From Language to Programs: Bridging Reinforcement Learning and Maximum Marginal Likelihood

Kelvin Guu, Ice Pasupat, Evan Liu, Percy Liang

Today I'll be talking about translating natural language into executable programs.

TASK: parse multi-step instructions into programs

Let's imagine that you're working in a chemistry lab, and you have a robot assistant.
You'd like to say things like "pour the last green beaker into beaker two", etc.

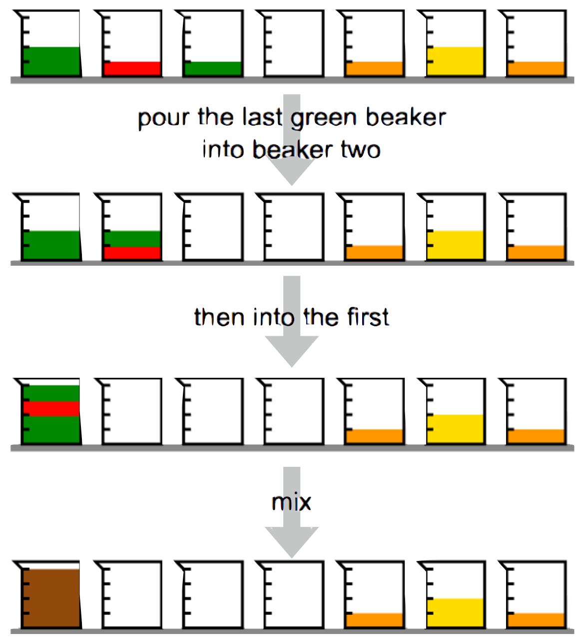
The robot then needs to translate each of these instructions into an executable program.

To train this robot, it is very time consuming to label each command with the right program, or the user may not even know how to write code. In contrast, it's quite easy to move the beakers yourself and demonstrate what should happen.

So, we consider a learning setup where the robot sees a demonstration, but does not actually observe the correct program.

This is really a classic "weak supervision", or "learning from denotation" task, which has had a rich history in the semantic parsing literature.

TASK: parse multi-step instructions into programs



Let's imagine that you're working in a chemistry lab, and you have a robot assistant.
You'd like to say things like "pour the last green beaker into beaker two", etc.

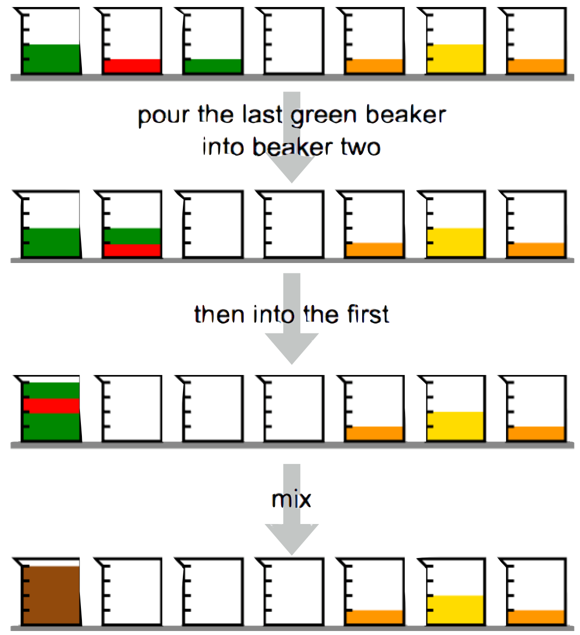
The robot then needs to translate each of these instructions into an executable program.

To train this robot, it is very time consuming to label each command with the right program, or the user may not even know how to write code. In contrast, it's quite easy to move the beakers yourself and demonstrate what should happen.

So, we consider a learning setup where the robot sees a demonstration, but does not actually observe the correct program.

This is really a classic "weak supervision", or "learning from denotation" task, which has had a rich history in the semantic parsing literature.

TASK: parse multi-step instructions into programs




```
pour(hasColor(green)[-1], beakers[2])
```

Let's imagine that you're working in a chemistry lab, and you have a robot assistant.
You'd like to say things like "pour the last green beaker into beaker two", etc.

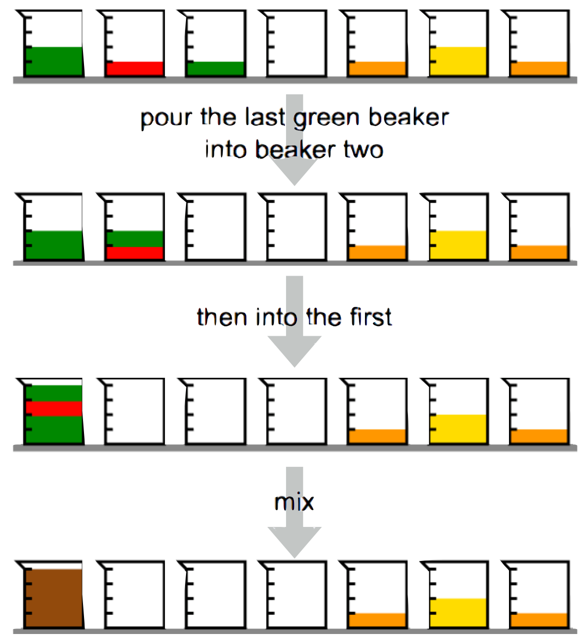
The robot then needs to translate each of these instructions into an executable program.

To train this robot, it is very time consuming to label each command with the right program, or the user may not even know how to write code. In contrast, it's quite easy to move the beakers yourself and demonstrate what should happen.

So, we consider a learning setup where the robot sees a demonstration, but does not actually observe the correct program.

This is really a classic "weak supervision", or "learning from denotation" task, which has had a rich history in the semantic parsing literature.

TASK: parse multi-step instructions into programs




```
pour(hasColor(green)[-1], beakers[2])

pour(prevArg(2), beakers[1])
```

Let's imagine that you're working in a chemistry lab, and you have a robot assistant.
You'd like to say things like "pour the last green beaker into beaker two", etc.

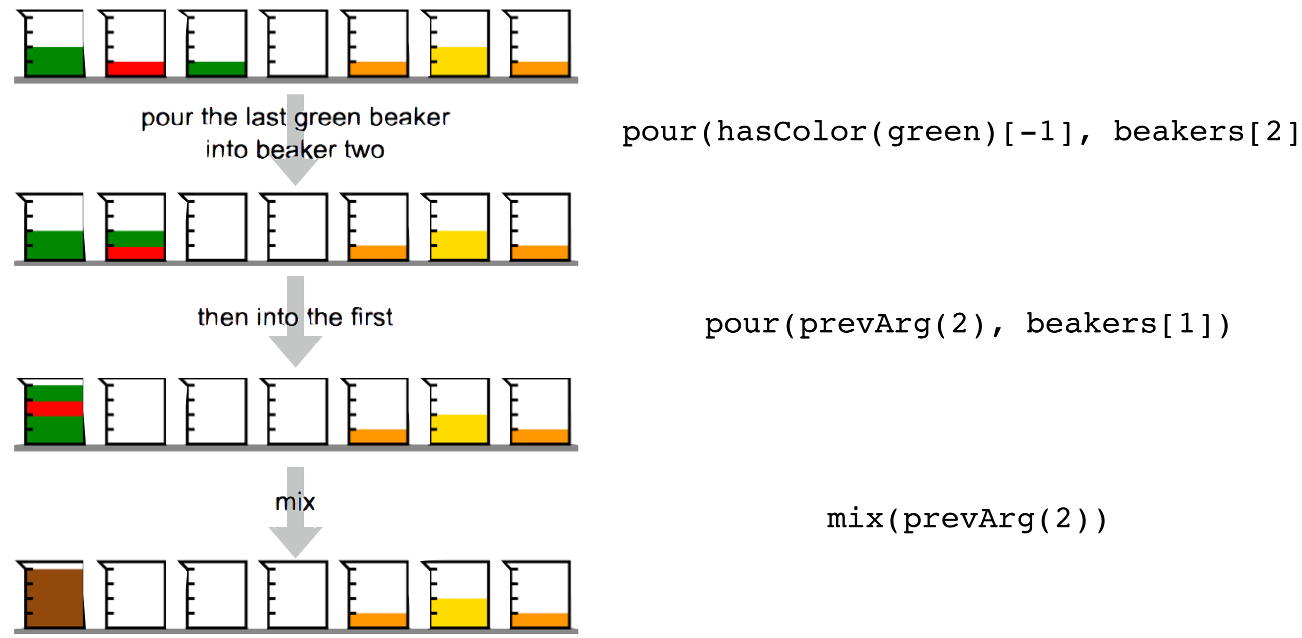
The robot then needs to translate each of these instructions into an executable program.

To train this robot, it is very time consuming to label each command with the right program, or the user may not even know how to write code. In contrast, it's quite easy to move the beakers yourself and demonstrate what should happen.

So, we consider a learning setup where the robot sees a demonstration, but does not actually observe the correct program.

This is really a classic "weak supervision", or "learning from denotation" task, which has had a rich history in the semantic parsing literature.

TASK: parse multi-step instructions into programs



Let's imagine that you're working in a chemistry lab, and you have a robot assistant.
You'd like to say things like "pour the last green beaker into beaker two", etc.

The robot then needs to translate each of these instructions into an executable program.

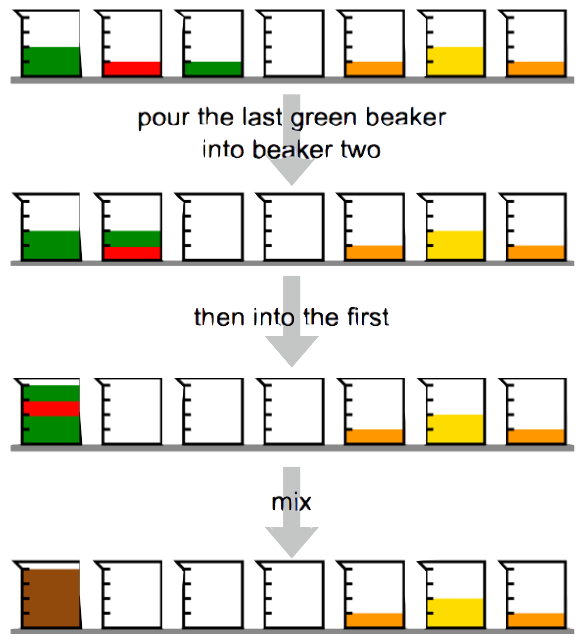
To train this robot, it is very time consuming to label each command with the right program, or the user may not even know how to write code. In contrast, it's quite easy to move the beakers yourself and demonstrate what should happen.

So, we consider a learning setup where the robot sees a demonstration, but does not actually observe the correct program.

This is really a classic "weak supervision", or "learning from denotation" task, which has had a rich history in the semantic parsing literature.

TASK: parse multi-step instructions into programs

OBSERVED




```
pour(hasColor(green)[-1], beakers[2])
pour(prevArg(2), beakers[1])
mix(prevArg(2))
```

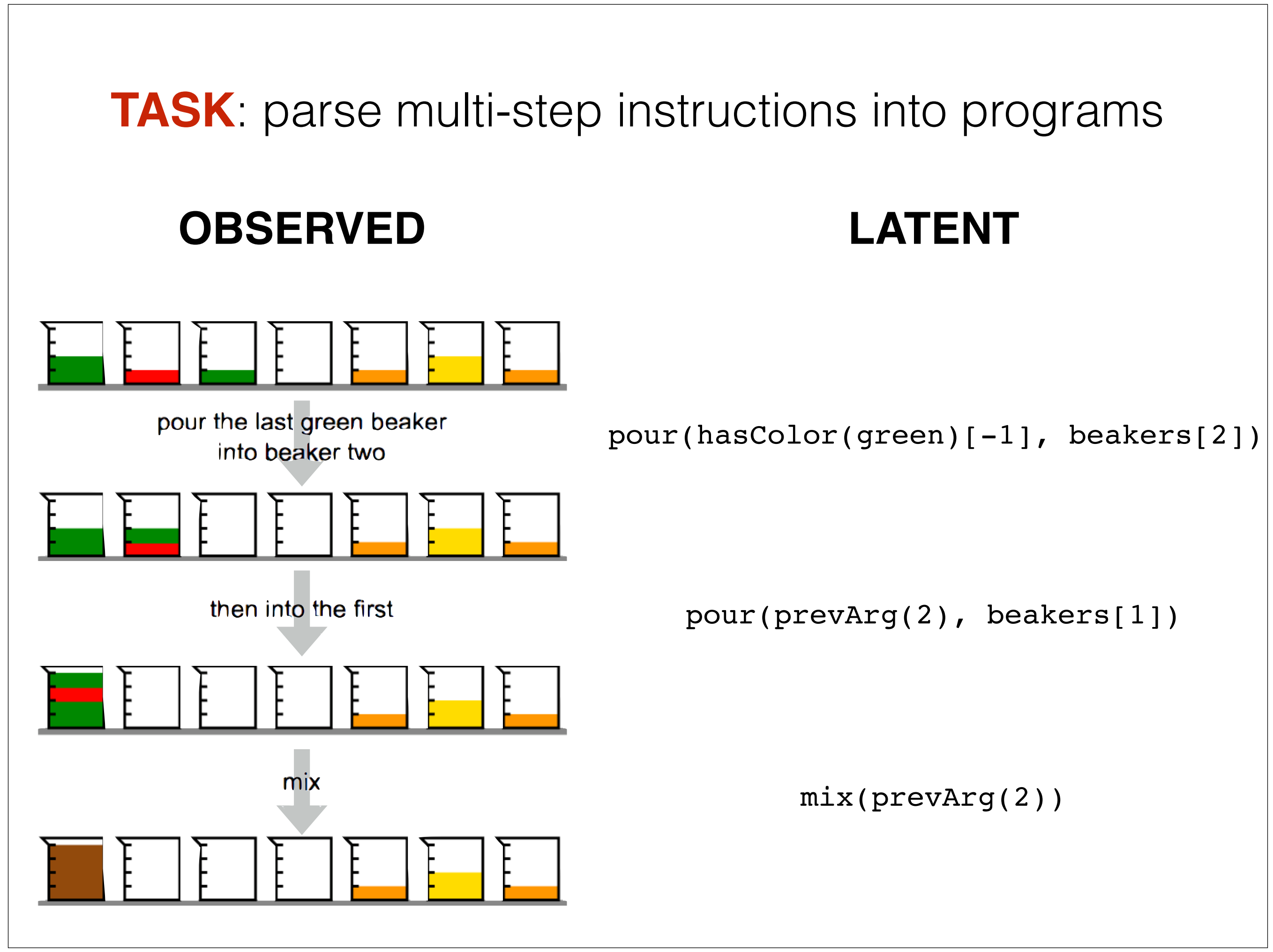
Let's imagine that you're working in a chemistry lab, and you have a robot assistant.
You'd like to say things like "pour the last green beaker into beaker two", etc.

The robot then needs to translate each of these instructions into an executable program.

To train this robot, it is very time consuming to label each command with the right program, or the user may not even know how to write code. In contrast, it's quite easy to move the beakers yourself and demonstrate what should happen.

So, we consider a learning setup where the robot sees a demonstration, but does not actually observe the correct program.

This is really a classic "weak supervision", or "learning from denotation" task, which has had a rich history in the semantic parsing literature.



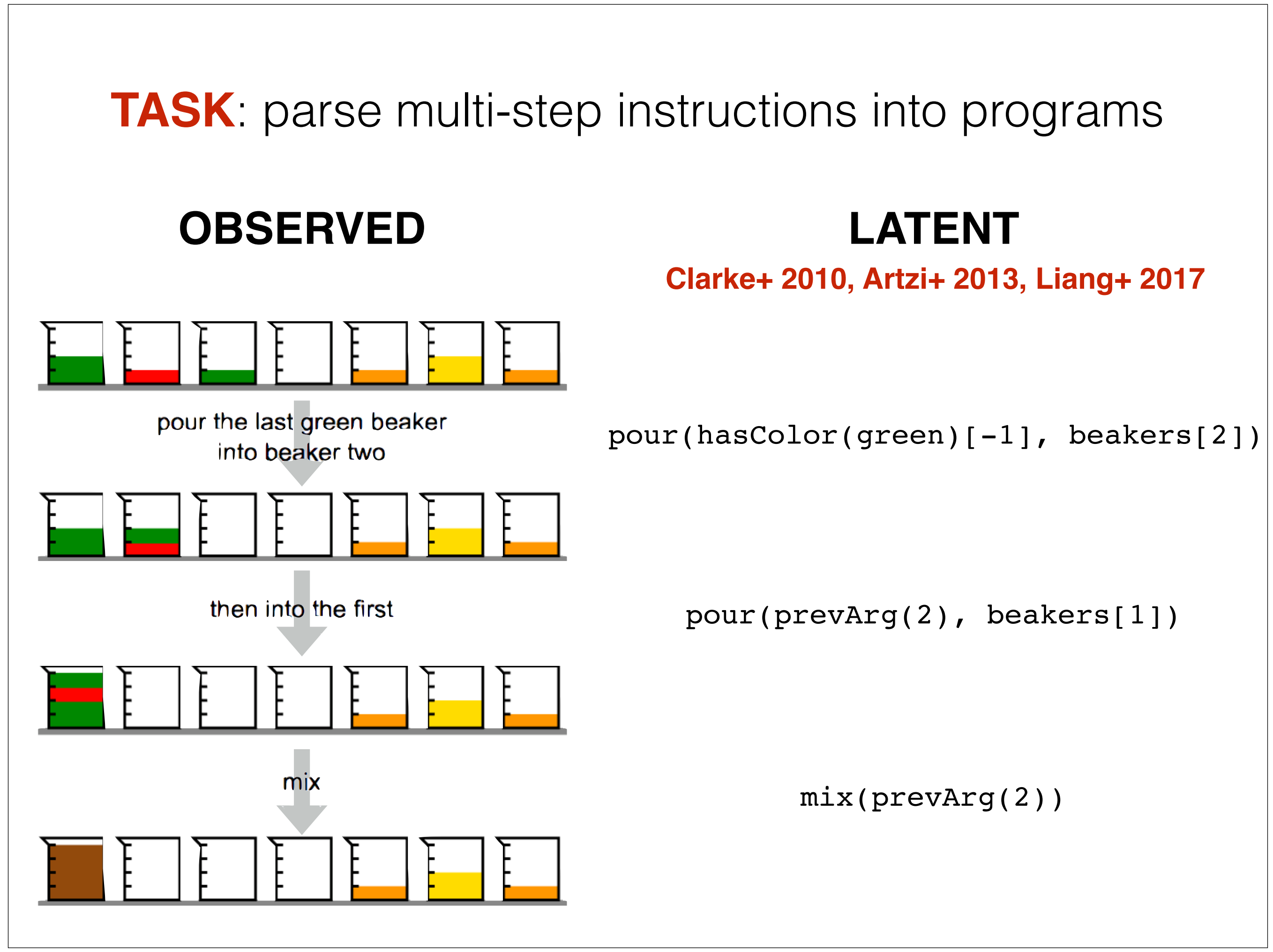
Let's imagine that you're working in a chemistry lab, and you have a robot assistant.
You'd like to say things like "pour the last green beaker into beaker two", etc.

The robot then needs to translate each of these instructions into an executable program.

To train this robot, it is very time consuming to label each command with the right program, or the user may not even know how to write code. In contrast, it's quite easy to move the beakers yourself and demonstrate what should happen.

