Multiagent Systems and Reflection

Lucius Gregory Meredith[1]
and Ben Goertzel[2]

[1] CEO, F1R3FLY.io 9336 California Ave SW, Seattle, WA 98103, USA
f1r3fly.ceo.com
[2] CEO, SingularityNet
ben@singularitynet.io

Abstract. We make the case for a reflective multi-agent formalism.

1 Multi-agent System and AI

Multi-agent systems are at least as old as AI, and have enjoyed active research in a number of fields. Among the various research programmes, concurrency theory has produced one of the most advanced versions of mult-agent systems. More specifically, the mobile process calculi formalize a notion of multi-agent systems that is simultaneously a foundation for computing, in general, and has been used in a wide range of successful applications. Surprisingly, the AI community has not adopted these formalisms, despite the fact that they enjoy features for reasoning about and executing multi-agent specifications not found in any other framework. Notable among these features are

- an effective notion of equality of ensembles of agents (bisimulation) that comes with powerful proof techniques;
- a notion of spatial and behavioral types allowing for exceptionally powerful and fine-grained classification of the structure and behavior of ensembles of agents;
- a theorem relating the notion of equality to classification, affording a Liskov-style substitution principle saying when an agent or ensemble of agents may be substituted in for another such without change to the behavior of the overall ensemble in which the substitution was effected.

We present a formalism for multi-agent systems sporting these features while also enjoying the kind of computational reflection Brian Smith argued for in 3-Lisp. This combination of powers allows for agents to reason effectively about ensembles of agents, that is to build internal models of the society of minds they observe in their environment. In fact, we will argue that these features constitute the smallest formalism necessary to develop a plausible theory of mind.

2 Concurrent process calculi and spatial logics

In the last thirty years the process calculi have matured into a remarkably powerful analytic tool for reasoning about concurrent and distributed systems. Process-calculus-based algebraic specification of processes began with Milner's Calculus for Communicating Systems (CCS) [19] and Hoare's Communicating Sequential Processes (CSP) [12], and continue through the development of the so-called mobile process calculi, e.g. Milner, Parrow and Walker's π-calculus [22], [23], Cardelli and Caires's spatial logic [4] [6] [7], or Meredith and Radestock's reflective calculi [17] [16]. The process-calculus-based algebraic specification of processes has expanded its scope of applicability to include the specification, analysis, simulation and execution of processes in domains such as:

- telecommunications, networking, security and application level protocols [1] [2] [13] [14];
- programming language semantics and design [13] [11] [10] [30];
- webservices [13] [14] [15];
- blockchain [18]
- and biological systems [8] [9] [25] [24].

Among the many reasons for the continued success of this approach are three central points.

Compositionality First, the process algebras provide a compositional approach to the specification, analysis and execution of concurrent and distributed systems. Owing to Milner's original insights into computation as interaction [20], the process calculi are so organized that the behavior —the semantics— of a system may be composed from the behavior of its components. This means that specifications can be constructed in terms of components —without a global view of the system— and assembled into increasingly complete descriptions.

Bisimulation The second central point is that process algebras have a potent proof principle, yielding a wide range of effective and novel proof techniques [28] [26] [27]. In particular, *bisimulation* encapsulates an effective notion of process equivalence that has been used in applications as far-ranging as algorithmic games semantics [3] and the construction of model-checkers [5]. The essential notion can be stated in an intuitively recursive formulation: a *bisimulation* between two processes P and Q is an equivalence relation E relating P and Q such that: whatever action of P can be observed, taking it to a new state P', can be observed of Q, taking it to a new state Q', such that P' is related to Q' by E and vice versa. P and Q are *bisimilar* if there is some bisimulation relating them. Part of what makes this notion so robust and widely applicable is that it is parameterized in the actions observable of processes P and Q, thus providing a framework for a broad range of equivalences and up-to techniques [28] all governed by the same core principle [27].

Spatial and behavioral logics The third central point is that process calculi have ushered in a new way of thinking about type systems and typing. Beginning with the Hennessy-Milner logics, the process calculi have joined Kripke's many world's interpretation of modal logic with the state transition behavior of processes to deliver a power tool for classifying and proving properties about ensembles of agents. Continuing from there the spatial logics have expanded the purview of typing to see into the structure of ensembles of agents, allowing for the classification of these ensembles in terms of the structure they enjoy. This extends to being able to determine which subcommunities are privy to which information, which has been widely employed in establishing security properties of protocols.

Mobility Another important feature of the mobile process calculi is that the concurrency model is explicitly mobile, meaning agents can discover each other. In other words, the communication topology (who knows whom and who is talking to whom) is evolving. This model is very different from a model where computational elements are soldered together like components on a motherboard. Mobile concurrency is more like the Internet or the telephony networks where people who just met for the first time learn each other's websites, email addresses, and phone numbers.

