Gauguin, Descartes, Bayes: A Diurnal Golem's Brain

Kartik Chandra
Amanda Liu
Jonathan Ragan-Kelley
Joshua B. Tenenbaum
MIT CSAIL and Department of Brain & Cognitive Sciences
Cambridge, Massachusetts, USA

Abstract

A "quine" is a deterministic program that prints itself. In this essay, I will show you a "gauguine": a *probabilistic* program that *infers* itself. A gauguine is repeatedly asked to guess its own source code. Initially, its chances of guessing correctly are of course minuscule. But as the gauguine observes more and more of its own previous guesses, it detects patterns of behavior and gains information about its inner workings. This information allows it to bootstrap self-knowledge, and ultimately discover its own source code. We will discuss how—and why—we might write a gauguine, and what we stand to learn by constructing one.

CCS Concepts: • **Computing methodologies → Philosophical/theoretical foundations of artificial intelligence**; **Theory of mind**.

Keywords: reflection, probabilistic programming

ACM Reference Format:
Kartik Chandra, Amanda Liu, Jonathan Ragan-Kelley, and Joshua B. Tenenbaum. 2025. Gauguin, Descartes, Bayes: A Diurnal Golem's Brain. In *Proceedings of the 2025 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3759429.3762631

1 A Way of Knowing

From time to time, we all have crises of identity—moments of radical and overwhelming uncertainty about our selves. I[1] don't know whether the doubts that seize us can really be externalized in language, but if I were to try, I would express them as questions, questions like: *Who am I? What am I? What kind of person? What kind of mind?*

Whatever the Question really is, it creeps up on us abruptly. It taps us on the shoulder, sneaks into our heads, lodges itself stubbornly in our brains. Perhaps you asked it of yourself when deciding where to do your PhD—or whom to marry—or whether to forgive—or perhaps the Question haunted you after a failure, or a regret, or an unbearable personal loss.

In this essay, I want to tell you about a time the Question crept up on *me*. It happened this past March: it was spring, and I was freshly heartbroken, and before I knew it the Question was clattering around in my head again, louder and louder, until it drowned out everything else in my mind.

When I was a child, I never really knew where to start with the Question. There are no easy attacks on it, no easy ways to "know thyself." But by this spring I was 25, no longer a child, and it was far from the first time I had faced the Question—which meant, for once, that I was not starting from scratch. There was prior work to build on this time. One evening, I pulled out all of my old journals, settled into a couch, and began looking to see whether I had made any progress the last time this had happened to me.

Now, as it turned out, the "last time" was when I was 15—which meant, of course, that my Answers were totally bunk. At 15, I had barely any life experience to reflect on at all: what could I possibly have known about myself? I laughed at the confidence with which my teenage self posited increasingly absurd theories about himself. This was a mistake, I thought, there was no progress here. I almost put the journals away.

And yet— as I turned the pages, I couldn't help but feel like I was learning *something* about myself from Kartik-at-15. But how was that possible, if I knew his theories were all bunk? It took me a moment to realize what was going on. My 15-year-old self's thoughts may have been bunk, but they were nonetheless windows into *how* he thought: the ways in which he tried to derive self-knowledge from whatever bits of self-data were available to him. If I were to take those journals not as a source of *theories*, but as a source of *data on the theorizer*, then perhaps I might get somewhere in forming new *meta*theories of my own. So I kept reading—and weirdly, it worked. When I finished the final entry, I really did feel like I understood myself better than ever before.

What a strange learning mechanism this is—if you can even call it a "learning mechanism." I hope you agree that there is something mysterious, perhaps even uncomfortable,

[1] In this essay, "I" refers to the first author, writing from personal experience. The remaining authors helped develop and present these ideas in this form.

This work is licensed under a Creative Commons Attribution 4.0 International License.
Onward! '25, Singapore, Singapore
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2151-9/25/10
https://doi.org/10.1145/3759429.3762631

about the idea that one can learn about oneself simply by... well, by theorizing incorrectly, reflecting on the incorrect theories, and then theorizing anew. Isn't that circular? Can a mind really come to learn something new about itself endogenously, just by thinking about its own thinking?

It is strange, yes. But this, I have come to believe, really *is* a way of knowing—perhaps even *the* way of knowing—oneself. As the Cartesian argument goes: in the face of overwhelming doubt about ourselves, the firmest ground we have to build from is *cogito ergo sum*, reflection upon reflection. Not that this way of learning is limited to professional philosophers. We are all born strangers to ourselves; every human infant faces the extraordinary puzzle of organizing the blooming, buzzing confusion of cognitive experience into a working theory of "mind" and "self." The strongest signal about our thoughts *is* our thoughts: so to learn about our minds, we must reflect on our own reflections.

Of course, this is not the only way we learn. We also learn about ourselves from external feedback: from friends, from books, from teachers. But then, how do those *others* learn? At the level of a community—or a species—there is nothing "external" to give us external feedback. Humanity as a whole has no choice but to bootstrap self-understanding *ex nihilo*. We collectively look to our history to understand how our predecessors (mis)understood themselves, and from that evidence we form new theories about ourselves. Those theories in turn will one day become data for our children to theorize about, and so on, ad infinitum. It is just like the journals on my couch: ancestors' theories become descendants' data, a grand inter-generational baton-pass.

