Reasoning about Reasoning by Nested Conditioning: Modeling Theory of Mind with Probabilistic Programs

A. Stuhlmüller[a], N. D. Goodman[b]

[a]*Department of Brain and Cognitive Sciences, Massachusetts Institute of Technology*
[b]*Department of Psychology, Stanford University*

Abstract

A wide range of human reasoning patterns can be explained as conditioning in probabilistic models; however, conditioning has traditionally been viewed as an operation *applied to* such models, not *represented in* such models. We describe how probabilistic programs can explicitly represent conditioning as part of a model. This enables us to describe reasoning about others' reasoning using *nested* conditioning. Much of human reasoning is about the beliefs, desires, and intentions of other people; we use probabilistic programs to formalize these inferences in a way that captures the flexibility and inherent uncertainty of reasoning about other agents. We express examples from game theory, artificial intelligence, and linguistics as recursive probabilistic programs and illustrate how this representation language makes it easy to explore new directions in each of these fields. We discuss the algorithmic challenges posed by these kinds of models and describe how Dynamic Programming techniques can help address these challenges.

1. Introduction

Reasoning about the beliefs, desires, and intentions of other agents—*theory of mind*—is a central part of human cognition and a critical challenge for human-like artificial intelligence. Reasoning about an opponent is critical in competitive situations, while reasoning about a compatriot is critical for cooperation, communication, and maintaining social connections. A variety of approaches have been suggested to explain humans' theory of mind. These include informal approaches from philosophy and psychology, and formal approaches from logic, game theory, artificial intelligence, and, more recently, Bayesian cognitive science. Many of the older approaches neglect a critical aspect of human reasoning—uncertainty—while recent probabilistic approaches tend to treat theory of mind as a special mechanism that cannot be described in a common representational framework with other aspects of mental representation. In this paper, we discuss how probabilistic programming, a recent merger of programming languages and Bayesian statistics, makes it possible to concisely represent complex multi-agent reasoning scenarios. This formalism, by representing reasoning itself as a program, exposes an essential contiguity with more basic mental representations.

Email addresses: ast@mit.edu (A. Stuhlmüller), ngoodman@stanford.edu (N. D. Goodman)

Probability theory provides tools for modeling reasoning under uncertainty: distributions formalize agents' beliefs, conditional updating formalizes updating of beliefs based on evidence or assertions. This approach can capture a wide range of reasoning patterns, including induction and non-monotonic inference. In cognitive science, probabilistic methods have been very successful at capturing aspects of human learning and reasoning [33]. However, the fact that conditioning is an operation *applied to* such models and not itself *represented in* such models makes it difficult to accommodate full theory of mind: We would like to view reasoning as probabilistic inference and reasoning about others' reasoning as inference about inference; however, if inference is not itself represented as a probabilistic model we cannot formulate inference about inference in probabilistic terms.

Probabilistic programming is a new, and highly expressive, approach to probabilistic modeling. A probabilistic program defines a stochastic generative process that can make use of arbitrary deterministic computation. In probabilistic programs, conditioning itself can be defined as an ordinary function within the modeling language. By expressing conditioning as a function in a probabilistic program, we represent knowledge about the reasoning processes of agents in the same terms as other knowledge. Because conditioning can be used in every way an ordinary function can, including composition with arbitrary other functions, we may easily express nested conditioning: we can condition any random variable, including random variables that are defined in terms of other conditioned random variables. Nested conditioning describes reasoning about reasoning and this makes theory of mind amenable to the kind of statistical analysis that has been applied to the study of mental representation more generally.

The probabilistic program view goes beyond other probabilistic views by extending compositionality from a restricted model specification language to a Turing-complete language, which allows arbitrary composition of reasoning processes. For example, the multi-agent influence diagrams proposed by Koller and Milch [18] combine the expressive power of graphical models with the analytical tools of game theory, but their focus is not on representing knowledge that players' might have about other players' reasoning.

From a philosophical perspective, the nested conditioning approach supports the idea that the mental machinery underlying theory of mind is not necessarily a specialized module. Instead, it can be expressed in, and potentially acquired from, more general primitives for representing the world and means of composing these primitives. Indeed, the primitives needed to express conditioning as a probabilistic program are quite elementary: control structure, random choice, and recursive functions. This suggests that representations needed for complex theory of mind may be more elementary than often thought—though constructing and reasoning with these representations can be tricky.

The probabilistic programs we use here are essentially declarative probabilistic knowledge: they describe representations of (social) knowledge and inferences that follow, but they do not describe the process of inference itself. That is, as models of human cognition, the models we present here should be seen as a computational-level descriptions [21]—a normative abstraction that helps understand what kind of problem the human mind solves in social reasoning, not a proposal for a particular process or strategy used to solve these problems. Indeed, there are significant challenges to even computing the distributions implied by these models. We suggest algorithms for addressing the practical problems of modeling without intending to make a strong algorithmic claim about human cognition.

In the following, we first present background on computational modeling using probabilistic programs, then show examples of how programs with nested conditioning concisely express reasoning about agents in game theory, artificial intelligence, and linguistics. We then describe the challenges in computing the predictions of these models. Finally, we sketch a generic Dynamic Programming inference algorithm for probabilistic programs and explain how it can help address some of these practical challenges.

2. Representing distributions as probabilistic programs

A probabilistic program is a program in a universal programming language with primitives for sampling from probability distributions, such as Bernoulli, Gaussian, and Poisson. Execution of such a program leads to a series of computations and random choices. Probabilistic programs thus describe models of the stochastic generation of results, implying a distribution on return values. In our examples, we use Church [13], a probabilistic programming language based on the stochastic lambda calculus. This calculus is universal in the sense that it can be used to define any computable discrete probability distribution [6] (and indeed, continuous distributions when encoded via rational approximation).

Church is a close relative of the functional programming language Scheme [1]. In this language, function application is written in prefix notation:

```
(+ 3 2)  →  5
```

The same applies to conditionals:

```
(if (> 3 2) true false)  →  true
```

Functions are first-class values and can be defined using λ. For example,

```
(λ (x) (* x 2))  →  <function object>
```

refers to a function that doubles its argument. Values can be bound to variables using `define` and, for explicit scope, with `let`:

```
(let ([y 3]) (+ y 4))  →  7
```

For function definitions,

```
(define (double x) (* x 2))
```

is short for:

```
(define double (λ (x) (* x 2)))
```

The random primitive `(flip` p`)` samples from a Bernoulli distribution: it returns `true` with probability p, `false` with probability $1-p$. By composing random primitives such as `flip` with deterministic computation, we can build complex distributions. For example, the expression

```
(sum (repeat 5 (λ () (if (flip .5) 0 1))))
```

induces the Binomial distribution shown in Figure 1. The meaning of a complex Church program can be understood via its *sampling semantics*: a single execution returns a sample from the distribution defined by the program; conversely, the histogram of (many) samples from the program defines its distribution on return values.

A Church program describes knowledge with uncertainty, and can be used to capture many effects of reasoning under uncertainty. As an example of reasoning under uncertainty, consider the following situation, depicted in Figure 2: We are presented with three

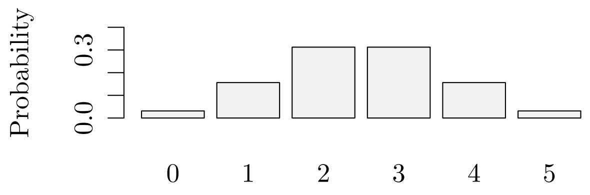


Figure 1: A Binomial(5, .5) distribution.

opaque urns, each of which contains some unknown number of red and black balls. We do not know the proportion of red balls in each urn, and we don't know how similar the proportions of red balls are between urns, but we have reason to suspect that the urns could be similar, as they all were filled at the same factory. We are asked to predict the proportion of red balls in the third urn (1) before making any observations, (2) after observing 15 balls drawn from the first urn, 14 of which are red, and (3) after observing in addition 15 balls drawn from the second urn, only one of which is red.

Intuitively, observing the first 15 balls, almost all of which are red, has two plausible explanations: either all urns have a high proportion of red balls, or all urns have differing proportions of red balls and this particular urn happens to have a high proportion of red balls. We don't know which of these is right and therefore have not learned about the overall bias of the urns yet, but we can predict a higher proportion of red balls for urn 3. After we observe in addition 15 balls drawn from urn 2, almost all of which are black, it becomes unlikely that all urns have similar proportions, hence our predicted proportion of red balls for urn 3 goes down again and we predict that the urns are biased towards very high and very low proportions of red balls. Notice that this reasoning is non-monotonic: our degree of confidence that "urns mostly contain red balls" goes up with the first evidence, but decreases with the second evidence.

Figure 3 shows how to model this situation in Church. Figure 4 shows the predictions derived from the model for conditions 1-3. In Church, conditional distributions are defined using the `query` operator, which takes as arguments a list of definitions describing a generative model, a query expression, and a condition, and which returns a sample from the conditional distribution. (This `query` syntax is much like the standard mathematical notation for conditionals, $P(q|c)$, but makes the model explicit.) The predictions of the model in Figure 3 match the non-monotonic intuition sketched above. More generally, a wide range of reasoning patterns are naturally modeled as conditioning in probabilistic models, including explaining away, screening off, and Occam's razor [see 14].

In sharp contrast to less expressive modeling languages, and traditional statistical notation, conditioning is not a special primitive in Church. Figure 5 shows how to define a generative model that samples from a conditional distribution. The procedure `joint` draws samples from a prior distribution. The predicate `condition?` takes a sample and returns true if the condition of interest holds. The function `rejection-query` simply keeps on drawing samples from the prior until it encounters a sample that satisfies the

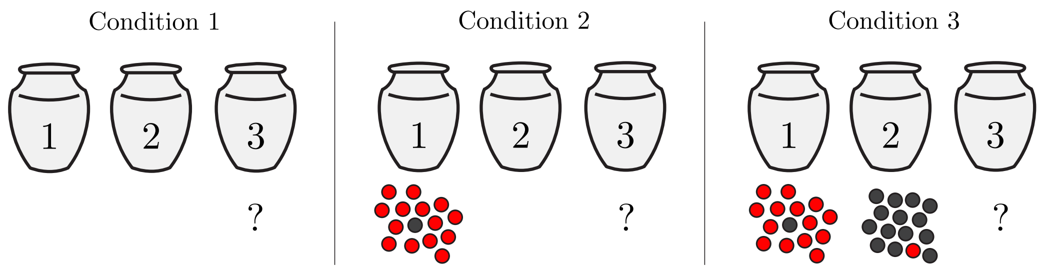


Figure 2: A simple scenario that requires reasoning under uncertainty: Drawing balls from urns, we want to guess for each condition (1) whether and how the urns are biased overall and (2) the likely proportion of red balls for the third urn.

condition, in which case it returns the sample. The critical difference from simpler languages is the ability to "keep sampling until": marrying random choice with universal computation enables *stochastic recursion*, which makes it simple to define the operation of conditioning. Of course, drawing conditional samples using `rejection-query` is very inefficient when the probability of satisfying the condition is low. However, we can distinguish model specification from the process that is actually used to *infer* the distribution implied by the model. From this point of view `rejection-query` is an elegant *definition* of the distribution of interest, while the *implementation* we will use to compute this distribution remains unspecified. Thus the practical problem of inference is the problem of efficiently computing the marginal distribution of a probabilistic program, i.e., its distribution on return values.

