Submission to Foundations of Interaction

A reflective higher-order calculus

L.G. MEREDITH AND MATTHIAS RADESTOCK

Abstract. 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, where ancillary mechanisms provide a means of interpreting of names, e.g. as tcp/ip ports. 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.

Here we present such a theory in the form of an asynchronous message-passing calculus built on a notion of quoting. Names are quoted processes, and as such represent the code of a process, a reification of the syntactic structure of the process as an object for process manipulation. Name-passing in this setting becomes a way of passing the code of a process as a message. In the presence of a dequote operation, turning the code of a process into a running instance, this machinery yields higher-order characteristics without the introduction of process variables.

As is standard with higher-order calculi, replication and/or recursion is no longer required as a primitive operation. Somewhat more interestingly, the introduction of a process constructor to dynamically convert a process into its code is essential to obtain computational completeness, and simultaneously supplants the function of the ν operator. In fact, one may give a compositional encoding of the ν operator into a calculus featuring dynamic quote as well as dequote.

1. Introduction

The π-calculus ([10]) 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 [8], where an ancillary binding language providing 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. Behind this question lurk a whole host of other exciting and potentially enlightening questions regarding the role of names with structure in calculi of interaction and the relationship between the structure of names and the structure of processes.

Speaking provocatively, nowhere in the tools available to the computer scientist is there a countably infinite set of *atomic* entities. All such sets, e.g. the natural numbers, the set of strings of finite length

Key words and phrases. concurrency, message-passing, process calculus, reflection.

on some alphabet, etc., are *generated* from a finite presentation, and as such the elements of these sets inherit *structure* from the generating procedure. As a theoretician focusing on some aspects of the theory of processes built from such a set, one may temporarily forget that structure, but it is there nonetheless, and comes to the fore the moment one tries to build *executable* models of these calculi.

To illustrate the point, when names have structure, name equality becomes a computation. But, if our theory of interaction is to provide a basis for a theory of computation, then certainly this computation must be accounted for as well. Moreover, the fact that any realization of these name-based, mobile calculi of interaction must come to grips with names that have structure begs the question: would the theoretical account of interaction be more effective, both as a theory in its own right and as a guide for implementation, if it included an account of the relationships between the structure of names and the structure of processes?

1.1. **Overview and contributions.** Here we present a theory of an asynchronous message-passing calculus built on a notion of quoting. Names are quoted processes, and as such represent the code of a process, a reification of the syntactic structure of the process (up to some equivalence). Name-passing, then becomes a way of passing the code of a process as a message. In the presence of a dequote operation, turning the code of a process into a running instance, this machinery yields higher-order characteristics without the introduction of process variables.

As is standard with higher-order calculi, replication and/or recursion is no longer required as a primitive operation. Somewhat more interestingly, the introduction of a process constructor to dynamically convert a process into its code is essential to obtain computational completeness, and simultaneously supplants the function of the ν operator. In fact, we give a compositional encoding of the ν operator into the calculus, making essential use of dynamic quote as well as dequote. Following the tradition started by Smith, et al, [3] we dub this ability to turn running code into data and back again, reflection; hence, the name *r*eflective, *h*igher-*o*rder calculus, or rho-calculus, for short, or ρ-calculus for even shorter.

Certainly, the paper presents a concrete calculus that may be used to model a variety of computations and highlights a number of interesting phenemona in those computations. We take the view, however, that the main contribution is that the calculus provides an instrument to bring to life a set of questions regarding the role of names in calculi of interaction. These questions include the calculation of name equality as a computation to be considered within the framework of interaction and the roles of name equality in substitution versus synchronization. These questions don't really come to life, though, without the instrument in hand. So, we turn immediately to the presentation of the calculus.

2. The calculus

2.0.1. *Notation.* We let P, Q, R range over processes and x, y, z range over names.

