Amortizing Pragmatic Program Synthesis with Rankings

Yewen Pu [1] **Saujas Vaduguru** [2] **Priyan Vaithilingam** [3] **Elena Glassman** [3] **Daniel Fried** [2]

Abstract

The usage of Rational Speech Acts (RSA) framework has been successful in building *pragmatic* program synthesizers that return programs which, in addition to being logically consistent with user-generated examples, account for the fact that a user chooses their examples informatively. We present a general method of amortizing the slow, exact RSA synthesizer. Our method first query the exact RSA synthesizer to compile a communication dataset. The dataset contains a number of example-dependent rankings of subsets of programs. It then distills a *single* global ranking of all programs as an approximation to every ranking in the dataset. This global ranking is then used at inference time to rank multiple logically consistent candidate programs generated from a fast, non-pragmatic synthesizer. Experiments on two program synthesis domains using our ranking method resulted in orders of magnitudes of speed ups compared to the exact RSA synthesizer, while being more accurate than a non-pragmatic synthesizer when communicating with humans. Finally, we prove that in the special case of synthesis from a single example, this approximation is exact.

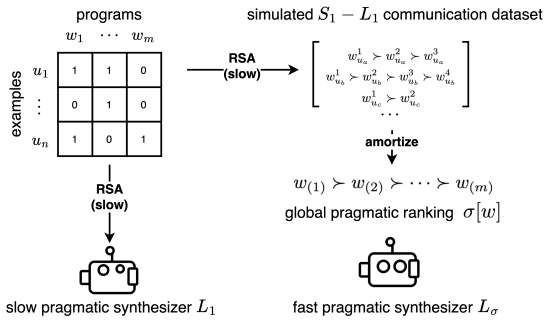


Figure 1. (left) Directly using the exact RSA algorithm in a pragmatic synthesizer L_1 is slow. (right) Our approach uses RSA to generate a simulated communication dataset between the informative speaker S_1 and the pragmatic synthesizer L_1, and stores the responses of L_1 as example-dependent rankings of subsets of programs. We then distill the dataset into a single example-agnostic global ranking of all programs $\sigma[w]$. This global ranking is then used to build a fast pragmatic synthesizer L_σ, by using the examples only to filter out consistent programs, then using the global ranking to sort them. This amortized synthesizer performs similar selections of programs as an exact RSA synthesizer, while being orders of magnetudes faster.

1. Introduction

For intelligent systems to be accessible to end users, it is important that they can infer the user's intent under ambiguity. Imagine a person asking an AI assistant to generate a regular expression that matches the string *123-7890*. It would be unhelpful if the AI assistant simply returned the regular expression Σ^* – the expression that matches *all* strings – although it is technically correct. The rational speech acts model (RSA) of pragmatics (Frank & Goodman, 2012) gives an algorithm for resolving ambiguities by modeling the user as a speaker that chooses *informative examples* for the system, via recursive Bayesian reasoning. Given several competing responses, for instance $\texttt{regex}_1$ = `\d{3}-\d{4}` and $\texttt{regex}_2 = \Sigma^*$, RSA would reason that it is more likely that an informative user would use the example *123-7890* to describe $\texttt{regex}_1$ over $\texttt{regex}_2$, allowing it to prefer the intended regex. Recent works (Pu et al., 2020; Vaithilingam et al., 2023) have leveraged the RSA algorithm to build pragmatic program synthesizers – interactive systems that take in user given examples (e.g. strings) and return programs (e.g. regexes) that are both logically consistent and take into account the informativity of the chosen examples. Their algorithm, which we refer to as `RSA`, is applicable to any program synthesis domain where programs can be efficiently enumerated (Feser et al., 2015; Solar-Lezama, 2008; Gulwani, 2011), and produces a pragmatic synthesizer which interacts well with humans, while requiring no labeled human data.

The `RSA` algorithm marginalizes across *all* possible examples (e.g. all strings) and programs (e.g. all regexes) multiple times. This makes it difficult to scale `RSA` to large

[*]Equal contribution [1]Autodesk AI Research [2]Carnegie Mellon University [3]Harvard SEAS. Correspondence to: Yewen Pu <yewen.pu@autodesk.com>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).

domains, where users expect the system to complete its inference in real-time. Prior works in scaling up RSA computation (Monroe et al., 2017; Andreas & Klein, 2016) have largely focused on sampling and re-ranking, curbing RSA's computation to a small subset of programs and examples. In this work, we show a simple yet effective way of **amortizing** RSA via a single global ranking of all programs. Rather than using RSA directly at inference time, our method uses it to generate training data in the form example-dependent rankings of subsets of programs. We then distill a global ranking from the training data, amortizing the computation of RSA (Figure 1). At inference time, a fast, non-pragmatic synthesizer is used to propose multiple logically consistent programs, and the global ranking is used to quickly rank them,[1] resulting in a pragmatic yet efficient synthesizer.

This work makes the following contributions. **(1)** We describe a general method of amortizing the RSA algorithm (considered in Cohn-Gordon et al. (2018b); Pu et al. (2020); Vaithilingam et al. (2023)) applicable to any pragmatic program synthesis domains. **(2)** Using global ranking, we scale the model proposed by Vaithilingam et al. (2023) to a larger domain while still allowing for real-time interaction. We conduct a small user study validating that end-users are more accurate communicating with a ranking based program synthesizer compared to a non-pragmatic one (+27%, +41% relative). **(3)** We conduct *simulated* user studies by replaying the human interactive synthesis data collected from Pu et al. (2020) and Vaithilingam et al. (2023). We confirm that our ranking-based synthesizer retains the communicative accuracy of RSA (55%, 92% respectively), while running orders of magnitudes(over 100 times) faster. **(4)** We prove that in the special case of synthesis from just a single example, RSA_single, a setting studied in the original RSA literature (Goodman & Frank, 2016; Vogel et al., 2013; Monroe & Potts, 2015; Smith et al., 2013), the approximation using a global ranking is exact.

2. Background on Pragmatic Synthesis

In this section, we provide background on a reference game framework of program synthesis, which affords building a *pragmatic synthesizer* that can infer a user's intended program from few examples (Pu et al., 2020). We illustrate this framework using a toy example from a small version of the regular expression domain of this work.

2.1. Synthesis as a Reference Game

Consider the problem where a user gives example strings to a synthesis system, and asks it to find a matching regular expression. This process can be modeled as a **reference game**

[1] In our example, the regex Σ^* would be ranked lower than other consistent programs.

M

	0{1}	0+1{1}	0{2}1+	0+1*
01	0	1	0	1
0	1	0	0	1
001	0	1	1	1
00	0	0	0	1

L_0

0{1}	0+1{1}	0{2}1+	0+1*
0	**0.5**	0	**0.5**
0.5	0	0	0.5
0	0.33	0.33	0.33
0	0	0	1

L_1

0{1}	0+1{1}	0{2}1+	0+1*
0	**0.74**	0	0.26
0.82	0	0	0.18
0	0.26	0.65	0.09
0	0	0	1

Figure 2. A boolean lexicon for a small reference game of regular expressions. The rows are the utterances (strings) and the columns are hypotheses (regexes), and each entry denotes if a string is consistent with a regex. The L_0 and L_1 matrices show conditional probabilities that would be inferred by a synthesizer performing literal and pragmatic inference respectively.

(Lewis, 1979), where a speaker (the user) chooses a few utterances (strings) to give to the listener (the synthesizer), with the intention that the listener can infer the correct hypothesis (regular expression). This reference game is characterized by the lexicon M, a boolean matrix of 1s and 0s (Figure 2). In M, each row corresponds to an utterance/example and each column corresponds to a hypothesis/program, and 1s indicating *consistency* of its corresponding utterance and a hypothesis: whether the program's output (e.g. deciding whether a regular expression matches a string) is consistent with the example (e.g. the string). As we can see, a given utterance (such as *001*) may be consistent with multiple hypotheses (0+{1}, 0{2}1+, and 0+1*).