3. Modeling theory of mind as nested conditioning

In the previous section we briefly sketched the probabilistic programming approach to capturing uncertain knowledge. We described how the operation of conditioning itself could be represented as an ordinary function in Church, `query`. This representation opens up an intriguing possibility: we can nest a call to the `query` function within other calls to this function. If we view `query` as capturing reasoning, then such nested models will capture reasoning about reasoning. We now illustrate how this approach can be used to capture multi-agent reasoning, using examples from game theory, linguistics, and artificial intelligence.

3.1. Schelling coordination games

As a first illustration of our approach to theory of mind, consider a coordination game of the kind discussed by Schelling [28]: Two agents, Alice and Bob, want to meet, and they share the common knowledge that there are two possible meeting locations, one of them slightly more popular than the other. Each agent is modeled as making an inference about where it would be best to go. This can be formalized as a conditional distribution: "my location conditioned on my partner choosing the same location." Since each agent starts with the bias that the other agent is more likely to go to the more popular place, they are yet more likely to go to this place themselves. This reasoning

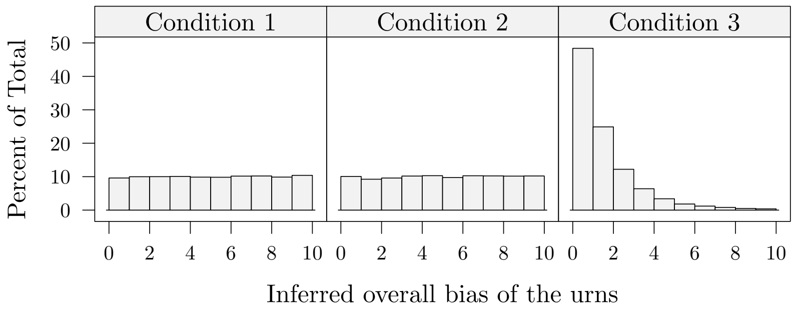
```
(query

  ;; model
  (define bias (uniform 0 10))
  (define red-bias (uniform 0 bias))
  (define black-bias (- bias red-bias))
  (define urn->proportion-red
    (mem
     (λ (urn)
       (first (beta (+ .4 red-bias) (+ .4 black-bias))))))
  (define (sample-urn urn)
    (if (flip (urn->proportion-red urn))
        'red
        'black))

  ;; query expression
  (urn->proportion-red 3)

  ;; condition
  (equal? (repeat 15 (λ () (sample-urn 1)))
          '(red red red red red red red red
            red red red red red red black)))
```

Figure 3: A hierarchical model of urn draws in Church. This program formalizes reasoning about condition 2 of the urn scenario in Figure 2, implemented as conditional sampling using the `query` operator. It takes as arguments a model (given as a sequence of definitions), an expression of interest, and a condition (an expression that evaluates to `true` or `false`). This model first samples a bias, splits the bias into red and black, and then defines how the urns' proportions of red balls depend on the biases. The function `urn->proportion-red` is *memoized* such that it always returns the same proportion when asked about the same urn. By contrast, the function `sample-urn` always flips a new coin to determine whether to return a red or black ball, with the coin weighted by the given urn's proportion of red balls. Given this model, we sample the proportion of red balls in the third urn conditioned on observing 1 black and 14 red draws from the first urn.



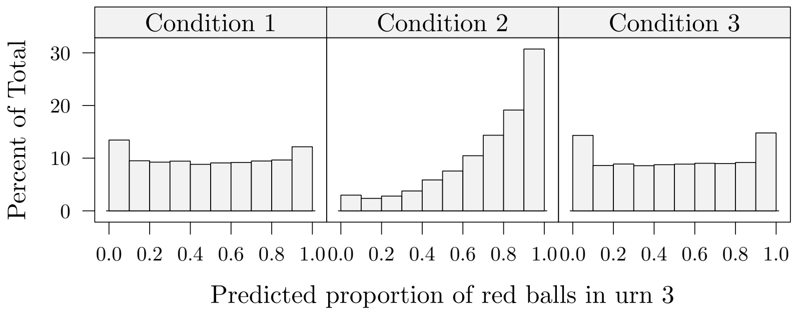


Figure 4: Urn scenario predictions derived from the Church model in Figure 3. Before seeing any draws, the model predicts uniform uncertainty about the overall bias, and mostly uniform uncertainty about the proportion of red balls in urn 3 (with extreme values being a bit more likely). After seeing 15 balls drawn from urn 1, 14 of which are red, the model still cannot judge how much the urns are biased, since it has seen only draws from a single urn. However, it would predict that, if the urns are biased, they are biased towards red, and therefore predicts that the proportion of red balls is likely to be high for urn 3. After seeing an additional 15 balls from urn 2, 14 of which are black, the model predicts that it is likely that the urns do not have directional bias, and predicts that the balls in urn 3 are most likely to be either all red or all black, but with significant probability on other proportions, since we have seen only two urns.

```
(define (rejection-query joint condition?)
  (let ([sample (joint)])
    (if (condition? sample)
        sample
        (rejection-query joint condition?))))
```

Figure 5: The conditioning operator `query` can be defined as a Church function. It takes as arguments `joint`, a stochastic function without arguments that samples from a prior distribution, and `condition?`, a function that checks whether a given sample satisfies the condition of interest. It calls itself recursively until it finds a sample that satisfies the condition, which it then returns. The `query` syntax used in the remainder of the paper—which takes as arguments a model, query expression, and condition—can be turned into a call to `rejection-query` using a simple syntactic transform.

```
(define (sample-location)
  (if (flip .55)
      'popular-bar
      'unpopular-bar))

(define (alice depth)
  (query
   (define alice-location (sample-location))
   alice-location
   (equal? alice-location (bob (- depth 1)))))

(define (bob depth)
  (query
   (define bob-location (sample-location))
   bob-location
   (or (= depth 0)
       (equal? bob-location (alice depth)))))
```

Figure 6: A Schelling coordination game in Church. Two agents, Alice and Bob, want to meet. They choose which bar to go to by recursively reasoning about one another.

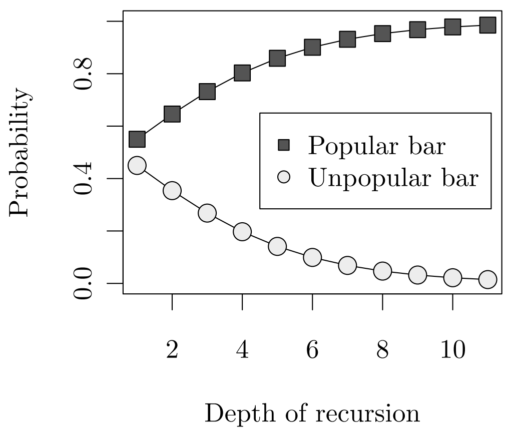


Figure 7: As the depth of recursive reasoning increases, the agents' actions converge to a focal point in the Schelling coordination game shown in Figure 6.

proceeds recursively, increasing the chance that the agents will go to the popular place. Each stage of this recursion is a conditional distribution.

This is an instance of *planning as inference*: we transform the problem of finding high-utility actions into the problem of computing a conditional distribution. The problem of maximizing expected utility can generally be translated into a likelihood maximization problem [34]. We are interested in modeling agents which choose only approximately optimally. This can be achieved using softmax-optimal decision-making, where an action a is chosen in proportion to its exponentiated expected utility under a belief distribution $P(s)$, i.e., $P(a) \propto \exp\left(\alpha \mathbb{E}_{P(s)}[U(a;s)]\right)$. We follow this approach throughout this paper, however we simplify by using a single-sample approximation to estimate the expected utility, and assume a shared prior belief distribution, except where we note otherwise.

Figure 6 shows how to use the planning-as-inference idea to formalize recursive reasoning in the Schelling coordination game as a Church program. The parameter `depth` controls the number of levels of recursive reasoning. Figure 7 shows how the marginal distribution changes as a function of this parameter. As the depth increases, the agents' actions converge on the focal point of the game—they always go to the popular location.

This illustrates a common pattern when modeling agents using probabilistic programs, namely the use of goal predicates instead of utility functions. It is possible to express choice in proportion to utility explicitly (by conditioning on a coin flip with a weight corresponding to the exponentiated utility of the outcome), but it is often equivalent to directly condition on the desired outcome (goal). This results in concise models that are intuitively appealing, since they match how we tend to think about our plans: not in terms of an explicit ordering on outcomes, but as aimed at achieving particular goals.

Uncertainty, including uncertainty about other players' uncertainty, can also be mod-

```
(define (speaker access state depth)
  (query
   (define sentence (sentence-prior))
   sentence
   (equal? (belief state access)
           (listener access sentence depth))))

(define (listener speaker-access sentence depth)
  (query
   (define state (state-prior))
   state
   (if (= 0 depth)
       (sentence state)
       (equal? sentence
               (speaker speaker-access state (- depth 1))))))
```

Figure 8: Pragmatic reasoning in language understanding as recursively nested conditioning. For the full model specification, see appendix.

eled using the standard tools of game theory, but the strengths of these tools lie elsewhere (e.g., amenability to analysis with respect to equilibria). In a discussion of games with incomplete information, Kreps [20] writes:

> "If we wanted something really complex, we could imagine that player 3 has an assessment concerning player 2's assessment of player 3's assessment of player 1's utility function. (If you think this is painful to read, imagine having to draw the corresponding extensive form.)"

Probabilistic programs make it easy to concisely express facts about other players' state of mind. For example, the coordination game can easily be modified to capture false beliefs (see appendix): Bob believes that Alice wants to meet him, and Alice knows this, but in fact Alice wants to avoid Bob. The compositional nature of probabilistic programs allows anything that can be expressed on its own to become the subject of others' reasoning, including agents' beliefs and reasoning steps. This is key for accurately modeling the interaction between agents who can represent other agents as intelligent reasoners.