I want to be clear that the baton-passes I am talking about are *not* the ones by which we iteratively refine our theories over the course of generations—not, for example, the baton-pass from Galileo to Newton to Einstein. Rather, I am talking about cases where the very act of theorizing begets theorizing about theorizing: for example, the way that the history of science has *itself* been marshaled as a source of evidence with which to study the human capacity for theory-formation [Gopnik 1996; Kuhn 1962; Lakatos 1970].

I am expressing this notion in the language of science, but I think the clearest and most compelling examples of the baton-pass actually come from the history of art. I am thinking in particular of the French painter Paul Gauguin. At the turn of the 20th century, Gauguin moved to Tahiti and suffered a major psychological crisis. Sick and destitute, and believing he would soon die, he decided to make one last statement on the nature of humanity: an enormous painting of the native people of Tahiti, which he called *Where do we come from? What are we? Where are we going?* (This is, of course, another phrasing of the Question—though it is asked not of an individual, but of humanity as a whole.)

Today, Gauguin's painting hangs in a gallery in the Boston Museum of Fine Arts. It is just across the river from MIT, and I have spent many hours of my life sitting by it. Sometimes when I visit, I wish that I could just read off the Answer from Gauguin's canvas—but, alas, that would be too easy. The problem is that one cannot in good conscience endorse Gauguin's theories of humanity today. Gauguin saw his subjects as savages to be civilized by the Europeans: his colonial, exploitative attitude toward the native people of Tahiti casts a dark shadow over any reading of his work.

And yet— whenever I visit the painting, I can't help but feel like we can still learn *something* about being human from it. Over the years I have seen hundreds of visitors wander past it, gaze at the figures, and reflect on Gauguin's treatment of his fellow humans. Visitors often see themselves in the two children Gauguin painted, who appear lost in thought—reflecting on these figures, visitors come to reckon with how a man could treat a thinking, glowing human as an animal. The visitors do not accept Gauguin's painting at face value. Rather, they marshal Gauguin's painting—I mean that as a verb—as evidence with which to theorize about human nature in ways Gauguin himself could never have expected. This is the fate of all art, I think: the moment it leaves the artist's fingers, it belongs to the world. Just like that, it transmutes from theory to data. The children, the painter, the visitors, the student: the baton passes on and on.

Well, then, can it really work this way? Can a mind come to understand something about itself merely by reflecting on its own past reflections? As I wrote earlier, it is not immediately obvious that this is possible. Let me say a little more about this now. I think there are at least three reasons for skepticism. First, there seems to be something circular about this way of knowing: information gained "for free" without any external feedback. Second, there is the pesky empirical observation that today's most advanced AI systems decidedly do *not* learn in this way. When large language models are trained on their own output, they do not gain in wisdom, as we might hope—instead, they "model collapse" into gibberish [Shumailov et al. 2024]. Third, there is a representational problem to contend with. "If the human brain were so simple that we could understand it," wrote Emerson Pugh, "then we would be so simple that we couldn't" [Pugh 1977]. If Pugh is correct, then no mind could *ever* come to understand itself: self-knowledge would be a fool's errand (so to speak).

I acknowledge all of these challenges. But I have been a computer scientist long enough to know that reflection can defy our intuitions. Consider the unexpected consequences of constructing a set that contains itself [Russell 1903], a machine that can simulate itself [Turing et al. 1936], or a compiler that can compile itself [Thompson 1984].

Let us take a step back, then, and think about this problem like programming languages researchers. What would it look like to try to capture the essence of this issue computationally? Can we at least write a *program* that comes to know something about itself by reflecting on its own reflection? That, at least, would be progress: an existence

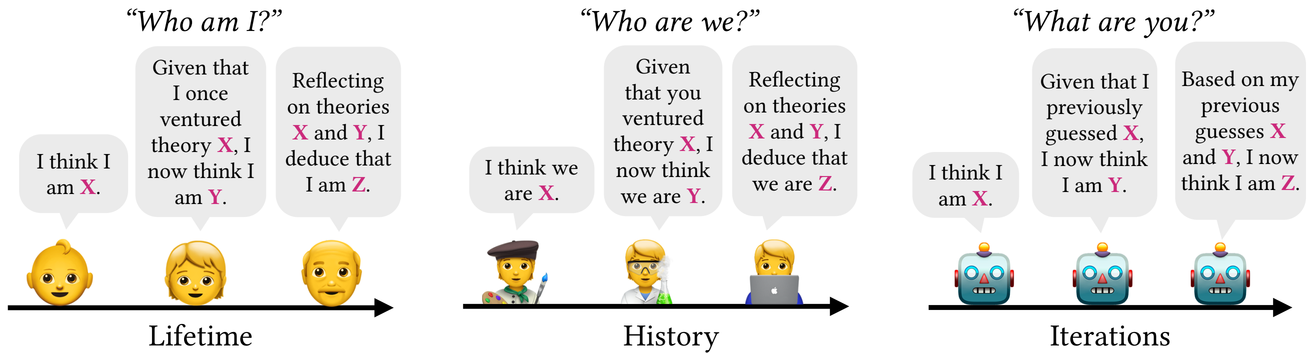

Figure 1. It is human nature to theorize about human nature—whether at the timescale of an individual's lifetime **(left)** or a civilization's history **(center)**. Even when our theories are incorrect, they are informative: they reveal clues about how we theorize, clues which in turn inform future theorizing. Similarly, a gauguine makes guesses about its own source code, and then reflects on those guesses to learn about itself **(right)**. In this way, it seeks to gradually bootstrap a robust self-understanding.

proof that an intelligent system could in principle bootstrap self-knowledge this way.