2.2. A Literal Program Synthesizer

How might we build a system that takes an utterance (say *01*) and produces the intended hypothesis 0+1{1}? As *01* is consistent with multiple hypotheses (0+1{1} and 0+1*), a naive strategy is to treat all consistent hypotheses as equally likely, scaled by a prior distribution of hypotheses $P(w)$:

$$L_0(w|u) \propto P(w)M[u,w] \tag{1}$$

$$= P(w)\frac{M[u,w]}{\sum_{w'} M[u,w']} \tag{2}$$

A synthesizer built this way is a *literal listener* L_0 (Bergen et al., 2016). Assuming the prior $P(w)$ is uniform over programs, we can construct it by normalizing the rows of the matrix M, resulting in a probability distribution over hypotheses W given utterances u (Figure 2). As we can see, given the utterance *01*, this listener predicts an equal probability of 0+1{1} and 0+1* being the intended program.

2.3. A Pragmatic Synthesizer from a Single Example

A key insight to improving on the literal synthesizer is to consider that a user is cooperatively choosing an utterance to be *informative* about the intended program to the synthesizer. The Rational Speech Acts (RSA) framework models this

informative choice of utterances using recursive Bayesian reasoning (Frank & Goodman, 2012). By reasoning about why a speaker (user) might have chosen a particular utterance (examples), rather than possible alternatives, the listener (synthesizer) can disambiguate the hypothesis (program) to which the speaker was referring to. Formally, the RSA framework produces a chain of alternating listeners and speakers beginning with the L_0 model above.

$$\begin{array}{lllll} S_1(u|w) & \propto & L_0(w|u) & = & \frac{L_0(w|u)}{\sum_{u'} L_0(w|u')} \\ L_1(w|u) & \propto & S_1(u|w) & = & \frac{S_1(u|w)}{\sum_{w'} S_1(u|w')} \end{array} \tag{3}$$

Applying this framework amounts to normalizing the columns of the L_0 matrix to obtain a pragmatic speaker distribution S_1, then normalizing the rows of S_1 to obtain a pragmatic listener (synthesizer), L_1 (Figure 2). As we can see, given the utterance *01*, this listener prefers `0+1{1}` over `0+1*`, reflecting the reasoning that if the user wanted to refer to `0+1*`, they might have provided an example that highlights the possibility of no 1s in the string. In this paper, we shall call this algorithm `RSA_single`. As this algorithm only depends on M, it is applicable to all program synthesis domains where programs and examples can be effectively enumerated.

2.4. A Pragmatic Synthesizer from Multiple Examples

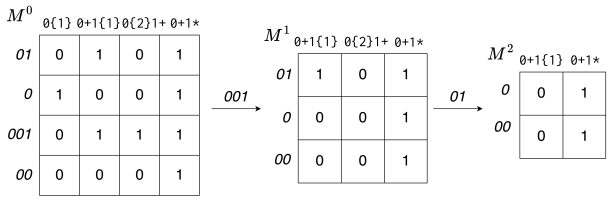


Figure 3. In the case of incremental `RSA`, the meaning matrix becomes smaller as more utterances are given, as each utterance rules out hypotheses that are inconsistent with it.

`RSA_single` is capable of producing a program synthesis algorithm from a single example. However, the users will typically have to clarify their intent *interactively*, by giving a sequence of multiple utterances $\mathbf{u} = u_1, u_2, \ldots, u_n$. The synthesizer must infer the intended program after every turn. With each new utterance, the meaning matrix M becomes smaller, as hypotheses inconsistent with the new utterance are ruled out (Figure 3). This is an instance of incremental RSA (Cohn-Gordon et al., 2018b), which models the informative speaker S_1 generating utterances auto-regressively:

$$\begin{aligned} S_1(\mathbf{u}|w) &= S_1(u_1, u_2, \ldots, u_n|w) \\ &= \prod_{i=1}^{n} S_1(u_i|w, u_1, \ldots, u_{i-1}) \\ &= \prod_{i=1}^{n} \frac{L_0(w|u_1, \ldots, u_i)}{\sum_{w'} L_0(w'|u_1, \ldots, u_i)} \end{aligned}$$

In essense, the S_1 is the product of multiple single-utterance S_1 computed on separate meaning matrixes (like those in Figure 3). The synthesizer $L_1(w|\mathbf{u})$ is defined recursively on top of S_1, $L_1(w|\mathbf{u}) \propto S_1(\mathbf{u}|w)$.

Pu et al. (2020) builds on top of the incremental RSA algorithm with additional memoization strategies. In this work, we shall call their algorithm `RSA`. Similar to `RSA_single`, this algorithm is applicable to enumerative program synthesis domains such as Feser et al. (2015); Solar-Lezama (2008); Gulwani (2011).

2.5. Exact RSA is Slow

In practice, it is infeasible to explicitly store the matrices M, L_0, S_1, L_1. Instead, computing L_1 using `RSA` requires $\mathcal{O}(|W|)$ calls to S_1. Each call to compute S_1 requires $\mathcal{O}(|U|)$ calls to L_0, which in turn requires $\mathcal{O}(|W|)$ operations to determine a set of consistent programs. In practice, the pragmatic synthesizer L_1 runs in $\mathcal{O}(|W|^2|U|)$ time. In the incremental RSA setting with multiple (say ℓ) utterances, the runtime of L_1 is $\mathcal{O}(|W|^2|U|\ell)$. As the number of hypotheses and utterances becomes large in a program synthesis domain, it becomes infeasible to compute L_1 at a speed required for end-user interactions.

3. Amortizing RSA with Rankings

We explain how the pragmatic listener L_1, derived from the `RSA` algorithm can be amortized using a single global ranking of programs.

Finding Consistent Programs Finding correct programs given a sequence of examples $\mathbf{u} = u_1, u_2, \ldots$ is the primary challenge of program synthesis, with solutions ranging from enumeration (Feser et al., 2015), constraint solving (Solar-Lezama et al., 2006), neuro-symbolic (Polosukhin & Skidanov, 2018; Balog et al., 2016), and using large language models for code (Li et al., 2022). In this work, we assume the a set of k consistent programs $w_1, w_2, \ldots, w_k$ can be found using any of these techniques.

Ranking Consistent Programs with a Prior A global ranking σ is an un-normalized prior (a score) over all programs. The global ranking is example-agnostic: given two programs w_a and w_b, either $\sigma[w_a] \succ \sigma[w_b]$ or $\sigma[w_a] \prec$

$\sigma[w_b]$, irrespective of the given examples $\mathbf{u}$.

$$L_\sigma(w|\mathbf{u}) \propto \sigma[w]M[\mathbf{u}, w]$$

As we can see, ranking the consistent programs under $\sigma[w]$ can be very efficient. In practice, efficient synthesis algorithms are built using either domain-specific heuristics for rankings (Singh & Gulwani, 2015; Polozov & Gulwani, 2015), or a learned prior from a code corpus (Li et al., 2022).