So, we consider a learning setup where the robot sees a demonstration, but does not actually observe the correct program.

This is really a classic "weak supervision", or "learning from denotation" task, which has had a rich history in the semantic parsing literature.



Let's imagine that you're working in a chemistry lab, and you have a robot assistant.
You'd like to say things like "pour the last green beaker into beaker two", etc.

The robot then needs to translate each of these instructions into an executable program.

To train this robot, it is very time consuming to label each command with the right program, or the user may not even know how to write code. In contrast, it's quite easy to move the beakers yourself and demonstrate what should happen.

So, we consider a learning setup where the robot sees a demonstration, but does not actually observe the correct program.

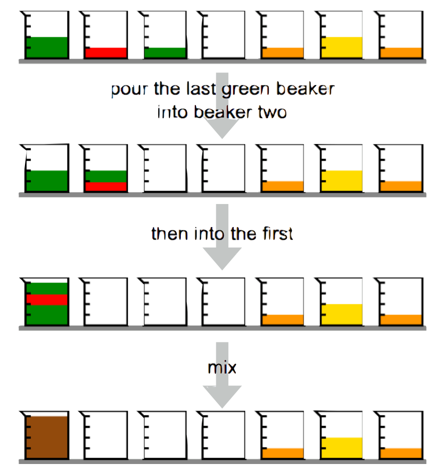
This is really a classic "weak supervision", or "learning from denotation" task, which has had a rich history in the semantic parsing literature.

DATASET: SCONE

(Sequential CONtext-dependent Execution dataset)

Long et al, ACL 2016

ALCHEMY

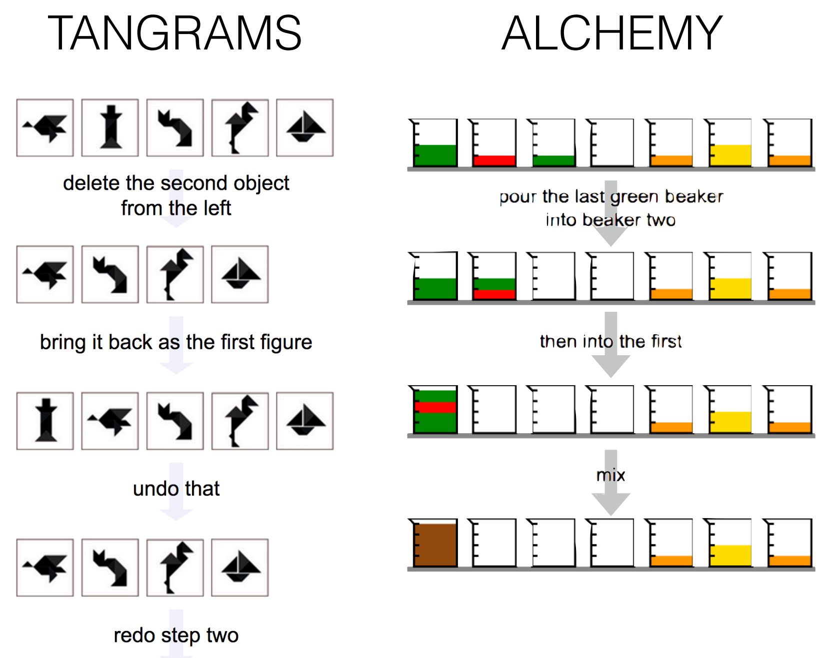


We'll study this kind of weakly supervised parsing task across three different domains, which were all part of a dataset called SCONE, released at ACL last year.

DATASET: SCONE

(Sequential CONtext-dependent Execution dataset)

Long et al, ACL 2016

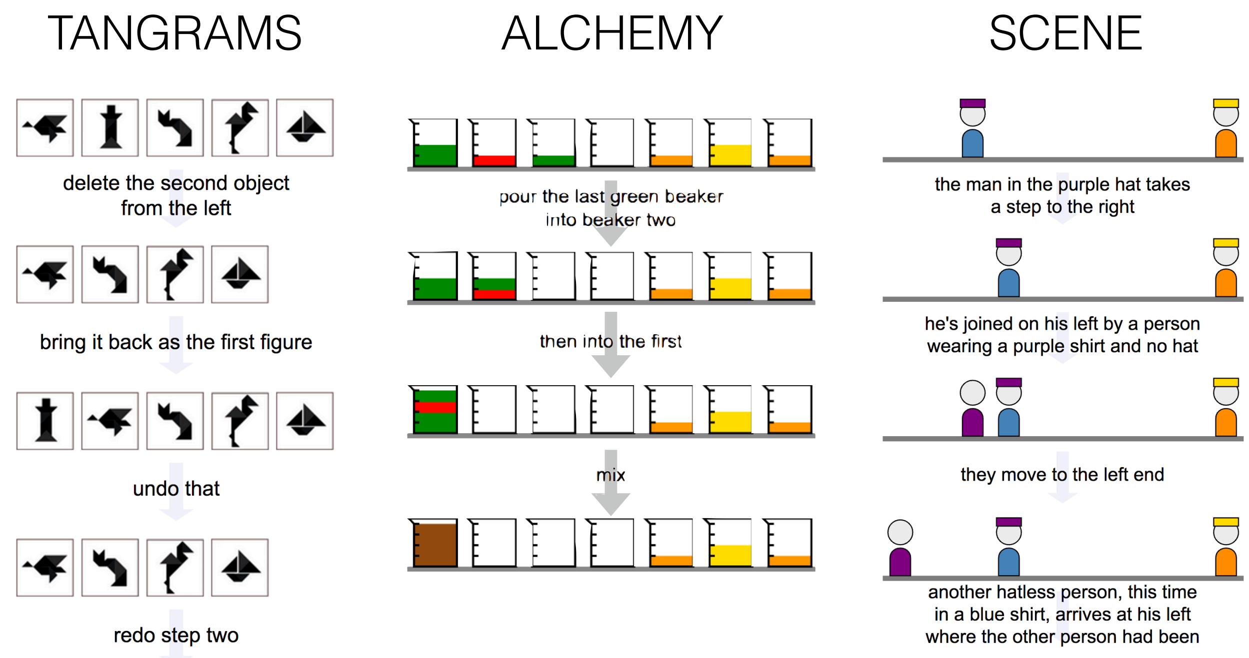


We'll study this kind of weakly supervised parsing task across three different domains, which were all part of a dataset called SCONE, released at ACL last year.

DATASET: **SCONE**

(Sequential CONtext-dependent Execution dataset)

Long et al, ACL 2016



We'll study this kind of weakly supervised parsing task across three different domains, which were all part of a dataset called SCONE, released at ACL last year.

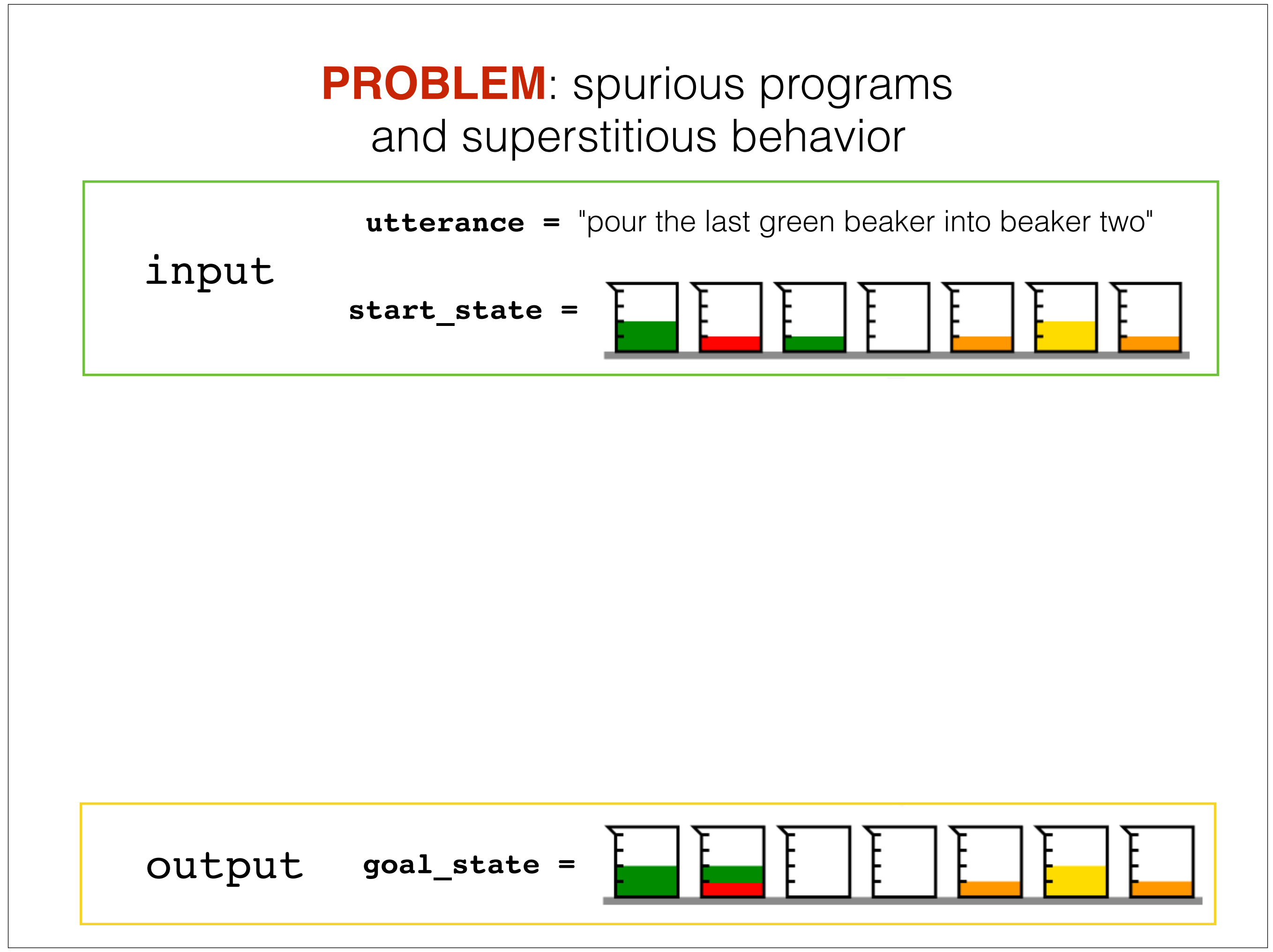
PROBLEM

"spurious programs"
and "superstitious behavior"

For this talk, I'll be highlighting one big problem with weak supervision, and how we solve it.

In particular, researchers have noted that weakly supervised models tend to learn spurious programs, and what we call "superstitious behavior".

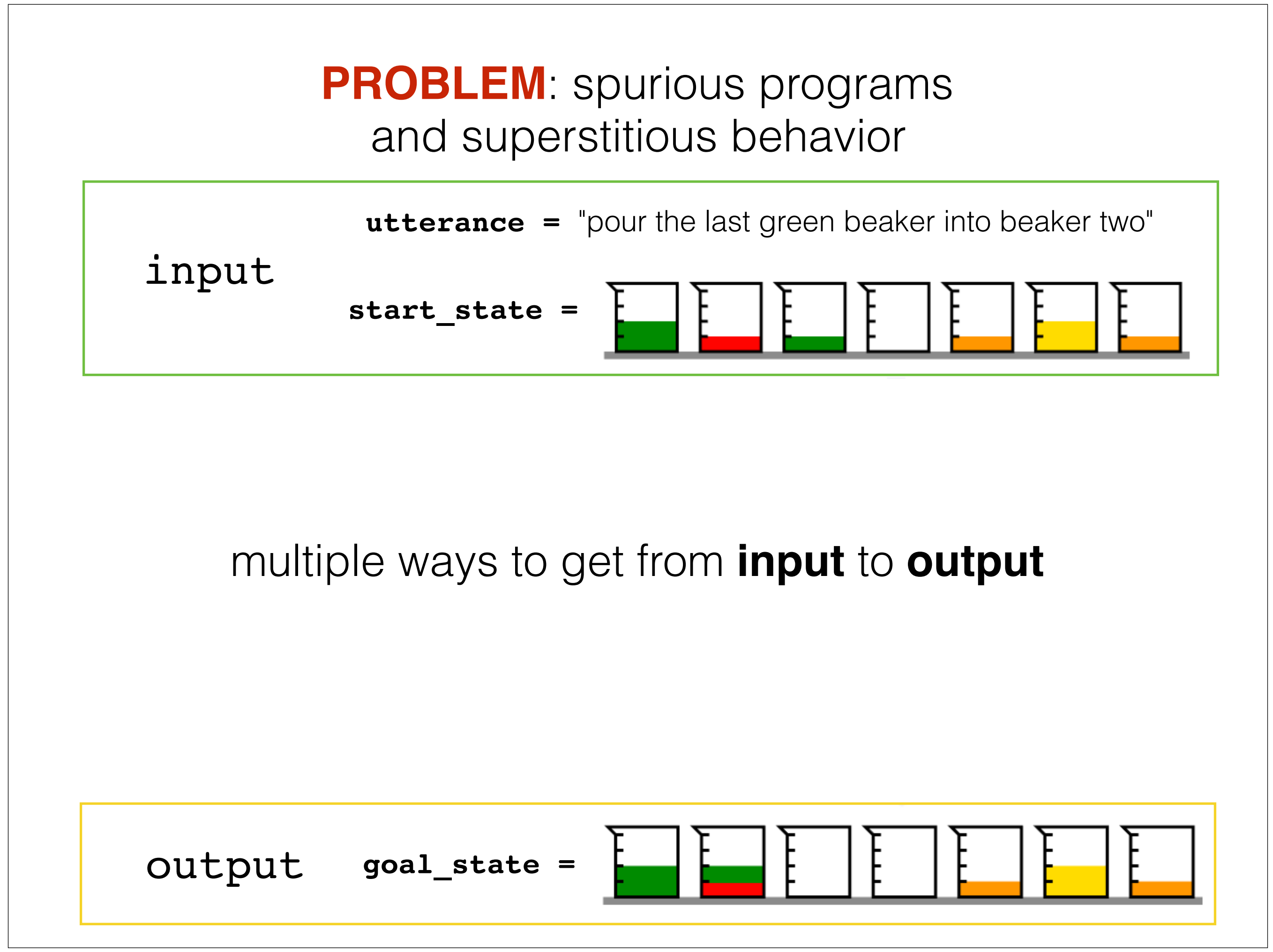
So, what I do mean by that?



At training time, this is what our model sees.

We give it an input, consisting of an utterance and a start_state.
It then has to find a program which produces the output goal_state.

The problem is that there are multiple ways to get from input to output.



At training time, this is what our model sees.

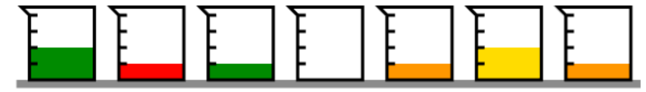
We give it an input, consisting of an utterance and a start_state.
It then has to find a program which produces the output goal_state.

The problem is that there are multiple ways to get from input to output.

PROBLEM: spurious programs and superstitious behavior

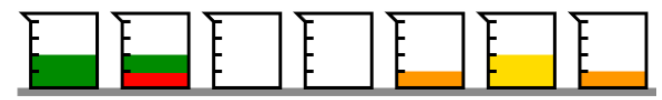
input

utterance = "pour the last green beaker into beaker two"

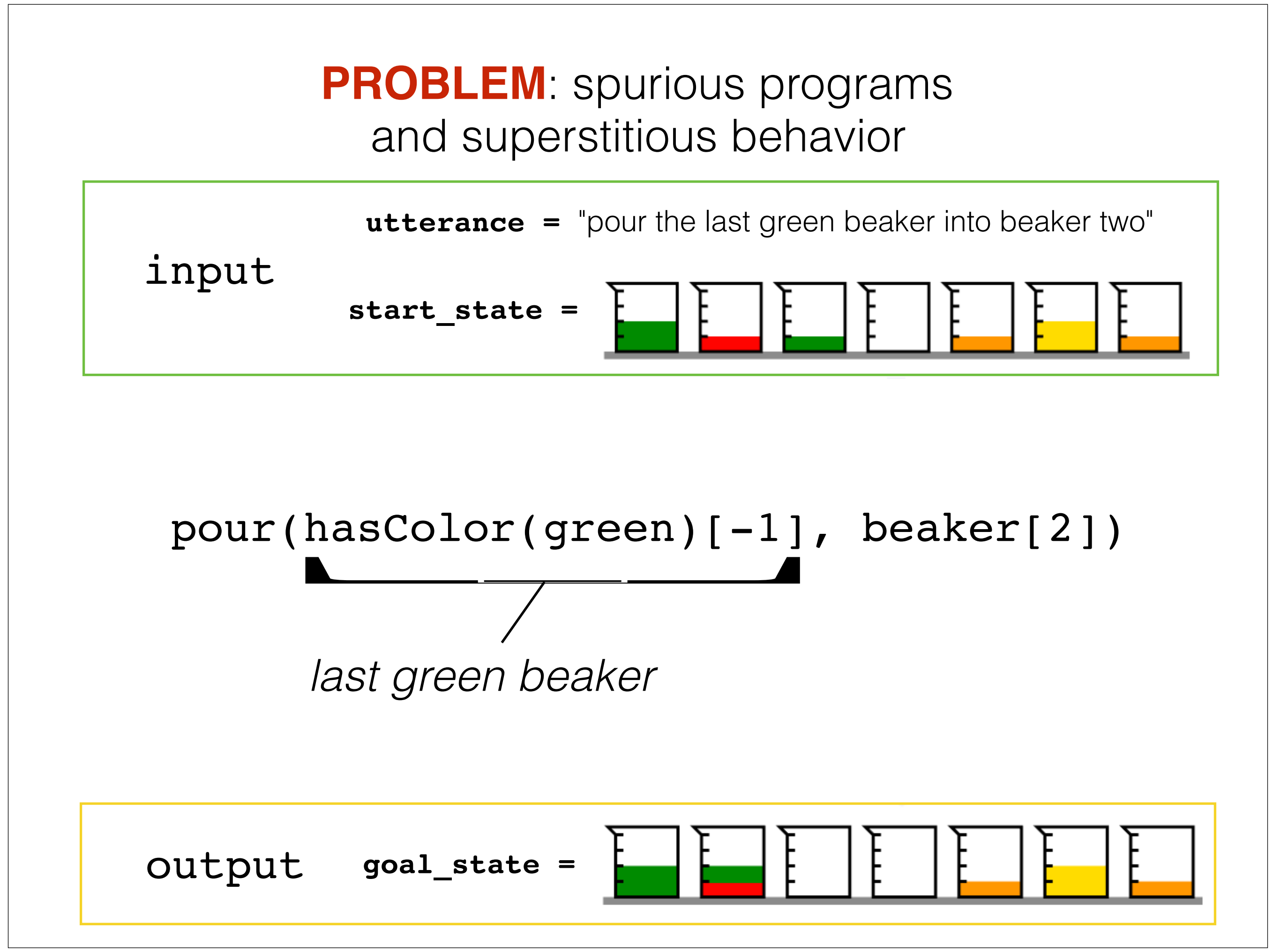
start_state = 

```
pour(hasColor(green)[-1], beaker[2])
```