3.2. Language understanding

Communication can be seen as a special case of decision-making in a multi-agent context: a speaker chooses utterances given an intended interpretation, and a listener chooses an interpretation given an utterance. We build on the *rational speech-act* theory of language understanding [8]: listeners model speakers as choosing their utterances approximately optimally based on social goals such as conveying information. Listeners then interpret utterances by "inverting" this model using Bayesian inference and drawing inferences about, among other things, the state of the world that caused the speaker to choose this utterance. This can again be seen as an instance of planning as inference, if we model the choice of utterances and interpretations as samples from conditional distributions.

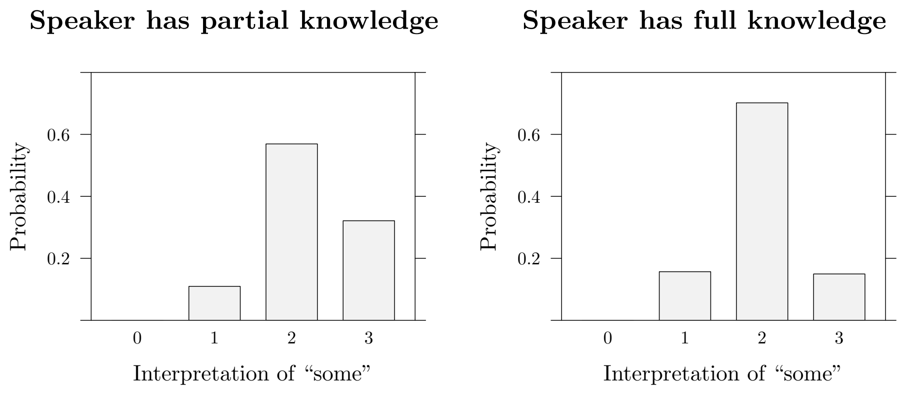


Figure 9: Predictions of the language understanding model shown in Figure 8. When the speaker has knowledge of the state of all 3 objects, the listener interprets "some" as likey to imply "not all". When the speaker has access to only 2 out of 3 objects, this implicature is cancelled and "some" is compatible with referring to all objects.

The basic tenet of pragmatics is that listeners sometimes make inferences that differ from those that directly follow from the literal meaning of an utterance. Consider the sentence "some of the apples are red." In general, we take the speaker to mean that not *all* of the apples are red. This effect is called a scalar implicature [16]. The rational speech-act theory predicts that such effects are sensitive to the listener's beliefs about the speaker's knowledge of the state of the world. For example, if there are three apples in total, and if the speaker has only seen one of the apples (and it was red), then the speaker's utterance "some of the apples are red" does not imply that not all of them are red. This effect has been confirmed experimentally in Goodman and Stuhlmüller [12].

Figure 8 shows one way to model speaker and listener in this situation. The speaker chooses a sentence conditioned on the listener inferring the intended state of the world when hearing this sentence; the listener chooses an interpretation conditioned on the speaker selecting the given utterance when intending this meaning. After a few iterations (determined by the `depth` parameter), this mutual recursion bottoms out and the speaker interprets the utterance literally, i.e., the speaker chooses an intended state conditioned on the given sentence being true of this state. Figure 9 shows model predictions that mirror the implicature-cancelling effect found in experiments—the probability that all three apples are red, after hearing "some of the apples are red," is much less when the speaker has seen all the apples.

The predicted interaction between the listener's interpretation and the speaker's knowledge does not depend on the particular choice of words and intended meanings, but is a more general result of the fact that the speaker is modeled as choosing words based on expected utility. We can easily replace the set of words used in the model to derive predictions about pragmatic inferences in other contexts. We can also change the shared background knowledge to derive predictions for interpretations in other contexts, all without changing the core model of language understanding. This is an example of the particular kind of modularity that results from representing agents and their mental

```
(define (exp-utility outcome player)
  (cond [(win? player outcome) 1.0]
        [(draw? outcome) 0.1]
        [else 0.01]))

(define (sample-action state player)
  (query
   (define action (action-prior state))
   (define outcome (sample-outcome state action player))
   action
   (flip (exp-utility outcome player))))

(define (sample-outcome state action player)
  (let ([next-state (transition state action player)])
    (if (terminal? next-state)
        next-state
        (let ([next-player (other-player player)])
          (sample-outcome next-state
                          (sample-action next-state next-player)
                          next-player)))))

(define start-state
  '((0 o 0)
    (o x x)
    (0 o 0)))

(sample-action start-state 'x)
```

Figure 10: Tic-tac-toe in Church. Each player chooses actions by sampling an action, simulating the game until the end, and choosing actions in proportion to how likely they are to lead to a successful outcome at the end of the game. This mental simulation includes reasoning about both players' reasoning at all future game steps, including their reasoning about the other player. For the definition of functions such as `action-prior` and `transition`, see appendix.

content as functional probabilistic programs: A priori, different model parts are independent and can thus easily be replaced with alternatives. A posteriori, i.e., as the result of conditioning, nontrivial dependencies between different model elements can arise, such as those between speaker's knowledge and listener's interpretation.

3.3. Game playing

Theory of mind is an essential part of playing games such as chess and Go. Each player chooses moves depending on their beliefs about the other players, including their beliefs about the other players' beliefs. As in the Schelling coordination game, this kind of reasoning can be modeled using the planning-as-inference transformation: actions are chosen conditional on ultimately leading to a successful outcome. Whereas the Schelling coordination game is a one-shot setting, many games require sequential decision-making.

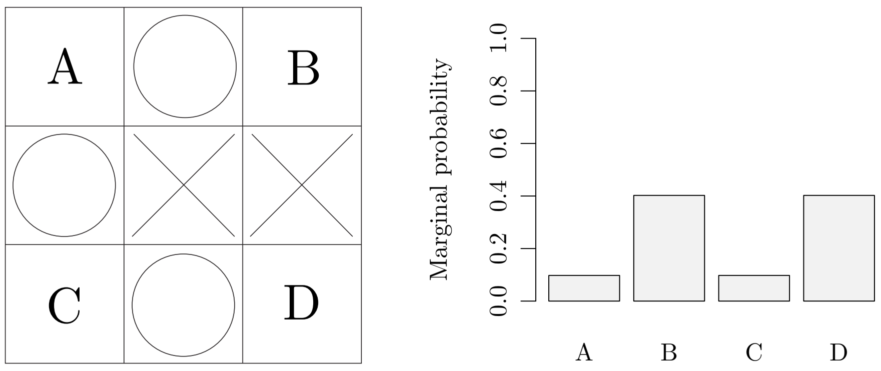


Figure 11: Where should **X** move next? This plot shows the predictions derived from the model in Figure 10. B and D are more likely to lead to a successful outcome, therefore they are more likely to be chosen.

Moves made in one turn depend on the expected consequences for subsequent turns, and on each player's decision-making strategy in subsequent turns.

As a simple example of a two-player game with sequential decision-making, consider Tic-tac-toe. Figure 10 shows the core of a Church implementation of a player. Figure 11 shows a simple prediction by this model that matches our intuitions: in a situation where **X** is not directly threatened and where **X** has two possible types of moves, one of which opens up a double threat, the forking move will be chosen, as it allows **X** to win two turns from now.

The player implemented in Church recursively reasons about the future moves of another player who implements the same strategy. In the first round, player X samples an action conditioned on the game that starts with player X taking this action ultimately leading to a win for player X. Sampling from this conditioned distribution requires a sample from another conditioned distribution, namely from the distribution on player O's action in round two, which is conditioned on player O ultimately winning. This in turn depends on player X's action in round three, and so on, until the game tree bottoms out at a win for one of the players or at a draw.

The functions `sample-action` and `sample-outcome` make no mention of Tic-tac-toe in particular—they are a fully generic implementation of approximately optimal decision-making in a setting where two players take turns. By supplying different implementations of functions such `action-prior` and `transition`, we can use the same framework to model other games. This suggests that, to some extent, representation of players and games can be studied independently, with interesting predictions arising from their interaction. For example, from a psychological perspective, one can ask whether and when a player model that judges actions by sampling a single outcome matches empirical data, and when a more strongly optimizing model is more accurate.

Representing sequential decision-making in games as probabilistic programs naturally lends itself to interesting extensions. Instead of modeling both players as identical thinkers, we can model their particular strategic tendencies, their potentially approximate evaluation of game states, their beliefs about who they are playing, and their process

```
(define (agent t raised-hands others-blue-eyes)
  (query
   (define my-blue-eyes (if (flip baserate) 1 0))
   (define total-blue-eyes (+ my-blue-eyes others-blue-eyes))
   my-blue-eyes
   (and (> total-blue-eyes 0)
        (! (λ () (= raised-hands (run-game 0 t 0 total-blue-eyes)))
           2))))

(define (get-raised-hands t raised-hands true-blue-eyes)
  (+ (sum-repeat (λ () (agent t raised-hands (- true-blue-eyes 1)))
                 true-blue-eyes)
     (sum-repeat (λ () (agent t raised-hands true-blue-eyes))
                 (- num-agents true-blue-eyes))))

(define (run-game start end raised-hands true-blue-eyes)
  (if (>= start end)
      raised-hands
      (run-game (+ start 1)
                end
                (get-raised-hands start raised-hands
                   true-blue-eyes)
                true-blue-eyes)))
```

Figure 12: Church implementation of a stochastic version of the blue-eyed islanders puzzle. For the full specification, see appendix.

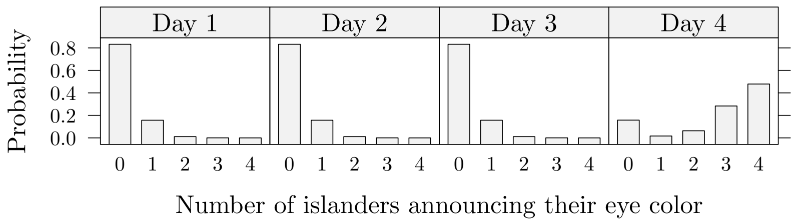
of learning about the other player from past moves. This could be used, for instance, to build computer opponents that reason about the human player's state of mind and that use sophisticated tactics such as "misleading the player," not because such tactics are built-in, but as a consequence of rational planning with a model that represents the human player's model of the situation. This would have relatively little effect in a game like Tic-tac-toe, but an enormous effect in games like Go and Poker.