2.1 Comparison to other formalisms

In the context of AGI it is worth a brief review of what singles the mobile process calculi out from other formalisms, notably the λ-calculus. The key point is driven home when we consider multi-agent systems, and in particular, multi-agent systems of agents that are themselves compositions of agents. These

λ-calculus The λ-calculus is sequential (Berry's theorem) and confluent. Neither of these facts lend it to modeling autonomous agents. Autonomy of execution has to be simulated in the λ-calculus, using techniques, like co-routines, and continuations. In the tower of encoding necessary to simulate autonomous execution it becomes difficult to sort out what is encoding and what is important agent behavior. Specifically, the encodings unavoidably leak into the types of agent behavior as the Morgan Stanley team recently reported. Likewise, the most basic property of multi-agent systems is that they are not confluent. First come, first serve is not just a property of airline reservation systems, but of biological systems at all scales. First come, first serve is fundamentally not confluent: it is a different state if Alice gets the last seat on the plane than if Bob does. Again, this makes the λ-calculus far from ideal to represent systems of autonomously executing agents.
Of course, these same comments apply to Turing machines as originally conceived. In fact, they apply generally to formalisms that do not have concurrent composition as a first class operation. Composition is the operative word. Petri nets, for example, are concurrent, but that concurrency is not expressed as a

composition of nets, which makes them awkward for modeling compositions of agents, and especially agents which are themselves compositions of agents.
Compositionality, or the lack of it, also applies to continuous classical formalisms, such as differential equations. For example, given a chemical solution in beaker A and a different chemical solution in beaker B, the stoichiometric differential models cannot combined to produce a model of pouring the contents of A and B into a third beaker. However, the stochastic π-calculus models, say $[\![A]\!]$ and $[\![B]\!]$ can be combined via parallel composition, $[\![A]\!]|[\![B]\!]$ to produce a model of the contents of the third beaker.
In what follows below, we argue that adding reflection to the primitives of the mobile process calculi yields a model specifically suited for AI. In particular, the model is the smallest such that accounts for a theory of mind.

3 Consciousness as the colonization of the brain

Joscha Bach recently argued that mobile concurrency is at odds with artificial general intelligence.[3] Joscha's argument is that brains are only plastic, i.e. the connections between neurons are only changing during learning, but not during general computation. Here we argue that this position assumes that the mind is not hosted in a logical model of computation that runs on the brain's hardware. After all, the Java virtual machine (JVM) is a very different model of computation than the hardware it runs on. Haskell's model of computation is an even more dramatic variation on the idea of computation than the one embodied in the hardware the glorious Haskell compiler (GHC) is typically hosted on. Why wouldn't the mind be organized like this? To use Joscha's graphic metaphor, why wouldn't the mind arise as a colonizing computational model that hosts itself on the brain's hardware.

In particular, we present a model of multi-agent systems, the rho-calculus, that does provide direct support for mobile concurrent computation. The communication topology amongst a society of processes executing in the rho-calculus is dynamic. Who knows whom and can talk to whom in this society changes over the course of the computation. This way of thinking about the rho-calculus foreshadows the argument presented here that there are very good reasons to suppose that a model like the rho-calculus may have hosted itself on and colonized the hardware of the human brain, in fact any brain that supports a theory of mind.

4 Code, data, and computation

To make this argument i want to make some distinctions that not every computer scientist, let alone every developer makes. i make a distinction between code, data, and computation. Arbitrary code can be treated as some data which is an instance of some type of data structure in which the model of computation is expressed, or hosted. For example, you can host a Turing-complete computational model, like Haskell, in a term language expressed via a context-free grammar, e.g. the grammar for well formed Haskell programs. Yet, we know that context-free grammars are not Turing-complete. How can this be? How can something provably less expressive than Turing-complete models represent Turing-complete computation?

It cuts to the heart of the distinction between syntax and semantics. The grammar of the term language expresses the *syntax* of programs, not the *dynamics of computation*, i.e, the semantics of code. Instead, the dynamics of computation arise by the interaction of rules (that operate on the syntax) with a particular piece of syntax, i.e. some code, representing the computation one wants to effect. In the lambda calculus (the model of computation on which Haskell is based) the workhorse of computation is a rule called beta reduction. This rule represents the operation of a function on data through the act of substituting the data for variables occurring in code. The data it operates on is a syntactic representation of the application of a function to data, but it is not the computation that corresponds to *applying the function to the data*. That computation happens when beta reduction operates on the syntax, transforming it to a new piece of syntax.

[3] In private conversation.

This distinction is how models that are less expressive than Turing-complete (e.g. context-free grammars) can host Turing-complete computation.