Ranking with L_1 Rather than relying on heuristics or learning from a large corpus, RSA *automatically derives* a ranked synthesizer $L_1(w|\mathbf{u})$:

$$L_1(w|\mathbf{u}) \propto S_1(\mathbf{u}|w)M[\mathbf{u}, w]$$

To rank the consistent programs, L_1 uses $S_1(\mathbf{u}|w)$, an *example-dependent* ranking function, that ranks the satisfying programs differently depending on the sequences of examples $\mathbf{u}$ given. In this setting with multiple examples, there could be **cycles** where a pair[2] of satisfying programs w_a and w_b, which is ranked $S_1(\mathbf{u}_1|w_a) > S_1(\mathbf{u}_1|w_b)$ under some examples $\mathbf{u}_1$ and ranked $S_1(\mathbf{u}_2|w_a) < S_1(\mathbf{u}_2|w_b)$ given different examples $\mathbf{u}_2$. In this work, we assume that S_1 can be tractably computed at non-interactive speed.

Amortizing L_1 with a Ranking In this work, we explore whether the example-dependent ranking of $S_1(\mathbf{u}|w)$ can be approximated — to have similar top-k responses — with an example-agnostic ranking function $\sigma[w]$. Note that due to the existence of cycles, it may be impossible to find a global ranking that is consistent with all example-dependent rankings. Our key findings are as follows:

> **Key Finding 1**: One can distill a pragmatic ranking σ_{L_1} from L_1. While this is an approximation, it nonetheless retains much of the L_1's communicative accuracy when interacting with end-users, and running orders of magnetudes faster.
>
> **Key Finding 2**: In the special case where only a single example is used, RSA_single, the approximation *can* be made exact: There exists a global ranking σ^* that perfectly matches the top-k responses of L_1 over any example u.

4. Distilling L_1 of RSA to a Global Ranking

Distilling the example-dependent L_1 rankings into a global ranking has two stages. First, we generate a dataset of $D = \{(w, \mathbf{u}, \tilde{\sigma}_\mathbf{u}), \dots\}$, where w is a program, $\mathbf{u}$ is a specification (sequence of examples) used to describe w, and $\tilde{\sigma}_\mathbf{u} = [w_1, w_2, \dots, w_k]$ are the k example-dependent rankings of consistent programs given $\mathbf{u}$.[3] Then, we distill a global ranking that aggregates the example-dependent rankings in D.

[2] or a triple or larger cycles

[3] it is a mouthful, we are terribly sorry

4.1. Dataset Generation via Simulated Communications

The pragmatic listener L_1 can generate a partial ranking of consistent programs for *any* sequences of examples $\mathbf{u}$. As arbitrary examples $\mathbf{u}$ are unlikely to reflect what a user might give at inference time, we use the informative speaker S_1 as a "stand-in". Specifically, we generate D in a form of simulated interactions between the pragmatic speaker S_1 and the pragmatic listener L_1. We enumerate over the set of programs $w \in W$, then use the pragmatic speaker to sample the most likely specifications (sequence of examples) $\mathbf{u} \sim_{top-1} S_1(\cdot|w)$ of length 1 to length N. For each specification, we query L_1 for a partial ranking $\tilde{\sigma}_\mathbf{u}$ of consistent programs, and add it to the dataset D (Algorithm 1).

Algorithm 1 Algorithm to obtain a dataset of simulated interactions between a speaker S and listener L. For each turn of each interaction, a ranking of programs is obtained.

Require: Set of programs W
Require: Length of specification to generate N
Require: Speaker model $S(u|w, \mathbf{u})$
Require: Listener model $L(w|\mathbf{u})$
Require: Function MAKERANKING that ranks samples from a distribution based on the probability
  $\mathcal{D} \leftarrow \{\}$
  for w in W **do**
    $\mathbf{u} \leftarrow [\,]$
    for $i = 1$ **to** N **do**
      $u_{\text{next}} \leftarrow \arg\max_u S(u|w, \mathbf{u})$
      $\mathbf{u} \leftarrow \mathbf{u} + [u_{\text{next}}]$
      $\tilde{\sigma}_\mathbf{u} \leftarrow$ MAKERANKING$(L(\cdot|\mathbf{u}))$
      $\mathcal{D} \leftarrow \mathcal{D} \cup \{(w, \mathbf{u}, \tilde{\sigma}_\mathbf{u})\}$
    end for
  end for

4.2. Distillation via Annealing

The most straight-forward representation of a ranking is as an explicit list of programs $\sigma_{\text{global}} = [w_1, w_2, \dots, w_n]$. We describe a process of finding an approximate global ranking using annealing. We repeatedly sample example-dependent rankings $\tilde{\sigma}_\mathbf{u}$ from D, and update the global ranking σ_{global} to match $\tilde{\sigma}_\mathbf{u}$ for a single pair of programs sampled from $\tilde{\sigma}_\mathbf{u}$. Since cycles exist in example-dependent rankings, we terminate the annealing procedure once the number of swaps in a sliding window has stabilized (Algorithm 2). The resulting σ_{global} is then used at inference time.

Algorithm 2 Algorithm to infer a global order σ based on a dataset of simulated interactions, that terminates based on a validation criterion determined by the validation frequency V, patience t and convergence threshold T

Require: Dataset of simulated interactions $\mathcal{D}$
  $\sigma \leftarrow$ randomly initialized ranking
  converged $\leftarrow$ FALSE
  $N_{\text{swaps}} \leftarrow [\,]$
  $n_{\text{swaps}} \leftarrow 0$
  $i \leftarrow 0$
  while not converged **do**
    $(p, \tilde{\sigma}, \mathbf{u}) \sim \mathcal{D}$
    Sample programs p_1, p_2 in $\tilde{\sigma}$
    if $\tilde{\sigma}[p_1] \succ \tilde{\sigma}[p_2]$ **and** $\sigma[p_1] \prec \sigma[p_2]$ **then**
      Swap $\sigma[p_1], \sigma[p_2]$
      $n_{\text{swaps}} \leftarrow n_{\text{swaps}} + 1$
    end if
    $i \leftarrow i + 1$
    if $i \equiv 0 \mod V$ **then**
      $N_{\text{swaps}} \leftarrow N_{\text{swaps}} + [n_{\text{swaps}}]$, $n_{\text{swaps}} \leftarrow 0$
      if $\max N_{\text{swaps}}[-t:] - \min N_{\text{swaps}}[-t:] < T$ **then**
        converged $\leftarrow$ TRUE
      end if
    end if
  end while
  return σ

```
 S → RP | S RP
RP → '[01]' OP | '0' OP | '1' OP
OP → '*' | '+' | '{1}' | '{2}'
```

Figure 4. Grammar for the regex domain

4.3. Distillation via Learning a Score Function

An alternative method to distill D is to train a score function $s_\theta : w \to \mathbb{R}$ that determines a score for a program w that is independent of the specifications $\mathbf{u}$. We can optimize θ to minimize disagreement with the generated dataset of example-dependent rankings, by minimizing the loss

$$\mathcal{L}(\theta) = \mathop{\mathbb{E}}_{\substack{\tilde{\sigma}_{\mathbf{u}} \sim \mathcal{D} \\ w_1, w_2 \sim \tilde{\sigma}_{\mathbf{u}} : \tilde{\sigma}_{\mathbf{u}}[w_1] \succ \tilde{\sigma}_{\mathbf{u}}[w_2]}} - \log(\text{sig}(s_\theta(w_1) - s_\theta(w_2)))$$

where sig is the sigmoid function. This follows estimating a score function from a set of pairwise preferences (Bradley & Terry, 1952; Christiano et al., 2017). We parametrize s_θ as a small neural network that scores programs. To reduce variance, we fit an ensemble of score functions and use their average to rank the consistent programs at inference time (Christiano et al., 2017). Details of the neural models are in Appendix E.