3.4. Induction puzzles

Induction puzzles are an instance of multi-agent reasoning that goes beyond the two-player case. Typically, induction puzzles describes a scenario involving multiple agents that are all assumed to go through similar reasoning steps. In such a scenario, the outcome can usually be determined inductively by first solving a simple case, then assuming that all agents know the solution to this simple case, which in turn makes another case simple to solve for all agents.

We consider a stochastic version of the *blue-eyed islanders* puzzle (also known as the *muddy children* and *cheating husbands* problem), a well-known problem in epistemic logic [9, 31]. The setup is as follows: There is a tribe on a remote island. Out of the n people in this tribe, m have blue eyes. Their religion forbids them to know their own eye color, or even to discuss the topic. Therefore, everyone sees the eye color of every other islander, but does not know their own eye color. If an islander discovers their eye

‘At least one of you has blue eyes.’



‘At least one of you has blue eyes and a twitchy hand.’

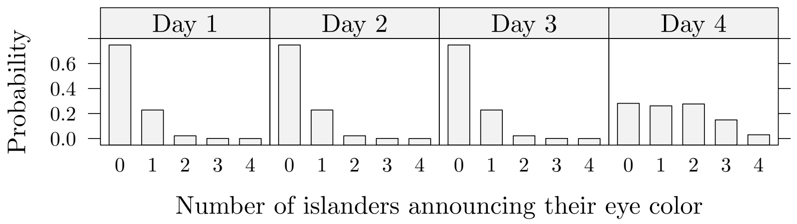


Figure 13: Model predictions for a stochastic version of the blue-eyed islanders puzzle with population size 4, all islanders blue-eyed. Four days after the foreigner makes his announcement, the islanders are likely to realize that they have blue eyes. However, if the foreigner (truthfully) states that one of the blue-eyed islanders has a twitchy hand and mistakenly announces that she has blue eyes 10% of the time, this inference becomes much less pronounced.

color, they have to publicly announce this the next day at noon. All islanders are highly logical. One day, a foreigner comes to the island and—speaking to the entire tribe—he truthfully says: "At least one of you has blue eyes." What happens next?

Intuitively, the solution is as follows. If there is only one islander with blue eyes, the islander will see that no other person has blue eyes and will announce their knowledge the next day. If no islander does so the next day, then everyone knows that there are at least two islanders with blue eyes. Since each of the two islanders with blue eyes only observes one other blue-eyed islander, they can deduce that they must have blue eyes themselves, and so the two blue-eyed islanders announce their eye colors on the second day. Generalizing this line of reasoning, all m blue-eyed islanders announce their eye color on the m-th day.

Figure 20 shows a formalization of a stochastic version of this puzzle. We have converted a multi-agent sequential planning problem into an inference problem that uses nested conditioning to model recursive reasoning. In this model, we have increased the optimization strength of action selection in softmax-optimal sampling (α) by conditioning on two successes instead of one, thus moving part of the way from probability matching to utility maximization.

Figure 13 shows the corresponding model predictions for population size 4 when all islanders are blue-eyed: on the 4th day, it is most likely that all islanders decide to announce their eye color. In this version, the probability of an islander making the announcement is proportional to their estimate for how likely they are to have blue eyes.

The induction that happens as a result of this conversion differs from the backwards-induction in the previous section. In playing games, the current action is chosen based on reasoning about the likely future outcome of the game, which depends on modeling the other agents' *future* reasoning. In this induction puzzle, the current action is chosen based on current beliefs ("do I have blue eyes or not?"), which are the result of explaining past observations ("how many islanders have made the announcement?") by reasoning about other agents' *past* reasoning.

One of the main attractions of probabilistic programming as a modeling framework is that it is easy to rapidly prototype complex probabilistic models, since implementations of probabilistic programming languages provide *generic* inference algorithms; it is not necessary to design model-specific algorithms. This is highly useful in multi-agent scenarios, where complex interdependencies between agents' conditional inferences can make it difficult to "see" what the outcome of a model change is. For example, what if the foreigner had said to the islanders: "At least one of you has blue eyes, and raises their hand by accident 10% of the time." Changing the model to account for this requires one additional line of code (see appendix). The predictions are shown in Figure 13. It still takes four days for the islanders to realize with some uncertainty what their eye color is, but each islander is less certain and therefore less likely to make the announcement.

The idea of common knowledge is central to both versions of this induction puzzle. Before the foreigner makes his announcement, every islander already knows that there is at least one other islander with blue eyes simply by observing the others' eye colors. By speaking in front of the entire tribe, the foreigner causes this fact to become common knowledge: now, everyone knows that everyone knows this fact, and everyone knows that everyone knows that everyone knows, and so on. This idea is reflected in the Church program by the condition `(> total-blue-eyes 0)`. This condition applies both to the islander whose reasoning is modeled, to his mental model of other islanders' reasoning, to the mental models of their mental models, and so on. If we remove this condition, the effect of all islanders inferring their eye color on the m-th day goes away. By representing this multi-agent reasoning scenario as a probabilistic program, we have formalized a hypothesis about what it means to have common knowledge, and we are able to explore the implications of this hypothesis.

4. Inference algorithms

Probabilistic programs that use stochastic recursion to express nested conditioning can represent a wide and flexible space of multi-agent reasoning, as described above, but it can be difficult to compute the predictions of these models. We now explain the challenges of computing the distributions defined by models with nested conditioning, and then sketch a Dynamic Programming algorithm that addresses some of these challenges.

4.1. Multiply-intractable distributions

Given a probabilistic program, our goal is to estimate its distribution on return values. This is often easy for probabilistic programs without conditioning (i.e., without `query`). For instance, when the expected program runtime is small, we can estimate the distribution on return values using samples we get by repeatedly running the program. However, when a program contains recursion the time it takes to run the program can grow large even for small programs. Programs that contain a single level of conditioning implemented by rejection sampling already require in expectation $1/p$ recursive calls when we condition on an event with probability p. Since it is common to condition on low-probability events, this is often infeasible. For this reason, many algorithms for approximately sampling from conditional distributions have been developed, including Markov Chain Monte Carlo (MCMC), importance sampling, and variational methods.

Instead of directly attempting to sample from a program's distribution on return values, consider the goal of sampling from a program's distribution on executions. An execution corresponds to a complete sequences of random choices and determines a return value. For an unconditioned program, the probability of an execution is the product of all random choices that occur within this execution, and thus is easy to compute. For a conditioned program, the probability of an execution is only proportional to this product, since the condition rules out some executions, redistributing their mass on the "allowed" executions. Computing this probability exactly is often intractable, as it requires integrating over all program executions. Many algorithms for approximate inference require only the unnormalized probability (i.e. the probability up to an unknown normalizing constant). However, in the setting of nested conditioning, even the unnormalized probability can be difficult to compute.

For example, consider the program in Figure 14. This program could model the following situation: Two agents play a game in which each agent needs to name a number between 0 and 9 and they win if their numbers add up to 13. The first player knows this, and he knows that the second player gets to see the number the first player chooses, but the second player mistakenly thinks that the two win if their numbers add up to any number greater than 8 (and the first player knows this as well). What number should the first player choose?

```
(query
  (define a (sample-integer 10))
  (define b
    (query
      (define c (sample-integer 10))
      c
      (> (+ a c) 8)))
  a
  (= (+ a b) 13))
```

Figure 14: A simple program with nested conditioning.

In this game, we can write the probability of a program state as follows:

$$\begin{aligned} p(a, b | a + b = 13) &= \frac{p(a)p(b|a)\delta_{a+b=13}(a, b)}{\sum_{a',b'} p(a')p(b'|a')\delta_{a'+b'=13}(a', b')} \\ &\propto p(a)p(b|a)\delta_{a+b=13}(a, b) \end{aligned}$$

Here, $\delta_{a+b=13}(a, b)$ is the delta function that returns 1 if $a + b = 13$ is satisfied, otherwise 0. The distribution $p(b|a)$ is itself defined in terms of a conditional distribution q:

$$p(b|a) = q(b|a, a + b > 8) = \frac{q(b|a)\delta_{a+b>8}(a, b)}{\sum_{b'} q(b'|a)\delta_{a+b'>8}(a, b')}$$

Making use of the fact that $p(a)$ and $q(b|a)$ are uniformly distributed, the *unnormalized* probability of a program state is

$$p(a, b|a + b = 13) \propto \frac{\delta_{a+b=13}(a, b)}{\sum_{b'} \delta_{a+b'>8}(a, b')}.$$

Intuitively, this means that the probability of a state (a, b) is inversely proportional to the number of assignments b' that the second player could have chosen to make the sum $a + b'$ greater than 8, since each such assignment reduces the chance that the second player chooses the assignment that makes $a + b = 13$ true. The first player is best off choosing $a = 4$, such that for the second player, the goal of making their sum greater than 8 coincides as much as possible with the goal of making their sum equal to 13.

From the perspective of inference, the relevant insight is that computing the unnormalized probability requires us to sum over the state space of the inner query. That is, even the unnormalized probability of the outer query depends on the normalizing constant of the inner query. While this is easy to compute for the given toy problem, it is typically difficult and makes inference in models with nested queries challenging.

To generalize this line of reasoning, assume that we are interested in sampling from a distribution

$$p(y|c_1) = \frac{p(y)\delta_{c_1}(y)}{\int p(y)\delta_{c_1}(y)\,\mathrm{d}y} \propto p(y)\delta_{c_1}(y).$$

Here, y is a program state, c_1 is a condition on that state, $p(y)\delta_{c_1}(y)$ is the unnormalized probability of y, and $Z_y = \int p(y)\delta_{c_1}(y)\,\mathrm{d}y$ is the normalization constant.

Suppose that the distribution on program states $p(y)$ factors as follows:

$$p(y) = p(y_1, y_2) = p(y_1)p(y_2|y_1)$$

Assume further that $p(y_2|y_1)$ is defined in terms of a conditional distribution itself, i.e., we are describing a distribution defined in terms of a query within a query[1]:

$$\begin{aligned} p(y_2|y_1) &= q(y_2|y_1, c_2) \\ &= \frac{q(y_2|y_1)\delta_{c_2}(y_2)}{\int q(y_2|y_1)\delta_{c_2}(y_2)\,\mathrm{d}y_2} \end{aligned}$$

Then, the *unnormalized* probability of a state y is:

$$\begin{aligned} p(y|c_1) &\propto p(y_1)p(y_2|y_1)\delta_{c_1}(y) \\ &= \frac{p(y_1)q(y_2|y_1)\delta_{c_2}(y_2)\delta_{c_1}(y)}{\int q(y_2|y_1)\delta_{c_2}(y_2)\,\mathrm{d}y_2} \end{aligned}$$

Note that the denominator depends on y_2 and thus on y. It affects the relative probabilities of different y and cannot be ignored, even if we are only interested in the unnormalized probability. As a consequence, evaluation of unnormalized $p(y|c_1)$ requires integrating over the domain of y_2, which is often intractable. Since inference methods such as MCMC are used to approximately sample from distributions that are "intractable" in the sense that we cannot compute the normalization constant, models with nested conditioning are called *doubly-intractable*.

