation of the ρ-calculus, where the continuation is now reified and may be manipulated by an outer interpreter. For example, a scheduler may be interposed between the lift of P to z and the corresponding evaluating listener. Likewise, a store may be added to the mix.

For example, let $Store(s)$ be a process, that takes storage requests on s and set

$$[\![x(y)\ .\ P]\!](s,z) \quad = \quad x(y)\ .\ (z(\!|[\![P]\!](s,z)|\!)\ |\ z(w)\ .\ \ulcorner w\urcorner\ |\ s[w])$$

Then, $[\![x(y)\ .\ P]\!](s,z)\ |\ Store(s)$ will sequester copies of the continuations of input-guarded processes, in the store. This serves as the basis for an approach to durability; although, of course, the story is much more complex than this. For example, how does one retrieve a continuation of a given shape or description? This question provides a segue into the next section.

6.2. **Complex data.** The key idea behind how complex data, expressed as processes, may be sent and received is to pattern match on the form of the process sent. We achieve this by allowing inputs to be process forms that are used to deconstruct the names supplied by output. Deconstruction is effected by modifying the communication rule to reflect the pattern matching.

ρ-calculus++

$$
\begin{array}{rcl}
\mathbf{P} & ::= & 0 \\
& | & \mathbf{x[x]} \\
& | & \mathbf{x(m)\ .\ P} \\
& | & \mathbf{x}\langle\!|\mathbf{P}|\!\rangle \\
& | & \mathbf{P\ |\ P} \\
& | & \ulcorner\mathbf{x}\urcorner \\
\mathbf{m} & ::= & 0 \\
& | & \mathbf{x[x]} \\
& | & \mathbf{x(x)\ .\ z} \\
& | & \mathbf{x}\langle\!|\mathbf{z}|\!\rangle \\
& | & \mathbf{z\ |\ z} \\
& | & \ulcorner\mathbf{x}\urcorner \\
\mathbf{z} & ::= & \mathbf{x} \\
& | & \ulcorner\mathbf{x}\urcorner \\
\mathbf{x} & ::= & \ulcorner\mathbf{P}\urcorner
\end{array}
$$

$$\frac{\mathbf{x}_0 \equiv_N \mathbf{x}_1 \qquad \mathbf{x}_2 \equiv_N \ulcorner 0 \urcorner}{\mathbf{x}_0[\mathbf{x}_2] \mid \mathbf{x}_1(0).\mathbf{P} \to \mathbf{P}} \quad \text{(Comm-zero)}$$

$$\frac{\begin{array}{c}\mathbf{x}_0 \equiv_N \mathbf{x}_3 \\ \exists \mathbf{P}_1, \mathbf{P}_2. \ulcorner \ulcorner \mathbf{P}_1 \urcorner [\ulcorner \mathbf{P}_2 \urcorner] \urcorner \equiv_N \mathbf{x}_2\end{array}}{\mathbf{x}_0[\mathbf{x}_2] \mid \mathbf{x}_3(\mathbf{x}_1[\mathbf{x}_4]).\mathbf{P} \to \mathbf{P}\{\ulcorner \mathbf{P}_1 \urcorner / \mathbf{x}_1, \ulcorner \mathbf{P}_2 \urcorner / \mathbf{x}_4\}} \quad \text{(Comm-write)}$$

$$\frac{\begin{array}{c}\mathbf{x}_0 \equiv_N \mathbf{x}_3 \\ \exists \mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3. \ulcorner \ulcorner \mathbf{P}_1 \urcorner (\ulcorner \mathbf{P}_2 \urcorner).\mathbf{P}_3 \urcorner \equiv_N \mathbf{x}_2\end{array}}{\mathbf{x}_0[\mathbf{x}_2] \mid \mathbf{x}_3(\mathbf{x}_1(\mathbf{x}_4).\mathbf{x}_5).\mathbf{P} \to \mathbf{P}\{\ulcorner \mathbf{P}_1 \urcorner / \mathbf{x}_1, \ulcorner \mathbf{P}_2 \urcorner / \mathbf{x}_4, \ulcorner \mathbf{P}_3 \urcorner / \mathbf{x}_5\}} \quad \text{(Comm-read)}$$