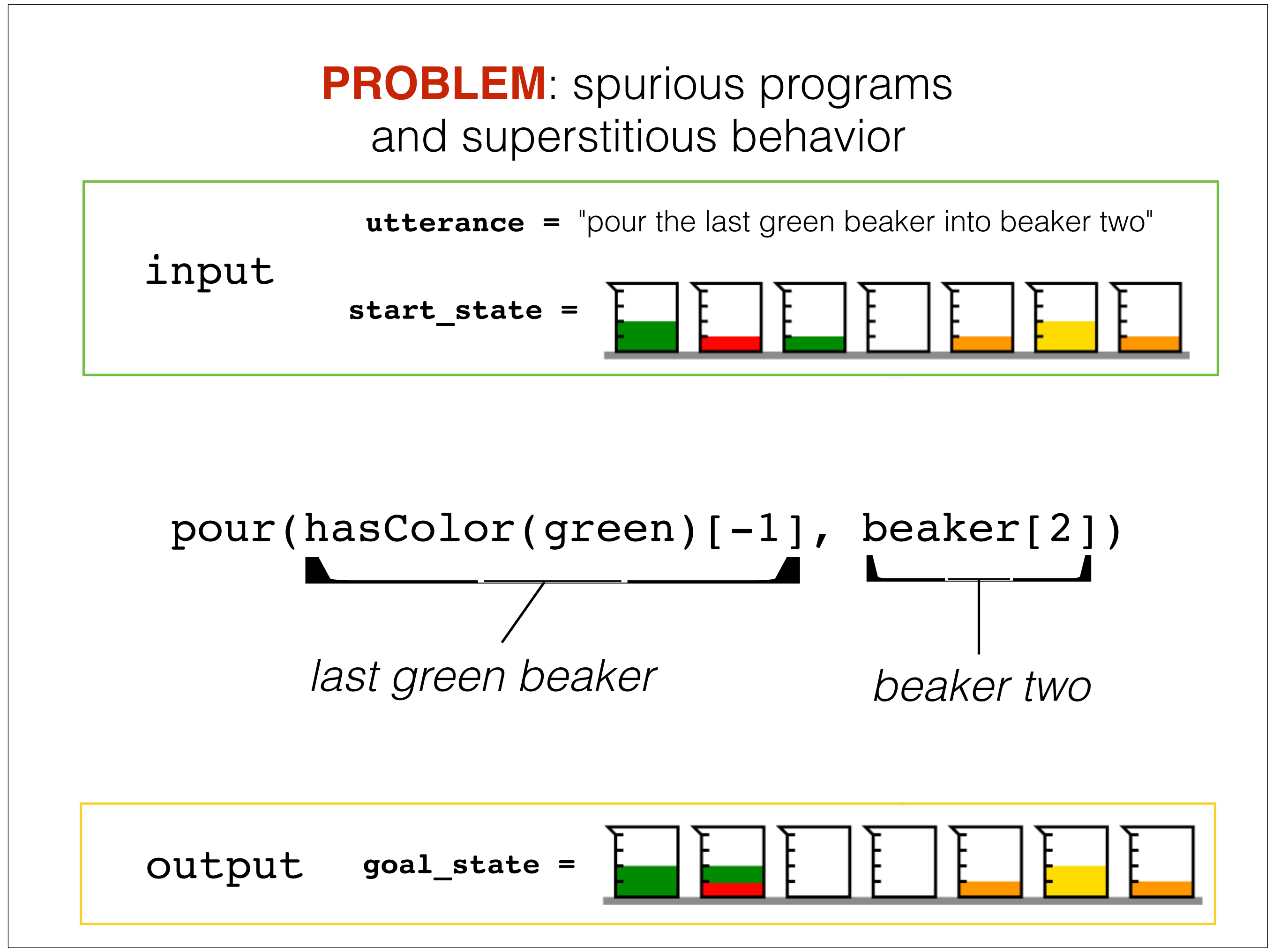
output

goal_state = 

For example, here is the correct program, which accurately reflects the utterance.



For example, here is the correct program, which accurately reflects the utterance.

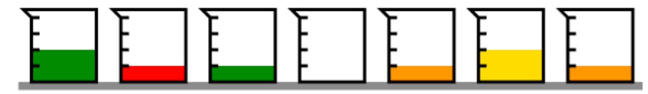


For example, here is the correct program, which accurately reflects the utterance.

PROBLEM: spurious programs and superstitious behavior

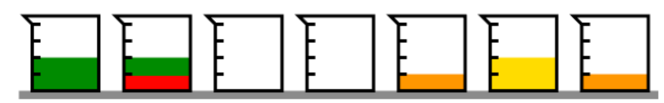
input

utterance = "pour the last green beaker into beaker two"

start_state = 

```
empty(beaker[3]); add(hasColor(red), green)
```

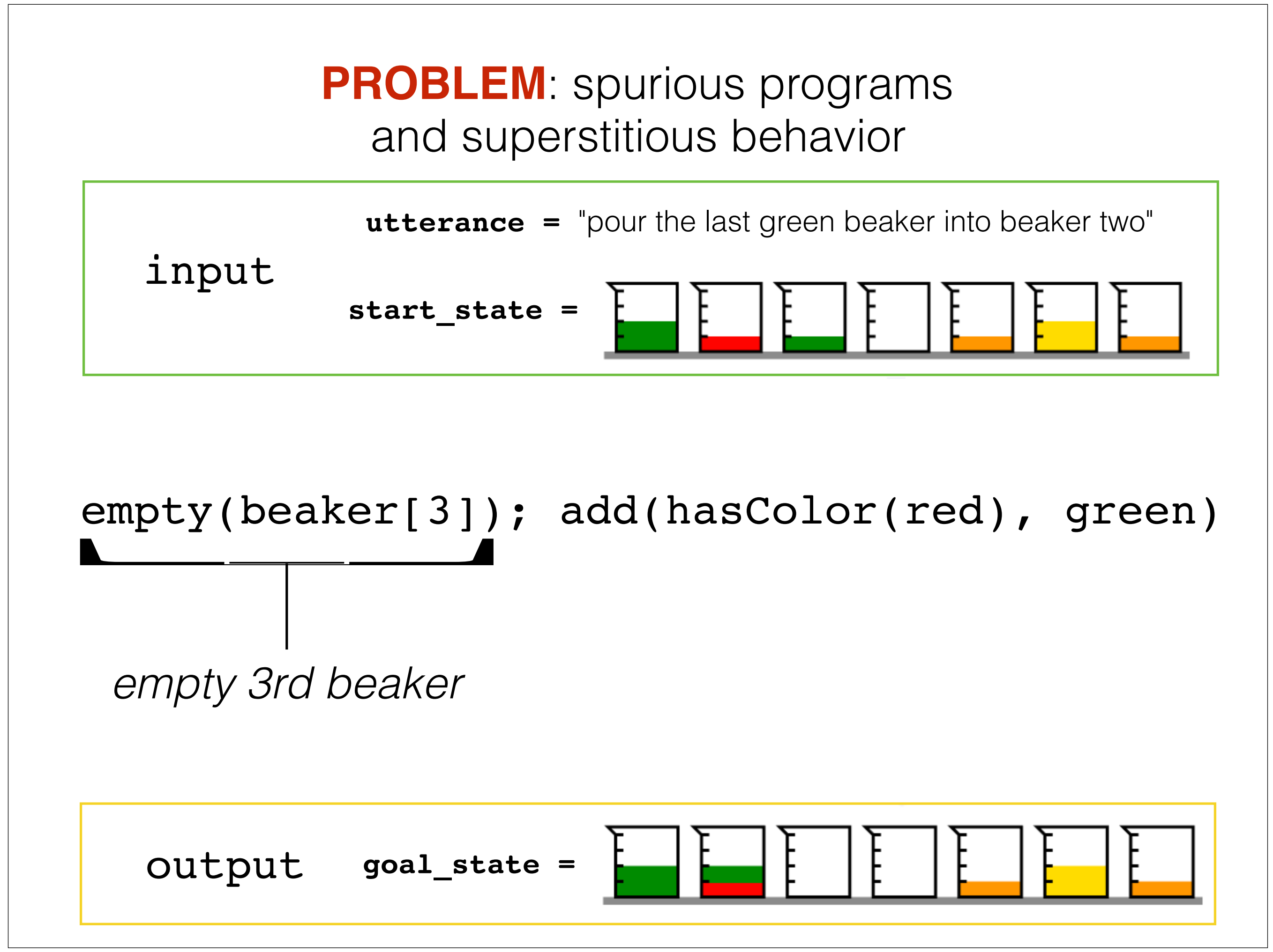
output

goal_state = 

But this program also gets us to the goal state.

If I switched to a different chemistry table and gave the same command, I would not want to run this program.

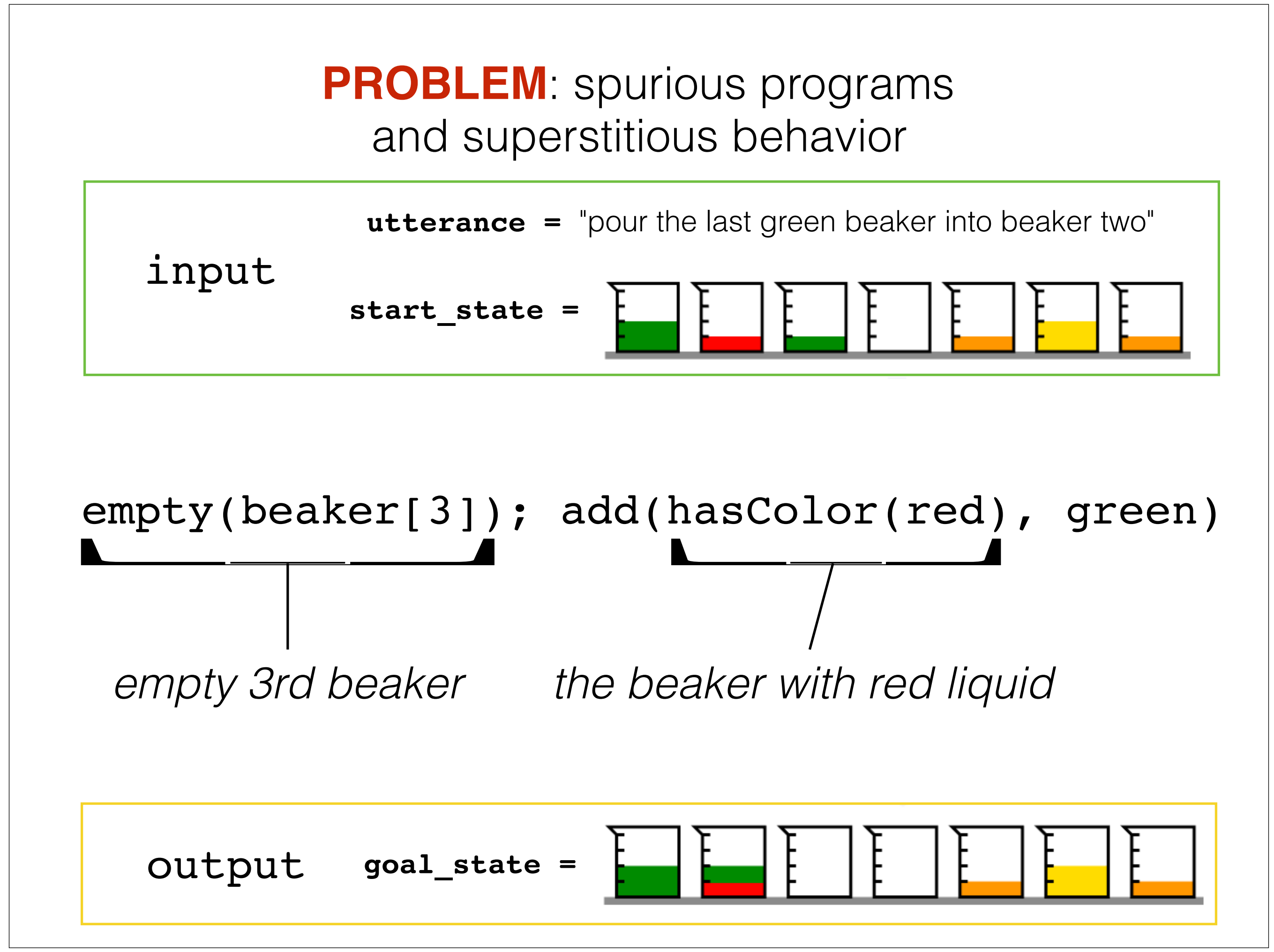
Ironically, this problem gets worse as your programming language becomes more powerful.
A more expressive language means there are more wrong ways to get to the goal_state.



But this program also gets us to the goal state.

If I switched to a different chemistry table and gave the same command, I would not want to run this program.

Ironically, this problem gets worse as your programming language becomes more powerful.
A more expressive language means there are more wrong ways to get to the goal_state.



But this program also gets us to the goal state.

If I switched to a different chemistry table and gave the same command, I would not want to run this program.

Ironically, this problem gets worse as your programming language becomes more powerful.
A more expressive language means there are more wrong ways to get to the goal_state.

Superstitious behavior

Icons made by *Freepik* and Creative Stall Premium from www.flaticon.com

So, this is really a problem you might encounter any time you are working with weak supervision.

In fact, humans certainly aren't immune to this either.

At some point, we've all studied hard for an exam. Maybe we did well on that exam, from which we rightly concluded that studying helps.

But maybe we were also wearing purple socks that day. And now we think, "hmm, my purple socks are lucky".

Superstitious behavior

study hard for exam

Icons made by *Freepik* and Creative Stall Premium from www.flaticon.com

So, this is really a problem you might encounter any time you are working with weak supervision.

In fact, humans certainly aren't immune to this either.

At some point, we've all studied hard for an exam. Maybe we did well on that exam, from which we rightly concluded that studying helps.

But maybe we were also wearing purple socks that day. And now we think, "hmm, my purple socks are lucky".

So, this is really a problem you might encounter any time you are working with weak supervision.

In fact, humans certainly aren't immune to this either.

At some point, we've all studied hard for an exam. Maybe we did well on that exam, from which we rightly concluded that studying helps.

But maybe we were also wearing purple socks that day. And now we think, "hmm, my purple socks are lucky".

Superstitious behavior

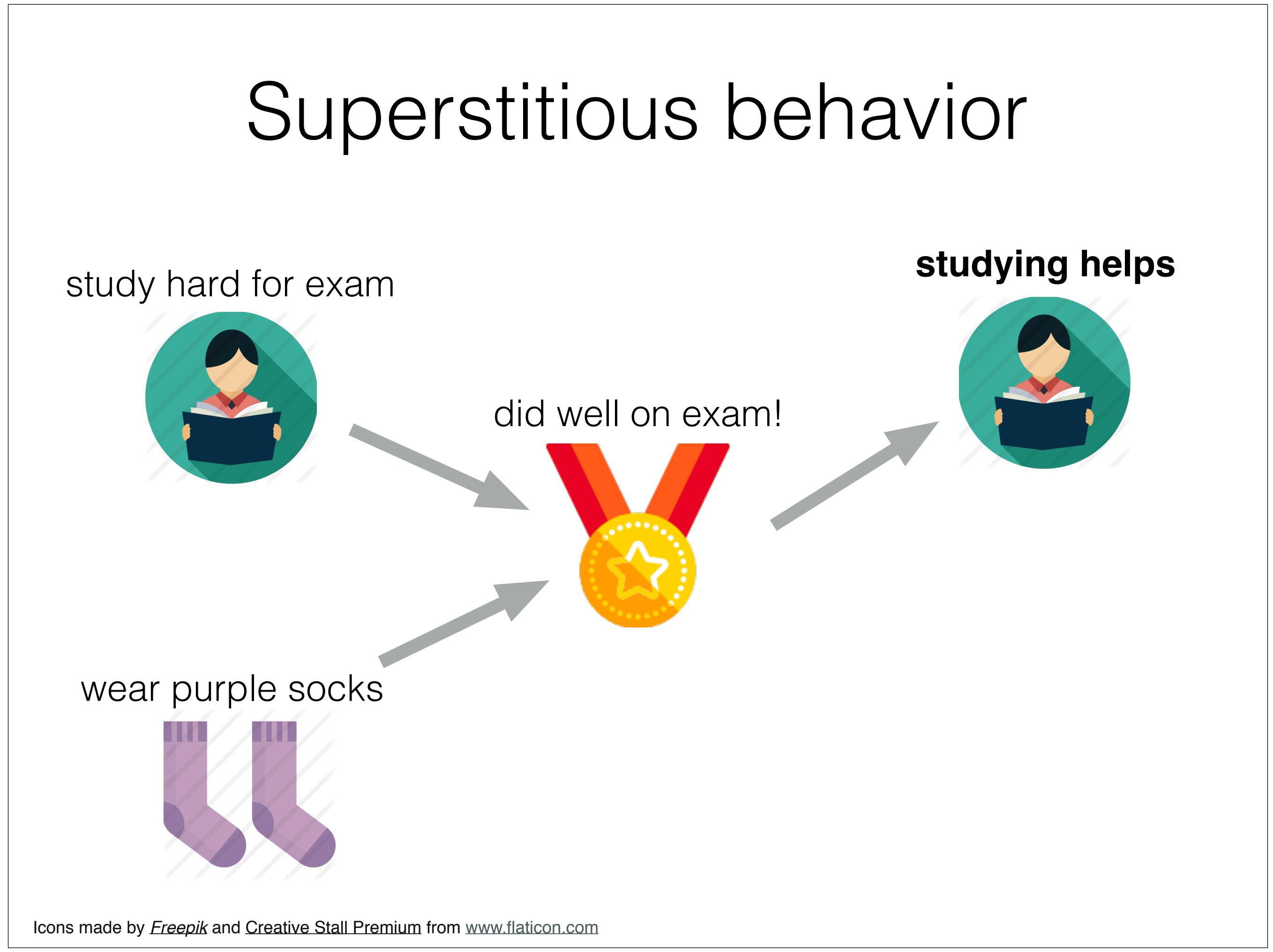
Icons made by *Freepik* and Creative Stall Premium from www.flaticon.com

So, this is really a problem you might encounter any time you are working with weak supervision.

In fact, humans certainly aren't immune to this either.

At some point, we've all studied hard for an exam. Maybe we did well on that exam, from which we rightly concluded that studying helps.

But maybe we were also wearing purple socks that day. And now we think, "hmm, my purple socks are lucky".