Not to belabor the point, but the same distinction happens in Java and JVM. The dynamics of computation in the JVM happen through rules that operate on a combination of registers in the virtual machine together with a representation of the code. A Java programmer staring at a piece of Java code is not looking at the computation. Far from it. The syntax of a Java program is a window into a whole range of possibly different computations that come about depending on the state of the registers of the JVM at the time the code is run. The difference between these two forms of evaluation, beta reduction in the lambda calculus versus the transitions of the JVM is very important and we will return to it.

For now, though, one way to think about this distinction between code and computation is through an analogy with physics. Traditionally, the laws of physics are expressed through three things: a representation of physical states (think of this as the syntax of programs); laws of motion that say how states change over time (think of this as the rules that operate on syntax); and initial conditions (think of this as a particular piece of code you want to run). In this light, physics is seen as a special purpose programming language whose execution corresponds in a particular fashion to the way the physical world evolves, based on our observations of it. Physics is *testable* because it lets us run a program and see if the evolution of some initial state to a state it reaches via the laws of motion matches our observations. In particular, when we see the physical world in a configuration that matches our initial state, does it go through a process of evolution that matches what our laws of motion say it should and does it land in a state that our laws of motion say it should. The fact that physics has this shape is why we can represent it effectively in code.

Once we see the distinction between code and computation, then the distinction between code and data is relatively intuitive, though somewhat subtle. Data in a computer program is also just syntax. In this sense it is no different than code, which is also just syntax. Every Lisp programmer understands this idea that somehow code is data and data is code. Even Java supports a kind of metaprogramming in which Java code can be manipulated as Java objects. The question is, is there any real dividing line between code and data?

The answer is a definitive yes. Data is code that has very specific properties, for example the code always provably runs to termination. Not all code does this. In fact, Turing's famous resolution of the Entscheidungsproblem shows us that we cannot, in general, know when a program will halt for a language enjoying a certain quality of expressiveness, i.e Turing-completeness. But, there are less expressive languages, and Turing-complete languages enjoy suitable sublanguages or fragments that are less expressive than the whole language. Data resides in syntax that allows proving that the computation associated with a piece of syntax will halt. Likewise, data resides in syntax that allows proving that the computation will only enjoy finite branching.

Programmers don't think about data like this, they just know data when they see it. But in models of computation like the lambda calculus that doesn't come equipped with built in data types, everything, even things like the counting numbers or the Boolean values true and false are represented as code. Picking out which code constitutes data and which constitutes general purpose programs has to do with being able to detect when code has the sorts of properties we discussed above. In general, there are type systems that can detect properties like this. Again, it's a subtle issue, but fortunately we don't need to understand all the subtleties, nor do we need to understand exactly where the dividing line between data and code is, just that there is one.

In summary code and data are both just syntax that represents a state upon which a rule, or many rules, will operate. Data is expressed in a less expressive fragment of the syntax than code, giving it a definite or finitary character that code doesn't always enjoy. Computation is the process of evolution arising when some rules interact with a representation of a state. Now, what does all this have to do with AI or the mind or even the rho-calculus?

5 Reflection as a defining characteristic of intelligence

The rho-calculus has a syntactic representation of the distinction between computation and code. It has an operation that expresses packaging up a computation as a piece of code so that it can be operated on, transforming it into new code. It also has an operation for turning a piece of code back into a computation. Whoah, you might say, that is some next level sh!t. But, as we mentioned Lisp programmers, and Java programmers have been doing this sort of metaprogramming for a long time. They have to. The reason has to do with scale. It is impossible for human teams to manage codebases involving millions and millions of lines of code without automated support. They use computer programs to write computer programs. They use computer programs to build computer program deployments. Metaprogramming is a necessity in today's world.

But way back in the '80's, still the early days of AI, a researcher named Brian Cantwell Smith made an observation that resonated with me and many other people in AI and AI-adjacent fields. Smith's argument is that introspection, the ability for the mind to look at the mind's own process, is a key feature of intelligence. For some this is even the defining feature of intelligence. To make this idea of introspection, which he called computational reflection, concrete, Smith designed a language called 3-Lisp that has the same operators that the rho-calculus has. Specifically, 3-Lisp has syntax to represent reifying a computation into code, and syntax for reflecting code back into running computation.

Now, there is a good reason to suspect that there is a connection between the problem of scale that today's developers face and the problem of modeling our reflective, introspective capacity as reasoning beings. In particular, managing the complexity of representing our own reasoning becomes tractable in the presence of computational reflection. We can apply all of our algorithmic tricks to representations of our own reasoning to obtain better reasoning. This observation is amplified in the context of what evolutionary biologists and psychologists call theory of mind.

Specifically, introspection arises from the evolutionary advantage gained by being able to computationally model the behaviors of others, in particular members of your own species. If Alice develops the ability to model Barbara's behavior, and Barbara is remarkably similar to Alice (as in same species, same tribe, even same extended family structure), then Alice is very close to being able to model Alice's behavior. And when Alice needs to model Barbara's behavior when Barbara is interacting with Alice, then Alice is directly involved in modeling Alice's behavior. Taking this up to a scale where Alice can model her family unit, or the behavior of her tribe is where things get really interesting. More on that shortly, but for now we can see that something about computational reflection has to do with improving reasoning at scale in two senses of that word: (the complexity scale) improving reasoning by applying reasoning to itself; and (the social scale) improving reasoning about large numbers of reasoning agents.