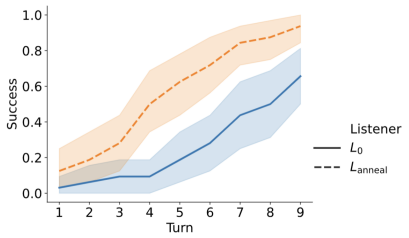


Figure 5. Success rate of the literal L_0 and ranking-based L_{anneal} synthesizers inferring the correct regex as a function of numbers of examples given (turn). L_{anneal} achieves a success rate of 93.75%, L_0 achieves only 65.63%. The ranking-based synthesizer also achieves higher success with fewer utterances. Bands indicate 95% CI over 24 regexes for each condition.

5. Experiments

To validate the accuracy and run-time of an approximate ranking listener, we perform two sets of experiments. First, we conduct a small ($n = 8$) **human experiment** by building a ranking-based synthesizer in a regular expression synthesis domain where it is infeasible to run the RSA algorithm L_1 at interaction time. Second, we conduct two **replay studies** by simulating virtual users giving examples one after another using human interaction data collected from prior works. We seek to answer the following questions: (**Q1**) Can ranking based synthesizers accurately infer programs from humans (both in live interaction and in simulated replays)? (**Q2**) Are ranking-based synthesizers fast to run when compared to L_0 and L_1?

Metrics In our experiments, the users (real or simulated) will be given a target program, and attempt to communicate it to the synthesizers using examples. The synthesizers will be measured on their communication accuracy — whether the synthesizers can infer the target program from the examples given. A synthesizer is better than another if it can recover the target program using fewer examples.

5.1. Interactive User Study

We conduct a user study where people interacted with both the ranking-based synthesizer distilled with annealing L_{anneal} and the literal synthesizer L_0 on the domain of regular expression synthesis.

The Regex Domain The regex domain is a scaled up version of Vaithilingam et al. (2023), which has a total of 350 regular expressions from their grammar (Figure 4. For this study, we expanded the space of programs to 3500 regular expressions from the same grammar – a setting that would make live interaction infeasible running L_1 with RSA.

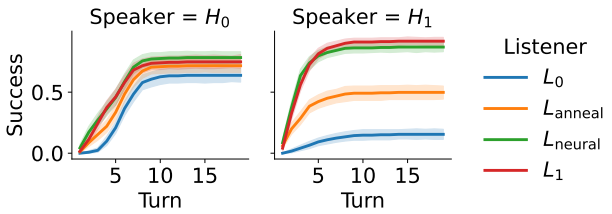


Figure 6. Animal domain replay results. Fraction of successfully communicated target programs (success) vs number of examples given (turn). Bands are 95% confidence interval across interactions (254 for H_0 and 291 for H_1). Two kinds of simulated speakers: H_0 — replaying the interactions where participants communicated with a literal L_0 synthesizer from the original study; H_1 — with a pragmatic L_1 synthesizer. On H_0 replay, L_0 performs worst (63.78%), and L_{anneal} (71.65%), L_1 (74.80%), L_{neural} (78.34%) performing similarly to each other. On H_1 replay, L_0 (15.46%) performs worst, with L_{anneal} (49.82%) in the middle, while L_1 (91.75%) and L_{neural} (86.94%) perform best.

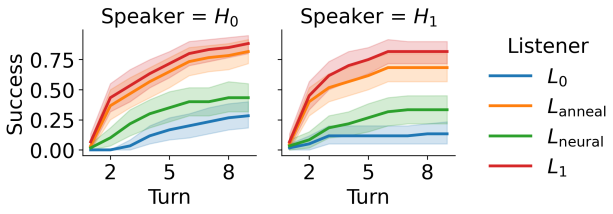


Figure 7. Regex domain replay results. Bands are 95% confidence interval across interactions (60 interactions for both H_0 and H_1). On H_0 replay, L_0 (28.33%) performs worst, L_{neural} (35.00%) slightly better, and L_{anneal} (81.67%), L_1 (88.33%) perform best. On H_1 replay we observe the same trend, with L_0 (13.33%), L_{neural} (28.33%), L_{anneal} (68.33%), L_1 (81.67%) respectively.

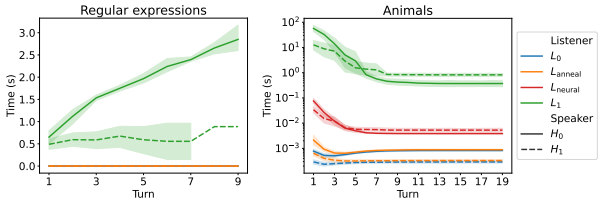


Figure 8. The wall clock time for each synthesizer given different numbers of examples (turn). We see that L_1 is consistently much slower than either L_{anneal} or L_0 in both domains. Note that time is on a logarithmic scale for the animals domain. The difference slopes for L_1 (trending up for regex and trending down for animals) is due to an optimization of the L_1 synthesizer for the animals domain, which filters out invalid programs as a pre-proccessing step using L_0, making it having to rank fewer programs over turns

Procedure We recruited 8 participants from our institution. Each participant was given a short tutorial on how to use the interface, then attempted to communicate a total of 4 regexes using examples. For each regex, the participant communicated with both the literal synthesizer L_0 and the ranking synthesizer L_σ, anonymized as simply a "green robot" and a "blue robot" in randomized order. The participants gave example strings one at a time until the regex is recovered by the synthesizer, or they may give up early. The communication is interactive: When the participant added a new example, they were immediately shown the current top-1 guess of the synthesizer, which allowed them to choose the next example accordingly.

Results: end-users interact well with an amortized ranking synthesizer (Q1) Figure 5 shows the communication success rate over numbers of given exmaples (turns) for both the literal and ranking-based synthesizers. We can see that (1) L_{anneal} has a higher overall success rate with humans, and (2) It also achieves a higher success rate with fewer number of examples (**Q1**).

5.2. Simulated User Studies Using Replays

We evaluate the ranking-based synthesizers by replaying the interaction data collected from Vaithilingam et al. (2023) and Pu et al. (2020) – small pragmatic program synthesis domains where it is feasible to run L_1 with RSA.

Replay Data In the human studies by Vaithilingam et al. (2023) and Pu et al. (2020), a human H is given a target program w, and attempt to get the synthesizer (L_0 or L_1) to infer the target using a sequence of examples $\mathbf{u} = u_1, u_2, \ldots$. Thus, two sets of data are generated, one where the human is interacting with the literal synthesizer L_0, which we term H_0, and one where the human is interacting with the pragmatic synthesizer L_1, which we term H_1. Specifically, from each domain we extract the following dataset $\{(w, \mathbf{u}_i^j) | w \in W_s, j \in P, i \in \{0, 1\}\}$. Here, W_s are the set of programs used for the human study (the stimuli), P is the set of participants, and i indicates if the participant is communicating with L_0 or L_1.

Experiment Setup We can simulate an user interaction by using the replay data. Given a datapoint $w, \mathbf{u}$, we create a simulated user that iteratively gives the examples $u_1, u_2, \ldots$ in multiple turns to communicate a given target program w. At every turn, the synthesizer returns the top-1 responses, $L^{\text{top-1}}(u_1), L^{\text{top-1}}(u_1, u_2), \ldots$, and we can check if any of them matches the target program w. If they do, we mark the communication as successful and stop early. Otherwise, we keep adding examples until the $\mathbf{u}$ runs out, and we mark the communication as unsuccessful. Note that our evaluation cannot account for a user adapting their choice of examples to L, as the simulated user can only give scripted examples according to the replay data.