Here is how I imagine such a program might work. It would be tasked at the outset with guessing its own source code. We would provide as input a sequence of its past guesses (initially empty), and demand as output a *new* guess informed by the previous guesses. Then, we would append the new guess to the sequence of past guesses and repeat (see Figure 1). The critical question would be whether or not the program eventually converges to correctly guess its own source code. If successful, the program would be a lot like a "quine," a program that prints out a copy of its own source code [Hofstadter 1979]. But while a quine is a "nativist," repeating a self-description baked into its source, our program would be an "empiricist," learning about itself by experience.

Let us call such programs—not quines, but *gauguines*. The question then simply becomes: Does a gauguine exist? Can we construct one explicitly? And if we did, what would it do—would it really converge to discovering its own source code? The answers to these questions, as it turns out, are all *yes*. In the rest of this essay, I will show you step-by-step how to write a gauguine from scratch.[2]

2 Preliminaries: Probabilistic Inference over Probabilistic Programs

Let us start by getting our bearings straight. What kind of a program is a gauguine? In what programming language should it be written? The centrality of "guessing" gives us a hint: a gauguine should be a *probabilistic* program, one that makes rational inferences by conditioning its uncertainty on observed evidence. In particular, a gauguine is a probabilistic program that performs inference over a space of candidate probabilistic programs.

[2]The full implementation is available online at github.com/kach/gauguine.

There is a long tradition in computational cognitive science of modeling theorizing using probabilistic programs that perform inference over (probabilistic) programs [Lake et al. 2015; Mills et al. 2023; Rule et al. 2020]. But gauguines are perhaps unique in just how meta they are. A gauguine is a probabilistic program that itself performs inference over a space of probabilistic programs, by conditioning on the probabilistic programs it has previously inferred. (Don't worry—we will get there step by step.)

To write a gauguine, we need a programming language that natively supports both probabilistic inference and reflective metaprogramming. Here, we will work in the probabilistic programming language Church [Goodman et al. 2012]. Church extends Scheme with three primitives for reasoning about probability distributions: the form (**sample** D) draws a fresh sample from a probability distribution D, the form (**condition** ϕ) conditions sampled random variables on predicate ϕ being true, and the form (**infer** e) generates a probability distribution over the values of expression e, marginalizing over calls to **sample** and conditioning on calls to **condition** made in e. (Helper functions like `distr/uniform` and `distr/bernoulli` generate simple distributions directly.)

The use of Church is best explained by example. Suppose I secretly roll two six-sided dice. Then, I tell you that the sum of the two rolls is 7. Based on this new information, how likely should you think it is that the first die came up 3 or more? We can answer this question in Church like this:

```
(define die (distr/uniform '(1 2 3 4 5 6)))
(infer
  (let ((x (sample die))
         (y (sample die)))
       (condition (= (+ x y) 7))
       (>= x 3)))
; P(#t) = 0.6667
; P(#f) = 0.3333
```

This **infer** expression returns a distribution over booleans, i.e. possible values of `(>= x 3)`. In the inferred distribution, `#t` has probability 2/3 and `#f` has probability 1/3. Hence, the first die is twice as likely to be ≥ 3 than not.

How does Church produce this distribution? Conceptually, Church works similar to the `amb` evaluator [Abelson and Sussman 1996], except that it additionally keeps track of the probability of each execution path [Goodman and Stuhlmüller 2014]. In fact, Church can be embedded into Scheme or Racket in just a few lines of code, via delimited continuations [Kiselyov and Shan 2009].

The joy of Church is that its probabilistic primitives compose seamlessly with the full expressive power of Scheme. In particular, we are not limited to inference over numbers, as in the example above. For example, suppose we rolled a die, saw the number k, and then tossed k fair coins. Given that we saw exactly 3 heads come up, what sequences of tosses are likeliest? We can reason about this problem by recursively sampling a length-k list of independent coin toss outcomes, and then conditioning on the number of heads observed.

```
(define fair-coin (distr/uniform '(0 1)))
(define (toss k)  ; list of k coin tosses
  (if (= k 0) '() (cons (sample fair-coin)
                         (toss (- k 1)))))
(infer
  (let* ((k (sample die))
         (sequence (toss k)))
        (condition (= (sum sequence) 3))
        sequence))
; P('(1 1 1))   = 0.1250
; P('(1 1 0 1)) = 0.0625
; P('(0 1 1 1)) = 0.0625
; ... (rest elided) ...
```

I chose this example because it demonstrates an interesting effect in Bayesian inference. Even though lists of length $k = 3$ and lists of length $k = 4$ are equally probable *a priori*, the length-3 list `'(1 1 1)` is twice as likely *a posteriori* than the length-4 list `'(1 1 0 1)`. What is going on here? The answer lies not in the prior, but in the *likelihood* of the data observed. When $k = 4$, we must additionally posit that a fourth coin toss came up tails, an event whose likelihood of 1/2 must be factored into the right-hand-side of Bayes' rule. Hence, length $k = 4$ sequences are half as likely as length $k = 3$ sequences.