In fact, Smith's ideas about computational reflection and its role in intelligence and the design of programming languages were an inspiration for the design of the rho-calculus, which takes reification and reflection as primitive computational operators. However, where 3-Lisp and the rho-calculus part company is that 3-Lisp is decidedly sequential. It has no way to reasonably represent a society of autonomous computational processes running independently while interacting and coordinating. But in the context of a theory of mind this is just what a reasoner needs to do. They need an explicit model of their social context, which is made up of autonomous agents acting independently while also communicating and coordinating.

6 The rho-calculus: from 3-Lisp to Society of Mind

Around the same time Smith was developing his ideas of computational reflection Marvin Minsky was developing his famous Society of Mind thesis. My take on Minsky's proposal is that the mind is something like the US Congress, or any other deliberative body. It consists of a bunch of independent agents who are all vying for different resources (such as funding from the tax base). What we think of as a conscious decision is more like the result of a long deliberative process amongst a gaggle of independent, autonomous agents that often goes on well below conscious experience. But, the deliberative process results in a binding vote, and that binding vote is what is experienced as a conscious decision.

6.1 The syntax and semantics of the notation system

We now summarize a technical presentation of the calculus that embodies the core of rholang. The typical presentation of such a calculus follows the style of giving generators and relations on them. The grammar, below, describing term constructors, freely generates the set of processes, Proc. This set is then quotiented by a relation known as structural congruence and it is over this set that the notion of dynamics is expressed. This presentation is essentially that of [17] with the addition of polyadicity and summation. For readability we have relegated some of the technical subtleties to an appendix.

Notational interlude when it is clear that some expression t is a sequence (such as a list or a vector), and a is an object that might be meaningfully and safely prefixed to that sequence then we write $a : t$ for the sequence with a prefixed (aka "consed") to t. We write $t(i)$ for the ith element of t.

Process grammar

$$\begin{array}{ll} \text{PROCESS} & \qquad\qquad \text{NAME} \\ P, Q ::= \mathsf{0} \ \mid\ \mathsf{for}(\overrightarrow{y} \leftarrow x)P \ \mid\ x!(\overrightarrow{Q}) \ \mid\ *x \ \mid\ P|Q & \qquad\qquad x, y ::= @P \end{array}$$

Note that $\overrightarrow{x}$ (resp. $\overrightarrow{P}$) denotes a vector of names (resp. processes) of length $|\overrightarrow{x}|$ (resp. $|\overrightarrow{P}|$). We adopt the following useful abbreviations.

$$\Pi\overrightarrow{P} := \Pi_{i=1}^{|\overrightarrow{P}|} P_i := P_1 | \dots | P_{|\overrightarrow{P}|}$$

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

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

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

where $\{\overrightarrow{x}\} := \{x_1, \dots, x_{|\overrightarrow{x}|}\}$ *and* $\mathsf{FN}(\overrightarrow{P}) := \bigcup \mathsf{FN}(P_i)$.
An occurrence of x in a process P is bound if it is not free. The set of names occurring in a process (bound or free) is denoted by $\mathsf{N}(P)$.

6.2 Substitution

We use Proc for the set of processes, $@\mathsf{Proc}$ for the set of names, and $\{\overrightarrow{y}/\overrightarrow{x}\}$ to denote partial maps, $s : @\mathsf{Proc} \to @\mathsf{Proc}$. A map, s lifts, uniquely, to a map on process terms, $\widehat{s} : \mathsf{Proc} \to \mathsf{Proc}$. Historically, it is convention to use σ to range over lifted subsitutions, $\widehat{s}$, to write the application of a substitution, σ to a process, P, with the substitution on the right, $P\sigma$, and the application of a substitution, s, to a name, x, using standard function application notation, $s(x)$. In this instance we choose not to swim against the tides of history. Thus,

Definition 2. *given $x = @P'$, $u = @Q'$, $s = \{u/x\}$ we define the lifting of s to $\widehat{s}$ (written below as σ) recursively by the following equations.*

$$0\sigma := 0$$

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

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

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

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

where

$$\{\widehat{@Q/@P}\}(x) = \{@Q/@P\}(x) = \begin{cases} @Q & x \equiv_{\mathsf{N}} @P \\ x & otherwise \end{cases}$$

and z is chosen distinct from $@P$, $@Q$, the free names in Q, and all the names in R. Our α-equivalence will be built in the standard way from this substitution.