$$\frac{\begin{array}{c}\mathbf{x}_0 \equiv_N \mathbf{x}_3 \\ \exists \mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3. \ulcorner \ulcorner \mathbf{P}_1 \urcorner (\ulcorner \mathbf{P}_2 \urcorner). \ulcorner \ulcorner \mathbf{P}_3 \urcorner \urcorner \equiv_N \mathbf{x}_2\end{array}}{\mathbf{x}_0[\mathbf{x}_2] \mid \mathbf{x}_3(\mathbf{x}_1(\mathbf{x}_4).\mathbf{x}_5).\mathbf{P} \to \mathbf{P}\{\ulcorner \mathbf{P}_1 \urcorner / \mathbf{x}_1, \ulcorner \mathbf{P}_2 \urcorner / \mathbf{x}_4, \ulcorner \mathbf{P}_3 \urcorner / \mathbf{x}_5\}} \quad \text{(Comm-read-drop)}$$

$$\frac{\begin{array}{c}\mathbf{x}_0 \equiv_N \mathbf{x}_3 \\ \exists \mathbf{P}_1, \mathbf{P}_2. \ulcorner \ulcorner \mathbf{P}_1 \urcorner \langle\!|\mathbf{P}_2|\!\rangle \urcorner \equiv_N \mathbf{x}_2\end{array}}{\mathbf{x}_0[\mathbf{x}_2] \mid \mathbf{x}_3(\mathbf{x}_1\langle\!|\mathbf{x}_4|\!\rangle) \to \mathbf{P}\{\ulcorner \mathbf{P}_1 \urcorner / \mathbf{x}_1, \ulcorner \mathbf{P}_2 \urcorner / \mathbf{x}_4\}} \quad \text{(Comm-lift)}$$

$$\frac{\begin{array}{c}\mathbf{x}_0 \equiv_N \mathbf{x}_3 \\ \exists \mathbf{P}_1, \mathbf{P}_2. \ulcorner \ulcorner \mathbf{P}_1 \urcorner \langle\!| \ulcorner \ulcorner \mathbf{P}_2 \urcorner \urcorner |\!\rangle \urcorner \equiv_N \mathbf{x}_2\end{array}}{\mathbf{x}_0[\mathbf{x}_2] \mid \mathbf{x}_3(\mathbf{x}_1\langle\!|\mathbf{x}_4|\!\rangle) \to \mathbf{P}\{\ulcorner \mathbf{P}_1 \urcorner / \mathbf{x}_1, \ulcorner \mathbf{P}_2 \urcorner / \mathbf{x}_4\}} \quad \text{(Comm-lift-drop)}$$

$$\frac{\begin{array}{c}\mathbf{x}_0 \equiv_N \mathbf{x}_3 \\ \exists \mathbf{P}_1, \mathbf{P}_2. \mathbf{P}_1 \not\equiv 0, \mathbf{P}_2 \not\equiv 0, \ulcorner \mathbf{P}_1 \mid \mathbf{P}_2 \urcorner \equiv_N \mathbf{x}_2\end{array}}{\mathbf{x}_0[\mathbf{x}_2] \mid \mathbf{x}_3(\mathbf{x}_1 \mid \mathbf{x}_4).\mathbf{P} \to \mathbf{P}\{\ulcorner \mathbf{P}_1 \urcorner / \mathbf{x}_1, \ulcorner \mathbf{P}_2 \urcorner / \mathbf{x}_4\}} \quad \text{(Comm-par)}$$

$$\frac{\begin{array}{c}\mathbf{x}_0 \equiv_N \mathbf{x}_3 \\ \exists \mathbf{P}_1, \mathbf{P}_2 . \mathbf{P}_1 \not\equiv 0, \mathbf{P}_2 \not\equiv 0, \ulcorner\urcorner\ulcorner \mathbf{P}_1 \urcorner\ulcorner \mid \urcorner\ulcorner \mathbf{P}_2 \urcorner\ulcorner\urcorner \equiv_N \mathbf{x}_2\end{array}}{\mathbf{x}_0[\mathbf{x}_2] \mid \mathbf{x}_3(\mathbf{x}_1 \mid \mathbf{x}_4).\mathbf{P} \to \mathbf{P}\{\ulcorner \mathbf{P}_1 \urcorner / \mathbf{x}_1, \ulcorner \mathbf{P}_2 \urcorner / \mathbf{x}_4\}} \qquad \text{(Comm-par-drop)}$$

The effect of this change is to make the language lexically reflective. Every syntactic form is now reified and communication is dispatched on the basis of these forms.