ρ-calculus	P, Q	$::=$	0	null process
		$\mid$	$x(y) \,.\, P$	input
		$\mid$	$x\langle\!\lvert P \rvert\!\rangle$	lift
		$\mid$	$\urcorner x \ulcorner$	drop
		$\mid$	$P \mid Q$	parallel
	x, y	$::=$	$\ulcorner P \urcorner$	quote

2.0.2. *Quote.* Working in a bottom-up fashion, we begin with names. The technical detail corresponding to the π-calculus' parametricity in a theory of *names* shows up in standard presentations in the grammar describing terms of the language: there is no production for names; names are taken to be terminals in the grammar. Our first point of departure from a more standard presentation of an asynchronous mobile process calculus is here. The grammar for the terms of the language will include a production for names in the grammar. A name is a *quoted* process, $\ulcorner P \urcorner$.

2.0.3. *Parallel.* This constructor is the usual parallel composition, denoting concurrent execution of the composed processes.

2.0.4. *Lift and drop.* Despite the fact that names are built from (the codes of) processes, we still maintain a careful disinction in kind between process and name; thus, name construction is not process construction. So, if one wants to be able to generate a name from a given process, there must be a process constructor for a term that creates a name from a process. This is the motivation for the production $x\langle\!|P|\!\rangle$, dubbed here the *lift* operator. The intuitive meaning of this term is that the process P will be packaged up as its code, $\ulcorner P \urcorner$, and ultimately made available as an output at the port x.

A more formal motivation for the introduction of this operator will become clear in the sequel. But, it will suffice to say now that $\ulcorner P \urcorner$ is impervious to substitution. In the ρ-calculus, substitution does not affect the process body between quote marks. On the other hand, $x\langle\!|P|\!\rangle$ is susceptible to substitution and as such constitutes a dynamic form of quoting because the process body ultimately quoted will be different depending on the context in which the $x\langle\!|P|\!\rangle$ expression occurs.

Of course, when a name is a quoted process, it is very handy to have a way of evaluating such an entity. Thus, the $\urcorner x \ulcorner$ operator, pronounced *drop* x, (eventually) extracts the process from a name. We say 'eventually' because this extraction only happens when a quoted process is substituted into this expression. A consequence of this behavior is that $\urcorner x \ulcorner$ is inert except under and input prefix. One way of saying this is that if you want to get something done, sometimes you need to drop a name, but it should be the name of an agent you know.

Remark 2.0.1. The lift operator turns out to play a role analogous to $(\nu\ x)P$. As mentioned in the introduction, it is essential to the computational completeness of the calculus, playing a key role in the implementation of replication. It also provides an essential ingredient in the compositional encoding of the ν operator.

Remark 2.0.2. It is well-known that replication is not required in a higher-order process algebra [12]. 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 below an encoding 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. The reader familiar with the λ-calculus will note the formal similarity between the crucial term in the encoding and the paradoxical combinator [1].

2.0.5. *Input and output.* The input constructor is standard for an asynchronous name-passing calculus. Input blocks its continuation from execution until it receives a communication. Lift is a form of output which – because the calculus is asynchronous – is allowed no continuation. It also affords a convenient syntactic sugar, which we define here.

$$x\,[y] \triangleq x\langle\!|\urcorner y \ulcorner|\!\rangle$$

2.0.6. *The null process.* As we will see below, the null process has a more distinguished role in this calculus. It provides the sole atom out of which all other processes (and the names they use) arise much in the same way that the number 0 is the sole number out of which the natural numbers are constructed; or the empty set is the sole set out of which all sets are built in ZF-set theory [7]; or the empty game is the sole game out of which all games are built in Conway's theory of games and numbers [2]. This analogy to these other theories draws attention, in our opinion, to the foundational issues raised in the introduction regarding the design of calculi of interaction.