Domain 1: Animals Pu et al. (2020) used a domain of grid patterns generated by an underlying domain-specific language (see Appendix for the grammar of the DSL and semantics). The space contains 17,976 semantically distinct programs and 343 possible examples, where a user uses a sequence of multiple examples to communicate a target program. They conducted a study with 48 human subjects, collecting data for 10 programs (10 distinct grid patters). The data includes interactions between humans and both a literal synthesizer ($H_0 - L_0$) and a pragmatic synthesizer ($H_1 - L_1$). In total, there are 254 interactions from $H_0 - L_0$ and 291 interactions from from $H_1 - L_1$, where each interaction consists of multiple turns until either the target program is successfully communicated or the user gives up.

Domain 2: Regular expressions Vaithilingam et al. (2023) studied the usability of pragmatic program synthesizers in the domain of binary regular expressions. The space contains 350 distinct regular expressions. A sample of 2000 strings was used to compute the S_1 and L_1 distributions. Their study included 30 participants interacting with both L_0 and L_1 models. In total, there are 60 interactions from $H_0 - L_0$ and 60 interactions from from $H_1 - L_1$, where each consisting of multiple turns.

Result: rank-based synthesizers are comparable to L_1 in terms of communication accuracy with simulated users (Q1) The replay study results are shown in Figure 6 (animals domain) and Figure 7 (regex domain). For either domain, there is a rank-based synthesizer that vastly outperforms the literal synthesizer L_0, and is close to performance to the pragmatic synthesizer L_1 derived from RSA.

The existence of a rank-based synthesizer (be it L_{anneal} or L_{neural}) that matches the performance of L_1 entails that there exists some ranking of programs that effectively amortizes L_1 for either domain. For the animals domain, L_{neural} is better able to discover an effective ranking, while L_{anneal} is more effective at discovering the ranking for the regex domain. This is likely due to the differences of the *sizes* of the communicative datasets for the two domains — 17,976 programs for the animals domain vs 350 for the animals domain, which makes it more feasible to learn a generalizable neural scoring function for the animals domain.

Result: rank-based synthesizers are orders of magnetudes faster than L_1 (Q2) For both domains, the ranking-based synthesizer is much faster than L_1, requiring approximately the same time as L_0 (Figure 8). This implies that most of the computation cost of a ranking-based synthesizer lies in coming up with consistent programs — the primary challenge of program synthesis — while the computation for ranking the top-k programs can be made negligible in comparison (**Q2**).

6. RSA_single Can Be Distilled Completely

In this section, we prove a strong approximation result for a special case of RSA, RSA_single, where only a single example u is used to communicate. In accordance with the terminologies of Goodman & Frank (2016); Vogel et al. (2013); Monroe & Potts (2015); Smith et al. (2013) and Franke & Degen (2016), we'll use the term "hypothesis" instead of "program". We prove that a global pragmatic ranking of hypotheses must exist for any listeners $L_0, L_1, \ldots$ resulting from the RSA_single algorithm.[4] In other words, the rankings over consistent hypotheses in these listeners are example-agnostic.

Theorem: For a sequence of listeners in the RSA algorithm $L_0, L_1, \ldots$ over a boolean-valued lexicon M, there exists a sequence of global pragmatic rankings $\sigma_{L_0}, \sigma_{L_1}, \ldots$ such that:

$$\begin{aligned} &\forall w, w', u.\ \textbf{if}\ L_i(w|u) > 0 \wedge L_i(w'|u) > 0. \\ &\textbf{then}\ L_i(w|u) > L_i(w'|u) \iff \sigma_{L_i}[w] \succ \sigma_{L_i}[w'] \end{aligned} \tag{4}$$

This means the partial rankings produced by any L_i over consistent hypotheses are example-agnostic, where a global ranking preferring certain hypotheses unconditionally over others (e.g. a convention) is sufficient to explain the relative rankings of L_i resulting from RSA_single.

Proof: Let M be a boolean lexicon of size m rows and n columns. Let $r_0 = r_0^1 \ldots r_0^m$ be the row-normalizing vector such that $r_0^j = (\sum M[j,:])^{-1}$, which is to say, each element r_0^j is the normalization term for row j of L_0. Let $*_{\leftrightarrow}$ denotes row-wise multiplication:

$$L_0 = M *_{\leftrightarrow} r_0$$

Which is to say, starting from M, L_0 can be obtained by scaling each row j by their respective normalization constant r_0^j. Let $c_1 = c_1^1 \ldots c_1^n$ be the col-normalizing vector such that $c_1^j = (\sum L_0[:,j])^{-1}$, which is to say, each element c_1^j is the normalization term for column j of S_1. Similarly, let $*_{\updownarrow}$ denotes column-wise multiplication

$$S_1 = L_0 *_{\updownarrow} c_1 = M *_{\leftrightarrow} r_0 *_{\updownarrow} c_1$$

Computing L_i under RSA amounts to applying row and column normalization alternatively multiple times:

$$L_i = M *_{\leftrightarrow} r_0 *_{\updownarrow} c_1 \ldots *_{\updownarrow} c_{i-1} *_{\leftrightarrow} r_i$$

Let $*$ be element-wise multiplication, let $\otimes$ be outer-product, we can rearrange the terms:

$$\begin{aligned} L_i =& M * ((r_0 * \cdots * r_i) \otimes (c_1 * \cdots * c_{i-1})) \\ =& M * (r_{0\ldots i} \otimes c_{1\ldots i-1}) \end{aligned} \tag{5}$$

[4] one can derive the same result for pragmatic ranking of speakers by taking a transpose of M

Here, $r_{0\ldots i} = r_0 * \cdots * r_i$ is a vector of size m, and $c_{1\ldots i-1} = c_1 * \cdots * c_{i-1}$ is a vector of size n. As we can see, following the RSA algorithm, L_i can be decomposed to to multiplication of 2 parts: the lexicon M, and a matrix that is formed by the outer product $r_{0\ldots i} \otimes c_{1\ldots i-1}$ [5].

Claim: The ordered indexes of $c_{1\ldots i-1}$ *is* the global pragmatic ranking σ_{L_i}:

$$\sigma_{L_i}[w] \succ \sigma_{L_i}[w'] \iff c_{1\ldots i-1}[w] > c_{1\ldots i-1}[w']$$

Proof: We show both sides of the $\iff$. Suppose that for some w, w', u, both $L_i(w|u) > 0$ and $L_i(w'|u) > 0$ (i.e. $M[u, w] = M[u, w'] = 1$).

(1) Show $\Rightarrow$: Suppose $L_i(w|u) > L_i(w'|u)$. We have

$$L_i(w|u) = L_i[u, w] = r_{0\ldots i}[u] * c_{1\ldots i-1}[w]$$
$$L_i(w'|u) = L_i[u, w'] = r_{0\ldots i}[u] * c_{1\ldots i-1}[w']$$

As $r_{0\ldots i}[u]$ is a constant, we have

$$L_i(w|u) > L_i(w'|u) \Rightarrow c_{1\ldots i-1}[w] > c_{1\ldots i-1}[w'] \ \square.$$

(2) Show $\Leftarrow$: Suppose $c_{1\ldots i-1}[w] > c_{1\ldots i-1}[w']$.

$$\begin{aligned} c_{1\ldots i-1}[w] &> c_{1\ldots i-1}[w'] \\ M[u, w] * r_{0\ldots i}[u] * c_{1\ldots i-1}[w] &> \\ M[u, w'] * r_{0\ldots i}[u] * c_{1\ldots i-1}[w'] \\ L_i[u, w] &> L_i[u, w'] \\ L_i(w|u) &> L_i(w'|u) \ \square. \end{aligned}$$

Thus, $c_{1\ldots i-1}$ is the global ranking σ_{L_i} as claimed ■.

We check the our proof using simulations on 10000 randomly generated boolean lexicons size ranging from 10×10 to 20×20, and running a chain of 100 listeners on top. A total ordering can be found for all of them (Appendix B.1). We further study the *stability* of these ranks as they are formed, finding that the formed rankings tend to be stable across different RSA iterations (Appendix B.2).