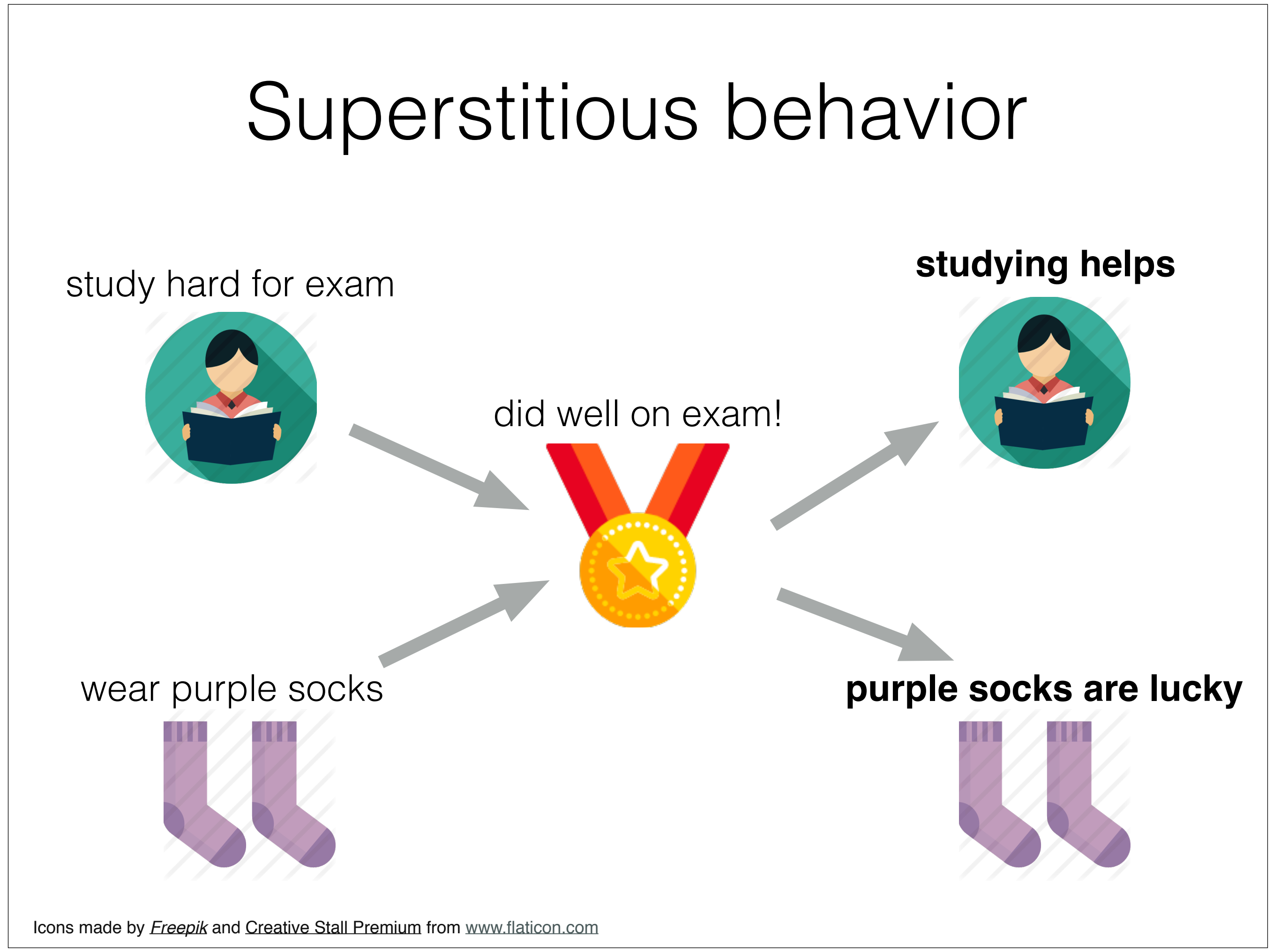


So, this is really a problem you might encounter any time you are working with weak supervision.

In fact, humans certainly aren't immune to this either.

At some point, we've all studied hard for an exam. Maybe we did well on that exam, from which we rightly concluded that studying helps.

But maybe we were also wearing purple socks that day. And now we think, "hmm, my purple socks are lucky".

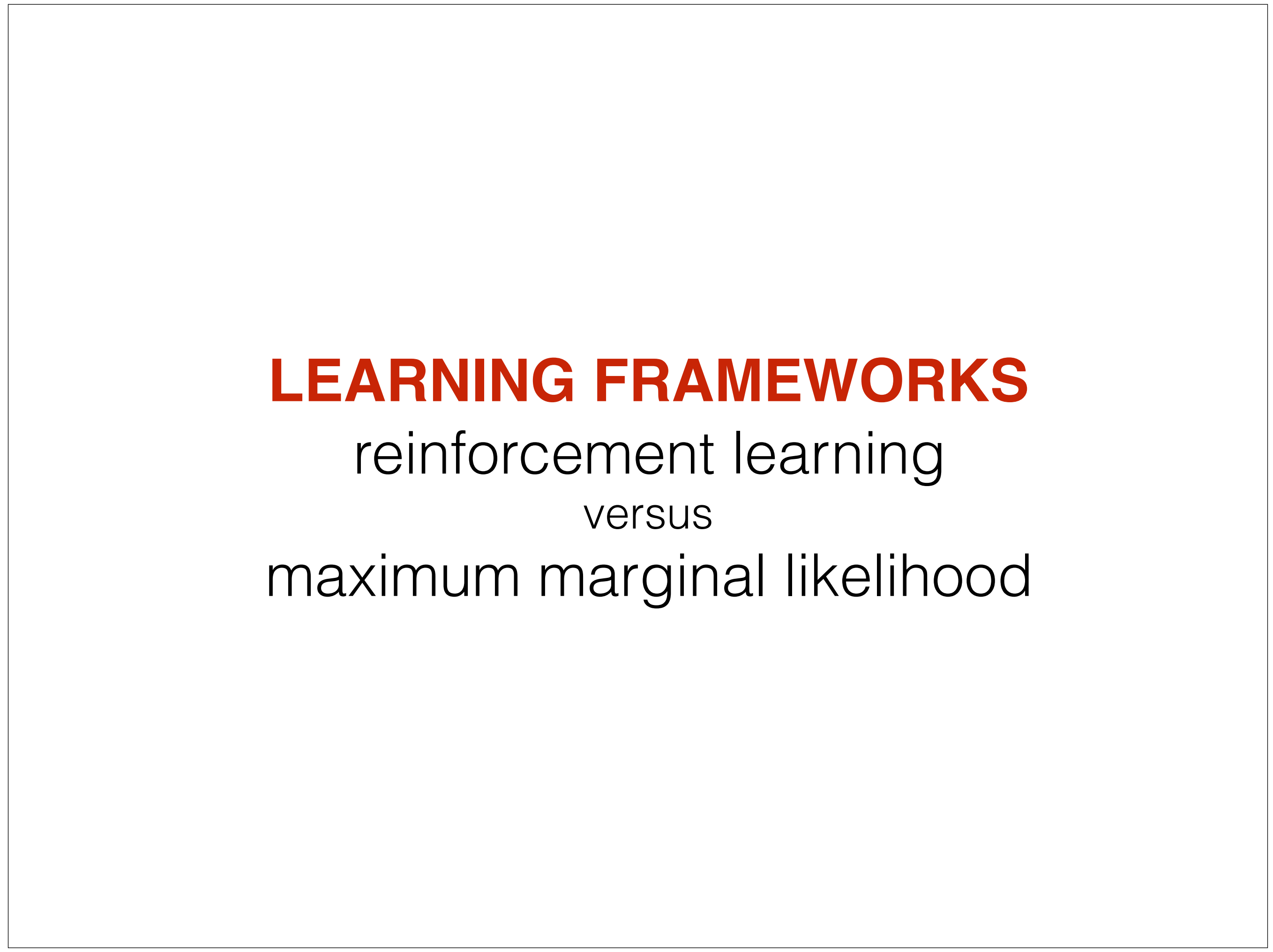


So, this is really a problem you might encounter any time you are working with weak supervision.

In fact, humans certainly aren't immune to this either.

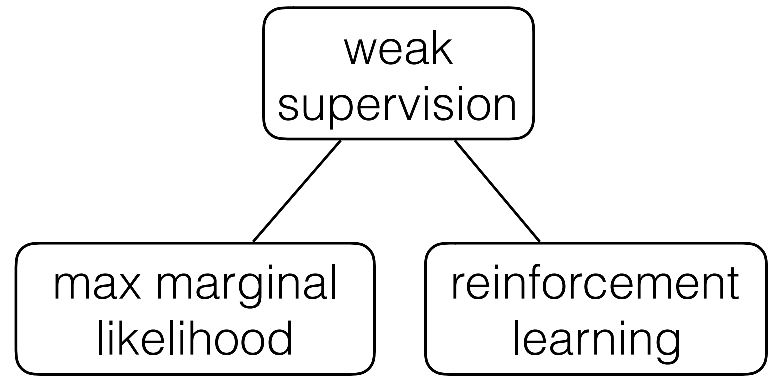
At some point, we've all studied hard for an exam. Maybe we did well on that exam, from which we rightly concluded that studying helps.

But maybe we were also wearing purple socks that day. And now we think, "hmm, my purple socks are lucky".



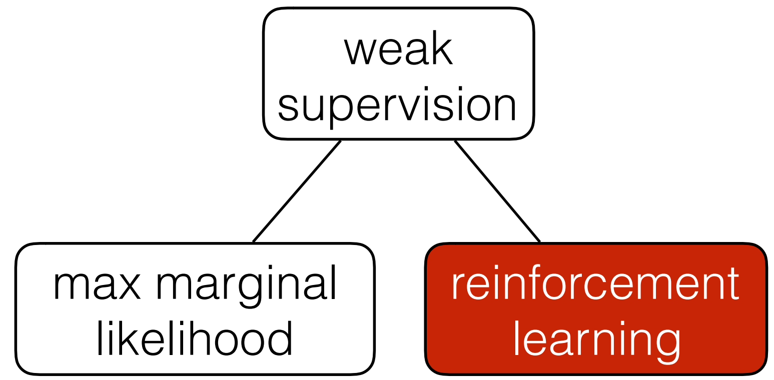
So, enough about humans. Let's now look at how learning algorithms get confused.

LEARNING FRAMEWORKS



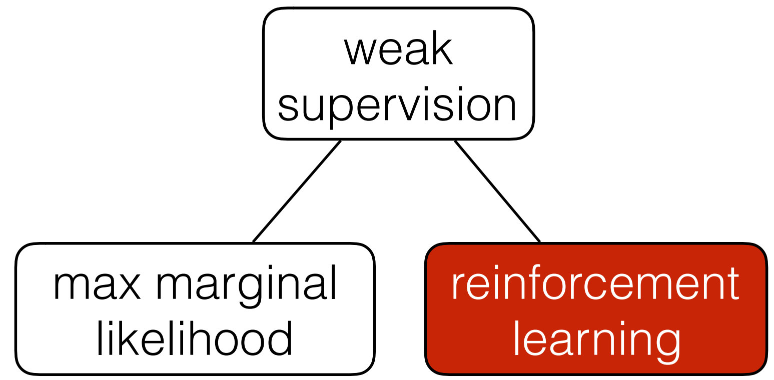
We'll look at two common approaches to the weak supervision problem.

LEARNING FRAMEWORKS



Starting with reinforcement learning.

LEARNING FRAMEWORKS



**Branavan 2009,
Norouzi+ 2016,
Liang+ 2017**

Starting with reinforcement learning.

program generation

```
pour(prevArg(2), beakers[1])
```

The first thing we need to do is convert program generation into a sequential decision making problem.

By reformatting the code in postfix notation, we can represent the program as a sequence of tokens.

And now the task is simply to sequentially generate tokens from left to right.

program generation

```
pour(prevArg(2), beakers[1])
```

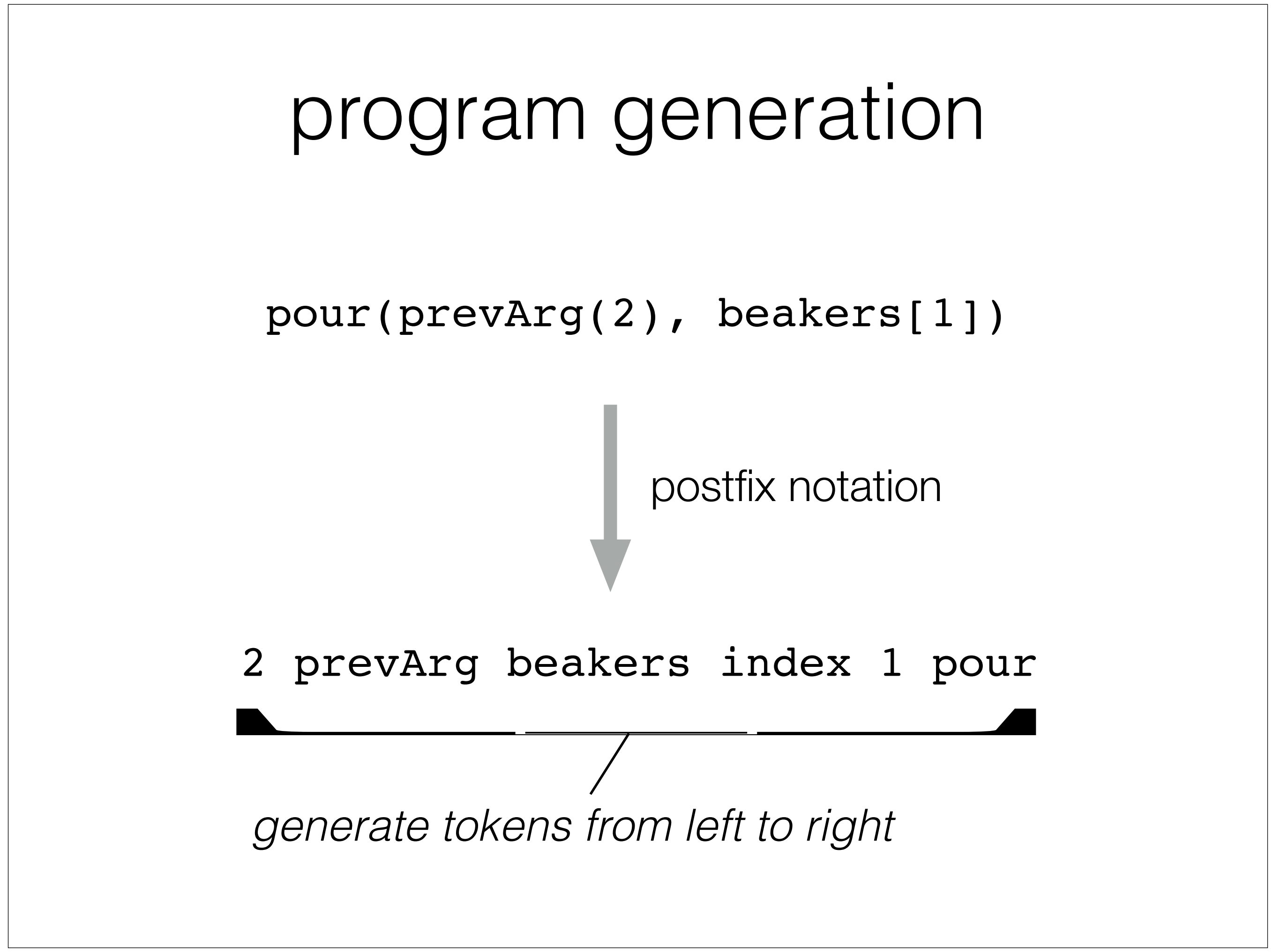
postfix notation

```
2 prevArg beakers index 1 pour
```

The first thing we need to do is convert program generation into a sequential decision making problem.

By reformatting the code in postfix notation, we can represent the program as a sequence of tokens.

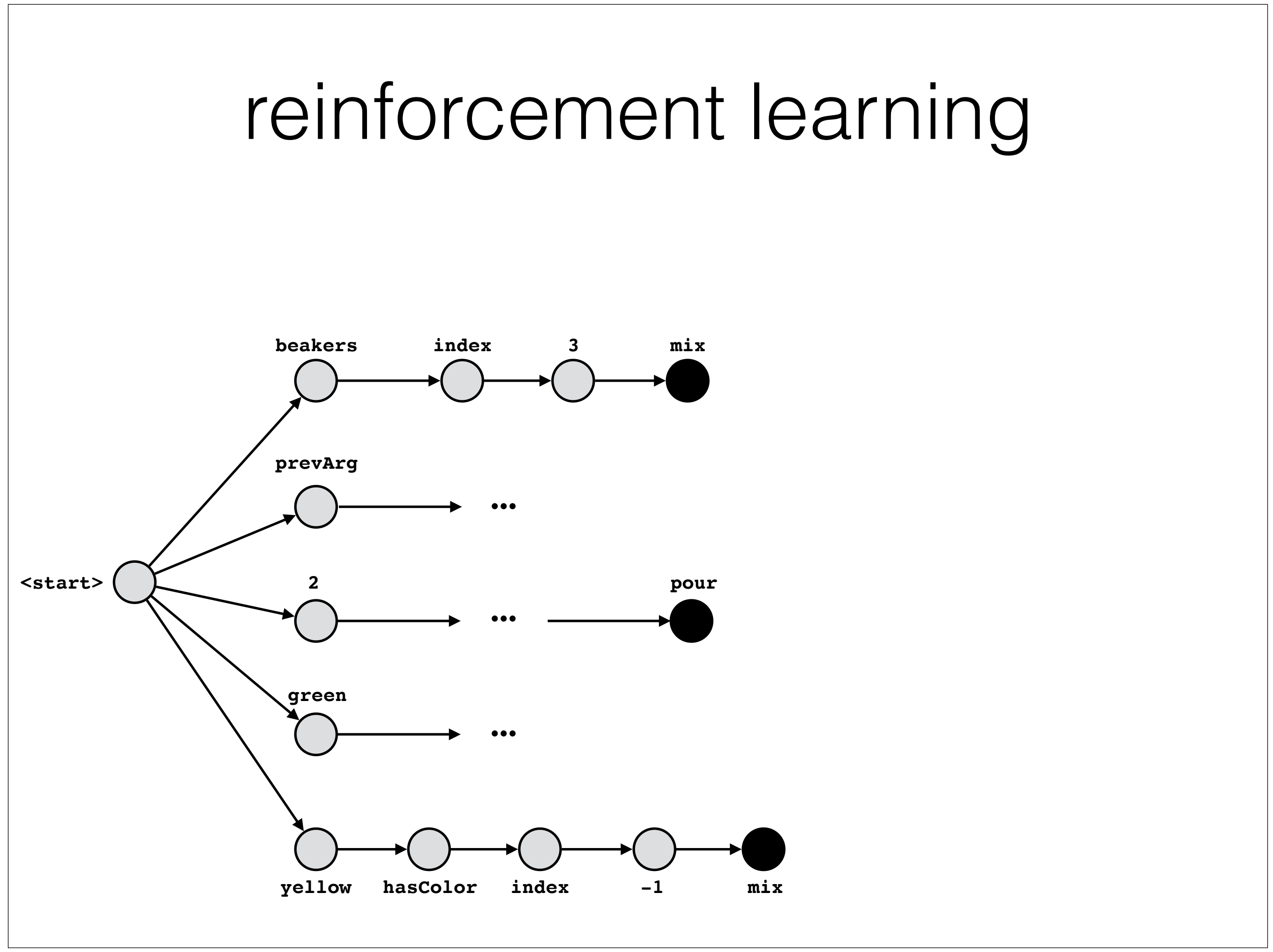
And now the task is simply to sequentially generate tokens from left to right.



The first thing we need to do is convert program generation into a sequential decision making problem.

By reformatting the code in postfix notation, we can represent the program as a sequence of tokens.

And now the task is simply to sequentially generate tokens from left to right.