Now imagine that our store from the previous example is parameterized on a port for get requests, $Store(s, g)$. While not realistic, we might imagine a protocol in which the client of the store gives it a port to which the store broadcasts the entire store. The client can then pick out those process forms that are of interest using the pattern matching from above.

A slightly more realistic version of this is that the client subscribes not to the entire store, but to a view of the store. Only the view is broadcast to the port on which the client is pattern-matching. An even more realistic version has the client quoting the pattern-matching process and shipping that to the store. The store evaluates the pattern-matching process which ships matches back to the client. We say this latter version is more realistic because it more closely mimics what happens in a relational database system where the query is sent from the client to the server, evaluated at the server and the results sent back.

To recapitulate, the client of the store requests of the store three ports: a port, m, on which to send the pattern-matching process, a port, v, for a view of the data, and a port, r, on which to ship back results to the client. Thus, a client reporting on a channel a the continuations of stored processes that were blocked on input would look like this:

$$client(s, g, c, a) \quad \triangleq \quad g\texttt{[}c\texttt{]} \mid c\texttt{(}m\texttt{(}v\texttt{)} \texttt{.} r\texttt{)} \texttt{.} (m\langle\!|\texttt{!}v\texttt{(}x\texttt{(}y\texttt{)} \texttt{.} z\texttt{)} \texttt{.} r\texttt{[}\ulcorner x\texttt{(}y\texttt{)} \texttt{.} z \urcorner\texttt{]}\rangle\!| \mid \texttt{!}r\texttt{(}x\texttt{(}y\texttt{)} \texttt{.} z\texttt{)} \texttt{.} a\texttt{[}z\texttt{]}\texttt{)}$$

6.3. **Macros.** The reader that has gotten this far will have already seen the basic ingredients of a macro system a la `Scheme` [?]. Quote and eval in the ρ-calculus work very much like quote and eval in `Scheme`. Interestingly, input-guarded lift works much like backquote and comma.

When these features are combined with the pattern-matching-based lexical reflection the system is fairly expressive. The reader is invited to try, as a test of this claim, writing a process that will convert a process into it's lift-saturated form.

7. Conclusions and future work

We have briefly studied a construction for adding reflection to name-passing calculi. The study was conducted in a concurrency setting, applying the construction to the async π-calculus. We were motivated to study this theory by the observation that many practical examples of features that might be desirable in a programming language based on mobile process algebras involve serialization and deserialization of program state, or program source data.

We considered how the asynchronous π-calculus may be embedded into such a theory. Additionally, we touched briefly on the chief features of some examples that indicate this as a promising direction.

We should also mention that there is a wider context in which we view this work. On the theoretical side, we note that Carbone and Maffeis observe a tower of expressiveness resulting from adding very simple structure to names [?]. In some sense, this may be view as approaching the phenomena of structured names 'from below'. By making names be processes, this work may be seen as approaching the same

phenomena 'from above'. But, both investigations are really the beginnings of a much longer and deeper investigation of the relationship between process structure and name structure.

On the applications side, such an investigation may be highly warranted in light of the recent connection between concurrency theory and biology. In particular, despite the interesting results achieved by researchers in this field, there is a fundamental difference between the kind of synchronization observed in the π-calculus and the kind of synchronization observed between molecules at the bio-molecular level. The difference is that interactions in the latter case occur at sites with extension and behavior of their own [**?**]. An account of this phenomenon may be revealed in a detailed study of the relationship between the structure of names and the structure of processes.

Acknowledgments. The authors thank Lucian Wischik for his thoughtful and careful reading of previous drafts of this paper.