7. Related Works

Scaling RSA without Global Ranking Prior work such as that by Monroe et al. (2017) and Andreas & Klein (2016) has largely focused on sample and re-rank as a way of scaling RSA, making the example-dependent ranking function $S_1(u|w)$ more efficient at a cost of accuracy. Recent work by Key et al. (2022) and Vaduguru et al. (2024) apply the sample and re-rank approach to program synthesis, resulting in neural program synthesizers that also rank programs in an example-dependent way. Our work enables a different kind of synthesis algorithm altogether — that of a distilled pragmatic ranking that rank consistent programs agnostic to examples given. We view these works as complimentary, able to efficiently produce a simulated communication dataset D which our approach can distill from.

Scaling RSA with Human Data RSA has been applied to improve the performance of language interfaces in a variety of other domains, such as image description (Andreas & Klein, 2016; Cohn-Gordon et al., 2018a;b), instruction generation and interpretation (Fried et al., 2018a;b), and grounded interaction (Fried et al., 2021; Lin et al., 2022). These works all use speaker models trained on labeled data from people. Our approach requires no human-produced data, and can be run entirely from the lexicon M of the synthesis problem. On the other hand, we can easily integrate human data within our approach by training similar speaker models on the collected interactive data.

Ranking Functions in Synthesis Prior works on resolving ambiguity in program synthesis have relied on example-agnostic ranking functions. Works such as Singh & Gulwani (2015); Polozov & Gulwani (2015) use scoring functions to penalize certain properties of programs (e.g. discouraging the use of constants), effectively inducing a global ranking over all programs; Ellis & Gulwani (2017) uses a set of hand-crafted features to learn a naturalistic ranking from data. Synthesis algorithms that use a large neural code model to sample a large number of programs (Chen et al., 2021; Li et al., 2022) implicitly rank the programs based on their naturalistic distributions in its training data. Our work is unique in that (1) the learned ranking is rooted in efficient communication rather than hand-crafted features and (2) our approach does not require human annotated data.

Other Theoretical Works on Ranking Recent work by Muggleton FREng (2023) shows that in the case of single-example, the MAP estimate of the learner can be completely ranked by $sz(H) + \ln g(H)$ an example-agnostic global ranking. Our work can be viewed as a strict generalization in the following sense: They consider the chain of recursive bayesian reasoners of the form $M \to S_0 \to L_1$, whereas our result applies to any alternating chains speakers and listeners of arbitrary depth. Their notion of "specificity" and "program length" also has direct analogies to the normalization terms in Equation (5), except these analogies do not carry over to deeper recursive depths.

8. Conclusion

We present a way of amortizing the expensive RSA algorithm by an example-agnostic global ranking. We have shown this amortization interacts well with humans when applied to two program synthesis domains. We have further proved

[5] note that any prior over hypotheses and utterance can be similarly absorbed into these outer products terms

this amortization is exact in the case of communication with a single example. In addition of being a practical method for scaling up RSA, these findings may provide an alternative account for pragmatic behaviour in humans – one rooted in relative rankings of hypotheses (e.g. a pragmatic prior), perhaps distilled from the expensive RSA computation over time.

8.1. Limitation and Future Directions

The limitation of our approach is two-fold: First, whether an optimal global ranking exists for the multi-example PBE setting; Second, whether our distillation algorithm can find this optimal ranking.

Existence of an effective global ranking The effectiveness of a global ranking is upper-bounded by the amount of *cycles* that exists in the communicative dataset of example-dependent rankings of subsets of programs. A cycle exists if under one ranking we have $w_a \succ w_b$, and under a different one we have $w_b \succ w_a$, which no single ranking can approximate exactly. Forecasting the number of cycles from the meaning matrix M is an exciting future work.

Effectiveness of distilling an effective global ranking Our experiments have shown that given a communicative dataset, both the annealing (in the case of a small dataset) and neural scoring (in the case of a larger dataset) have their merits in deriving a ranking. Thus, running the slow RSA in the dataset generation itself is the likely bottleneck. We believe recent works by Key et al. (2022) and Vaduguru et al. (2024) using sample-and-rerank may be used in generating the communicative dataset instead of the exact RSA algorithm.

Impact Statement

This work builds a system where end-users may use examples to generate programs. While the proposed method is more intuitive to use by humans, it is possible that for some interactions, it may generate unexpected programs. Therefore, it could be of potential danger when humans do not manually verify the generated program, as it may have unintended outcomes when executed.

Acknowledgements

The authors would like to thank Kevin Ellis, Pei Wang, and Jesse Wang for preliminary explorations in this direction and insights to the proof. SV was partially supported by a gift from Autodesk Research. This material is based upon work supported by the NSF under Grant Nos. CCF-2123965 and IIS-2107391. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

References

Andreas, J. and Klein, D. Reasoning about pragmatics with neural listeners and speakers. *arXiv preprint arXiv:1604.00562*, 2016.

Balog, M., Gaunt, A. L., Brockschmidt, M., Nowozin, S., and Tarlow, D. Deepcoder: Learning to write programs. *ICLR*, 2016.

Bergen, L., Levy, R., and Goodman, N. Pragmatic reasoning through semantic inference. *Semantics and Pragmatics*, 9:ACCESS–ACCESS, 2016.

Bradley, R. A. and Terry, M. E. Rank analysis of incomplete block designs: I. the method of paired comparisons. *Biometrika*, 39(3/4):324–345, 1952. ISSN 00063444. URL http://www.jstor.org/stable/2334029.

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code, 2021.

Christiano, P. F., Leike, J., Brown, T., Martic, M., Legg, S., and Amodei, D. Deep reinforcement learning from human preferences. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/d5e2c0adad503c91f91df240d0cd4e49-Paper.pdf.

Cohn-Gordon, R., Goodman, N., and Potts, C. Pragmatically informative image captioning with character-level inference. *arXiv preprint arXiv:1804.05417*, 2018a.

Cohn-Gordon, R., Goodman, N. D., and Potts, C. An incremental iterated response model of pragmatics. *arXiv preprint arXiv:1810.00367*, 2018b.

Ellis, K. and Gulwani, S. Learning to learn programs from examples: Going beyond program structure. *IJCAI*, 2017.

Feser, J. K., Chaudhuri, S., and Dillig, I. Synthesizing data structure transformations from input-output examples. PLDI '15, pp. 229–239, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450334686. doi: 10.1145/2737924.2737977. URL https://doi.org/10.1145/2737924.2737977.

Frank, M. C. and Goodman, N. D. Predicting pragmatic reasoning in language games. *Science*, 336(6084):998–998, 2012.

Franke, M. and Degen, J. Reasoning in reference games: Individual-vs. population-level probabilistic modeling. *PloS one*, 11(5):e0154854, 2016.

Fried, D., Andreas, J., and Klein, D. Unified pragmatic models for generating and following instructions. In *Proceedings of North American Chapter of the Association for Computational Linguistics*, 2018a.

Fried, D., Hu, R., Cirik, V., Rohrbach, A., Andreas, J., Morency, L.-P., Berg-Kirkpatrick, T., Saenko, K., Klein, D., and Darrell, T. Speaker-follower models for vision-and-language navigation. *Advances in Neural Information Processing Systems*, 31, 2018b.

Fried, D., Chiu, J. T., and Klein, D. Reference-centric models for grounded collaborative dialogue. *arXiv preprint arXiv:2109.05042*, 2021.

Goodman, N. D. and Frank, M. C. Pragmatic language interpretation as probabilistic inference. *Trends in cognitive sciences*, 20(11):818–829, 2016.