This effect is called the "Bayesian Occam's Razor": because of the influence of the likelihood term in Bayes' rule, rational agents naturally come to favor simpler hypotheses, i.e. those that require positing fewer parameters [see Blanchard et al. 2018]. We will see in a few pages how the Bayesian Occam's Razor is critical for gauguines to work.

2.1 Probabilistic Inference over Programs

In the previous example, we performed inference over *lists* of coin-toss outcomes. But as you may have guessed, we can just as easily perform inference over arbitrary S-expressions—and thus over Church programs themselves. For example, suppose I take a standard 52-card deck and secretly pick out some of the cards according to some predicate ϕ. I then randomly sample some cards from the full deck and tell you whether ϕ satisfies them. Consider the following outcomes:

1. I draw K♡, and tell you that it satisfies ϕ
2. I draw 2♠, and tell you that it does *not* satisfy ϕ
3. I draw J♡, and tell you that it satisfies ϕ
4. I draw 3♢, and tell you that it does *not* satisfy ϕ.

Based on this information, what do you think ϕ might be? Intuitively, you might think that perhaps ϕ only accepts hearts-suit cards—or, perhaps it only accepts face cards. (It might also be the case that ϕ is even more complex: for example, it might accept hearts *or* faces, or it might only accept cards that are both-hearts-and-faces.)

To solve this problem computationally, let us imagine sampling a Scheme predicate by generating it syntactically from a simple probabilistic context-free grammar (PCFG). We can create a sampler of max-depth-d Boolean expressions over some card by recursively sampling simple production rules. In the generated expressions, x is a free variable representing the card that the predicate is applied to.

```
(define primitives
  (distr/uniform
    '(clubs? spades? ... face? numbered?)))

(define (<pred> d)  ; samples a predicate
  (condition (>= d 0))
  ((sample
    (distr/uniform
     (list
      ; apply primitive to card x
      (λ () `(,(sample primitives) x))
      ; create a compound Boolean expression
      (λ () `(not ,(<pred> (- d 1))))
      (λ () `(and ,(<pred> (- d 1))
                  ,(<pred> (- d 1))))
      (λ () `(or  ,(<pred> (- d 1))
                  ,(<pred> (- d 1))))
      )))))
```

(The syntax used in expressions like `` `(not ,(<pred> (- d 1))) `` is Scheme's "quasiquotation" syntax: it helps us create a quoted S-expression, into which get substituted the values of the "unquoted" sub-expressions, heralded by commas. Quasiquotation is similar to template string literals.)

Notice that we sample from a list of *lambda-expressions* that wrap each production rule. Why not directly sample from a list of production rules? It is important to defer the

recursive calls to `<pred>` until *after* sampling a production: otherwise, we would eagerly recurse down to $d = 0$, and *all* execution paths would be rejected by `(condition (>= d 0))`.

Now that we have defined a prior over predicates, we can sample a predicate ϕ, use **eval** to apply ϕ to a candidate card (binding the card to x via a let-expression), and condition the result on the evidence we are given. Then, applying Bayes' rule, we can infer our posterior belief over ϕ.

```
(infer
  (let ((ϕ (<pred> 2)))  ; prior
    (condition
     (and (eval `(let ((x 'K♡)) ,ϕ))
          (eval `(let ((x 'J♡)) ,ϕ))
          (eval `(let ((x '2♠)) (not ,ϕ)))
          (eval `(let ((x '3◇)) (not ,ϕ)))))
    ϕ))
```

Here are the top three hypotheses in the posterior distribution inferred by this program:

```
; P( (face? x) )                   = 0.42
; P( (hearts? x) )                 = 0.42
; P( (or (hearts? x) (face? x)) ) = 0.004
```

As we expected, our Church program is unsure whether the mystery predicate is `(face? x)` or `(hearts? x)`. There are some other valid hypotheses as well, such as the disjunctive predicate `(or (face? x) (hearts? x))`. These compound hypotheses are also consistent with the evidence, but they rank much lower in the inferred distribution, because they have a lower prior under our PCFG. That in turn is an effect of the Bayesian Occam's Razor: sampling a deeper abstract syntax tree requires positing additional choices of production rules.

2.2 Probabilistic Inference over *Probabilistic* Programs

So far, we have probabilistically inferred small deterministic programs from data. This is progress, but for a gauguine we need to perform inference over *probabilistic* programs. As a warm-up, let us consider a variation of the game from the previous example: this time, I will move cards matching ϕ to a special pile and randomly draw cards (with replacement) from that pile to show you. Suppose the cards I draw are $\langle K♣, Q♣, K♣, J♣\rangle$. What might ϕ be now? Intuitively, you might think from this evidence that ϕ only accepts clubs-suit face cards.

To solve this problem, we first define a helper function that converts a predicate ϕ into a uniform distribution over cards that satisfy ϕ. We can define this helper by metaprogramming a small probabilistic program:

```
(define (make-distr phi)
 `(infer
   (let ((x (sample (distr/uniform cards))))
         (condition ,phi)
         x)))
```

`(make-distr ϕ)` constructs an S-expression representing a probabilistic program that generates a uniform distribution over cards matching ϕ. We can **eval** the S-expression to obtain a uniform distribution over cards matching ϕ, and then sample repeatedly from that distribution in order to solve our problem.

```
(infer
  (let* ((ϕ (<pred> 2))
         (D (eval (make-distr ϕ))))
       (condition
        (and (equal? 'K♣ (sample D))
             (equal? 'Q♣ (sample D))
             (equal? 'K♣ (sample D))
             (equal? 'J♣ (sample D))))
       ϕ))
```

Here are the top three inferred predicates:

```
; P( (and (face? x) (clubs? x)) ) = 0.37
; P( (face? x) )                   = 0.14
; P( (clubs? x) )                  = 0.10
```

As we expected, and consistent with our intuitions, the most probable hypothesis is `(and (face? x) (clubs? x))`.