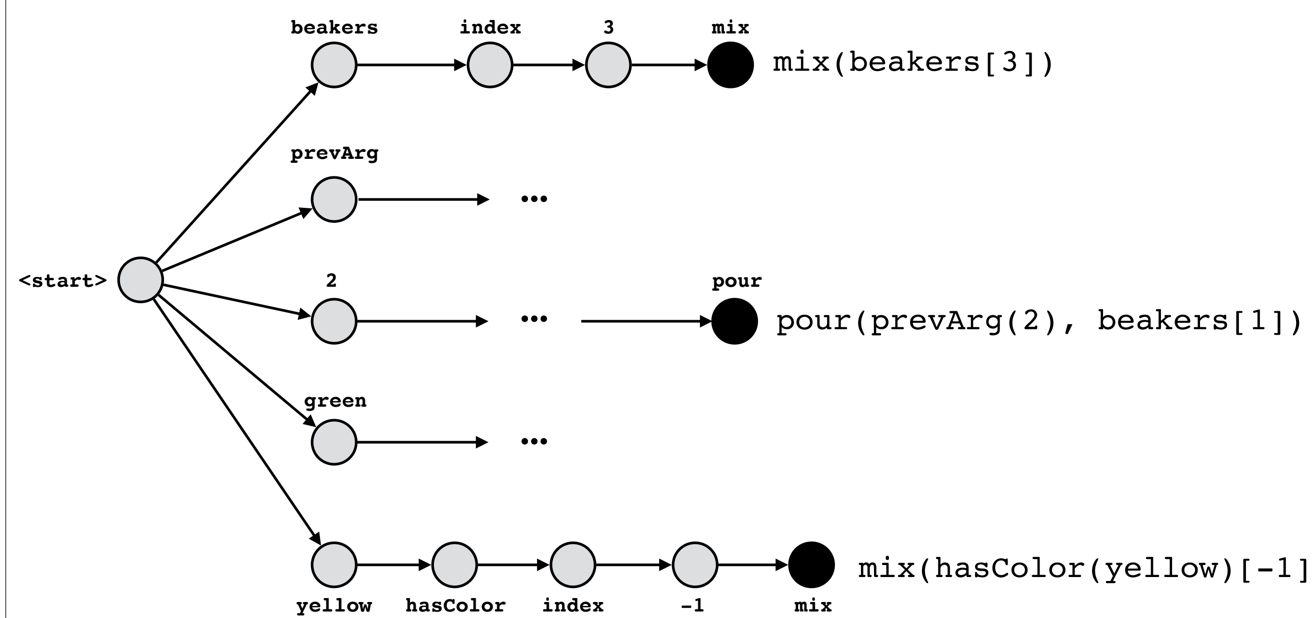
If we organize all possible program sequences in a prefix trie, we get this picture.

From the RL point of view, each node in the tree is a state, and the arrows are actions which transition you to new states.

When you hit a leaf node, the episode terminates.
And the path that you took uniquely defines a complete program.

Given this utterance, the right response is to take the path highlighted in yellow.

reinforcement learning

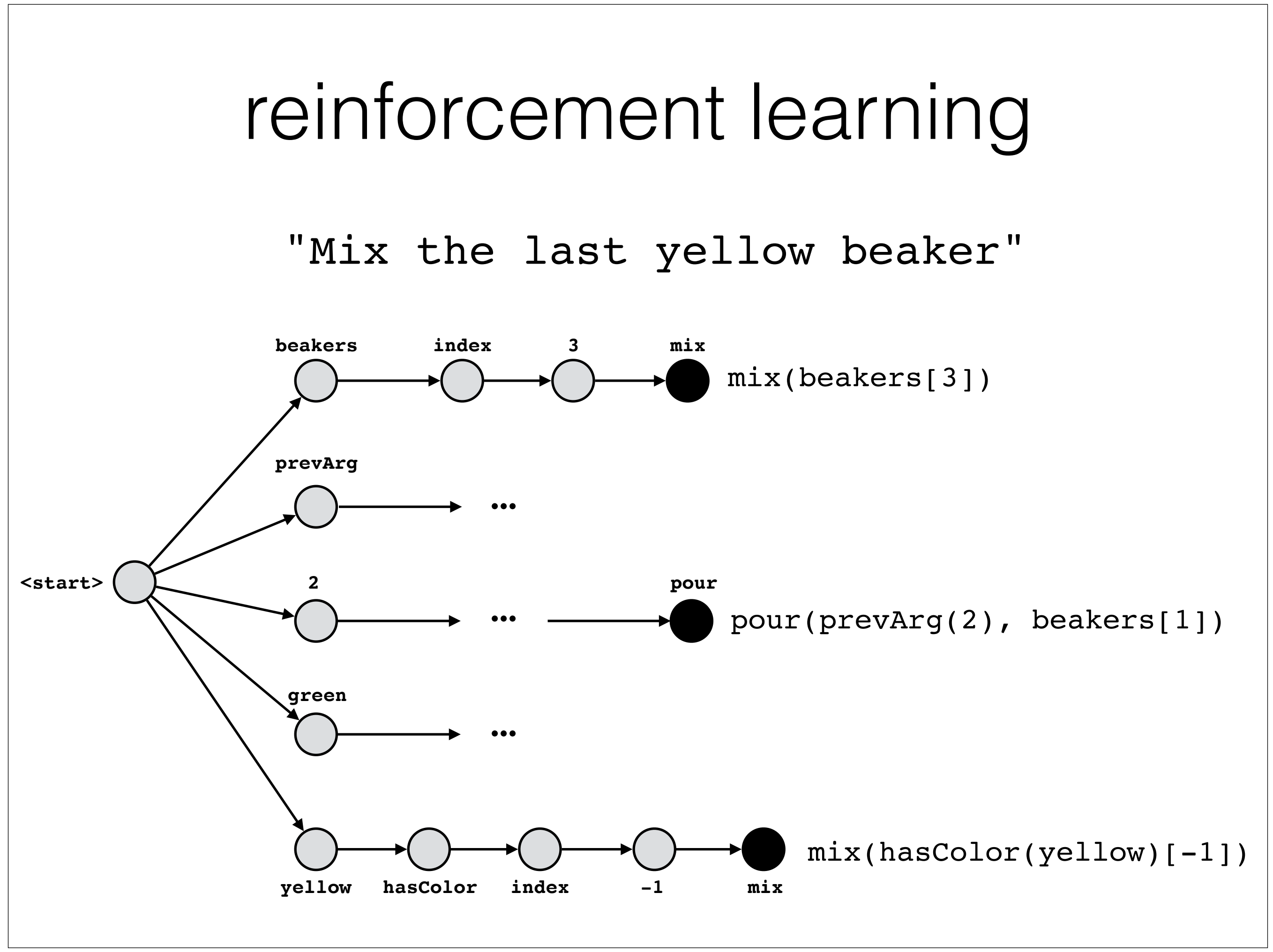


If we organize all possible program sequences in a prefix trie, we get this picture.

From the RL point of view, each node in the tree is a state, and the arrows are actions which transition you to new states.

When you hit a leaf node, the episode terminates.
And the path that you took uniquely defines a complete program.

Given this utterance, the right response is to take the path highlighted in yellow.

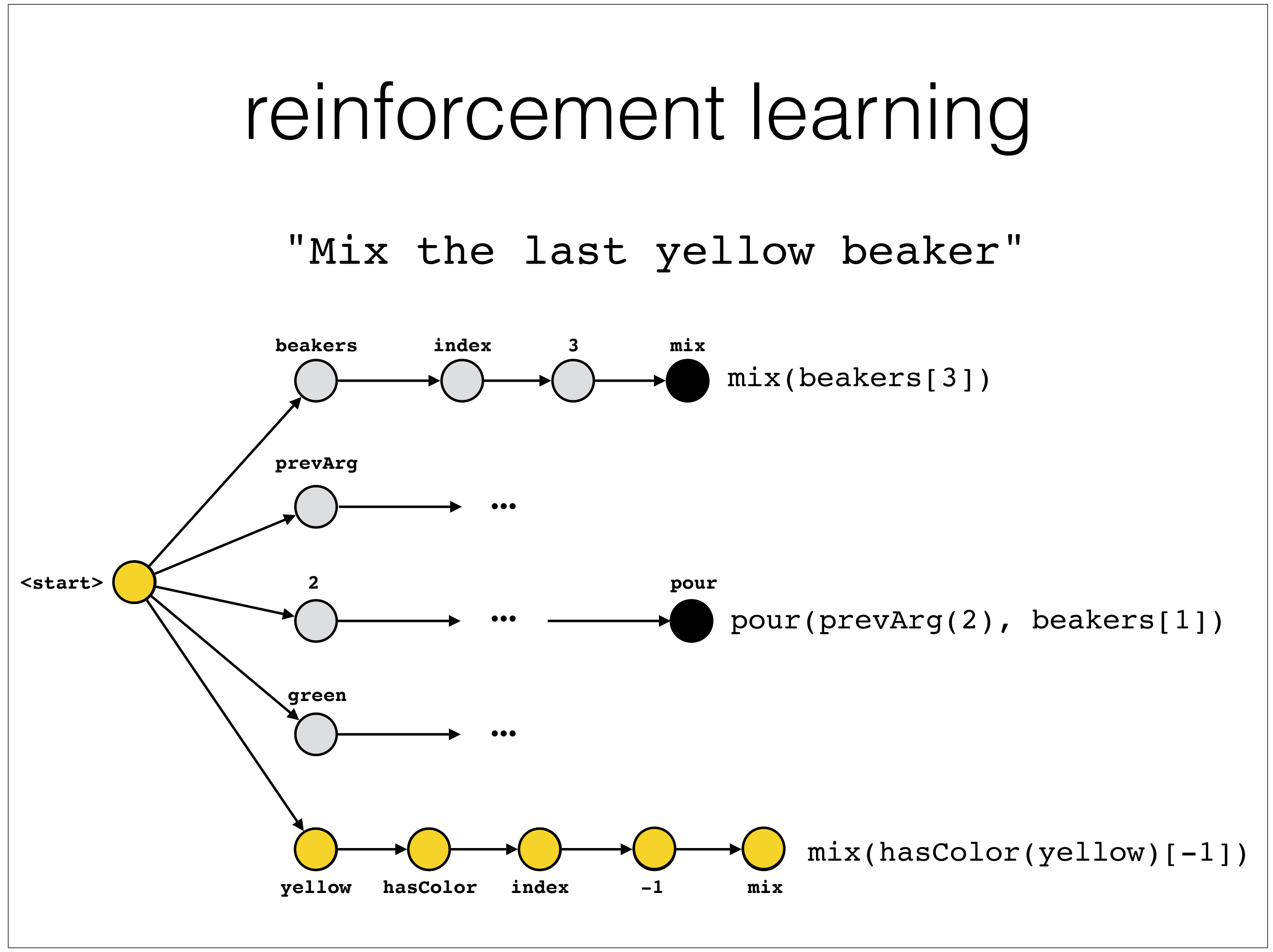


If we organize all possible program sequences in a prefix trie, we get this picture.

From the RL point of view, each node in the tree is a state, and the arrows are actions which transition you to new states.

When you hit a leaf node, the episode terminates.
And the path that you took uniquely defines a complete program.

Given this utterance, the right response is to take the path highlighted in yellow.



If we organize all possible program sequences in a prefix trie, we get this picture.

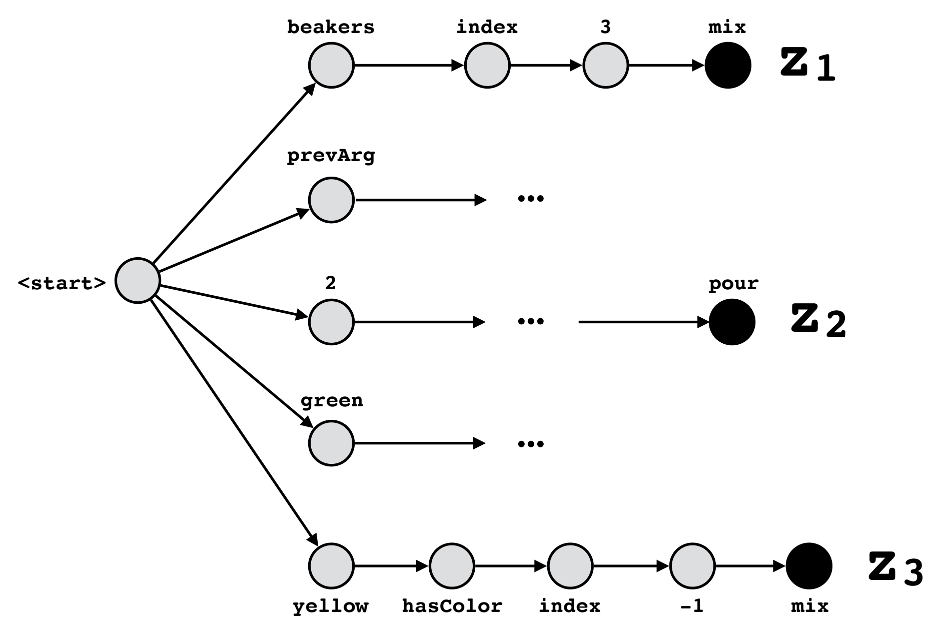
From the RL point of view, each node in the tree is a state, and the arrows are actions which transition you to new states.

When you hit a leaf node, the episode terminates.
And the path that you took uniquely defines a complete program.

Given this utterance, the right response is to take the path highlighted in yellow.

Reinforcement learning

x = "Mix the last yellow beaker"



Notation-wise, we'll call the utterance x, and we'll call each program z.

Reinforcement learning

```
x = "Mix the last yellow beaker"
```

Once the agent generates a program, we then execute the program and give it reward 1 if the output is right, 0 otherwise.

The agent itself has a stochastic policy, meaning that there is some randomness in the actions it takes.

We then train the agent to maximize its expected reward.

Reinforcement learning

`x = "Mix the last yellow beaker"`

RL agent generates program

z

Once the agent generates a program, we then execute the program and give it reward 1 if the output is right, 0 otherwise.

The agent itself has a stochastic policy, meaning that there is some randomness in the actions it takes.

We then train the agent to maximize its expected reward.

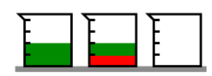
Reinforcement learning

`x = "Mix the last yellow beaker"`

RL agent generates program

z

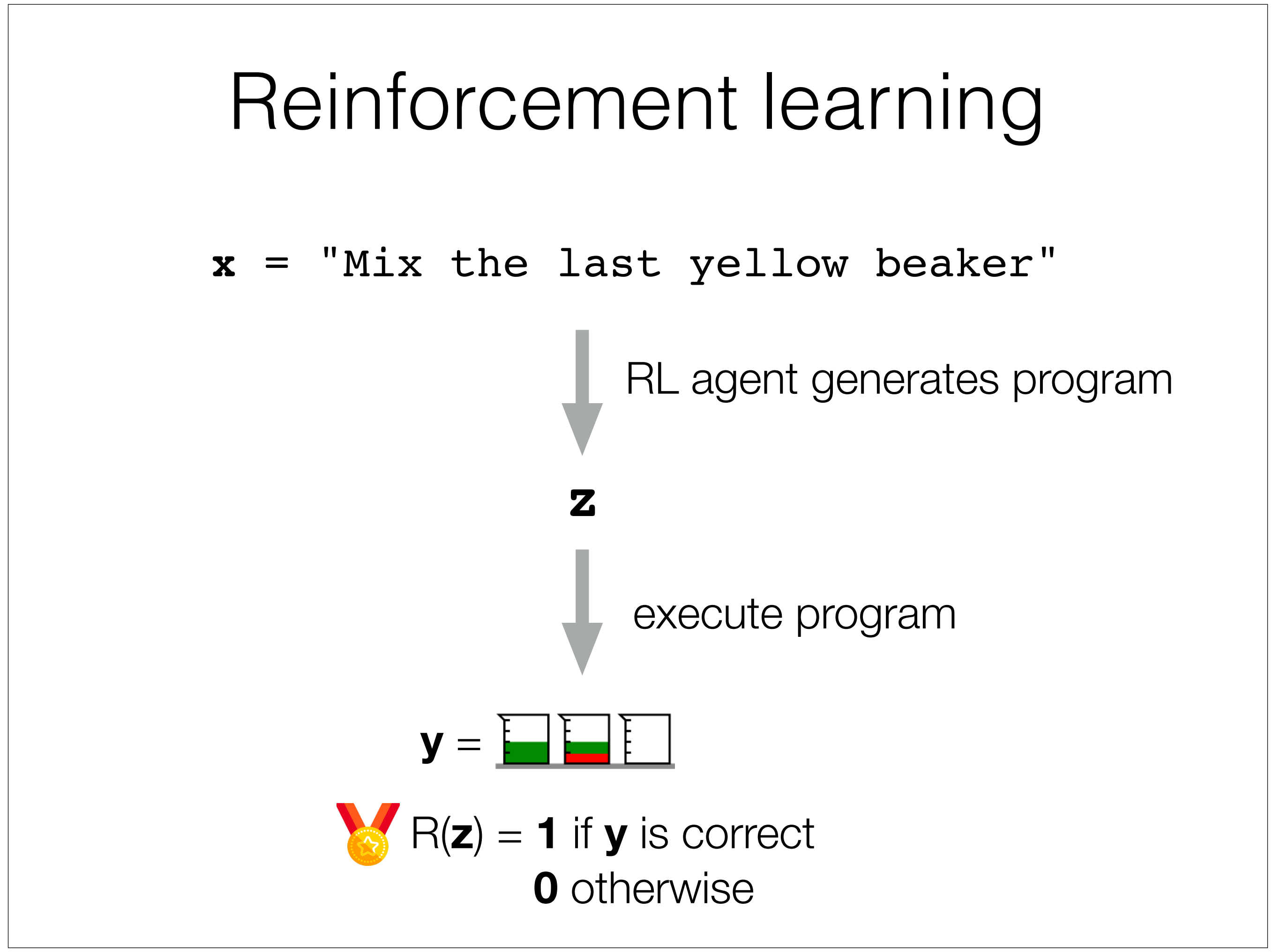
execute program

y = 

Once the agent generates a program, we then execute the program and give it reward 1 if the output is right, 0 otherwise.

The agent itself has a stochastic policy, meaning that there is some randomness in the actions it takes.

We then train the agent to maximize its expected reward.



Once the agent generates a program, we then execute the program and give it reward 1 if the output is right, 0 otherwise.

The agent itself has a stochastic policy, meaning that there is some randomness in the actions it takes.

We then train the agent to maximize its expected reward.