Learning the parameters of a graphical model is an example of such a problem: for each parameter setting, we need to integrate over the space of all explanations of the data in order to compute a value proportional to the likelihood of the data. While auxiliary variable techniques can make sampling tractable in special cases [23], Murray and Ghahramani [22] conjecture that for general undirected models, there exists no tractable MCMC scheme that results in the correct stationary distribution.

Most of the examples we have seen define a nesting of conditional distributions of depth greater than 2. For such *multiply-intractable* distributions, the difficulty of evaluating f increases exponentially in the depth. If we assume that $q(y_2|y_1)$ factors into an unconditioned and a conditioned distribution, and if we reason analogously to the steps above, we conclude that naive computation of the unnormalized probability of a single program execution y requires us to solve a multiple integral, thwarting existing methods for approximate inference.

4.2. Dynamic Programming

Since existing methods of approximate inference are intractable for models with nested query, we instead pursue exact inference methods that exploit shared subcomputation to perform tractable inference for modestly sized state spaces. For example, Figure 15 shows that the inference time for the language understanding model presented in Section 3.2

[1]Note that in the term $q(y_2|y_1, c_2)$ there is an ambiguity: y_1 is a parameter to this function, while c_2 is a condition. This is a common ambiguity in probability notation that is clarified in probabilistic program notation.

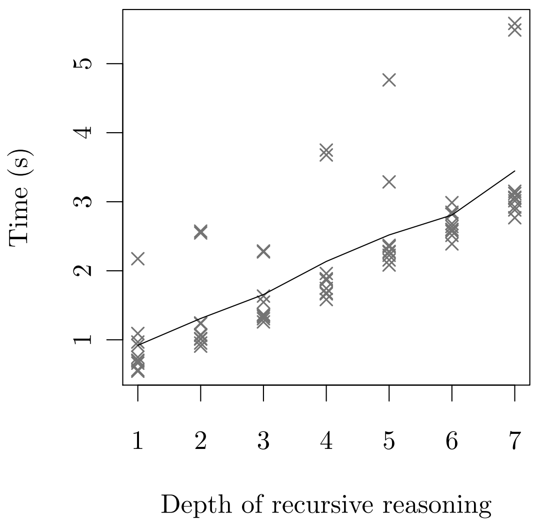


Figure 15: Increase in dynamic programming inference time for the scalar implicature model (Figure 8) as a function of nested conditioning depth. Each point on the plot corresponds to a run of our algorithm on the model with a given depth.

grows linearly in the depth of nested conditioning, whereas expected inference time for rejection, and for MCMC with rejection for the inner queries, would grow exponentially. All of the model predictions shown in Section 3 have been computed using this algorithm. We present this dynamic programming algorithm as a practical tool for modeling, not as an hypothesis about human processing; the structural insights this algorithm leverages, however, may prove useful in future exploration of the processes of social cognition. We next sketch this inference technique; for more technical background see Stuhlmüller and Goodman [30].

4.2.1. Approach

Our starting point is this: we are given an interpreter for a probabilistic programming language and a probabilistic program, and we want to compute the marginal distribution of this program. In other words, we want to compute the distribution on return values that we would sample from if we executed the program using this interpreter.

We assume that the interpreter is functional and defined recursively, i.e., in the process of evaluating a particular program, it calls itself and thus reduces the overall problem of evaluation to the problem of evaluating a number of smaller objects. Each of these subproblems has a unique distribution on solutions, and in the process of marginalization, some subproblems may occur multiple times. Our algorithm is based on the idea that we can detect repeated, identical subproblems and that we can then solve each subproblem only once, reusing the results for future occurrences of the same problem.

In Church, subproblems correspond to subexpressions with environments. For example, computing the marginal distribution of the Church program `(and (flip) (or` `(flip) (flip)))` involves computing the distribution of `(flip)` and of `(or (flip) (flip))`. In the setting of nested conditioning, the problem of computing the marginal distribution of a query that contains other queries may be such that the same inner query can be reused for many different settings of the parameters of the outer query.

To understand what makes marginalization with reuse of computation challenging, compare to simplified versions of the problem. If we were trying to ensure reuse of repeated subcomputations for a deterministic interpreter, we could simply memoize it. For a stochastic language, we could imagine a similar approach: use the interpreter to recursively compute and cache the distribution for each unique interpreter call, computing all subdistributions required by a call before we compute the distribution of the call itself. However, consider the program shown in Figure 16. In order to compute the distribution of the first `if`-branch of `(game true)`, we need to know the distribution of `(game (not true))`, but this in turn depends on the distribution of `(game true)`. These *self-recursive* dependencies make direct caching impossible, as it would lead to infinite regress. A similar pattern is present in queries that are defined via rejection sampling. Our algorithm addresses this challenge by first transforming the program into an intermediate structure that makes these dependencies explicit, then computing the marginal distribution from this structure in a way that is sensitive to the dependencies.

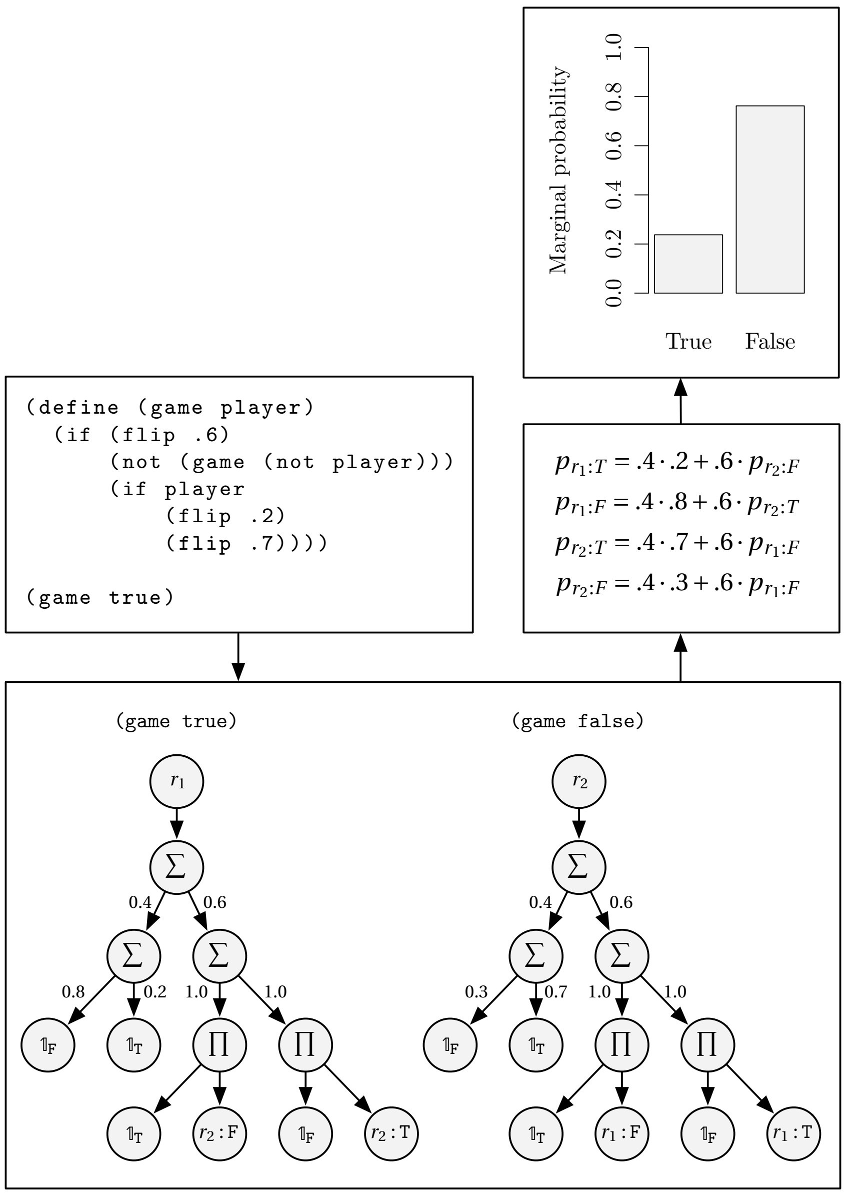
We first describe the interface the interpreter needs to satisfy such that we can use it to acquire sufficient information about the program to build our intermediate structure. We then present this structure and the steps of the algorithm.

By default, an interpreter takes a program and evaluates it, finally returning a single value. In order to control the evaluation process in a way that lets us acquire information about the structure of a program's distribution, we require that the interpreter is a *coroutine* that interrupts its evaluation and returns its current state whenever it (1) makes a recursive subcall, (2) samples a primitive random choice, and (3) returns a terminal value. Each of these cases will correspond to a particular way of building structure in our intermediate representation.

For our intermediate representation, we desire that it factors out all "deterministic" computation, leaving only probability calculations, i.e., sums and products, and that it makes explicit the dependencies between the distributions resulting from subcomputations. Sum-product networks [26] satisfy the former requirement. We extend the formalism to *factored* sum-product networks in order to satisfy the latter: A *factored sum-product network* (FSPN) over variables $x_1, \dots, x_d$ is defined as a directed graph with a uniquely labeled root node r. The internal nodes are sums and products. The leaves are indicators $x_1, \dots, x_d$ and $\bar{x}_1, \dots, \bar{x}_d$, and reference nodes $(y, \vec{x})$, where y is another node and $\vec{x}$ a vector of indicator values. Each edge (i, j) from a sum node i has a non-negative weight w_{ij}.

Let $\mathrm{Ch}(y)$ denote the children of node y. The value $\mathcal{V}(y, \vec{x})$ of a node y is defined as $\sum_{z \in \mathrm{Ch}(y)} w_{yz} \mathcal{V}(z, \vec{x})$ if y is a sum, as $\prod_{z \in \mathrm{Ch}(y)} \mathcal{V}(z, \vec{x})$ if y is a product, as $\mathbb{1}_{\vec{x}_j = x_j}$ if y is an indicator x_j, and as $\mathcal{V}(z, \vec{w})$ if y is a reference $(z, \vec{w})$. We denote the factored sum-product network F as a function of the indicator variables $\vec{x} = (x_1, \dots, x_d, \bar{x}_1, \dots, \bar{x}_d)$ by $F(\vec{x}) = \mathcal{V}(r, \vec{x})$. For any given $\vec{x}$, the value of the FSPN is the solution to the system of equations $F(\vec{x})$ (if a unique solution exists).