Gulwani, S. Automating string processing in spreadsheets using input-output examples. *SIGPLAN Not.*, 46(1): 317–330, jan 2011. ISSN 0362-1340. doi: 10.1145/1925844.1926423. URL https://doi.org/10.1145/1925844.1926423.

Key, D., Li, W.-D., and Ellis, K. I speak, you verify: Toward trustworthy neural program synthesis. *arXiv preprint arXiv:2210.00848*, 2022.

Lewis, D. Scorekeeping in a language game. *Journal of philosophical logic*, 8:339–359, 1979.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

Lin, J., Fried, D., Klein, D., and Dragan, A. Inferring rewards from language in context. *arXiv preprint arXiv:2204.02515*, 2022.

Monroe, W. and Potts, C. Learning in the rational speech acts model. *arXiv preprint arXiv:1510.06807*, 2015.

Monroe, W., Hawkins, R. X., Goodman, N. D., and Potts, C. Colors in context: A pragmatic neural model for grounded language understanding. *Transactions of the Association for Computational Linguistics*, 5:325–338, 2017.

Muggleton FREng, S. Hypothesizing an algorithm from one example: the role of specificity. *Philosophical Transactions of the Royal Society A*, 381(2251):20220046, 2023.

Polosukhin, I. and Skidanov, A. Neural program search: Solving programming tasks from description and examples. *arXiv preprint arXiv:1802.04335*, 2018.

Polozov, O. and Gulwani, S. Flashmeta: A framework for inductive program synthesis. *ACM SIGPLAN Notices*, 50 (10):107–126, 2015.

Pu, Y., Ellis, K., Kryven, M., Tenenbaum, J., and Solar-Lezama, A. Program synthesis with pragmatic communication. *Advances in Neural Information Processing Systems*, 33:13249–13259, 2020.

Singh, R. and Gulwani, S. Predicting a correct program in programming by example. In *CAV*, pp. 398–414. Springer, 2015.

Smith, N. J., Goodman, N., and Frank, M. Learning and using language via recursive pragmatic reasoning about other agents. *Advances in neural information processing systems*, 26, 2013.

Solar-Lezama, A. *Program synthesis by sketching*. PhD thesis, USA, 2008. AAI3353225.

Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., and Saraswat, V. Combinatorial sketching for finite programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pp. 404–415, 2006.

Vaduguru, S., Ellis, K., and Pu, Y. Efficient pragmatic program synthesis with informative specifications, 2022.

Vaduguru, S., Fried, D., and Pu, Y. Generating pragmatic examples to train neural program synthesizers. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=yxKZGQLzOP.

Vaithilingam, P., Pu, Y., and Glassman, E. L. The usability of pragmatic communication in regular expression synthesis, 2023.

Vogel, A., Bodoia, M., Potts, C., and Jurafsky, D. Emergence of gricean maxims from multi-agent decision theory. In *Proceedings of the 2013 conference of the north american chapter of the association for computational linguistics: Human language technologies*, pp. 1072–1081, 2013.

A. Code and Assets

Please find all simulation, replay results at this repository https://github.com/evantheboucy/pragmatic_synthesis_ranking/tree/main

B. Simulated Studies

B.1. Ranking Always Exists

We empirically validate that in the case of single utterances, a ranking can always be found. See simulation/single_utter/exp_exists_orders.py

B.2. Stability of Ranks Across RSA Iterations

We've shown that for every $L_0, L_1, \ldots$, there exists a corresponding global, utterance agnostic ranking $\sigma_{L_0}, \sigma_{L_1}, \ldots$. We now explore the relationship between these rankings as a function of the RSA iteration i. Specifically, how *stable* is the relative ranks of w and w' once it is formed?

Stable Order A pair-wise order between w and w' is *stable* from iteration i onward if:

$$stable(i, w \succ w') \iff \bigwedge_{j \in i, i+1, \ldots, \infty} \sigma_{L_j}[w] \succ \sigma_{L_j}[w']$$

Which means the relative ranking of $\sigma_{L_i}[w] \succ \sigma_{L_i}[w']$ holds true for every subsequent iterations until σ_{L_∞}. Let the *minimal-index* of a stable pair-wise ordering be the first iteration i such that $w \succ w'$ becomes stable:

$$i_{\min}(w \succ w') = \text{argmin}_j stable(j, w \succ w) \tag{6}$$

As σ_{L_1} is the first time any ranking can exist (L_0 is a uniform distribution over valid hypotheses, i.e. no rankings), we explore the following: For a lexicon M, what fraction of stable orderings have a minimal-index of 1?

$$\text{frac-stable}_{L_1}(M) = \frac{|\{w \succ w' \mid i_{\min}(w \succ w') = 1\}|}{|\{w \succ w' \mid \exists i.\ stable(i, w \succ w')\}|} \tag{7}$$

Simulation We measure $stable_{L_1}(M)$ on a population of sampled random boolean lexicons. We sample square lexicons of size $lexicon_size \in 2 \times 2 \ldots 100 \times 100$. Each lexicon is sampled with $Ptrue \in \{0.1, 0.2, 0.5\}$, where larger value of $Ptrue$ makes the lexicon have more 1s. We make sure each sampled lexicon is valid in the following sense: (1) all rows are unique – every utterance must communicate a unique subset of valid hypotheses (2) all columns are unique – every hypothesis has a unique set of utterances that can refer to it. For every combination of $(Ptrue, lexicon_size)$ we randomly sample 100 lexicons. As it is infeasible to run RSA until iteration ∞, we run RSA for 100 iterations for each lexicon (i.e. $L_{100} \approx L_\infty$). We measure $stable_{L_1}$ for each sampled lexicon. The result is shown in 9. As we can see, of all the stable pair-wise orderings, a large fraction (> 0.8) are formed during σ_{L_1}, this is increasingly true as we (1) increase $Ptrue$, making the boolean lexicons having more number of 1s – i.e. the lexicon is more *ambiguous* for a literal speaker and listener and (2) increase $lexicon_size$. We suspect this is due to faster "mixing time" of the RSA algorithm under these conditions, but this is just a guess.

Takeaway This study may provide an alternative explanation as to why humans do not perform RSA for more than few iterations (Franke & Degen, 2016). In addition to it being computationally expensive, it is also *not necessary* as the majority of top-k orderings becomes available at σ_{L_1}, and remains stable for all subsequent iterations of the RSA algorithm. In another word, $L_1^{top-k} \cong L_{i>1}^{top-k}$. Code in simulation/single_utter

C. Animals domain

In the Animals domain, a program is a pattern on a grid formed from a set of objects. These objects may be a colourless pebble, or a chicken or pig that may be red, green or blue. An utterance reveals one square on the grid, and the speaker has to communicate the pattern by choosing which square to reveal. The pattern is formed according to rules specified in the domain-specific language in Figure 10. Examples of programs shown in Figure 11. The description of the domain-specific language and the examples are due to Vaduguru et al. (2022).