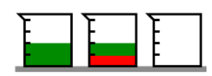
Reinforcement learning

```
x = "Mix the last yellow beaker"
```

RL agent generates program
p(**z** | **x**), **stochastic policy**

z

execute program

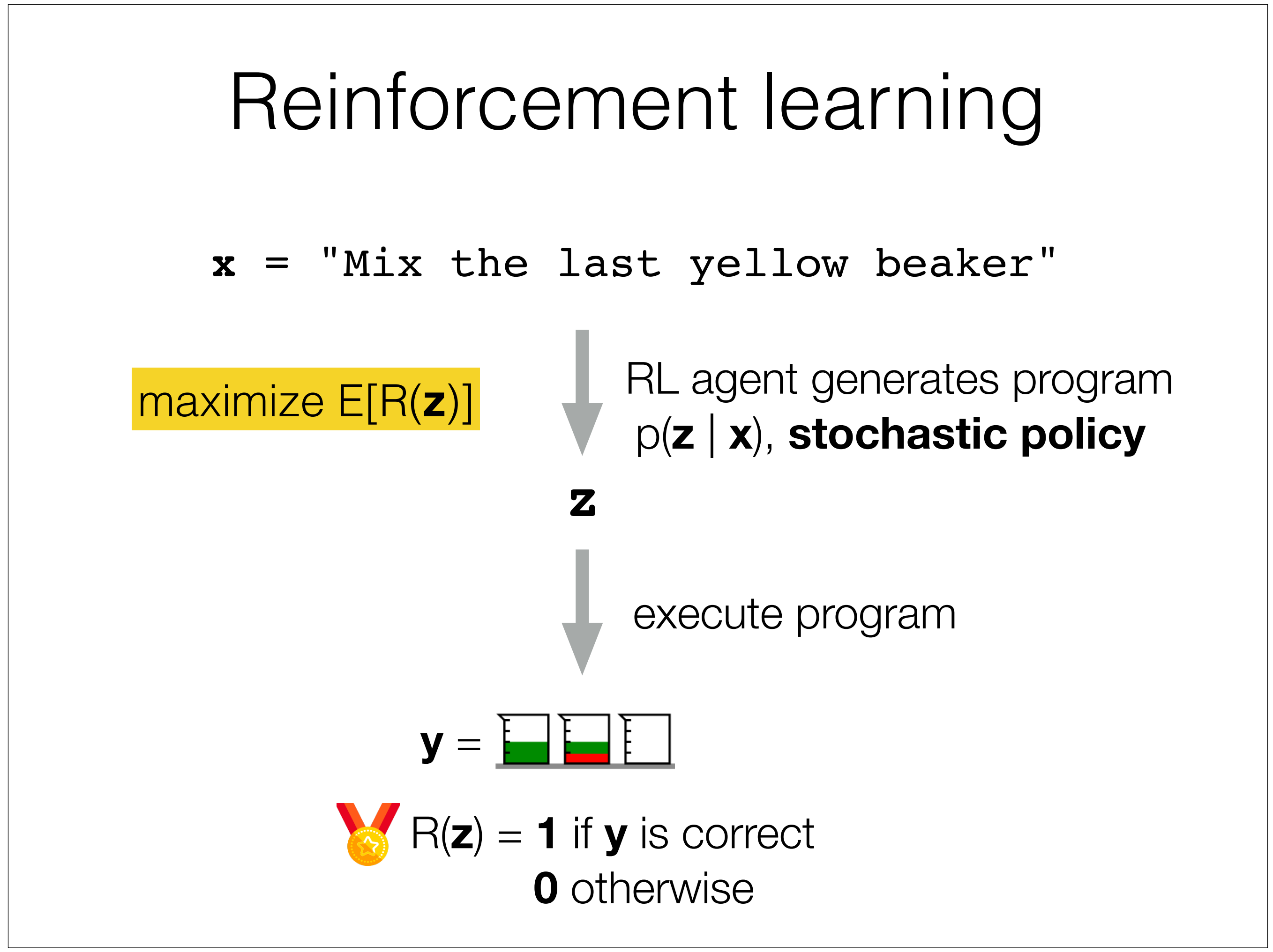
y = 

R(**z**) = **1** if **y** is correct
0 otherwise

Once the agent generates a program, we then execute the program and give it reward 1 if the output is right, 0 otherwise.

The agent itself has a stochastic policy, meaning that there is some randomness in the actions it takes.

We then train the agent to maximize its expected reward.

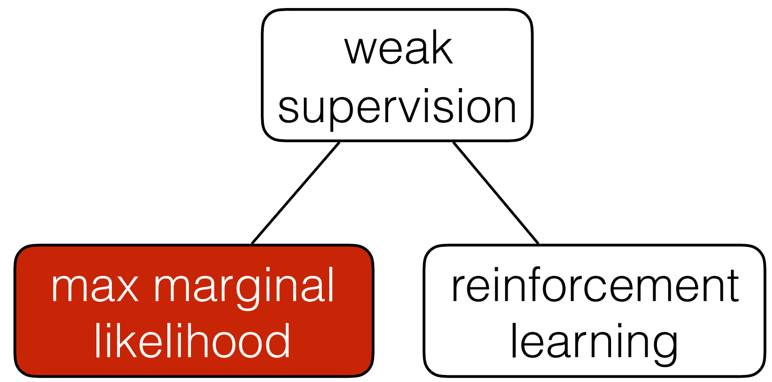


Once the agent generates a program, we then execute the program and give it reward 1 if the output is right, 0 otherwise.

The agent itself has a stochastic policy, meaning that there is some randomness in the actions it takes.

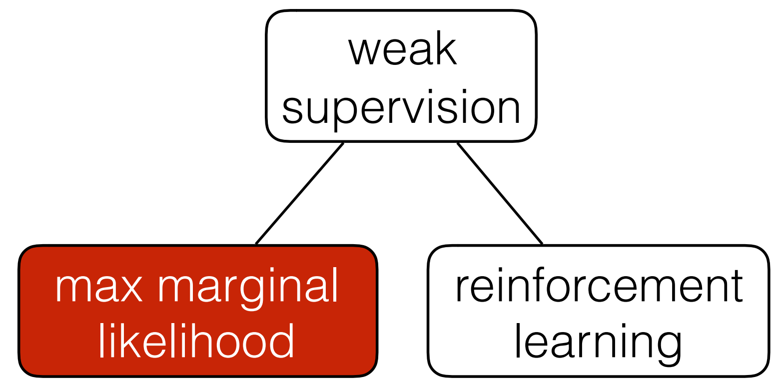
We then train the agent to maximize its expected reward.

LEARNING FRAMEWORKS



We can contrast this RL perspective with a different angle, one that has actually been more dominant in the semantic parsing literature.

LEARNING FRAMEWORKS



Clarke+ 2010, Artzi+ 2013
Krishnamurthy+ 2012
Liang+ 2017

We can contrast this RL perspective with a different angle, one that has actually been more dominant in the semantic parsing literature.

maximum marginal likelihood

$\mathbf{x}$ = "Mix the last yellow beaker"

In the marginal likelihood view, we put on our statistics hat and imagine a generative model of our data.
First, there is some conditional distribution over programs, given the utterance
Then, there is some conditional distribution over outcomes, given the program.

We then want to maximize the overall likelihood of this generative model
And it's marginal likelihood, because we have to marginalize out over the latent program, z

maximum marginal likelihood

x = "Mix the last yellow beaker"

p(**z** | **x**), generate program

z

In the marginal likelihood view, we put on our statistics hat and imagine a generative model of our data.
First, there is some conditional distribution over programs, given the utterance
Then, there is some conditional distribution over outcomes, given the program.

We then want to maximize the overall likelihood of this generative model
And it's marginal likelihood, because we have to marginalize out over the latent program, z

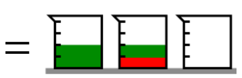
maximum marginal likelihood

x = "Mix the last yellow beaker"

p(**z** | **x**), generate program

z

p(**y** | **z**), generate outcome

y = 

In the marginal likelihood view, we put on our statistics hat and imagine a generative model of our data.
First, there is some conditional distribution over programs, given the utterance
Then, there is some conditional distribution over outcomes, given the program.

We then want to maximize the overall likelihood of this generative model
And it's marginal likelihood, because we have to marginalize out over the latent program, z

maximum marginal likelihood

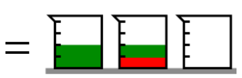
x = "Mix the last yellow beaker"

maximize p(**y** | **x**)

p(**z** | **x**), generate program

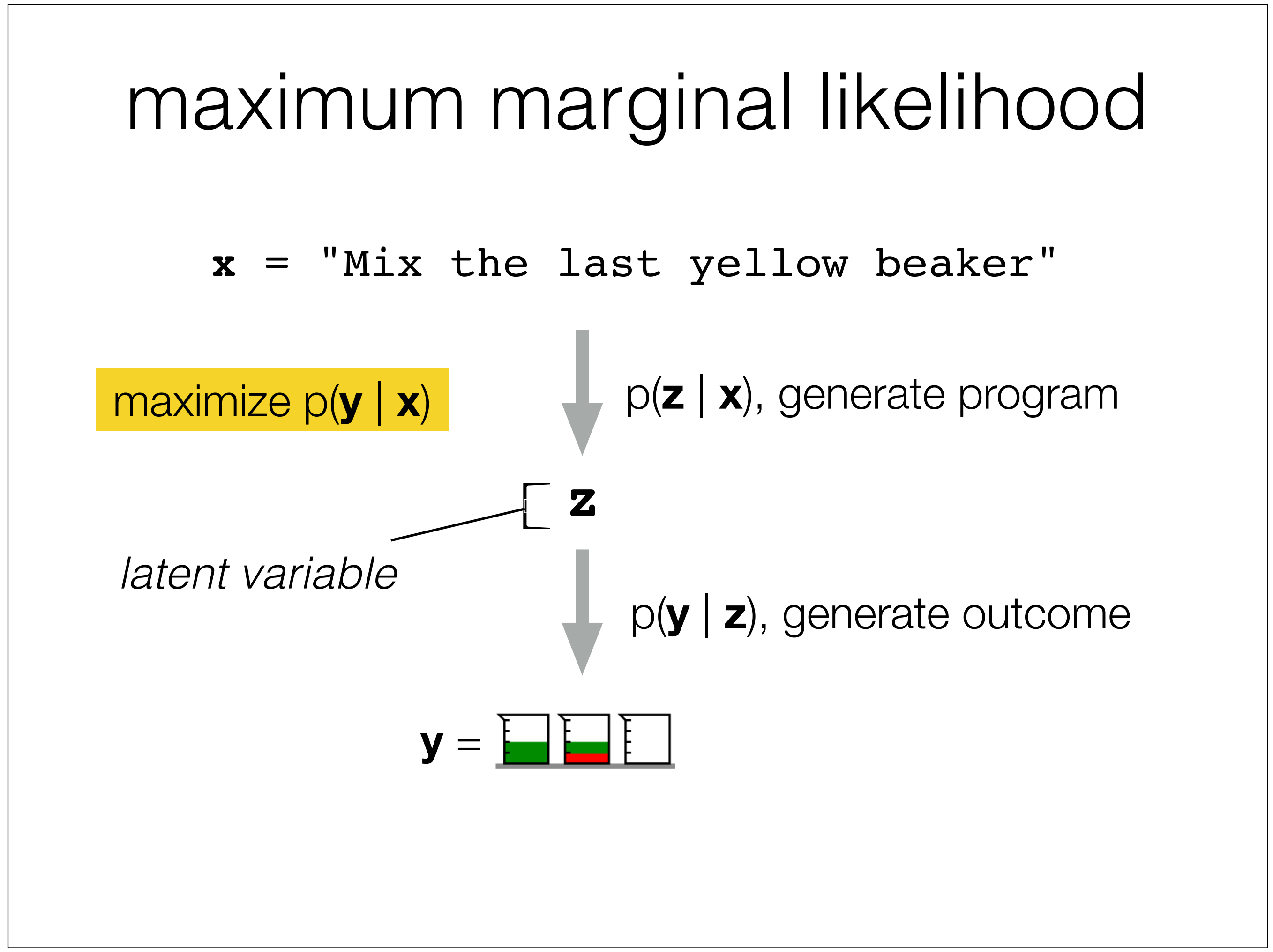
z

p(**y** | **z**), generate outcome

y = 

In the marginal likelihood view, we put on our statistics hat and imagine a generative model of our data.
First, there is some conditional distribution over programs, given the utterance
Then, there is some conditional distribution over outcomes, given the program.

We then want to maximize the overall likelihood of this generative model
And it's marginal likelihood, because we have to marginalize out over the latent program, z



In the marginal likelihood view, we put on our statistics hat and imagine a generative model of our data.
First, there is some conditional distribution over programs, given the utterance
Then, there is some conditional distribution over outcomes, given the program.

We then want to maximize the overall likelihood of this generative model
And it's marginal likelihood, because we have to marginalize out over the latent program, z

Comparing RL and MML

Those two descriptions actually sounded pretty similar

Comparing RL and MML

RL

$$E_{\mathbf{z}}\left[R\left(\mathbf{z}\right)\right] = \sum_{\mathbf{z}\in\mathcal{Z}} R\left(\mathbf{z}\right) p_{\theta}\left(\mathbf{z} \mid x\right)$$

MML

$$p\left(y \mid x\right) = \sum_{\mathbf{z}\in\mathcal{Z}} p\left(y \mid \mathbf{z}\right) p_{\theta}\left(\mathbf{z} \mid x\right)$$

y = **`goal_state`** z = **`program`** x = **`utterance`**

And in fact, if you write out both objectives, we see that they're almost the same
The only difference is right here.
But now let's think about p(y | z). It's the distribution over outcomes, given the program.
Since programs execute deterministically, this term is always 0 or 1, and is in fact identical to R(z).

Comparing RL and MML

RL

$$E_{\mathbf{z}}\left[R\left(\mathbf{z}\right)\right] = \sum_{\mathbf{z}\in\mathcal{Z}} R\left(\mathbf{z}\right) p_\theta\left(\mathbf{z} \mid x\right)$$

MML

$$p\left(y \mid x\right) = \sum_{\mathbf{z}\in\mathcal{Z}} p\left(y \mid \mathbf{z}\right) p_\theta\left(\mathbf{z} \mid x\right)$$

y = **`goal_state`** z = **`program`** x = **`utterance`**

And in fact, if you write out both objectives, we see that they're almost the same
The only difference is right here.
But now let's think about p(y | z). It's the distribution over outcomes, given the program.
Since programs execute deterministically, this term is always 0 or 1, and is in fact identical to R(z).

Comparing RL and MML

RL

$$E_{\mathbf{z}}\left[R\left(\mathbf{z}\right)\right] = \sum_{\mathbf{z} \in \mathcal{Z}} \underbrace{R\left(\mathbf{z}\right)}_{\text{same}} p_\theta\left(\mathbf{z} \mid x\right)$$

MML

$$p\left(y \mid x\right) = \sum_{\mathbf{z} \in \mathcal{Z}} \underbrace{p\left(y \mid \mathbf{z}\right)}_{\text{same}} p_\theta\left(\mathbf{z} \mid x\right)$$

y = **`goal_state`** z = **`program`** x = **`utterance`**

And in fact, if you write out both objectives, we see that they're almost the same
The only difference is right here.
But now let's think about p(y | z). It's the distribution over outcomes, given the program.
Since programs execute deterministically, this term is always 0 or 1, and is in fact identical to R(z).

Comparing RL and MML

RL $E_{\mathbf{z}}\left[R\left(\mathbf{z}\right)\right]$

MML $p\left(y \mid x\right)$

y = **`goal_state`** $\mathbf{z}$ = **`program`** x = **`utterance`**

So you might now think that RL and MML are the same.

But they are only the same when thinking about a single training example. Let's look at their objectives for multiple examples.

In RL, we maximize the average reward over examples.
In MML, we maximize the total log likelihood.

It turns out that the MML objective has an extra log. In a few slides, we'll see why that's important.

Comparing RL and MML

	single example	multiple examples
RL	$E_{\mathbf{z}}\left[R\left(\mathbf{z}\right)\right]$	
MML	$p\left(y \mid x\right)$	

y = **`goal_state`** $\quad$ z = **`program`** $\quad$ x = **`utterance`**

So you might now think that RL and MML are the same.

But they are only the same when thinking about a single training example. Let's look at their objectives for multiple examples.

In RL, we maximize the average reward over examples.
In MML, we maximize the total log likelihood.

It turns out that the MML objective has an extra log. In a few slides, we'll see why that's important.

Comparing RL and MML

	single example	multiple examples
RL	$E_{\mathbf{z}}\left[R\left(\mathbf{z}\right)\right]$	$\frac{1}{n}\sum_{i=1}^{n} E_{\mathbf{z}_i}\left[R\left(\mathbf{z}_i\right)\right]$
MML	$p\left(y \mid x\right)$	

y = **`goal_state`** z = **`program`** x = **`utterance`**

So you might now think that RL and MML are the same.

But they are only the same when thinking about a single training example. Let's look at their objectives for multiple examples.

In RL, we maximize the average reward over examples.
In MML, we maximize the total log likelihood.

It turns out that the MML objective has an extra log. In a few slides, we'll see why that's important.

Comparing RL and MML

	single example	multiple examples
RL	$E_{\mathbf{z}}\left[R\left(\mathbf{z}\right)\right]$	$\frac{1}{n}\sum_{i=1}^{n} E_{\mathbf{z}_i}\left[R\left(\mathbf{z}_i\right)\right]$
MML	$p\left(y \mid x\right)$	$\frac{1}{n}\sum_{i=1}^{n} \log p\left(y_i \mid x_i\right)$

y = **`goal_state`** z = **`program`** x = **`utterance`**

So you might now think that RL and MML are the same.

But they are only the same when thinking about a single training example. Let's look at their objectives for multiple examples.

In RL, we maximize the average reward over examples.
In MML, we maximize the total log likelihood.

It turns out that the MML objective has an extra log. In a few slides, we'll see why that's important.

Comparing RL and MML

	single example	**multiple examples**
RL	$E_{\mathbf{z}}\left[R\left(\mathbf{z}\right)\right]$	$\frac{1}{n}\sum_{i=1}^{n} E_{\mathbf{z}_i}\left[R\left(\mathbf{z}_i\right)\right]$
MML	$p\left(y \mid x\right)$	$\frac{1}{n}\sum_{i=1}^{n} \log p\left(y_i \mid x_i\right)$

same

y = **`goal_state`** z = **`program`** x = **`utterance`**

So you might now think that RL and MML are the same.

But they are only the same when thinking about a single training example. Let's look at their objectives for multiple examples.

In RL, we maximize the average reward over examples.
In MML, we maximize the total log likelihood.

It turns out that the MML objective has an extra log. In a few slides, we'll see why that's important.

Comparing RL and MML

	single example	**multiple examples**
RL	$E_{\mathbf{z}}\left[R\left(\mathbf{z}\right)\right]$	$\frac{1}{n}\sum_{i=1}^{n} E_{\mathbf{z}_i}\left[R\left(\mathbf{z}_i\right)\right]$
MML	$p\left(y \mid x\right)$	$\frac{1}{n}\sum_{i=1}^{n} \overbrace{\log}^{\textit{extra log!}}\, \overbrace{p\left(y_i \mid x_i\right)}^{\text{same}}$

y = **`goal_state`** z = **`program`** x = **`utterance`**

So you might now think that RL and MML are the same.

But they are only the same when thinking about a single training example. Let's look at their objectives for multiple examples.

In RL, we maximize the average reward over examples.
In MML, we maximize the total log likelihood.

It turns out that the MML objective has an extra log. In a few slides, we'll see why that's important.

Comparing RL and MML

	single example	multiple examples
RL	$E_{\mathbf{z}}\left[R\left(\mathbf{z}\right)\right]$	$\frac{1}{n}\sum_{i=1}^{n} \underbrace{E_{\mathbf{z}_i}\left[R\left(\mathbf{z}_i\right)\right]}_{\text{same}}$
MML	$p\left(y \mid x\right)$	$\frac{1}{n}\sum_{i=1}^{n} \overbrace{\log}^{\text{extra log!}} \overbrace{p\left(y_i \mid x_i\right)}^{\text{same}}$

Levine+ 2013
Ziebart 2010

y = **`goal_state`** z = **`program`** x = **`utterance`**

So you might now think that RL and MML are the same.

But they are only the same when thinking about a single training example. Let's look at their objectives for multiple examples.

In RL, we maximize the average reward over examples.
In MML, we maximize the total log likelihood.

It turns out that the MML objective has an extra log. In a few slides, we'll see why that's important.

Why do we get spurious programs in RL and MML?

Gradients for RL and MML

This is the last math slide you will see in this talk, but it contains the main idea, so I'll try to break this equation down.