We use FSPNs to describe marginal distributions and indicators to query the probabilities of marginal values. To simplify notation, we write indicators as $\mathbb{1}_v$, denoting the x_i



```
(define (game player)
  (if (flip .6)
      (not (game (not player)))
      (if player
          (flip .2)
          (flip .7))))

(game true)
```

$$p_{r_1:T} = .4 \cdot .2 + .6 \cdot p_{r_2:F}$$
$$p_{r_1:F} = .4 \cdot .8 + .6 \cdot p_{r_2:T}$$
$$p_{r_2:T} = .4 \cdot .7 + .6 \cdot p_{r_1:F}$$
$$p_{r_2:F} = .4 \cdot .3 + .6 \cdot p_{r_1:F}$$

Figure 16: An example of applying the Dynamic Programming algorithm to a simple Church program. The program is compiled to a factored sum-product network, which directly corresponds to a system of equations that can be solved to infer the marginal distribution.

corresponding to value v. We write reference nodes as $r : v$, denoting $(r, [0, 0, \dots, 0, 1, 0, \dots, 0])$, where the only indicator entry that is 1 corresponds to value v. Negated indicators $\bar{x}_i$ are not used by our algorithm.

In the following, we refer to "root nodes" in addition to the node types introduced above. These nodes always have exactly one child and can hence be of either type sum or product. The networks we build are sets of trees. Each node will be associated with a unique (ancestor) root node. Root nodes will correspond to subproblems. One of these root nodes is the root node referred to in the definition of a FSPN; it corresponds to the overall problem of computing the marginal distribution of the starting state of the given probabilistic program. In the illustration of the process our algorithm uses to build a FSPN, we depict in addition to the previously built node n_{prev} and the weight of the edge from this node to the current node, w_{prev}, the root node r that is associated with the previous and current node.

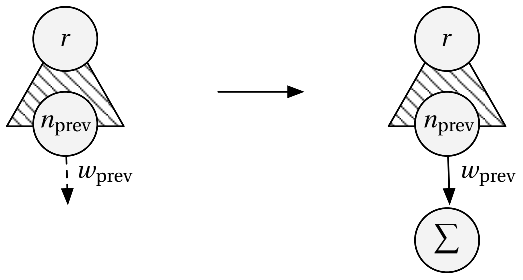
4.2.2. Algorithm

The algorithm proceeds in three steps. Figure 16 shows the result of applying each of the steps to a simple self-recursive program. For pseudocode, see the appendix.

1. Compile program to FSPN.

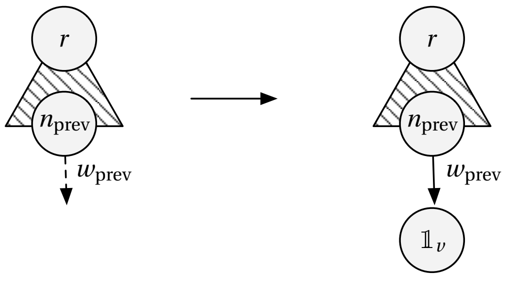
We initialize the FSPN with a single node r. Our algorithm maintains a queue of tasks, each of which is a tuple of a thunk f (a function without arguments), a previous node n_{prev}, and an edge weight w_{prev} (a probability). The queue is initialized to a task for the first interpreter call, $(\lambda.\mathcal{I}(s), r, 1)$. While the queue is not empty, the algorithm takes the first task in the queue and evaluates the function call $f()$. There are three types of values that this function call can return: random choices, terminal values, and subcalls. We manipulate FSPN and queue depending on the type of this value:

A **random choice** is a tuple $(c, \vec{v}, \vec{p})$ of a continuation c, a list of values $\vec{v}$, and a list of probabilities $\vec{p}$. The continuation c is a formal object that captures the current state of the computation; it can be called with a value to resume evaluation. $\vec{v}$ and $\vec{p}$ specify the distribution of the random choice where we interrupted evaluation. We modify the FSPN by connecting a sum node $\sum$ to n_{prev} using w_{prev}:

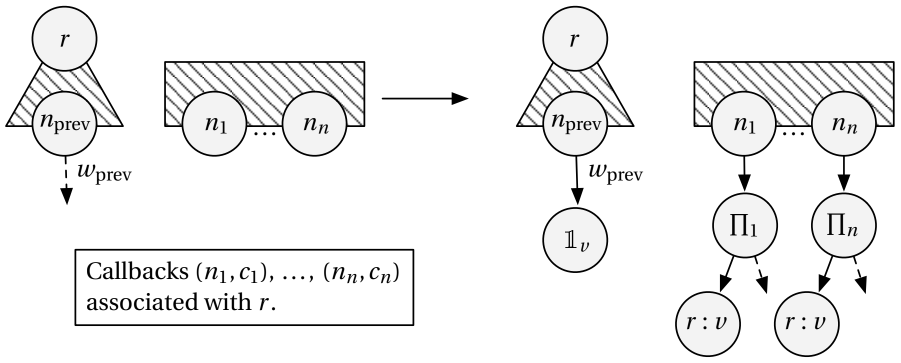


For each value-probability pair, we then add to the queue the task of exploring $c(v)$ with previous node $\sum$ and edge weight p. This reflects the fact that the marginal probability of any value under the current subproblem r is the sum of the probabilities of this value being returned for each of the ways of proceeding from the current program state, weighted by the probability of choosing each such way.

A **terminal value** v is treated depending on whether it is a new value for current root node r or whether it has been encountered before. If it has been encountered before, we simply add an indicator node for this variable to the FSPN, reflecting that for a program state that directly returns value v, the marginal probability is 1 for this value and 0 otherwise:

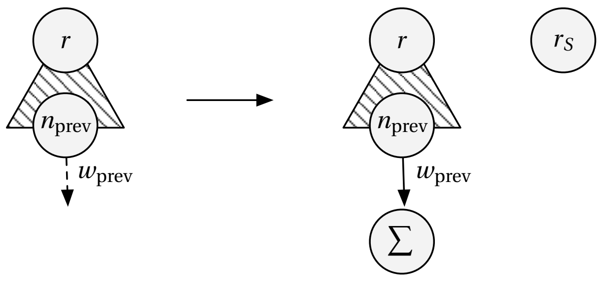


If we encounter v for the first time under root node r, this means that we just learned something about the set of support values of the subproblem corresponding to r. If this subproblem is used elsewhere, new parts of the program's state space just became accessible to us. To store where a subproblem is used, we associate a list of *callbacks* with each root node. A callback is a pair of a node n (the node corresponding to the program state that referred to r's subproblem) and a continuation c (for proceeding from that program state, given a return value for r's subproblem). Given a new value v, we can build graph structure and store a queue entry for each such callback, reflecting that it is possible to continue from n with the marginal probability of v under r. For every callback (n_i, c_i), we add a product node $\prod_i$ and a reference node $r : v$ under node n_i, and add to the queue the task of exploring $c_i(v)$ with previous node $\prod_i$:



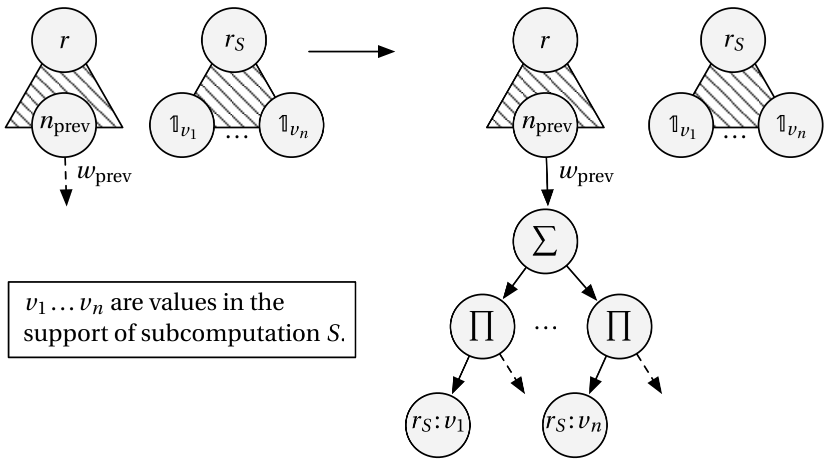
A **subcall** is a pair of a continuation c and an interpreter argument s. This is where we share repeated subcomputations: depending on whether we have seen s before or not, we proceed to explore this subcall and build graph structure, otherwise we simply refer to existing structure.

If we encounter s for the first time, we connect a sum node $\sum$ to n_{prev}, reflecting that the probability of any marginal value under r will be a sum of the probability of this value under all ways of continuing using different return values from subproblem s, weighted by the probability of these return values. We also add a new root node r_S to the graph which represents the subproblem s and its marginal distribution:



We add to the queue the task of exploring the interpreter call $\mathcal{I}(s)$ with previous node r_S, and add $(\sum, c)$ to the callbacks of r_S such that graph structure under $\sum$ will be built if marginal values for $\mathcal{I}(s)$ are found.

If we have seen the interpreter argument s before, we also add a sum node $\sum$ under n_{prev}, but then look up the existing root node r_S for subproblem S. If marginal values for this subproblem are already known, we can immediately build graph structure under $\sum$ that refers to the marginal probability of these values, and we can explore the current continuation c using these values. For each known return value v_i supported by S, we build a product node $\prod_i$ and reference node $r_S : v_i$ under $\sum$ and add to the queue the task of exploring $c(v_i)$ with previous node $\prod$.



As before, we also add $(\sum, c)$ to the callbacks of r_S in order to continue building graph structure under $\sum$ if more marginal values of r_S are found in the future.

This process terminates once the queue is empty. Alternatively, it is possible to bound the graph size, which results in lower bounds on the true marginal probabilities.

2. Convert FSPN to equations.

Following the value equations given in Section 4.2.1, we can directly convert a factored sum-product network into a system of polynomial equations. In addition, it is sometimes possible to reduce the time spent in the next step by applying a simple substitution-based equation simplifier.