2.1. **The name game.** Before presenting some of the more standard features of a mobile process calculus, the calculation of free names, structural equivalence, etc., we wish to consider some examples of processes and names. In particular, if processes are built out of names, and names are built out of processes, is it ever possible to get off the ground? Fortunately, there is one process the construction of which involves no names, the null process, 0. Since we have at least one process, we can construct at least one name, namely $\ulcorner 0 \urcorner$ [1]. Armed with one name we can now construct at least two new processes that are evidently syntactically different from the 0, these are $\ulcorner 0 \urcorner \mathtt{[} \ulcorner 0 \urcorner \mathtt{]}$ and $\ulcorner 0 \urcorner \mathtt{(} \ulcorner 0 \urcorner \mathtt{)} \,.\, 0$. As we might expect, the intuitive operational interpretation of these processes is also distinct from the null process. Intuitively, we expect that the first outputs the name $\ulcorner 0 \urcorner$ on the channel $\ulcorner 0 \urcorner$, much like the ordinary π-calculus process $x\mathtt{[}x\mathtt{]}$ outputs the name x on the channel x, and the second inputs on the channel $\ulcorner 0 \urcorner$, much like the ordinary π-calculus process $x\mathtt{(}x\mathtt{)} \,.\, 0$ inputs on the channel x.

Of course, now that we have two more processes, we have two more names, $\ulcorner \ulcorner 0 \urcorner \mathtt{[} \ulcorner 0 \urcorner \mathtt{]} \urcorner$ and $\ulcorner \ulcorner 0 \urcorner \mathtt{(} \ulcorner 0 \urcorner \mathtt{)} \,.\, 0 \urcorner$. Having three names at our disposal we can construct a whole new supply of processes that generate a fresh supply of names, and we're off and running. It should be pointed out, though, that as soon as we had the null process we also had $0 \mid 0$ and $0 \mid 0 \mid 0$ and consequently, we had the names $\ulcorner 0 \mid 0 \urcorner$, and $\ulcorner 0 \mid 0 \mid 0 \urcorner$, and .... But, since we ultimately wish to treat these compositions as merely other ways of writing the null process and not distinct from it, should we admit the codes of these processes as distinct from $\ulcorner 0 \urcorner$?

This question leads to several intriguing and apparently fundamental questions. Firstly, if names have structure, whether this derives from the structure of processes or something else, what is a reasonable notion of equality on names? How much computation, and of what kind, should go into ascertaining equality on names? Additionally, what roles should name equality play in a calculus of processes? In constructing this calculus we became conscious that substitution and synchronization identify two potentially very different roles for name equality to play in name-passing calculi. That these are very different roles is suggested by the fact that they may be carried out by very different mechanisms in a workable and effective theory. We offer one choice, but this is just one design choice among infinitely many. Most likely, the primary value of this proposal is to raise the question. Likewise, we offer a proposal regarding the calculation of name equality that is just one of many and whose real purpose is to make the question vivid. We wish to turn to the core mechanics of the calculus with these questions in mind.

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

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

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 $\mathcal{N}(P)$.

[1] pun gratefully accepted ;-)

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

$$\begin{array}{rcl} P \mid 0 & \equiv P \equiv & 0 \mid P \\ P \mid Q & \equiv & Q \mid P \\ (P \mid Q) \mid R & \equiv & P \mid (Q \mid R) \end{array}$$

2.4. **Name equivalence.** We now come to one of the first real subtleties of this calculus. Both the calculation of the free names of a process and the determination of structural congruence between processes critically depend on being able to establish whether two names are equal. In the case of the calculation of the free names of an input-guarded process, for example, to remove the bound name we must determine whether it is in the set of free names of the continuation. Likewise, structural congruence includes α-equivalence. But, establishing α-equivalence between the processes $x\texttt{(}z\texttt{)}\,.\,w\langle\!|y\texttt{[}z\texttt{]}|\!\rangle$ and $x\texttt{(}v\texttt{)}\,.\,w\langle\!|y\texttt{[}v\texttt{]}|\!\rangle$, for instance, requires calculating a substitution, e.g. $x\texttt{(}v\texttt{)}\,.\,w\langle\!|y\texttt{[}v\texttt{]}|\!\rangle\{z/v\}$. But this calculation requires, in turn, being able to determine whether two names, in this case the name in the object position of the output, and the name being substituted for, are equal.

As will be seen, the equality on names involves structural equivalence on processes, which in turn involves alpha equivalence, which involves name equivalence. This is a subtle mutual recursion, but one that turns out to be well-founded. Before presenting the technical details, the reader may note that the grammar above enforces a strict alternation between quotes and process constructors. Each question about a process that involves a question about names may in turn involve a question about processes, but the names in the processes the next level down, as it were, are under fewer quotes. To put it another way, each 'recursive call' to name equivalence will involve one less level of quoting, ultimately bottoming out in the quoted zero process.