Definition 3. *Then two processes, P, Q, are alpha-equivalent if $P = Q\{\overrightarrow{y}/\overrightarrow{x}\}$ for some $\overrightarrow{x} \in \mathsf{BN}(Q)$, $\overrightarrow{y} \in \mathsf{BN}(P)$, where $Q\{\overrightarrow{y}/\overrightarrow{x}\}$ denotes the capture-avoiding substitution of $\overrightarrow{y}$ for $\overrightarrow{x}$ in Q.*

Definition 4. *The* structural congruence $\equiv$ *between processes [29] is the least congruence containing alpha-equivalence and satisfying the commutative monoid laws (associativity, commutativity and* $\mathsf{0}$ *as identity) for parallel composition* $|$.

Definition 5. *The* name equivalence $\equiv_{\mathsf{N}}$ *is the least congruence satisfying these equations*

$$\frac{}{@*x \equiv_{\mathsf{N}} x}\ \textsc{Quote-drop} \qquad \frac{P \equiv Q}{@P \equiv_{\mathsf{N}} @Q}\ \textsc{Struct-equiv}$$

The astute reader will have noticed that the mutual recursion of names and processes imposes a mutual recursion on alpha-equivalence and structural equivalence via name-equivalence. Fortunately, all of this works out pleasantly and we may calculate in the natural way, free of concern. The reader interested in the details is referred to the appendix **??**.

Remark 1. One particularly useful consequence of these definitions is that $\forall P.@P \notin \mathsf{FN}(P)$. It gives us a succinct way to construct a name that is distinct from all the names in P and hence fresh in the context of P. For those readers familiar with the work of Pitts and Gabbay, this consequence allows the system to completely obviate the need for a fresh operator, and likewise provides a canonical approach to the semantics of freshness.

Equipped with the structural features of the term language we can present the dynamics of the calculus.

6.3 Operational semantics

Finally, we introduce the computational dynamics. What marks these algebras as distinct from other more traditionally studied algebraic structures, e.g. vector spaces or polynomial rings, is the manner in which dynamics is captured. In traditional structures, dynamics is typically expressed through morphisms between such structures, as in linear maps between vector spaces or morphisms between rings. In algebras associated with the semantics of computation, the dynamics is expressed as part of the algebraic structure itself, through a reduction reduction relation typically denoted by $\rightarrow$. Below, we give a recursive presentation of this relation for the calculus used in the encoding.

$$\text{COMM}\ \frac{x_t \equiv_{\mathsf{N}} x_s, \quad |\overrightarrow{y}| = |\overrightarrow{Q}|}{\mathsf{for}(\overrightarrow{y} \leftarrow x_t)P \mid x_s!(\overrightarrow{Q}) \rightarrow P\{\overrightarrow{@Q}/\overrightarrow{y}\}} \qquad \text{PAR}\ \frac{P \rightarrow P'}{P|Q \rightarrow P'|Q} \qquad \text{EQUIV}\ \frac{P \equiv P' \qquad P' \rightarrow Q' \qquad Q' \equiv Q}{P \rightarrow Q}$$

We write $P \rightarrow$ if $\exists Q$ such that $P \rightarrow Q$ and $P \not\rightarrow$, otherwise.

6.4 Dynamic quote: an example

Anticipating something of what's to come, let $z = @P$, $u = @Q$, and $x = @y!(*z)$. Now consider applying the substitution, $\widehat{\{u/z\}}$, to the following pair of processes, $w!(y!(*z))$ and $w!(*x) = w!(*@y!(*z))$.

$$w!(y!(*z))\widehat{\{u/z\}} = w!(y!(Q))$$
$$w!(*x)\widehat{\{u/z\}} = w!(*x)$$

The body of the quoted process, $@y!(*z)$, is impervious to substitution, thus we get radically different answers. In fact, by examining the first process in an input context, e.g. $\mathsf{for}(z \leftarrow x)w!(y!(*z))$, we see that the process under the output operator may be shaped by prefixed inputs binding a name inside it. In this sense, the combination of input prefix binding and output operators will be seen as a way to dynamically construct processes before reifying them as names.

7 Replication

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

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

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

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.

$$!\mathsf{for}(v \leftarrow u)P := x!(\mathsf{for}(v \leftarrow u)(D(x)|P))|D(x)$$

Remark 2. Note that the lazier definition still does not deal with summation or mixed summation (i.e. sums over input and output). The reader is invited to construct definitions of replication that deal with these features.

Further, the definitions are parameterized in a name, x. Can you, gentle reader, make a definition that eliminates this parameter and guarantees no accidental interaction between the replication machinery and the process being replicated – i.e. no accidental sharing of names used by the process to get its work done and the name(s) used by the replication to effect copying. This latter revision of the definition of replication is crucial to obtaining the expected identity $!!P \sim !P$.

Remark 3. The reader familiar with the lambda calculus will have noticed the similarity between D and the paradoxical combinator.