3. Solve equations to get marginal distribution.

For probabilistic programs, the generated system of equations tends to be sparse, reflecting the fact that most interpreter calls that occur within the possible executions of a program do not depend on most other calls. We therefore cluster the equations into strongly connected components and solve the clusters of equations in topological order. Computing a topological order of strongly connected components is linear in the size of the graph [32]. By solving in topological order we know that all probabilities required to compute the solution of a component have been computed once we reach this component. To solve components, we use fixed-point iteration and Newton's method. Exploring the use of other solution methods is a potential venue for future performance improvements.

In sum, the method described here involves creating an intermediate representation, a FSPN, from a probabilistic program, which can then be efficiently solved to compute the required distribution. This method often makes nested query models useful in practice: we have used it for all of the examples above, and further evaluate it in comparison to other algorithms in Stuhlmüller and Goodman [30].

5. Discussion

Our approach to theory of mind is based on a long history of both informal and computational accounts of theory of mind [e.g., 35, 15, 5, 11]. For an outline of existing logical and probabilistic approaches from the perspective of building cognitive architectures, see Bello [4]. While much of the relevant work on cognitive architectures aims to take into account resource constraints of the agents that are modeled, this is not our goal; we have presented tools for *computational-level modeling* of agents' reasoning in multi-agent scenarios. Such models constitute a normative standard that can be compared to actual patterns in human reasoning, which in turn may help us devise process-level descriptions of the same phenomena.

Most directly, we build on the Bayesian approach to theory of mind [2, 3], which takes the *principle of rational action* to be central to the concept of intentional agency: all else being equal, agents are expected to choose actions that satisfy their goals as effectively as possible. Given a probabilistic generative model of an agent's planning process that has this property, and given observations, Bayesian theory of mind simply proposes that we can infer the hidden internal states of the agent—beliefs, desires, and goals—by inverting this model, i.e., by conditioning. By viewing conditioning as a function that is represented in a probabilistic program, we have presented a simple, unified formalism for implementing Bayesian theory of mind, and for extending it to settings such as the recursive reasoning scenarios presented in this paper.

Philosophically, our approach suggests that it may be possible to acquire the mental machinery for theory of mind from simpler primitives such as control structure, random choice, and recursive functions. Specialized mental modules—such as a "belief box" and a "desire box" [24]—might arise simply as a consequence of an agent's attempt to explain her observations in a parsimonious way, or might not turn out to be necessary to explain our cognitive capabilities in the first place.

The proposed inference algorithm is related to and inspired by a long tradition of algorithms which use dynamic programming to exploit reusable structure in the natural language processing, logic programming, and functional programming literatures. For example, it is known that, in general, exactly solving problems such as marginalization for arbitrary recursive programs leads to systems of nonlinear equations [see, e.g., comments in 7]. Klein and Manning [17] exploit strongly connected components of the computation graph for PCFGs to perform efficient exact marginalization in a way similar to the present algorithm. It is beyond the scope of this paper to review the many connections with individual algorithms presented in the literature. Instead, we focus on three systems which attempt to use dynamic programming to provide general inference algorithms for universal, probabilistic (or, more generally, weighted) programming languages: IBAL, PRISM, and Dyna.

The most closely related system to the present work is the functional programming language IBAL, a probabilistic variant of ML [25]. IBAL provides an exact marginalization algorithm for discrete probabilistic models, which is based on a generalization of variable elimination applied to computation graphs. The graph used by this algorithm also exploits sharable subcomputations across the evaluation of the probabilistic program. However, the present algorithm is more general than the IBAL algorithm in an important way. The IBAL algorithm relies on *acyclic* computation graphs; this is equivalent to the requirement that the computation be *evidence-finite* [19]—there must only be a finite number of computations which can give rise to the observed evidence. By contrast, our algorithm handles many cases of *evidence-infinite* computation. For example, the simple recursive program shown in Figure 16, which has finite support {`true`, `false`} but an infinite number of computations which give rise to each support value, cannot be marginalized by IBAL, but is correctly handled by our algorithm. Practical examples of such evidence-infinite computations include many of the nested-query models for multi-agent reasoning that we have described above.

Another system which is similar to the present work is PRISM, a probabilistic generalization of Prolog, which also makes use of dynamic programming to provide a general inference algorithm. Although PRISM is able to recover many standard algorithms for problems such as PCFG estimation (e.g., the inside-outside algorithm), like IBAL, it cannot handle evidence-infinite computations [27].

A somewhat different approach is Dyna [7], a programming language for expressing weighted deductive logic programs. Dyna makes use of generalizations of parsing-as-deduction [29] and semi-ring parsing [10] to compile weighted logic programs into highly optimized dynamic programs. Dyna differs from our algorithm in the target level of abstraction. Our algorithm is focused on the problem of rapid prototyping of models for which no standard dynamic programming algorithm exists. The programmer simply provides an interpreter, and our algorithm automatically exploits whatever sharing is exposed by the structure of the recursive calls made in the process of computing the marginal distribution for a particular model. By contrast, Dyna is a language for abstractly expressing *specific* dynamic programming *algorithms* and compiling these algorithms to highly efficient code. It allows the programmer lower-level control over algorithm specification, but it also *requires* the programmer to specify these algorithmic details.

6. Conclusion

We have described how probabilistic programs can represent multi-agent scenarios by modeling each agent's reasoning as conditional sampling. Since conditioning is an operation that can be defined *within* such models, it can become the subject of other agents' reasoning. This representation goes beyond existing formalisms, such as graphical models, where conditioning is more naturally seen as an operation that is *applied to* a model, but not itself represented. We have discussed inference challenges posed by this modeling technique and sketched a Dynamic Programming algorithm that can provide tractable inference for small state spaces by merging identical subproblems.

There are many research opportunities related to this way of studying theory of mind, both regarding modeling and algorithms. For modeling, one avenue is to translate existing models used in fields such as game theory and linguistics into the framework of probabilistic programs. Extensions of these models that are difficult to express using existing modeling languages, but that are natural using this new approach, will suggest themselves. Further, integration of diverse aspects of social cognition will be fostered by representing them in a common representational system—libraries of probabilistic program components for cognition can be naturally integrated into a more complete architecture.

At the algorithmic level, it seems likely that exact inference will not scale to models with realistic state spaces. This poses the practical question of finding approximate inference algorithms that work in the setting of nested queries, and the scientific question of understanding how humans cope with the same. For some versions of these problems, scalable algorithms have already been introduced; for example, Monte Carlo tree search for playing the board game Go. It may be possible to construct inference algorithms for probabilistic programs that reduce to such known algorithms in restricted settings. Conversely, studying the psychological process of inference—which may implicate parallel processing, simulation, and significant resource bounds—may suggest new algorithms to solve the engineering challenges of inference.

Viewing theory of mind as reasoning about reasoning and formalizing this as nested conditioning is appealing on both philosophical and scientific grounds. By realizing theory of mind in a uniform representational framework for reasoning under uncertainty, we expose essential assumptions and provide opportunities for constructing more complex, and realistic, models of social reasoning.

Acknowledgements

We thank John McCoy, Thomas Icard, and Lewis Fishgold for comments; Daniel Roy and Timothy O'Donnell for contributions to early stages of work on the proposed algorithm; and ONR grant N00014-09-0124 for funding.

References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA, USA, 2nd edition, 1996. ISBN 0262011530.

[2] C. Baker, J. Tenenbaum, and R. Saxe. Bayesian models of human action understanding. *Advances in Neural Information Processing Systems*, 18:99, 2006.

[3] C. L. Baker, R. R. Saxe, and J. Tenenbaum. Bayesian theory of mind: Modeling joint belief-desire attribution. *Proceedings of the Thirty-Second Annual Conference of the Cognitive Science Society*, 2011.

[4] P. Bello. Cognitive Foundations for a Computational Theory of Mindreading. *Advances in Cognitive Systems.*

[5] P. Bello and N. Cassimatis. Developmental accounts of theory-of-mind acquisition: Achieving clarity via computational cognitive modeling. *28th Annual Conference of the Cognitive Science Society, Vancouver, Canada*, 2006.

[6] U. Dal Lago and M. Zorzi. Probabilistic Operational Semantics for the Lambda Calculus. *RAIRO - Theoretical Informatics and Applications*, (46):413–450, 2012.

[7] J. Eisner, E. Goldlust, and N. A. Smith. Compiling comp ling: Practical weighted dynamic programming and the dyna language. In *Proceedings of Human Language Technologies and the Conference on Empirical Methods in Natural Language Processing (HLT/EMNLP)*, 2005.

[8] M. C. Frank and N. Goodman. Predicting Pragmatic Reasoning in Language Games. *Science*, 336 (6084):998–998, May 2012.

[9] G. Gamow and M. Stern. Puzzle-math, 1958.

[10] J. Goodman. Semiring parsing. *Computational Linguistics*, 25(4), 1999.

[11] N. Goodman, C. L. Baker, E. B. Bonawitz, V. K. Mansinghka, A. Gopnik, H. Wellman, L. Schulz, and J. Tenenbaum. Intuitive theories of mind: A rational approach to false belief. *Proceedings of the twenty-eighth annual conference of the cognitive science society*, pages 1382–1387, 2006.

[12] N. D. Goodman and A. Stuhlmüller. Knowledge and implicature: Modeling language understanding as social cognition. In *Proceedings of the Thirty-Fourth Annual Conference of the Cognitive Science Society*, 2012.

[13] N. D. Goodman, V. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: a language for generative models. *Proceedings of the 24th Conference in Uncertainty in Artificial Intelligence*, pages 220–229, 2008.

[14] N. D. Goodman, J. B. Tenenbaum, T. J. O'Donnell, and the Church Working Group. Probabilistic models of cognition, 2011. URL http://projects.csail.mit.edu/church/wiki/Church.

[15] A. Gopnik, A. N. Meltzoff, NetLibrary, and Inc. Words, thoughts, and theories. 1998.

[16] L. R. Horn. Implicature. In *The Handbook of Pragmatics.* Blackwell Publishing Ltd, Oxford, UK, Jan. 2006.

[17] D. Klein and C. D. Manning. An O(n^3) agenda–based chart parser for arbitrary probabilistic context–free grammars. Technical report, Stanford University, 2001.

[18] D. Koller and B. Milch. Multi-agent influence diagrams for representing and solving games. *Games and Economic Behavior*, 45(1):181–221, 2003.

[19] D. Koller, D. McAllester, and A. Pfeffer. Effective bayesian inference for stochastic programs. In *Proceedings of the National Conference on Artificial Intelligence*, pages 740–747, 1997.

[20] D. M. Kreps. A Course in Microeconomic Theory. pages 1–859, Oct. 2011.