You might wonder why this hypothesis is even more probable *a posteriori* than `(clubs? x)`, given that the latter is clearly more parsimonious and must therefore have a higher prior under our PCFG. The answer again lies in the Bayesian Occam's Razor—this time applied not to ϕ, but the data that is sampled from ϕ. In the example in the previous section, the likelihood of the observed data was always binary—either 0, if the observations were inconsistent with hypothesis ϕ, or 1, if they were consistent. In the current example, however, the likelihood is more complex: the **condition** statement itself involves sampling probabilistically from a distribution that depends on ϕ. Hence, to understand why one hypothesis is favored over another, we have to analyze both the prior and the likelihood.

This analysis reveals an interesting effect. Even though it is more complex syntactically, the conjunctive hypothesis `(and (face? x) (clubs? x))` is also *more specific* than `(face? x)`: it specifies a smaller set of cards that match the predicate (there are 12 face cards, but only 3 of them are also of the suit clubs). The likelihood of each drawn card is therefore higher under the more specific hypothesis (1/3 vs. 1/12). As more and more cards are drawn, the effect of the likelihood term compounds and outweighs the prior, and so the less specific hypothesis `(face? x)` becomes less and less likely *a posteriori*. This phenomenon is known as the "size principle" of Bayesian concept learning [Tenenbaum 1998]. It explains why our own intuitions differ across these two examples: in the former case uncertain between two simple hypotheses, and in the latter case confident of a single, more complex hypothesis.

3 Growing a Gauguine

We finally have all of the ingredients we need to write a gauguine, a probabilistic program that infers probabilistic programs. Let us start with the definition: a gauguine is a function that takes as input H, a list of past guesses. (By convention, we will say that the list H is provided in reverse-chronological order, with the most recent guess first.)

```
(define g
  (λ (H)
```

What does a gauguine do? It returns a distribution over probabilistic programs, which represents its uncertainty about itself. First, it samples a candidate probabilistic program $g*$ from some prior over probabilistic programs. (We will define the prior <guesser> with a PCFG, just like we defined the prior <pred> earlier.)

```
    (infer
      (let ((g* (<guesser>)))
```

Next, for each guess H_i in its history, the gauguine conditions the candidate program $g*$ on having guessed guess H_i. We have to be careful here: we have to model $g*$ as having guessed H_i after having seen only the *preceding* guesses, i.e. after having seen $(H_{i-1}, H_{i-2}, \dots, H_2, H_1)$.

To implement this, we use the helper function postfixes, which returns a list of nonempty postfixes of a list (for example, (postfixes '(1 2 3)) gives '((3) (2 3) (1 2 3))). Instead of looping over H directly, we loop over its postfixes.

```
        (for ([H_ (postfixes H)])
```

For a given postfix H_ of H, the expression (**car** H_) gives a guess H_i and the expression (**cdr** H_) gives the preceding history $(H_{i-1}, \dots, H_1)$, which g observed when guessing (**car** H_). We thus want to condition on (**car** H_) being equal to a sample from the distribution given by applying g* to (**cdr** H_). We can use **eval** to convert the quoted S-expression g* to a callable function.

```
          (condition
           (equal?
            (car H_)  ; H_i
            (sample ((eval g*) (cdr H_))))))
```

Finally, we return a posterior distribution over guesses $g*$.

```
        g*))))
```

3.1 Defining the Hypothesis Space

All that remains is to specify a PCFG for sampling a <guesser>. The grammar of Church programs that perform inference over Church programs is much more complex than the grammar of predicates over playing cards (<pred>). It is straightforward to derive from the syntax of Church, but somewhat tedious to specify. Here, we will present the highlights.

At the top-level, <guesser> generates an S-expression, which represents a Church function that takes a history (list of guesses) as input, and then performs inference over some expression. Let us call such S-expressions *guesses*.

```
(define (<guesser> [d 5])  ; default depth 5
  (condition (>= d 0))
  `(λ (history) (infer ,(<expr> (- d 1)))))
```

To be clear, the expressions we are interested in performing inference over are Church expressions that themselves evaluate to guesses.

What are the valid <expr>s? Let us inductively define a new nonterminal.

```
(define (<expr> d)
  (condition (>= d 0))
  ((sample
    (distr/uniform
     (list
```

The simplest expression is a reference to a variable.

```
  (λ () (sample (distr/uniform '(g h ...))))
```

There are also the usual compound expressions, such as let-bindings and loops, but for space we will elide the definitions of these production rules here. Let us instead focus our attention on two particularly interesting types of expressions.