Bisimulation The computational dynamics gives rise to another kind of equivalence, the equivalence of computational behavior. As previously mentioned this is typically captured *via* some form of bisimulation. The notion we use in this paper is derived from weak barbed bisimulation [21].

Definition 6. *An* observation relation, *$\downarrow_{\mathcal{N}}$, over a set of names, $\mathcal{N}$, is the smallest relation satisfying the rules below.*

$$\frac{y \in \mathcal{N},\ x \equiv_{\mathsf{N}} y}{x!(v) \downarrow_{\mathcal{N}} x} \qquad (\textsc{Out-barb})$$

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

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

Definition 7. *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 $\mathcal{S}_{\mathcal{N}}$.

Contexts One of the principle advantages of computational calculi from the λ-calculus to the π-calculus is a well-defined notion of context, contextual-equivalence and a correlation between contextual-equivalence and notions of bisimulation. The notion of context allows the decomposition of a process into (sub-)process and its syntactic environment, its context. Thus, a context may be thought of as a process with a "hole" (written $\Box$) in it. The application of a context K to a process P, written $K[P]$, is tantamount to filling the hole in K with P. In this paper we do not need the full weight of this theory, but do make use of the notion of context in the proof the main theorem.

$$\begin{array}{l}\textsc{context}\\ K ::= \Box \mid \mathsf{for}(\overrightarrow{y} \leftarrow x)K \mid x!(\overrightarrow{P}, K, \overrightarrow{Q}) \mid K|P\end{array}$$

Definition 8 (contextual application). *Given a context K, and process P, we define the* contextual application*, $K[P] := K\{P/\Box\}$. That is, the contextual application of K to P is the substitution of P for $\Box$ in K.*

Remark 4. Note that we can extend the definition of free and bound names to contexts.

7.1 Additional notation

We achieve some notational compression with the following convention

$$\mathsf{for}(y_1 \leftarrow x_1;\ \ldots;\, y_n \leftarrow x_n)P := \mathsf{for}(y_1 \leftarrow x_1)\mathsf{for}(y_2 \leftarrow x_2)\ldots\mathsf{for}(y_n \leftarrow x_n)P$$

Even though we already have the notation $x = @P$, allowing us to pick out the process a name quotes, it will be convenient to introduce an alternate notation, $\dot{x}$, when we want to emphasize the connection to the use of the name. Note that, by virtue of name equivalence, $@\ \dot{x}\ \equiv_{\mathsf{N}} x$; so, the notation is consistent with previous definitions.
In [17] an interpretation of the new operator is given. It turns out that there are several possible interpretations all enjoying the requisite algebraic properties of the operator (see [21]). We will therefore make liberal use of $(\mathsf{new}\ \overrightarrow{x})P$.

7.2 Extensions to the calculus

Values While it is standard in calculi such as the λ-calculus to define a variety of common values such as the natural numbers and booleans in terms of Church-numeral style encodings, it is equally common to simply embed values directly into the calculus. Not being higher-order, this presents some challenges for the π-calculus, but for the rho-calculus, everything works out very nicely if we treat values, e.g. the naturals, the booleans, the reals, etc as processes. This choice means we can meaninfully write expressions like $x!(\mathsf{5})$ or $u!(\mathsf{true})$, and in the context $\mathsf{for}(y \leftarrow x)P[*y]|x!(\mathsf{5})$ the value $\mathsf{5}$ will be substituted into P. Indeed, since operations like addition, multiplication, etc. can also be defined in terms of processes, it is meaningful to write expressions like $\mathsf{5}|{+}|\mathsf{1}$, and be confident that this expression will reduce to a process representing $\mathsf{6}$. Thus, we can also use standard mathematical expressions, such as $\mathsf{5}+\mathsf{1}$, as processes, and know that these will evaluate to their expected values. Further, when combined with for-comprehensions, we can write algebraic expressions, such as $\mathsf{for}(y \leftarrow x)\mathsf{5}{+}{*y}$, and in contexts like $(\mathsf{for}(y \leftarrow x)\mathsf{5}{+}{*y})|x!(\mathsf{1})$ this will evaluate as expected, producing the process (aka value) $\mathsf{6}$.
With these conventions in place it is useful to reduce the proliferation of $*$'s, by adopting a pattern-matching convention. Thus, we write $\mathsf{for}(@v \leftarrow x)P$ to denote binding v to the value passed and not the *name* of the value. Hence, we may write $\mathsf{for}(@v \leftarrow x)\mathsf{5} + \mathsf{v}$ without any loss of clarity, confident that this translates unambiguously into the formal calculus presented above. We achieve even greater compression and a more familiar notation if we also adopt the notation

$$\mathsf{let}\ x = v\ \mathsf{in}\ P := (\mathsf{new}\ u)(\mathsf{for}(@x \leftarrow u)P)|u!(v)$$

and generalized to nested expressions via

$$\mathsf{let}\ x_1 = v_1;\ \ldots;\, x_n = v_n\ \mathsf{in}\ P := (\mathsf{new}\ \overrightarrow{u})\mathsf{for}(x_1 \leftarrow u_1;\ \ldots;\, x_n \leftarrow u_n)P|\Pi u_i!(v_i)$$

Annihilation Another important variation has to do with the rewrite rules. There is a recursive version of the COMM-rule that aligns with intuitions about the recursive nature of behavior in compositionally defined agents. The maths make it clearer than the English. First, we define what it means for two processes to annihilate each other.