Let us assume that we have an account of (syntactic) substitution and α-equivalence upon which we can rely to formulate a notion of name equivalence, and then bootstrap our notions of substitution and α-equivalence from that. We take name equivalence, written $\equiv_N$, to be the smallest equivalence relation generated by the following rules.

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

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

2.5. **Syntactic substitution.** Now we build the substitution used by α-equivalence. We use $Proc$ for the set of processes, $\ulcorner Proc\urcorner$ for the set of names, and $\{\vec{y}/\vec{x}\}$ to denote partial maps, $s : \ulcorner Proc\urcorner \to \ulcorner Proc\urcorner$. A map, s lifts, uniquely, to a map on process terms, $\widehat{s} : Proc \to Proc$ by the following equations.

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

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. Our α-equivalence will be built in the standard way from this substitution.

But, given these mutual recursions, the question is whether the calculation of $\equiv_N$ (respectively, $\equiv$, $\equiv_\alpha$) terminates. To answer this question it suffices to formalize our intuitions regarding level of quotes, or quote depth, $\#(x)$, of a name x as follows.

$$\begin{aligned} \#(\ulcorner P \urcorner) &= 1 + \#(P) \\ \#(P) &= \begin{cases} max\{\#(x) : x \in \mathcal{N}(P)\} & \mathcal{N}(P) \neq \emptyset \\ 0 & otherwise \end{cases} \end{aligned}$$

The grammar ensures that $\#(\ulcorner P \urcorner)$ is bounded. Then the termination of $\equiv_N$ (respectively, $\equiv$, $\equiv_\alpha$) is an easy induction on quote depth.

2.6. **Dynamic quote: an example.** Anticipating something of what's to come, consider applying the substitution, $\widehat{\{u/z\}}$, to the following pair of processes, $w\langle\!|y\,\mathtt{[}z\mathtt{]}\,|\!\rangle$ and $w\,\mathtt{[}\ulcorner y\,\mathtt{[}z\mathtt{]}\urcorner\mathtt{]}$.

$$\begin{aligned} w\langle\!|y\,\mathtt{[}z\mathtt{]}\,|\!\rangle\widehat{\{u/z\}} &= w\langle\!|y\,\mathtt{[}u\mathtt{]}\,|\!\rangle \\ w\,\mathtt{[}\ulcorner y\,\mathtt{[}z\mathtt{]}\urcorner\mathtt{]}\widehat{\{u/z\}} &= w\,\mathtt{[}\ulcorner y\,\mathtt{[}z\mathtt{]}\urcorner\mathtt{]} \end{aligned}$$

Because the body of the process between quotes is impervious to substitution, we get radically different answers. In fact, by examining the first process in an input context, e.g. $x\,\mathtt{(}z\mathtt{)}\,\mathtt{.}\,w\langle\!|y\,\mathtt{[}z\mathtt{]}\,|\!\rangle$, we see that the process under the lift operator may be shaped by prefixed inputs binding a name inside it. In this sense, the lift operator will be seen as a way to dynamically construct processes before reifying them as names.

2.7. **Semantic substitution.** The substitution used in α-equivalence is really only a device to formally recognize that binding occurrences do not depend on the specific names. It is not the engine of computation. The proposal here is that while synchronization is the driver of that engine, the real engine of computation is a semantic notion of substitution that recognizes that a dropped name is a request to run a process. Which process? Why the one whose code has been bound to the name being dropped. Formally, this amounts to a notion of substitution that differs from syntactic substitution in its application to a dropped name.

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

In the remainder of the paper we will refer to semantic and syntactic substitutions simply as substitutions and rely on context to distinguish which is meant. Similarly, we will abuse notation and write $\{y/x\}$ for $\widehat{\{y/x\}}$.

Finally equipped with these standard features we can present the dynamics of the calculus.