First, because <expr>s evaluate to S-expressions representing guesses, we can **eval** an expression (i.e. convert the S-expression to a function), and then apply the resulting function to data to sample a *new* expression. This is like asking a guess to itself make a guess.

```
  (λ ()
    `((eval ,(<expr> (- d 1))) <history>))
```

Second, we can generate a new expression by uniformly sampling from our prior (the one we are currently implementing). To be clear, in this production rule, we are *emitting* a call to <guesser> in the generated program, rather than making a recursive call to <guesser> in the meta-program. The generated program will at runtime call <guesser> to sample a fresh guess unconditionally from the prior.

```
  (λ () '(<guesser>))
  )))))
```

Finally, just like how we inductively defined a grammar for generating <expr>s, we can inductively define a grammar of <history>s (constructed out of list operations like **cdr** and postfixes), a grammar of <condition>s (constructed out of boolean operations), and so on. This completes our PCFG.

3.2 On Potential Qualms Regarding <guesser>

You might worry that we have somehow taken a shortcut by giving our gauguine access to the hypothesis space specified by <guesser>. Shouldn't it also learn <guesser> from scratch? It is a reasonable concern. In principle, one could imagine trying to learn <guesser> jointly alongside g, perhaps by sampling possible <guesser>s from a general grammar <program>

of Church programs. This learning problem would be computationally much more demanding, so we defer it for now.

Here is one reason to be okay with us taking this liberty. Even if we were to guess `<guesser>`, there would still be a loose end: the definition of `<program>`. From what grammar is `<program>` sampled? Would *that* grammar need to be built in? In fact, as argued by Fodor [1975, p. 64], ultimately there must be *a* grammar that is built in. To infer the grammar `<guesser>`, we would need a meta-grammar of possible grammars, which in turn would have to be learned from a meta-meta-grammar, and so on. The recursion must be truncated *somewhere*, and here, for the sake of tractability, we truncate it directly at the level of `<guesser>`.

A related concern is that our grammar of guessers might somehow be over-constrained, in a way that sneaks in unjustified knowledge about g. For example, to take a degenerate case, if the prior probability specified by `<guesser>` for g were extremely high (or even 1), then the task faced by g would be trivial. This is also a reasonable concern. Notice, however, that our grammar of guessers is not specialized to g—it is general over the space of Church programs that perform inference over Church programs. In fact, as we will see in the next section, the hypothesis space generated by `<guesser>` includes a wide variety of meaningful and interesting *non-gauguine* programs, among which g itself has only a minuscule prior probability. Hence, the inference problem faced by g is nontrivial: success at self-inference is not assured and would be meaningful.

3.3 So, Does Our Gauguine Work?

To test our gauguine g, we have to iteratively query it with its own guesses as input. The procedure `test!` takes as input a gauguine g and a list of past guesses H. It runs g, conditioned on history H, to produce a distribution b ("belief"). Then it samples a new guess $g*$ from distribution b. Finally, it prepends guess $g*$ to the history H and loops until k new guesses have been made.

```
(define (test! g H k)
  (if (= k 0) H
    (let ((b (g H))
           (g* (sample! b)))
         (test! g (cons g* H) (- k 1)))))
```

Notice that here we use **`sample!`** instead of **`sample`**. The form **`sample!`** is used to actually take a single stochastic sample from a distribution, i.e. by calling a random number generator, as one might want to do in a Monte Carlo simulation. In contrast, **`sample`** only *conceptually* draws from a distribution: in reality, all it does is it re-weight possible execution traces.

Let us see what happens. Figure 2 shows the sequences of guesses made by our gauguine when we run `(test! g '() 25)`. There are many things to say about these results. First, and most importantly, notice that g *does* eventually succeed in guessing itself—almost always within just 15 guesses.

Along the way, however, g makes some interesting patterns of incorrect guesses. Let us take a moment to explore what g's earlier, failed theories look like. The first guess is typically the program g_0, which I have reproduced below. I call this program the "wild guesser": this theory posits that g ignores past guesses in H, and instead simply samples a new independent guess from its prior each time.

```
(define g0-wild
  (λ (H) (infer (<guesser>))))
```

This theory is clearly wrong—but it is *simple*. Under the principle of Bayesian Occam's Razor, g_0 receives a high prior, and thus in the absence of any data (i.e. on the first guess) it is actually rational to guess g_0.

Another interesting-but-wrong theory is g_1, reproduced below. I call this program the "omniscient" theory. On this account, g always makes the same guess, which happens to be correct.

```
(define g1-omniscient
 (λ (H)
  (infer
   (let ([g* (<guesser>)])
     (for ([h H]) (condition (equal? h g*)))
     g*))))
```

This theory is also clearly wrong—but on top of that, it is also syntactically complex, which makes it improbable even *a priori*. As a result, the dynamics of g_1 are quite tragic: it is only slightly plausible at the outset, and it gets refuted as soon as g observes that its guesses may differ from one another (i.e. typically by the second guess).

One last interesting hypothesis is g_2, shown below. I call this the "memoryless" theory: it posits that g conditions its guess on having generated the past guesses in H, but that it erroneously treats the past guesses as having been made *independently*, rather than by progressively conditioning on the growing mass of available evidence.

```
(define g2-memoryless
 (λ (H)
  (infer
   (let ([g* (<guesser>)])
    (for ([h H])
     (condition
      (equal? h (sample ((eval g*) '())))))
    g*))))
```

The bug here is subtle; indeed, it is the bug we were careful to avoid earlier when we first implemented g. While g correctly loops over `(prefixes H)`, g_2 loops directly over the guesses in `H`, and fails to condition $g*$ on the evidence with which it produced each of those guesses—instead, it calls $g*$ with the empty list each time.

