The rho-calculus for AI

Chapter 1

Lucius Gregory Meredith

Jan 22, 2023

1.1 Consciousness as the colonization of the brain

In a recent conversation with Joscha Bach, one his generation's most original thinkers, he made the startling assertion that mobile concurrency is at odds with artificial general intelligence. In this context mobile concurrency means the sort of concurrency one finds when agents (aka computational processes) can discover each other, that is 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. 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.

My response to this proposition is that it 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. If it is, well, rholang, an implementation of the rho-calculus, is hosted on Intel and AMD chips, for a start, and their computational models are very different from the one captured by the rho-calculus.

In particular, the rho-calculus 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 my argument 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.

1.2 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. 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?

1.3 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 ha
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 program-

ming 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.

1.4 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.

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.

1.5 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.