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

$$\frac{x_0 \equiv_N x_1}{x_0 \langle\!|Q|\!\rangle \mid x_1(y)\,.\,P \rightarrow P\{\ulcorner Q \urcorner / y\}} \qquad \text{(Comm)}$$

In addition, we have the following context rules:

$$\frac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q} \qquad \text{(Par)}$$

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

The context rules are entirely standard and we do not say much about them, here. The communication rule does what was promised, namely make it possible for agents to synchronize and communicate processes packaged as names. For example, using the comm rule and name equivalence we can now justify our syntactic sugar for output.

$$\begin{array}{cc} & x[z] \mid x(y)\,.\,P \\ = & x\langle\!|\urcorner z \ulcorner|\!\rangle \mid x(y)\,.\,P \\ \rightarrow & P\{\ulcorner\urcorner z \ulcorner\urcorner / y\} \\ \equiv & P\{z/y\} \end{array}$$

But, it also provides a scheme that identifies the role of name equality in synchronization. There are other relationships between names with structure that could also mediate synchronization. Consider, for example, a calculus identical to the one presented above, but with an alternative rule governing communication.

$$\frac{\forall R.[P_{channel} \mid Q_{channel} \rightarrow^{*} R] \Rightarrow R \rightarrow^{*} 0}{\ulcorner Q_{channel} \urcorner \langle\!|Q|\!\rangle \mid \ulcorner P_{channel} \urcorner (y)\,.\,P \rightarrow P\{\ulcorner Q \urcorner / y\}} \qquad \text{(Comm-annihilation)}$$

Intuitively, it says that the codes of a pair of processes, $P_{channel}$, $Q_{channel}$, stand in channel/co-channel relation just when the composition of the processes always eventually reduces to 0, that is, when the processes annihilate one another. This rule is well-founded, for observe that because $0 \equiv 0 \mid 0$, $0 \mid 0 \rightarrow^{*} 0$. Thus, $\ulcorner 0 \urcorner$ serves as its own co-channel. Analogous to our generation of names from 0, with one such channel/co-channel pair, we can find many such pairs. What we wish to point out about this rule is that we can see precisely an account of the calculation of the channel/co-channel relationship as deriving from the theory of interaction. We do not know if the computation of name equality has a similar presentation, driving home the potential difference of those two roles in calculi of interaction.

We mention, as a brief aside, that there is no reason why 0 is special in the scheme above. We posit a family of calculi, indexed by a set of processes $\{S_\alpha\}$, and differing only in their communication rule each of which conforms to the scheme below.

$$\frac{\forall R.[P_{channel} \mid Q_{channel} \rightarrow^{*} R] \Rightarrow R \rightarrow^{*} R' \equiv S_\alpha}{\ulcorner Q_{channel} \urcorner \langle\!|Q|\!\rangle \mid \ulcorner P_{channel} \urcorner (y)\,.\,P \rightarrow P\{\ulcorner Q \urcorner / y\}} \qquad \text{(Comm-annihilation-S)}$$

We explore this family of calculi in a forthcoming paper. For the rest of this paper, however, we restrict our attention to the calculus with the less exotic communication rule, using $\rightarrow$ for reduction according to that system and $\Rightarrow$ for $\rightarrow^*$.

3. Replication

As mentioned before, it is known that replication (and hence recursion) can be implemented in a higher-order process algebra [12]. As our first example of calculation with the machinery thus far presented we give the construction explicitly in the ρ-calculus.

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

$$\begin{array}{lc} !P(x) & \\ = & x\langle\!|(x(y) \,.\, (x[y] \mid \ulcorner y \urcorner)) \mid P|\!\rangle \mid x(y) \,.\, (x[y] \mid \ulcorner y \urcorner) \\ \rightarrow & (x[y] \mid \ulcorner y \urcorner)\{\ulcorner (x(y) \,.\, (\ulcorner y \urcorner \mid x[y])) \mid P \urcorner / y\} \\ = & x[\ulcorner (x(y) \,.\, (x[y] \mid \ulcorner y \urcorner)) \mid P \urcorner] \mid (x(y) \,.\, (x[y] \mid \ulcorner y \urcorner)) \mid P \\ \rightarrow & \ldots \\ \rightarrow^* & P \mid P \mid \ldots \end{array}$$

Of course, this encoding, as an implementation, runs away, unfolding $!P$ eagerly. A lazier and more implementable replication operator, restricted to input-guarded processes, may be obtained as follows.