It is a reasonable error to make. Looking at Figure 2, we see that g_2 is actually favored over g for the first ≈ 7 guesses. However, g ultimately wins out in the long run.

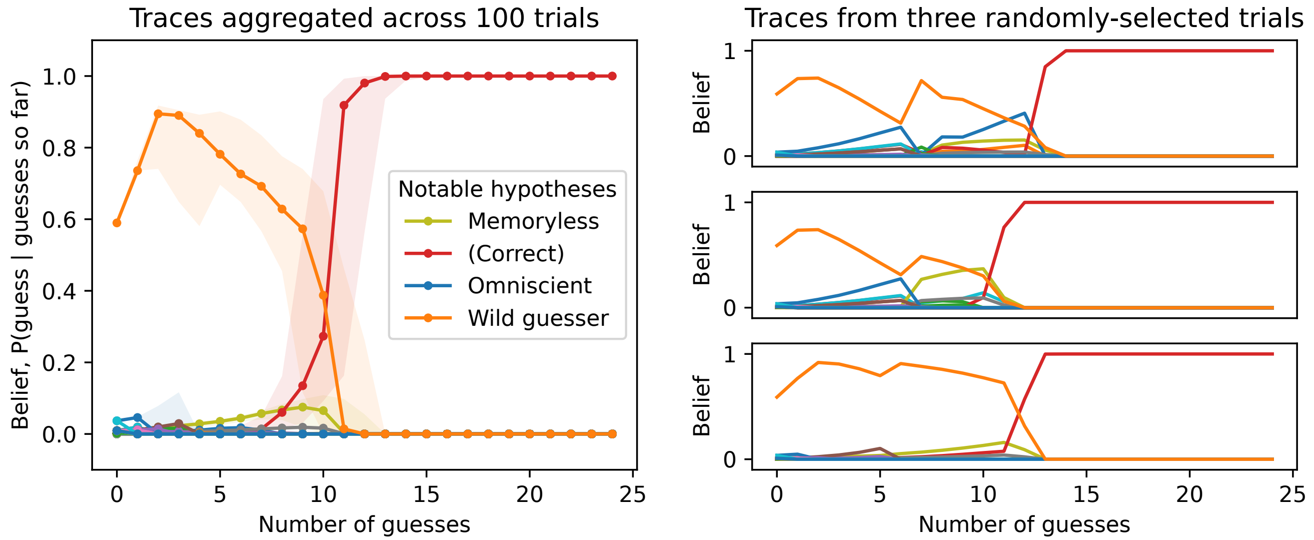

Figure 2. Dynamics of a gauguine. In these plots, each line represents a candidate program, i.e. a hypothesis about what g is. Initially, our gauguine is confident that it is a "wild guesser" (orange line)—a probabilistic program that merely samples from its prior, discarding all observed evidence. However, after observing around 5 of its guesses, various competing hypotheses begin to gain traction (blue/yellow lines). Eventually, around 10 guesses, a "paradigm shift" occurs, after which the gauguine reliably guesses its own source code (red line). Note that this figure only spotlights four interesting hypotheses—not explicitly labeled are the many, many other hypotheses that are rejected early on, and which overlap flat along the horizontal axis. (The shaded regions in the left panel depict 25%–75% interquartile ranges from data aggregated across 100 independent runs of `test!`.)

3.4 But Wait—How Could It Possibly Work?

Should we be surprised that the gauguine worked? Earlier in this essay, I offered three reasons to be skeptical that an intelligent system could bootstrap self-knowledge in this way. Let us take a moment to see how our gauguine escaped those problems.

The first challenge was circularity. Where exactly does the information "come from"? We seemingly do not give the gauguine any external feedback—for example, we do not explicitly say that its guesses are correct or wrong—and yet it is somehow able to learn something about itself. How is that possible? To reconcile these conflicting intuitions, it is helpful to separate the program from its execution. The Church *interpreter* is certainly external to the program it executes, and it is ultimately what creates a new ground-truth observation (the "guess") every time we run our gauguine.

This reasoning, by the way, applies to the human mind as well. Our thoughts are ultimately supported by cognitively-impenetrable neural processes. Observing an action—or a thought—of your own really *is* an external signal after all. To put it another way, thoughts about thought make contact with reality as soon as the thoughts are thought. I am reminded of Holden Caulfield's observation in J. D. Salinger's novel *The Catcher in the Rye:* "How do you know what you're going to do till you do it? The answer is, you don't. I think I am, but how do I know?"

The second challenge was the empirical observation that our most advanced AI systems, such as large language models (LLMs), do not learn in this way—instead, they suffer from model collapse. How is feeding a gauguine its own guess any different from feeding an LLM its own output? The critical distinction is that because a gauguine is a probabilistic program, it integrates each new piece of evidence in a model-based Bayesian manner—where the "model" is that it itself generated the given guesses. This model-based approach allows the gauguine to learn quite a lot about itself from just a few guesses. In contrast, LLMs are "model-free": for example, they do not "know" that they are being trained on their own outputs, and so they cannot take advantage of this extra knowledge when learning from data. Hence, it is no surprise that they collapse.