First of all, the gradient involves a sum over all programs. Since there are millions of possible programs, this sum is approximated in practice.

Each term is weighted by the reward. Since most programs get zero reward, a lot of terms in this sum disappear.

For the remaining programs which do get reward, we take a gradient step to increase their log probability.
Finally, we weight the gradient by how much we already like the program.

The MML gradient is actually just the RL gradient, but rescaled by the expected reward, so it's almost the same.

We will argue that this term is really the main culprit responsible for spurious programs.

Gradients for RL and MML

$$g^{\mathrm{RL}} = \sum_{\mathbf{z}} p_\theta(\mathbf{z} \mid x)\, R(\mathbf{z})\, \nabla_\theta \log p_\theta(\mathbf{z} \mid x)$$

This is the last math slide you will see in this talk, but it contains the main idea, so I'll try to break this equation down.

First of all, the gradient involves a sum over all programs. Since there are millions of possible programs, this sum is approximated in practice.

Each term is weighted by the reward. Since most programs get zero reward, a lot of terms in this sum disappear.

For the remaining programs which do get reward, we take a gradient step to increase their log probability.
Finally, we weight the gradient by how much we already like the program.

The MML gradient is actually just the RL gradient, but rescaled by the expected reward, so it's almost the same.

We will argue that this term is really the main culprit responsible for spurious programs.

Gradients for RL and MML

sum over all programs

$$g^{\mathrm{RL}} = \overbrace{\sum_{\mathbf{z}}}^{} p_\theta\left(\mathbf{z} \mid x\right) R\left(\mathbf{z}\right) \nabla_\theta \log p_\theta\left(\mathbf{z} \mid x\right)$$

This is the last math slide you will see in this talk, but it contains the main idea, so I'll try to break this equation down.

First of all, the gradient involves a sum over all programs. Since there are millions of possible programs, this sum is approximated in practice.

Each term is weighted by the reward. Since most programs get zero reward, a lot of terms in this sum disappear.

For the remaining programs which do get reward, we take a gradient step to increase their log probability.
Finally, we weight the gradient by how much we already like the program.

The MML gradient is actually just the RL gradient, but rescaled by the expected reward, so it's almost the same.

We will argue that this term is really the main culprit responsible for spurious programs.

Gradients for RL and MML

$$g^{\mathrm{RL}} = \overbrace{\sum_{\mathbf{z}}}^{\textit{sum over all programs}} p_\theta\left(\mathbf{z} \mid x\right) \overbrace{R\left(\mathbf{z}\right)}^{\textit{\textbf{0} or \textbf{1} reward}} \nabla_\theta \log p_\theta\left(\mathbf{z} \mid x\right)$$

This is the last math slide you will see in this talk, but it contains the main idea, so I'll try to break this equation down.

First of all, the gradient involves a sum over all programs. Since there are millions of possible programs, this sum is approximated in practice.

Each term is weighted by the reward. Since most programs get zero reward, a lot of terms in this sum disappear.

For the remaining programs which do get reward, we take a gradient step to increase their log probability.
Finally, we weight the gradient by how much we already like the program.

The MML gradient is actually just the RL gradient, but rescaled by the expected reward, so it's almost the same.

We will argue that this term is really the main culprit responsible for spurious programs.

Gradients for RL and MML

$$g^{\mathrm{RL}} = \overbrace{\sum_{\mathbf{z}}}^{\textit{sum over all programs}} p_\theta\left(\mathbf{z} \mid x\right) \overbrace{R\left(\mathbf{z}\right)}^{\textit{\textbf{0} or \textbf{1} reward}} \underbrace{\nabla_\theta \log p_\theta\left(\mathbf{z} \mid x\right)}_{\textit{upweight the program}}$$

This is the last math slide you will see in this talk, but it contains the main idea, so I'll try to break this equation down.

First of all, the gradient involves a sum over all programs. Since there are millions of possible programs, this sum is approximated in practice.

Each term is weighted by the reward. Since most programs get zero reward, a lot of terms in this sum disappear.

For the remaining programs which do get reward, we take a gradient step to increase their log probability.
Finally, we weight the gradient by how much we already like the program.

The MML gradient is actually just the RL gradient, but rescaled by the expected reward, so it's almost the same.

We will argue that this term is really the main culprit responsible for spurious programs.

Gradients for RL and MML

$$g^{\mathrm{RL}} = \overbrace{\sum_{\mathbf{z}}}^{\textit{sum over all programs}} \underbrace{p_\theta\left(\mathbf{z} \mid x\right)}_{\textit{upweight proportional to how much you already like it}} \overbrace{R\left(\mathbf{z}\right)}^{\textit{\textbf{0} or \textbf{1} reward}} \underbrace{\nabla_\theta \log p_\theta\left(\mathbf{z} \mid x\right)}_{\textit{upweight the program}}$$

This is the last math slide you will see in this talk, but it contains the main idea, so I'll try to break this equation down.

First of all, the gradient involves a sum over all programs. Since there are millions of possible programs, this sum is approximated in practice.

Each term is weighted by the reward. Since most programs get zero reward, a lot of terms in this sum disappear.

For the remaining programs which do get reward, we take a gradient step to increase their log probability.
Finally, we weight the gradient by how much we already like the program.

The MML gradient is actually just the RL gradient, but rescaled by the expected reward, so it's almost the same.

We will argue that this term is really the main culprit responsible for spurious programs.

Gradients for RL and MML

$$g^{\mathrm{RL}} = \overbrace{\sum_{\mathbf{z}}}^{\text{sum over all programs}} \underbrace{p_\theta\left(\mathbf{z} \mid x\right)}_{\substack{\text{upweight proportional to} \\ \text{how much you already like it}}} \overbrace{R\left(\mathbf{z}\right)}^{\mathbf{0} \text{ or } \mathbf{1} \text{ reward}} \underbrace{\nabla_\theta \log p_\theta\left(\mathbf{z} \mid x\right)}_{\text{upweight the program}}$$

$$g^{\mathrm{MML}} = g^{\mathrm{RL}} / E_{\mathbf{z}}[R(\mathbf{z})]$$

This is the last math slide you will see in this talk, but it contains the main idea, so I'll try to break this equation down.

First of all, the gradient involves a sum over all programs. Since there are millions of possible programs, this sum is approximated in practice.

Each term is weighted by the reward. Since most programs get zero reward, a lot of terms in this sum disappear.

For the remaining programs which do get reward, we take a gradient step to increase their log probability.
Finally, we weight the gradient by how much we already like the program.

The MML gradient is actually just the RL gradient, but rescaled by the expected reward, so it's almost the same.

We will argue that this term is really the main culprit responsible for spurious programs.

Gradients for RL and MML

$$g^{\mathrm{RL}} = \overbrace{\sum_{\mathbf{z}}}^{\text{sum over all programs}} \underbrace{p_\theta\left(\mathbf{z} \mid x\right)}_{\substack{\text{upweight proportional to}\\ \text{how much you already like it}}} \overbrace{R\left(\mathbf{z}\right)}^{\mathbf{0} \text{ or } \mathbf{1} \text{ reward}} \underbrace{\nabla_\theta \log p_\theta\left(\mathbf{z} \mid x\right)}_{\text{upweight the program}}$$

$$g^{\mathrm{MML}} = g^{\mathrm{RL}} / E_{\mathbf{z}}[R(\mathbf{z})]$$

This is the last math slide you will see in this talk, but it contains the main idea, so I'll try to break this equation down.

First of all, the gradient involves a sum over all programs. Since there are millions of possible programs, this sum is approximated in practice.

Each term is weighted by the reward. Since most programs get zero reward, a lot of terms in this sum disappear.

For the remaining programs which do get reward, we take a gradient step to increase their log probability.
Finally, we weight the gradient by how much we already like the program.

The MML gradient is actually just the RL gradient, but rescaled by the expected reward, so it's almost the same.

We will argue that this term is really the main culprit responsible for spurious programs.

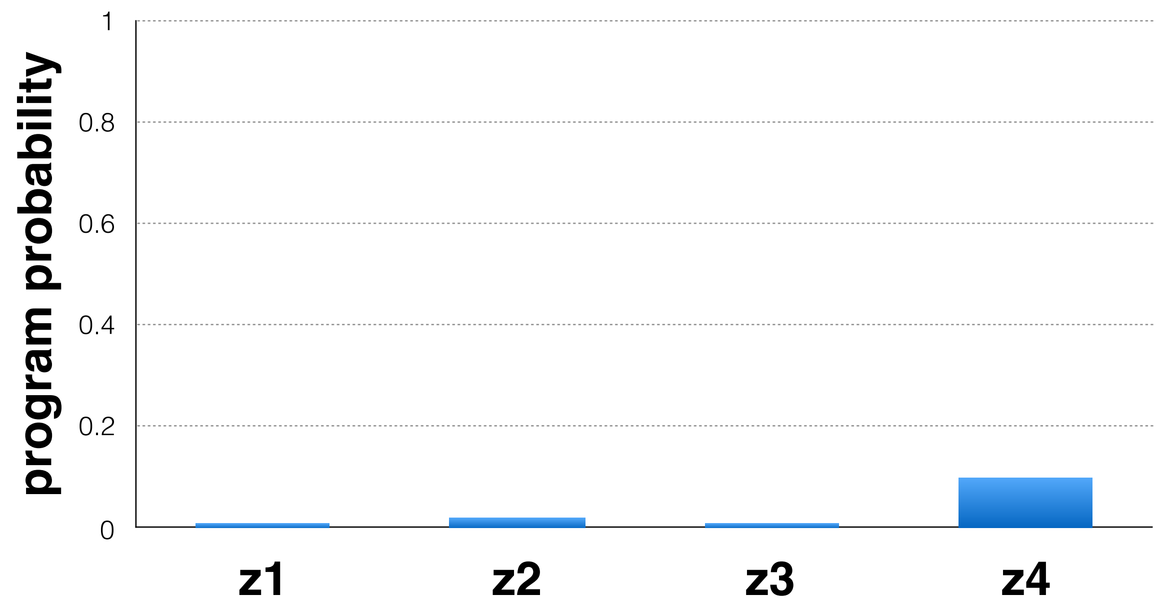
Gradients for RL and MML

$$g^{\mathrm{RL}} = \sum_{\mathbf{z}} \underbrace{p_\theta(\mathbf{z} \mid x)\, R(\mathbf{z})}_{\textit{gradient weight}} \nabla_\theta \log p_\theta(\mathbf{z} \mid x)$$

To make that argument, we'll focus on this quantity, which we call the gradient weight.

The rich get richer, the poor stay poor

$$g^{\mathrm{RL}} = \sum_{\mathbf{z}} p_\theta(\mathbf{z} \mid x)\, R(\mathbf{z}) \nabla_\theta \log p_\theta(\mathbf{z} \mid x)$$

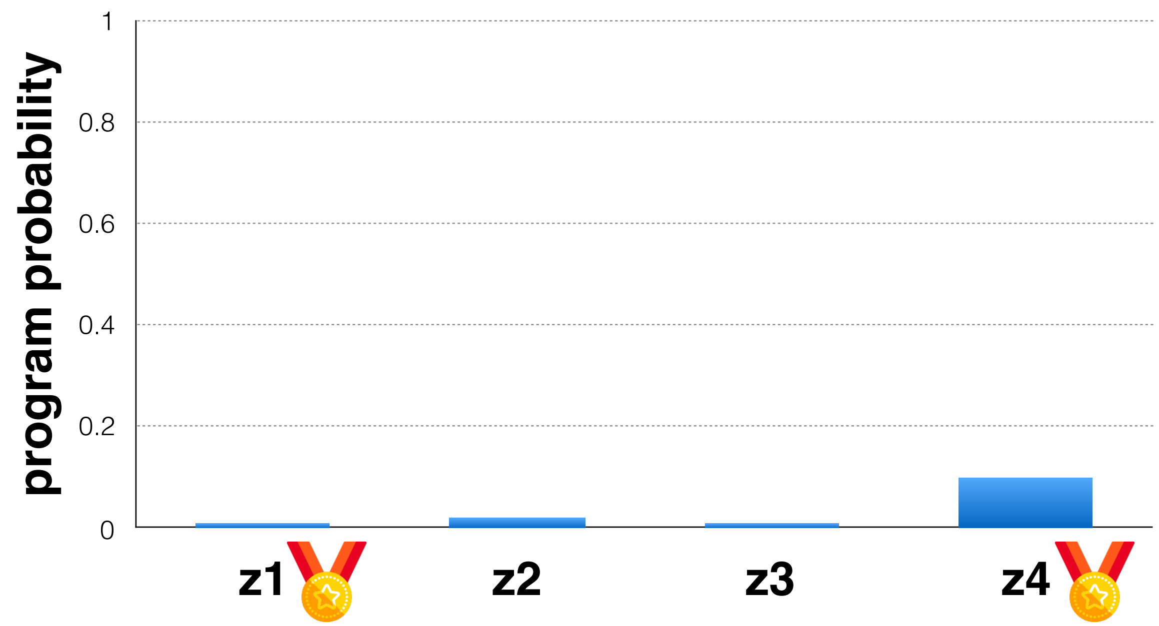


Suppose we have a model which places fairly low probability on all programs to begin with.

But we find out that two of them get reward.

The rich get richer, the poor stay poor

$$g^{\mathrm{RL}} = \sum_{\mathbf{z}} p_\theta(\mathbf{z} \mid x)\, R(\mathbf{z}) \nabla_\theta \log p_\theta(\mathbf{z} \mid x)$$

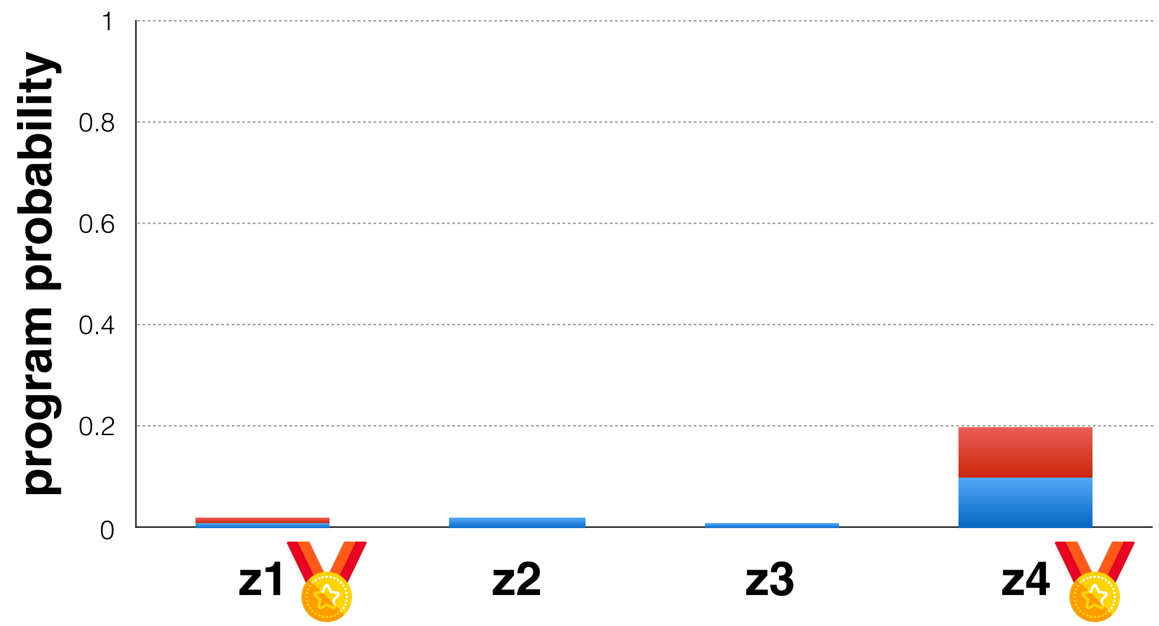


Suppose we have a model which places fairly low probability on all programs to begin with.

But we find out that two of them get reward.

The rich get richer, the poor stay poor

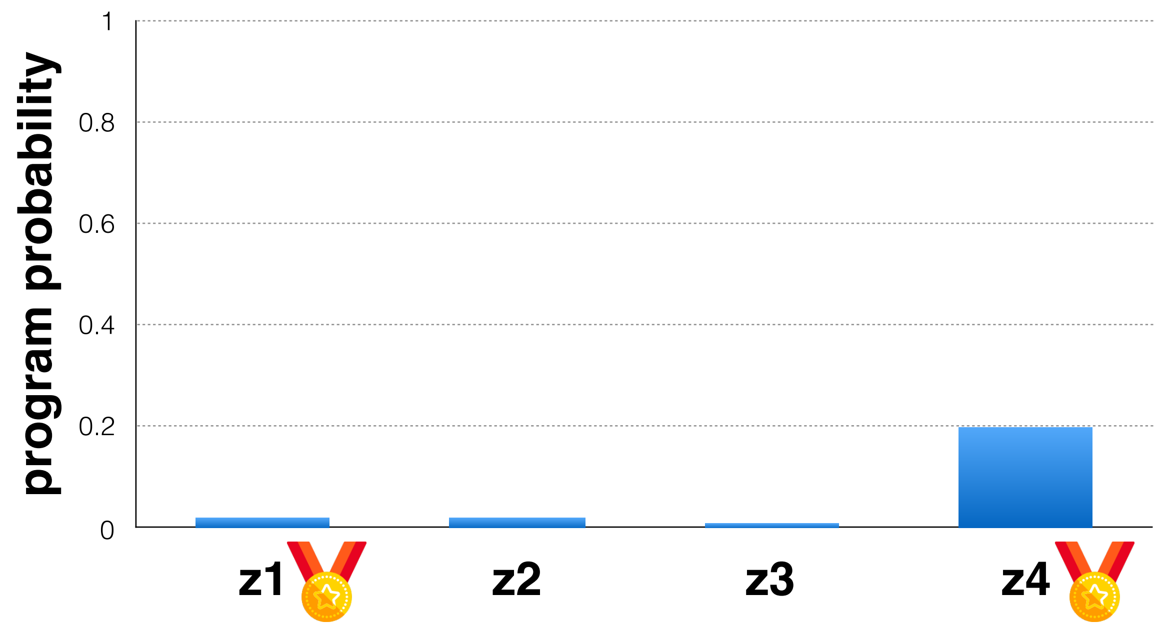
$$g^{\mathrm{RL}} = \sum_{\mathbf{z}} p_\theta(\mathbf{z} \mid x)\, R(\mathbf{z}) \nabla_\theta \log p_\theta(\mathbf{z} \mid x)$$



In our gradient step, we upweight them. The red bars show the size of the gradient weight.