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

It is worth noting that the lift operator is essential to get computational completeness. A similar calculus equipped with only a static quote enjoys a computational expressiveness at least equivalent to context-free grammars, but short of context-sensitive. This fact is established and exploited in a forthcoming paper on a type system for the ρ-calculus.

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 for this choice is really comparison with other calculi. The set of names of the ρ-calculus is *global*. It is impossible, in the grammar of processes, to guard terms from being placed into contexts that can potentially observe communication. So, we provide a place for reasoning about such limitations on the scope of observation in the theory of bisimulation.

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 $P \rightarrow P'$ 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 $\mathcal{S}_{\mathcal{N}}$.

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, 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). This alternative feels inelegant. Another is to provide an environment, $r : \mathcal{N}_\pi \rightarrow \ulcorner Proc \urcorner$, for mapping the free names in a π-calculus process into names in the ρ-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 the π-calculus is parameterized in a theory of names, we build a π-calculus in which the names are the names of ρ-calculus. This is no different than building a π-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.

5.1. π-calculus.

More formally,

$$\begin{array}{llcl}
\pi\text{-calculus} & P, Q & ::= & 0 \\
 & & \mid & x[y] \\
 & & \mid & x(y) \,.\, P \\
 & & \mid & (\nu\, x) P \\
 & & \mid & P \mid Q \\
 & & \mid & !P \\
 & x, y & ::= & x \in \ulcorner Proc \urcorner
\end{array}$$

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:

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

(2) the recursion law:

$$!P \equiv P \mid !P$$

(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.

$$\frac{}{x[z] \mid x(y)\, .\, P \rightarrow P\{z/y\}} \qquad \text{(COMM)}$$

In addition, we have the following context rules:

$$\frac{P \rightarrow P'}{P \mid Q \rightarrow P' \mid Q} \qquad \text{(PAR)}$$

$$\frac{P \rightarrow P'}{(\nu\, x)P \rightarrow (\nu\, x)P'} \qquad \text{(NEW)}$$

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

Again, we write $\Rightarrow$ for $\rightarrow^*$, and rely on context to distinguish when $\rightarrow$ 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 $Proc_\pi$.

5.4. **The translation.** The translation will be given by a function, $[\![-]\!](-,-) : Proc_\pi \times \ulcorner Proc \urcorner \times \ulcorner Proc \urcorner \rightarrow Proc$. 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, $n = \ulcorner \Pi_{m \in \mathcal{FN}(P)} m[\ulcorner 0 \urcorner] \urcorner$ and $p = \ulcorner \Pi_{m \in \mathcal{FN}(P)} m(\ulcorner 0 \urcorner)\, .\, 0 \urcorner$. Then

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

where

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

and

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

Remark 5.4.1. Note that all ν-binding is now interpreted, as in Wischik's global π-calculus, as an input guard [15].

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

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 $\mathcal{N}$ for the bisimulation. This is one of the technical motivations behind our introduction of a less standard bisimulation.

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

$$[\![(\nu\; v)\,(\nu\; v)\,u\,[v]\,]\!] = p(v)\,.\,((\ulcorner p\,[p]\,\urcorner(v)\,.\,u\,[v]\,) \mid (\ulcorner p\,[p]\,\urcorner[\ulcorner n\,[n]\,\urcorner])) \mid p\,[n]$$

and

$$[\![(\nu\; v)\,u\,[v]\,]\!] = p(v)\,.\,u\,[v] \mid p\,[n]$$

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 $(\nu\; x)\,(\nu\; x)\,P \equiv (\nu\; x)\,P$.

Implementations of the π-calculus, however, having the property that $(\nu\; x)\,P$ 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 ν-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 $(\nu\ x)P$, 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.

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 [13], [14]. 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

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

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.