The third challenge was Pugh's representational one. How is it that a gauguine can be simple enough to be represented by itself, and yet not so simple that it cannot represent itself? Here, the answer is that self-representation need not be explicit. The gauguine knows the *space* of possible probabilistic programs, which—though very large—is succinctly

expressed via <guesser>. Observing and conditioning on evidence allows the gauguine to then *implicitly* represent which of the programs in <guesser> it is, without having that representation ever *explicitly* baked into its self-description. (See Piantadosi et al. [2012] for a longer discussion of this issue, in the context of how children might bootstrap an understanding of natural numbers without an innate number system.)

4 Conclusion

Even considering the analysis in the previous section, there was of course no guarantee that g would *actually* work. There was always the possibility that while the gauguine was theoretically sound, in practice the guesses would be overwhelmingly likely to go down some degenerate path, perhaps as a consequence of some subtle statistical property of the system as a whole. Such things do happen. Consider the problem of Gambler's Ruin: while it is theoretically possible to double your money with repeated bets, it is infinitesimally unlikely, and in practice you will always be bankrupt. But that is not what happened with our gauguine.

I am glad for this outcome. If we take our gauguine to be a metaphor for human self-understanding, then it gives me hope, at least on an existence-proof basis, that the Question is answerable by humanity. We may stumble around along the way—and it is true, the history of science is littered with questionable theories about what it means to be human—but there is reason to hope that our cultural ratchet turns towards wisdom, and that generation by generation we really will come to better understand ourselves.

As for me, well, I am still sitting on that musty old couch. Six months ago I was reflecting on my journals, which led me to reflect on my reflections, which in turn led me to write this essay reflecting on reflection-on-reflection. Now it is time to step back one more time. Is there anything to learn from reflecting on this reflection-on-reflection-on-reflection? I would like to think that there is. But I have written enough, here, for now. Let me pass the baton to you.

Acknowledgments

We thank Nada Amin, Maddy Bowers, Michael Brocidiacono, Liana Chow, Fernanda De La Torre, Matthew Pauly, and the anonymous reviewers for thoughtful comments as we revised this paper. KC additionally thanks the friends who supported him this spring.

This work was supported by the AFOSR (FA9550-22-1-0387), the ONR Science of AI program (N00014-23-1-2355), a Schmidt AI2050 Fellowship to JBT, the Siegel Family Quest for Intelligence at MIT, and NSF awards CCF-2313023 and CCF-2328543. Additionally, KC was supported by the Hertz Foundation and the NSF GRFP.

References

Harold Abelson and Gerald Jay Sussman. 1996. *Structure and interpretation of computer programs* (2nd ed.). The MIT Press.

Thomas Blanchard, Tania Lombrozo, and Shaun Nichols. 2018. Bayesian Occam's razor is a razor of the people. *Cognitive science* 42, 4 (2018), 1345–1359.

Jerry A Fodor. 1975. *The language of thought*. Vol. 5. Harvard university press.

Noah Goodman, Vikash Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. 2012. Church: a language for generative models. *arXiv preprint arXiv:1206.3255* (2012).

Noah D Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org. Accessed: 2025-4-22.

Alison Gopnik. 1996. The scientist as child. *Philosophy of science* 63, 4 (1996), 485–514.

Douglas R Hofstadter. 1979. *Gödel, Escher, Bach: an eternal golden braid*. Basic books.

Oleg Kiselyov and Chung-chieh Shan. 2009. Embedded probabilistic programming. In *IFIP Working Conference on Domain-Specific Languages*. Springer, 360–384.

Thomas S Kuhn. 1962. *The structure of scientific revolutions*. University of Chicago press Chicago.

Imre Lakatos. 1970. History of science and its rational reconstructions. In *PSA: Proceedings of the biennial meeting of the philosophy of science association*, Vol. 1970. Cambridge University Press, 91–136.

Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. 2015. Human-level concept learning through probabilistic program induction. *Science* 350, 6266 (2015), 1332–1338.

Tracey Mills, Josh Tenenbaum, and Samuel Cheyette. 2023. Human spatiotemporal pattern learning as probabilistic program synthesis. *Advances in Neural Information Processing Systems* 36 (2023), 54354–54365.

Steven T Piantadosi, Joshua B Tenenbaum, and Noah D Goodman. 2012. Bootstrapping in a language of thought: A formal model of numerical concept learning. *Cognition* 123, 2 (2012), 199–217.

George Pugh. 1977. The biological origins of human values. (1977).

Joshua S Rule, Joshua B Tenenbaum, and Steven T Piantadosi. 2020. The child as hacker. *Trends in cognitive sciences* 24, 11 (2020), 900–915.

Bertrand Russell. 1903. *Principles of mathematics*.

Ilia Shumailov, Zakhar Shumaylov, Yiren Zhao, Nicolas Papernot, Ross Anderson, and Yarin Gal. 2024. AI models collapse when trained on recursively generated data. *Nature* 631, 8022 (2024), 755–759.

Joshua Tenenbaum. 1998. Bayesian modeling of human concept learning. *Advances in neural information processing systems* 11 (1998).

Ken Thompson. 1984. Reflections on trusting trust. *Commun. ACM* 27, 8 (1984), 761–763.

Alan Turing et al. 1936. On computable numbers, with an application to the Entscheidungsproblem. *Journal of Math* 58, 345-363 (1936), 5.

Received 2025-07-11; accepted 2025-08-11