Definition 9. *Annihilation: Processes P and Q are said to annihilate one another, written $P \perp Q$, just when $\forall R. P|Q \rightarrow^* R \Rightarrow R \rightarrow^* \mathsf{0}$.*

Thus, when $P \perp Q$, all rewrites out of $P|Q$ eventually lead to $\mathsf{0}$. Evidently, $P \perp Q \iff Q \perp P$, and $\mathsf{0} \perp \mathsf{0}$. Naturally, we can extend annihilation to names: $x \perp y \iff \dot{x} \perp \dot{y}$.
Annihilation affords a new version of the COMM-rule:

COMM
$$\frac{x_t \perp x_s, \quad |\overrightarrow{y}| = |\overrightarrow{Q}|}{(R_1 + \mathsf{for}(\overrightarrow{y} \leftarrow x_t)P) \mid (x_s!(\overrightarrow{Q}).P' + R_2) \rightarrow P\{\overrightarrow{@Q}/\overrightarrow{y}\}|P'}$$

All annihilation-based reduction happens in terms of reductions that happen at a lower degree of quotation, and grounds out in the fact that $\mathsf{0}\perp\mathsf{0}$ and thus $@\mathsf{0}\perp@\mathsf{0}$.

How can this view, which puts most of the computation outside of conscious reasoning, be reconciled with a view of the mind as essentially, indeed definitionally reflective? The rho-calculus was designed with an answer to this question in mind. The rho-calculus says that computational agents come in just six shapes:

– 0 - the stopped or null agent that does nothing;
– for(y <- x)P - the agent that is listening on the channel x waiting for data that it will bind to the variable y before becoming the agent P;
– x!(Q) - the agent that is sending a piece of code/data on the channel x;
– P|Q - the agent that is really the parallel composition of two agents, P and Q, running concurrently, autonomously;
– ∗x - the agent that is reflecting the code referred to by x back into a running computation

Notice how three of these constructs use the symbol x. Two of them use x as though it were a channel for communication between agents, and one of them uses x as though it were a reference to a piece of code. The one magic trick that the rho-calculus has up its sleeve is that channels are references to a piece of code. It takes a bit of getting used to, but it comes with time.

Armed with just this much information about the rho-calculus we can return to our narrative about Alice and find parsimonious representations of all of the challenges facing her developing social and introspective intelligence. As outside observers of Alice's social context we can write down its behavior as a parallel composition of the behavior of each individual. In symbols that's P1 | P2 | . . . | Pn where Pi is the model of the ith individual in Alice's social context. Now, a model of Alice's behavior needs a representation of that parallel composition for her own behavior to represent reasoning about it. In symbols that's @(P1 | P2 | . . . | Pn). For Alice to have this data located somewhere she has access to it she puts the model on a channel x!(P1 | P2 | . . . | Pn), and when she needs to retrieve it she executes

for(y <- x)AliceThinkingAboutHerColleagues(y) | x!(P1 | P2 | . . . | Pn)

The workhorse rule of computation in the rho-calculus, which is very similar in spirit to the lambda calculus' beta reduction, is that an expression like this evolves to

AliceThinkingAboutHerColleagues(@(P1 | P2 | . . . | Pn))

So, now Alice's thoughts about her colleagues have an explicit representation of their behavior available to Alice. With it, she can simulate her colleagues behavior by simulating the behavior of P1 | P2 | ... | Pn through operations on @(P1 | P2 | ... | Pn). We can model Alice observing her colleague's actual behavior with an expression like Alice | P1 | P2 | ... | Pn. Alice can compare her simulation with her observations. In fact, whatever we can model is also available to Alice both to run as well as to reify into data and compare the code and her simulations of it to what she observes of the actual behavior of her social context. This includes Alice's own behavior.

This may have gone by a little fast, but think about it. This is the smallest set of operations needed for Alice to simultaneously model her social context and herself in it. In particular, threads are 'consciously available' to Alice just when her own behavior reifies those threads into data and her processing interacts with that data. This argument is part of what went into the design deliberations for the rho-calculus. It is the smallest model of computation that reconciles Smith's arguments for computational reflection with Minsky's arguments for a Society of Mind that fits with evolutionary biology's account of organisms with a theory of mind. Anything smaller misses a key component of the situation.