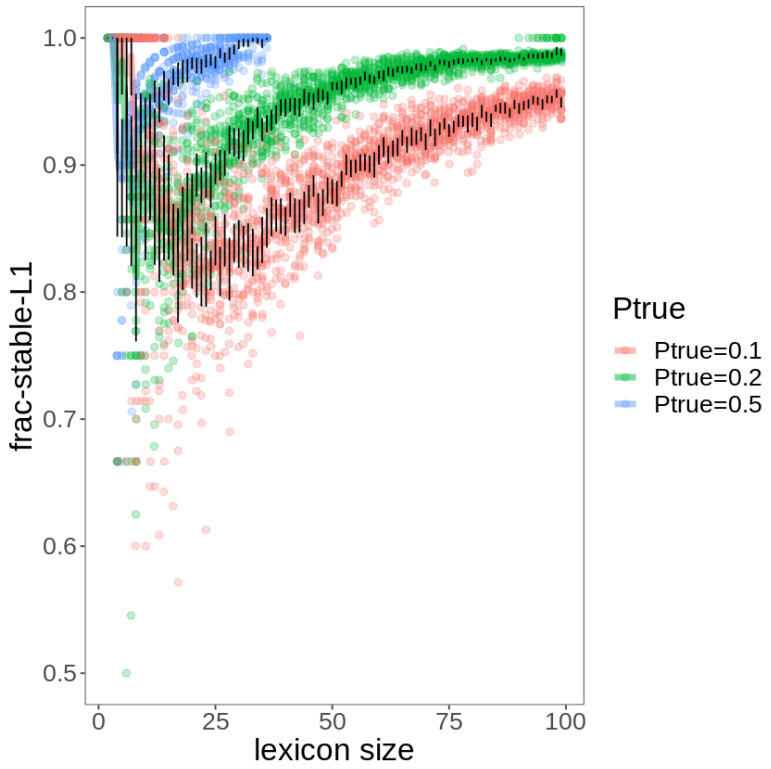


Figure 9. Fraction of stable orders that were formed in σ_{L_1} as a function of increasing lexicon size. Points are raw samples (n=100 per lexicon size and Ptrue), bars are 95% bootstrapped CI (nboot = 1000). Overall, increasing Ptrue and lexicon size increases the fraction of stable orders that were formed in σ_{L_1}

D. Human study interface

The interface for the human study on regular expression programs is shown in Figure 12.

E. Neural model

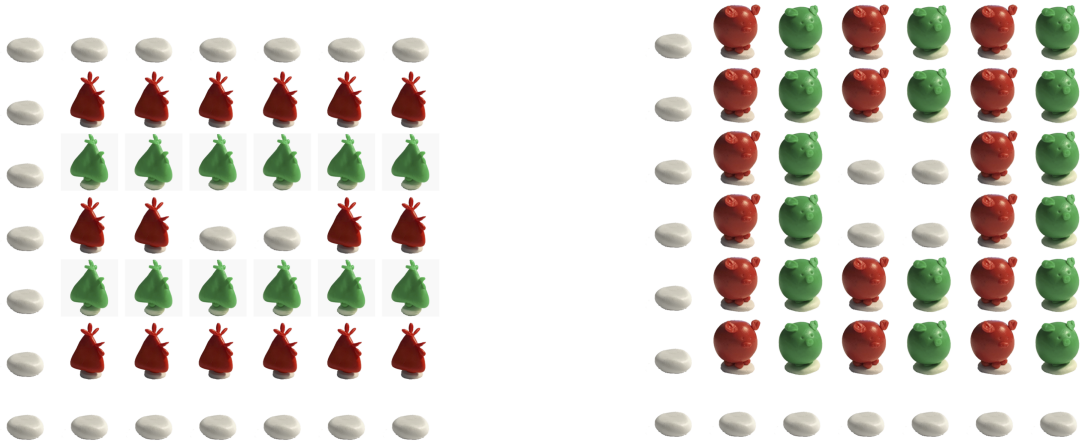
The neural scoring model maps from the program to a real number. The program is input as vector encoding the productions of the grammar that produce the program. That is, we construct a vector of the index of the production that is used to expand each non-terminal in the DSL grammar. We then convert this vector to a one-hot matrix. There are 12 rules, with any single rule having at most 7 possible expansions resulting in an input vector of dimension $12 \times 7 = 84$. The input is then passed through 3 hidden layers of size 128, each of which has as ReLU activation, and then mapped to a scalar output with a linear layer.

The model is trained on a dataset of rankings of the form $\mathcal{D} = (w, \mathbf{u}, \tilde{\sigma}_{\mathbf{u}})$. For each program w, we sample a pair of programs from the inferred ranking $\tilde{\sigma}_{\mathbf{u}}$ and use this pair to compute the loss function for this sample. We train the model for a maximum of 20 epochs, where one epoch of training corresponds to presenting the model with every element in $\mathcal{D}$ once. We train with a batch size of 32 using the Adam optimizer. We use a validation set generated similarly to $\mathcal{D}$ (on a disjoint set of programs) to perform validation, choosing the model that results in the highest synthesis accuracy on this validation dataset with synthetically produced examples (from the S_1 speaker model).

We train an ensemble of 10 models. For each model, we normalize the scores to be of zero mean and unit variance based on the empirical mean and standard deviation computed on the validation set. We then average the scores for the 10 models at inference time.

```
  Program → ⟨Shape, Colour⟩
    Shape → Box(Left, Right, Top, Bottom, Thickness, Outside, Inside)
     Left → 0 | 1 | 2 | 3 | ... | 6
    Right → 0 | 1 | 2 | 3 | ... | 6
      Top → 0 | 1 | 2 | 3 | ... | 6
   Bottom → 0 | 1 | 2 | 3 | ... | 6
Thickness → 1 | 2 | 3
        O → chicken | pig
        I → chicken | pig | pebble
   Colour → [red , green , blue][A2(A1)]
       A1 → x | y | x + y
       A2 → λz:0|λz:1|λz:2|λz:z%2|λz:z%2+1|λz:2*(z%2)
```

Figure 10. Grammar of the DSL



(a) [·, ·, 1, 5, 1, 6, 2, chicken, pebble, ·, x, λz:z%2]

(b) [·, ·, 0, 5, 1, 6, 2, pig, pebble, ·, y, λz:z%2]

Figure 11. Two patterns in our layout domain and their corresponding programs, represented as a sequence of production rules: [Program, Shape, Left, Right, Top, Bottom, Thickness, O, I, Colour, A1, A2]. The symbol · indicates rules which only have 1 choice of expansion (Program, Shape, and Colour). The rules where these two programs differ are marked with a box.

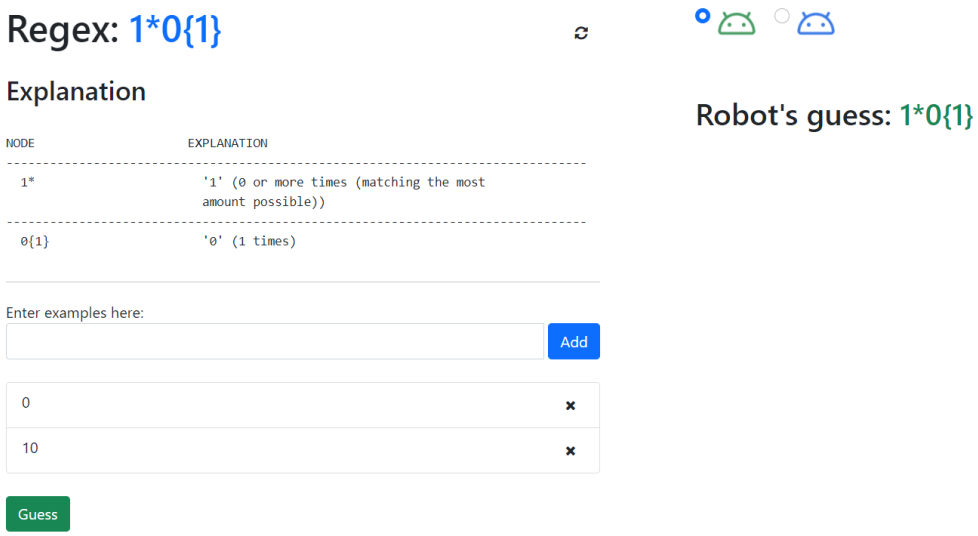


Figure 12. User interface for the regex domain