[21] D. Marr. Vision: A computational investigation into the human representation and processing of visual information. *Henry Holt and Co., Inc. New York, NY, USA*, Jan 1982.

[22] I. Murray and Z. Ghahramani. Bayesian learning in undirected graphical models: approximate MCMC algorithms. *Proceedings of the 20th conference on Uncertainty in artificial intelligence*, pages 392–399, 2004.

[23] I. Murray, Z. Ghahramani, and D. MacKay. Mcmc for doubly-intractable distributions. *Proceedings of the 22nd Annual Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 359–366, 2006.

[24] S. Nichols and S. P. Stich. *Mindreading: An Integrated Account of Pretence, Self-Awareness, and Understanding Other Minds.* Oxford University Press, USA, Oct. 2003.

[25] A. Pfeffer. IBAL: A probabilistic rational programming language. In *Proceedings of the International Joint Conferences on Artificial Intelligence*, 2001.

[26] H. Poon and P. Domingos. Sum-product networks: A new deep architecture. *Proc. 12th Conf. on Uncertainty in Artificial Intelligence*, pages 337–346, 2011.

[27] T. Sato. Generative Modeling by PRISM. In P. Hill and D. Warren, editors, *Logic Programming*, volume 5649 of *Lecture Notes in Computer Science*, chapter 4, pages 24–35. Springer, Berlin, Heidelberg, 2009. ISBN 978-3-642-02845-8. doi: 10.1007/978-3-642-02846-5_4.

[28] T. C. Schelling. *The Strategy of Conflict.* Harvard University Press, 1960.

[29] S. M. Shieber, Y. Schabes, and F. C. N. Pereira. Principles and implementation of deductive parsing. *The Journal of Logic Programming*, 24(1-2):3–36, 1995.

[30] A. Stuhlmüller and N. D. Goodman. A dynamic programming algorithm for inference in recursive

probabilistic programs. 2012.

[31] T. Tao. *Poincaré's Legacies.* Pages from Year Two of a Mathematical Blog. Amer Mathematical Society, 2009.

[32] R. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2): 146–160, 1972.

[33] J. B. Tenenbaum, C. Kemp, T. L. Griffiths, and N. D. Goodman. How to grow a mind: Statistics, structure, and abstraction. *Science*, 331(6022):1279–1285, 2011.

[34] M. Toussaint, S. Harmeling, and A. Storkey. Probabilistic inference for solving (PO) MDPs. *Institute for Adaptive and Neural Computation*, 2006.

[35] H. M. Wellman. *The child's theory of mind.* The MIT Press, Oct. 1992.

Appendix

```
(define (sample-location)
  (if (flip .55)
      'popular-bar
      'unpopular-bar))

(define (alice* depth)
  (query
   (define alice*-location (sample-location))
   alice*-location
   (not (equal? alice*-location (bob (- depth 1))))))

(define (alice depth)
  (query
   (define alice-location (sample-location))
   alice-location
   (equal? alice-location (bob (- depth 1)))))

(define (bob depth)
  (query
   (define bob-location (sample-location))
   bob-location
   (or (= depth 0)
       (equal? bob-location (alice depth)))))

(alice* 5)
```

Figure 17: A version of the Schelling coordination game with false beliefs: Bob believes that Alice wants to meet him, and Alice knows this, but in fact Alice wants to avoid Bob.

```
(define (belief actual-state access)
  (map (λ (ac st pr) (if ac st (sample pr)))
       access
       actual-state
       (substate-priors)))

(define (baserate) 0.8)

(define (substate-priors)
  (list (λ () (flip (baserate)))
        (λ () (flip (baserate)))
        (λ () (flip (baserate)))))

(define (state-prior)
  (map sample (substate-priors)))

(define (sentence-prior)
  (uniform-draw (list all-p some-p none-p)))

(define (all-p state) (all state))
(define (some-p state) (any state))
(define (none-p state) (not (some-p state)))

(define (speaker access state depth)
  (query
   (define sentence (sentence-prior))
   sentence
   (equal? (belief state access)
           (listener access sentence depth))))

(define (listener speaker-access sentence depth)
  (query
   (define state (state-prior))
   state
   (if (= 0 depth)
       (sentence state)
       (equal? sentence
               (speaker speaker-access state (- depth 1))))))

(define (num-true state)
  (sum (map (λ (x) (if x 1 0)) state)))

;; without full knowledge:
(num-true (listener '(#t #t #f) some-p 5))

;; with full knowledge:
(num-true (listener '(#t #t #t) some-p 5))
```

Figure 18: A model for pragmatic inferences like “some implies not-all.”

```
(define (valid-move? move state)
  (equal? (list-ref (list-ref state (first move))
                    (second move))
          0))

(define (action-prior state)
  (query
   (define action (list (sample-integer 3) (sample-integer 3)))
   action
   (valid-move? action state)))

(define (transition state action player)
  (map (λ (row i)
         (map (λ (s j)
                (if (equal? (list i j) action)
                    player
                    s))
              row
              (iota (length row))))
       state
       (iota (length state))))

(define (diag1 state)
  (map (λ (x i) (list-ref x i))
       state (iota (length state))))

(define (diag2 state)
  (let ([len (length state)])
    (map (λ (x i) (list-ref x (- len (+ i 1))))
         state (iota len))))

(define (win? player state)
  (let ([check (λ (elts) (all (map (λ (x) (eq? x player)) elts)))])
    (any (map check
              (list (first state) (second state) (third state)
                    (map first state) (map second state) (map third state)
                    (diag1 state) (diag2 state))))))

(define (flatten lst)
  (apply append lst))

(define (terminal? state)
  (all (map (λ (x) (not (equal? x 0)))
            (flatten state))))

(define (other-player player)
  (if (eq? player 'x) 'o 'x))

(define (draw? outcome)
  (and (not (win? 'x outcome))
       (not (win? 'o outcome))))
```

Figure 19: Helper functions for the Tic-tac-toe game.

```
(define num-agents 4)

(define baserate .045)

(define (! thunk n)
  (if (= n 0)
      true
      (and (thunk) (! thunk (- n 1)))))

(define (%sum-repeat proc n s)
  (if (= n 0)
      s
      (%sum-repeat proc
                   (- n 1)
                   (+ s (proc)))))

(define (sum-repeat proc n)
  (%sum-repeat proc n 0))

(define (agent t raised-hands others-blue-eyes)
  (query
   (define my-blue-eyes (if (flip baserate) 1 0))
   (define total-blue-eyes (+ my-blue-eyes others-blue-eyes))
   my-blue-eyes
   (and (> total-blue-eyes 0)
        (! (λ () (= raised-hands (run-game 0 t 0 total-blue-eyes)))
           2))))

(define (get-raised-hands t raised-hands true-blue-eyes)
  (+ (sum-repeat (λ () (agent t raised-hands (- true-blue-eyes 1)))
                 true-blue-eyes)
     (sum-repeat (λ () (agent t raised-hands true-blue-eyes))
                 (- num-agents true-blue-eyes))))

(define (get-raised-hands/twitch t raised-hands true-blue-eyes)
  (+ (sum-repeat (λ () (agent t raised-hands (- true-blue-eyes 1)))
                 (- true-blue-eyes 1))
     (sum-repeat (λ () (agent t raised-hands true-blue-eyes))
                 (- num-agents (+ true-blue-eyes)))
     (if (flip .1) 1 (agent t raised-hands (- true-blue-eyes 1)))))

(define (run-game start end raised-hands true-blue-eyes)
  (if (>= start end)
      raised-hands
      (run-game (+ start 1)
                end
                (get-raised-hands start raised-hands
                   true-blue-eyes)
                true-blue-eyes)))

(run-game 0 4 0 4)
```

Figure 20: Church implementation of a stochastic version of the blue-eyed islanders puzzle. This code includes an alternative implementation of `get-raised-hands` for the setting where the foreigner announces (truthfully) that one of the blue-eyed islanders sometimes raises her hand by accident.

```
procedure BuildFSPN($\mathcal{I}$, $x_{\text{init}}$)
    $G$ = Graph()
    $r = G$.addNode(*root*)
    $Q = [(\lambda.\mathcal{I}(x_{\text{init}}), r, 1.0)]$
    terminals, callbacks, subproblem = {}, {}, {}
    while $Q$ is not empty do
        $(f, n_{\text{prev}}, w_{\text{prev}}) = Q$.pop()
        $x = f()$
        if $x$ is a value $v$ then
            $n_{\text{cur}} = G$.addNode(*indicator*, $v$)
            $r = G$.root[$n_{\text{prev}}$]
            if $v \notin$ terminals[$r$] then
                for all $(n', c)$ in callbacks[$r$] do
                    processTerminal($G$, $Q$, $r$, $v$, $n'$, $c$)
                end for
                terminals[$r$].add($v$)
            end if
        else if $x$ is a random choice $(c, \vec{v}, \vec{p})$ then
            $n_{\text{cur}} = G$.addNode(*sum*)
            for all $v, p \in \vec{v}, \vec{p}$ do
                $Q$.enqueue($\lambda.c(v)$, $n_{\text{cur}}$, $p$)
            end for
        else if $x$ is a subcall $(c, s)$ then
            $n_{\text{cur}} = G$.addNode(*sum*)
            if $s \notin$ subproblem.keys() then
                $r = G$.addNode(*root*)
                subproblem[$s$] = $r$
                $Q$.enqueue($\lambda.\mathcal{I}(s)$, $r$, 1.0)
            else
                $r$ = subproblem[$s$]
                for all $v \in$ terminals[$r$] do
                    processTerminal($G$, $Q$, $r$, $v$, $n_{\text{cur}}$, $c$)
                end for
            end if
            callbacks[$r$].add(($n_{\text{cur}}, c$))
        end if
        $G$.addEdge($n_{\text{prev}}$, $n_{\text{cur}}$, $w_{\text{prev}}$)
    end whilereturn G
end procedure

procedure ProcessTerminal($G$, $Q$, $n_{\text{root}}$, $v$, $n_{\text{prev}}$, $c$)
    $n_{\text{prod}} = G$.addNode(*product*)
    $n_{\text{ref}} = G$.addNode(*ref*, $n_{\text{root}}$, $v$)
    $G$.addEdge($n_{\text{prev}}$, $n_{\text{prod}}$, 1.0)
    $G$.addEdge($n_{\text{prod}}$, $n_{\text{ref}}$, 1.0)
    $Q$.enqueue($\lambda.c(v)$, $n_{\text{prod}}$, 1.0)
end procedure
```

Figure 21: Compiling probabilistic programs to factored sum-product networks.