8 We have seen the enemy and we are they

This argument is why it is plausible for a model of computation like the rho-calculus to find purchase on the hardware in Alice's brain. She needs all the elements of this model to compete with other members of her species who are likewise racing to model the behavior of their social context. This is why, very much to the contrary of Joscha's position, i would argue that mobile concurrency is at the heart of artificial general intelligence.

References

1. Martín Abadi and Bruno Blanchet. Analyzing security protocols with secrecy types and logic programs. In *POPL*, pages 33–44. ACM, 2002.
2. Martín Abadi and Bruno Blanchet. Secrecy types for asymmetric communication. *Theor. Comput. Sci.*, 298(3):387–415, 2003.
3. Samson Abramsky. Algorithmic game semantics and static analysis. In *SAS*, volume 3672 of *Lecture Notes in Computer Science*, page 1. Springer, 2005.
4. Luís Caires. Behavioral and spatial observations in a logic for the pi-calculus. In *FoSSaCS*, pages 72–89, 2004.
5. Luis Caires. Spatial logic model checker, Nov 2004.
6. Luís Caires and Luca Cardelli. A spatial logic for concurrency (part I). *Inf. Comput.*, 186(2):194–235, 2003.
7. Luís Caires and Luca Cardelli. A spatial logic for concurrency - II. *Theor. Comput. Sci.*, 322(3):517–565, 2004.
8. Luca Cardelli. Brane calculi. In *CMSB*, volume 3082 of *Lecture Notes in Computer Science*, pages 257–278. Springer, 2004.
9. Vincent Danos and Cosimo Laneve. Core formal molecular biology. In *ESOP*, volume 2618 of *Lecture Notes in Computer Science*, pages 302–318. Springer, 2003.
10. Cédric Fournet, Fabrice Le Fessant, Luc Maranget, and Alan Schmitt. Jocaml: A language for concurrent distributed and mobile programming. In *Advanced Functional Programming*, volume 2638 of *Lecture Notes in Computer Science*, pages 129–158. Springer, 2002.
11. Cédric Fournet, Georges Gontheir, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR 1996*, volume 1119 of *Lecture Notes in Computer Science*, pages 406–421. Springer-Verlag, 1996.
12. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
13. Allen L. Brown Jr., Cosimo Laneve, and L. Gregory Meredith. Piduce: A process calculus with native XML datatypes. In *EPEW/WS-FM*, volume 3670 of *Lecture Notes in Computer Science*, pages 18–34. Springer, 2005.
14. Cosimo Laneve and Gianluigi Zavattaro. Foundations of web transactions. In *FoSSaCS*, volume 3441 of *Lecture Notes in Computer Science*, pages 282–298. Springer, 2005.
15. Greg Meredith. Documents as processes: A unification of the entire web service stack. In *WISE*, pages 17–20. IEEE Computer Society, 2003.
16. L. Gregory Meredith and Matthias Radestock. Namespace logic: A logic for a reflective higher-order calculus. In *TGC* [17], pages 353–369.
17. L. Gregory Meredith and Matthias Radestock. A reflective higher-order calculus. *Electr. Notes Theor. Comput. Sci.*, 141(5):49–67, 2005.
18. Lucius Meredith, Jan 2017.
19. Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.
20. Robin Milner. Elements of interaction - turing award lecture. *Commun. ACM*, 36(1):78–89, 1993.
21. Robin Milner. The polyadic π-calculus: A tutorial. *Logic and Algebra of Specification*, Springer-Verlag, 1993.
22. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992.
23. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, II. *Inf. Comput.*, 100(1):41–77, 1992.

24. Corrado Priami, Aviv Regev, Ehud Shapiro, and William Silverman. Application of a stochastic name-passing calculus to representation and simulation of molecular processes. *Inf. Process. Lett.*, 80(1):25–31, 2001.
25. Aviv Regev, William Silverman, and Ehud Shapiro. Representation and simulation of biochemical processes using the pi-calculus process algebra. In *Pacific Symposium on Biocomputing*, pages 459–470, 2001.
26. Davide Sangiorgi. Beyond bisimulation: The "up-to" techniques. In *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 161–171. Springer, 2005.
27. Davide Sangiorgi. On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst.*, 31(4):15:1–15:41, 2009.
28. Davide Sangiorgi and Robin Milner. The problem of "weak bisimulation up to". In *CONCUR*, volume 630 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 1992.
29. Davide Sangiorgi and David Walker. *The Pi-Calculus - a theory of mobile processes.* Cambridge University Press, 2001.
30. Peter Sewell, Pawel T. Wojciechowski, and Asis Unyapoth. Nomadic pict: Programming languages, communication infrastructure overlays, and semantics for mobile computation. *ACM Trans. Program. Lang. Syst.*, 32(4):12:1–12:63, 2010.