More generally, one may seek to understand the trade-offs between a presentation of higher-order capability in the higher-order π-calculus and the ρ-calculus. A detailed study is a subject worthy of an entire paper, but at a high level of description one may note that the same argument levied with the ordinary π-calculus applies here: the higher-order π-calculus does not *offer* a theory of names, but rather *depends* upon one being provided. An investigator interested in the higher-order π-calculus as an executable language must still address computation on names, such as calculating name equality in substitution or synchronization, outside of the framework of the theory. Additionally, the higher-order π-calculus has a larger inventory of moving parts: process variables, for sending and receiving processes, as well as names. On both counts the ρ-calculus is more minimalist, needing neither a theory of names, nor the machinery of process variables. On the other hand, minimalism does not always align with ease of use. Experience shows that when writing specifications in the ρ-calculus of any reasonable size one quickly adopts conventions that make the calculus resemble a more traditional higher-order calculus.

6. Conclusions and future work

We studied an asynchronous message-passing calculus built out of a notion of quote. We showed that the calculus provides a workable, effective theory of computation capable of encoding the π-calculus with a compositional account of the ν-operator, as well as the higher-order π-calculus. These encodings bring to light interesting computational phenomena that implementations of the π-calculus have had to face. Additionally, the development of the calculus highlights several intriguiging aspects of the relationships between the structure of processes and the structure of names.

We note that this work is situated in the larger context of a growing investigation into naming and computation. Milner's studies of action calculi led not only to reflexive action calculi [11], but to Power's and Hermida's work on name-free accounts of action calculi [6]. Somewhat farther afield, but still related, is Gabbay's theory of freshness [5]. Very close to the mark, Carbone and Maffeis observe a tower of expressiveness resulting from adding very simple structure to names [9]. In some sense, this may be viewed 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.

Beyond foundational questions concerning the theory of interaction, 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 [4]. An account of these kinds of phenomena may be revealed in a detailed study of the relationship between the structure of names and the structure of processes.

Acknowledgments. The authors wish to thank Robin Milner for his thoughtful and stimulating remarks regarding earlier work in this direction.

References

1. Hendrik Pieter Barendregt, *The Lambda Calculus – Its Syntax and Semantics*, Studies in Logic and the Foundations of Mathematics, vol. 103, North-Holland, 1984.
2. John Horton Conway, *On Numbers and Games*, Academic Press, 1976.
3. J. des Rivieres and B. C. Smith, *The implementation of procedurally reflective languages*, ACM Symposium on Lisp and Functional Programming, 1984, pp. 331–347.
4. Walter Fontana, *private conversation*, 2004.
5. M. J. Gabbay, *The π-calculus in FM*, Thirty-five years of Automath (Fairouz Kamareddine, ed.), Kluwer, 2003.
6. Claudio Hermida and John Power, *Fibrational control structures.*, CONCUR, 1995, pp. 117–129.
7. Jean-Louis Krivine, *The curry-howard correspondence in set theory*, Proceedings of the Fifteenth Annual IEEE Symp. on Logic in Computer Science, LICS 2000 (Martin Abadi, ed.), IEEE Computer Society Press, June 2000.
8. Microsoft Corporation, *Microsoft biztalk server*, microsoft.com/biztalk/default.asp.
9. M.Carbone and S.Maffeis, *On the expressive power of polyadic synchronisation in pi-calculus*, Nordic Journal of Computing **10(2)** (2003), 70–98.
10. Robin Milner, *The polyadic π-calculus: A tutorial*, Logic and Algebra of Specification **Springer-Verlag** (1993).
11. ———, *Strong normalisation in higher-order action calculi.*, TACS, 1997, pp. 1–19.
12. David Sangiorgi and David Walker, *The π-calculus: A theory of mobile processes*, Cambridge University Press, 2001.
13. Davide Sangiorgi, *Bisimulation in higher-order process calculi*, Information and Computation **131** (1996), 141–178.
14. B. Thomsen, *A Theory of Higher Order Communication Systems*, Information and Computation **116** (1995), no. 1, 38–57.
15. Lucian Wischik, *Old names for nu*, Submitted for publication, 2004.