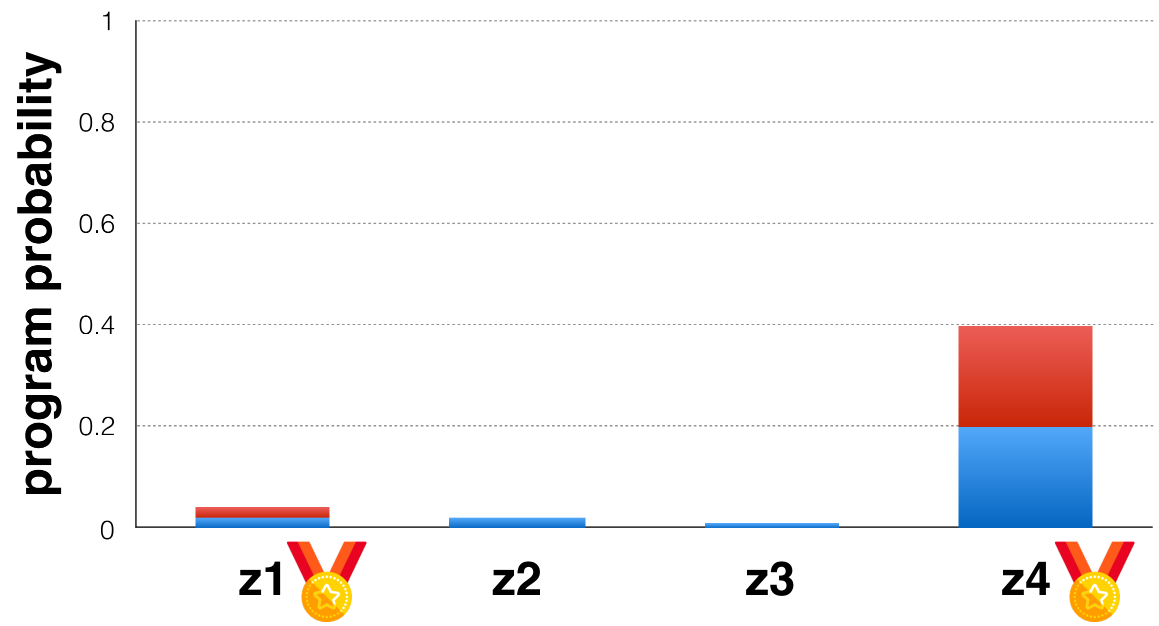
The rich get richer, the poor stay poor

$$g^{\mathrm{RL}} = \sum_{\mathbf{z}} p_\theta(\mathbf{z} \mid x)\, R(\mathbf{z}) \nabla_\theta \log p_\theta(\mathbf{z} \mid x)$$



The rich get richer, the poor stay poor

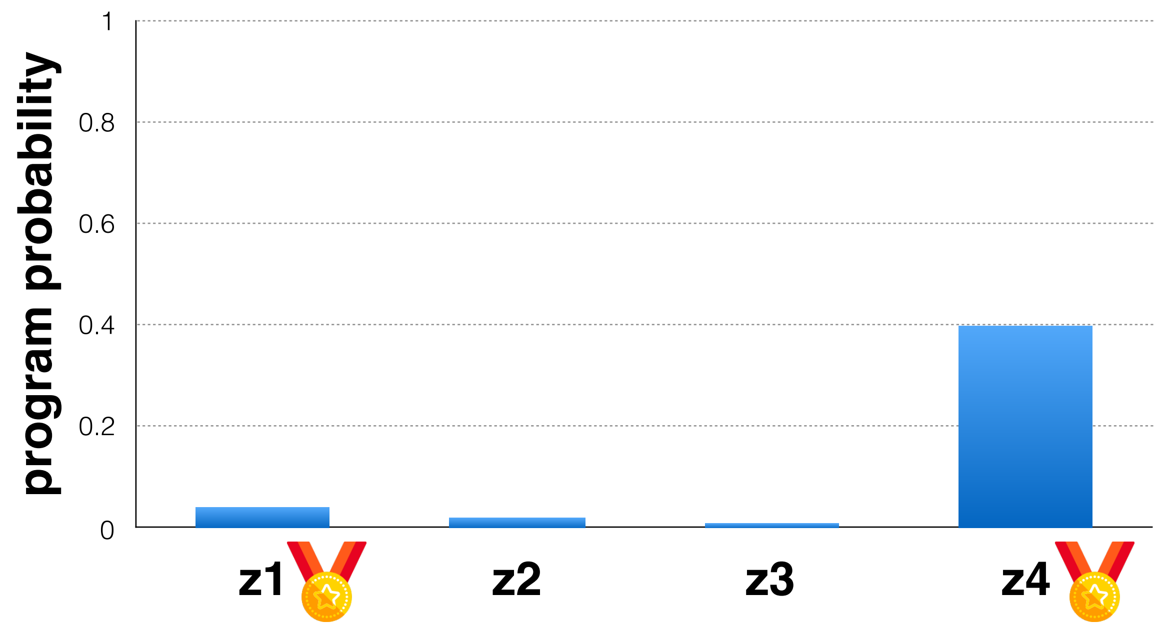
$$g^{\mathrm{RL}} = \sum_{\mathbf{z}} p_\theta(\mathbf{z} \mid x)\, R(\mathbf{z}) \nabla_\theta \log p_\theta(\mathbf{z} \mid x)$$



We then repeat the gradient step a few times.

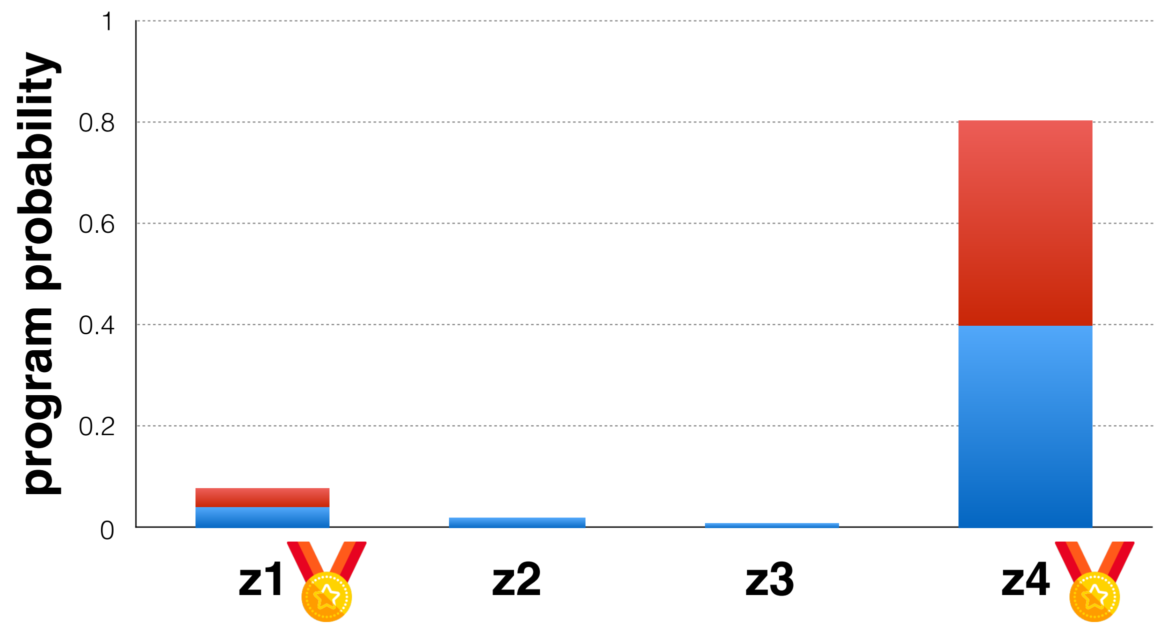
The rich get richer, the poor stay poor

$$g^{\mathrm{RL}} = \sum_{\mathbf{z}} p_\theta(\mathbf{z} \mid x)\, R(\mathbf{z}) \nabla_\theta \log p_\theta(\mathbf{z} \mid x)$$



The rich get richer, the poor stay poor

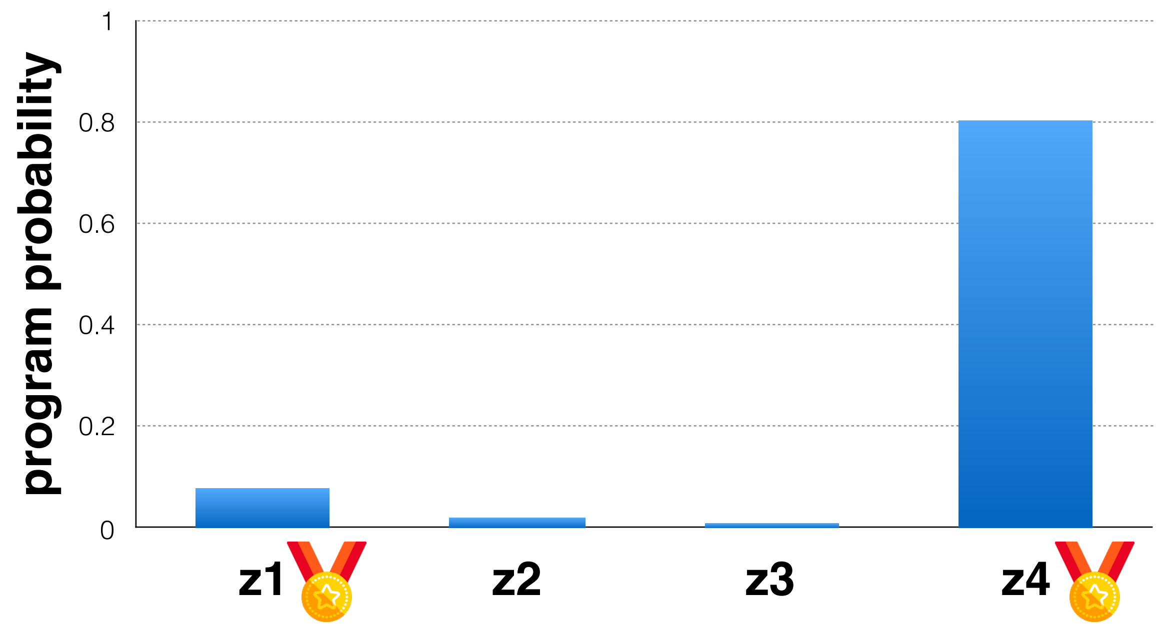
$$g^{\mathrm{RL}} = \sum_{\mathbf{z}} p_\theta(\mathbf{z} \mid x)\, R(\mathbf{z}) \nabla_\theta \log p_\theta(\mathbf{z} \mid x)$$



After a while, z4 dominates z1 by a lot, just because it had a head start.

The rich get richer, the poor stay poor

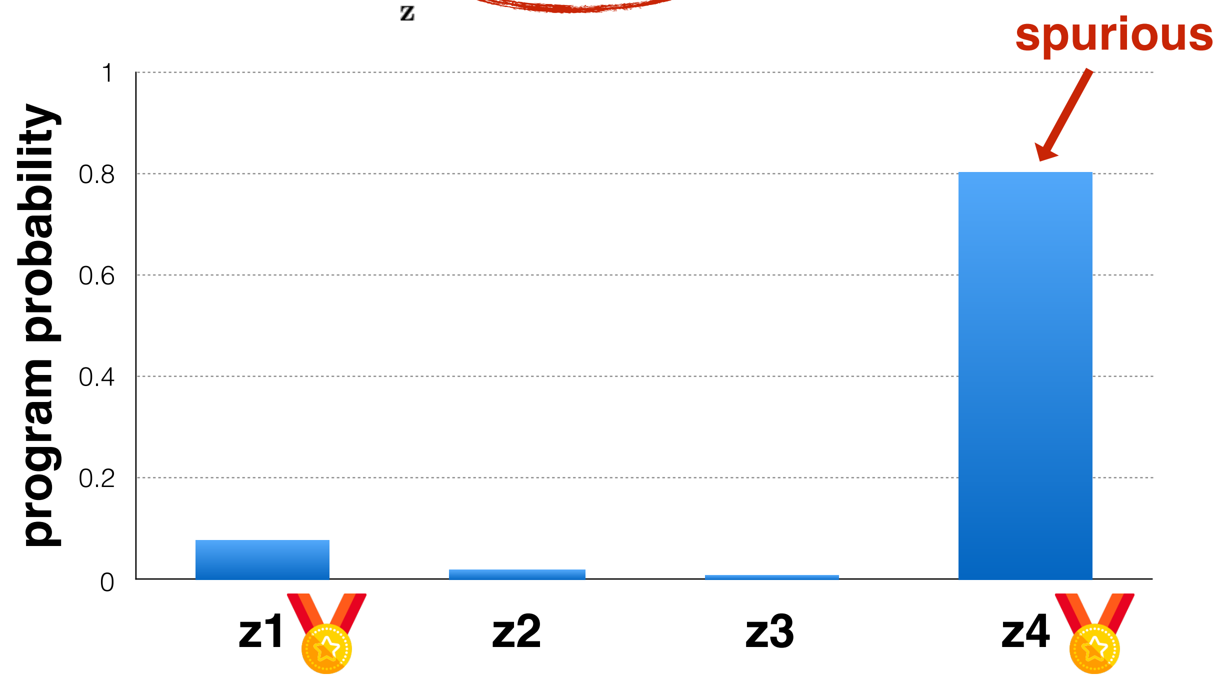
$$g^{\mathrm{RL}} = \sum_{\mathbf{z}} p_\theta(\mathbf{z} \mid x)\, R(\mathbf{z}) \nabla_\theta \log p_\theta(\mathbf{z} \mid x)$$



And if z4 were spurious, we would be in trouble

The rich get richer, the poor stay poor

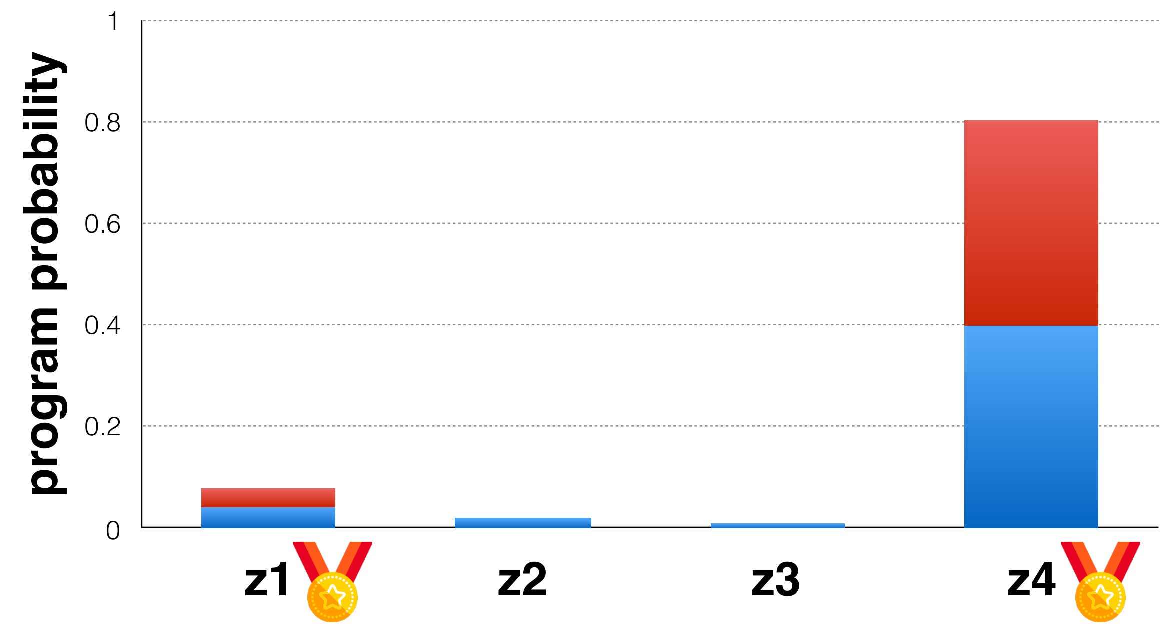
$$g^{\mathrm{RL}} = \sum_{\mathbf{z}} p_\theta(\mathbf{z} \mid x)\, R(\mathbf{z}) \nabla_\theta \log p_\theta(\mathbf{z} \mid x)$$



And if z4 were spurious, we would be in trouble

The rich get richer, the poor stay poor

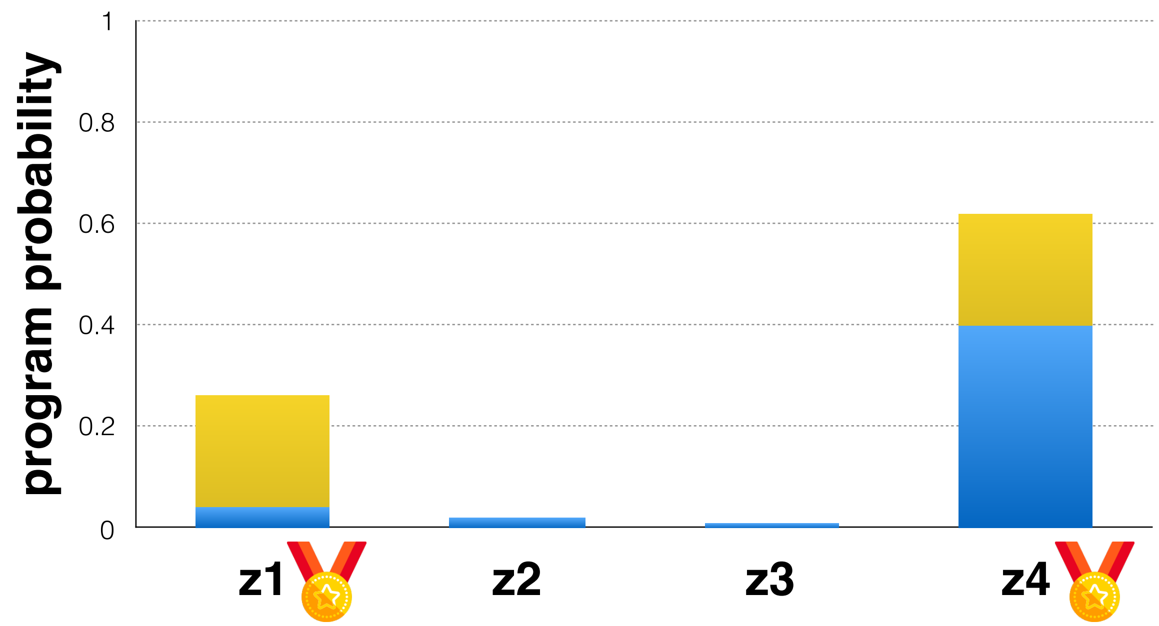
$$g^{\mathrm{RL}} = \sum_{\mathbf{z}} p_\theta\left(\mathbf{z} \mid x\right) R\left(\mathbf{z}\right) \nabla_\theta \log p_\theta\left(\mathbf{z} \mid x\right)$$



So let's rewind and think about what we could have done instead.

Meritocratic gradient weight

$$g^{\mathrm{RL}} = \sum_{\mathbf{z}} p_\theta(\mathbf{z} \mid x)\, R(\mathbf{z})\, \nabla_\theta \log p_\theta(\mathbf{z} \mid x)$$



Intuitively, we'd just like to give both programs the same boost.

SOLUTION

combining the best of RL and MML

1. meritocratic gradient weights

2. randomized beam search

And we can do exactly that, which brings us to our first solution

Meritocratic gradient weight

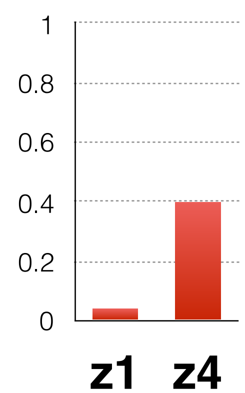
What we do is take the initial gradient weights
and renormalize them so that they sum to 1

Then we think of the gradient weight as a probability distribution, and raise the temperature of that distribution.
If you raise the temperature to infinity, you end up with equal weights.

Interestingly, the weights on the left correspond to the original RL gradient
while the renormalized weights correspond exactly to the MML gradient

We go one step further and adjust the temperature of that distribution, yielding what we call a meritocratic gradient.

Meritocratic gradient weight

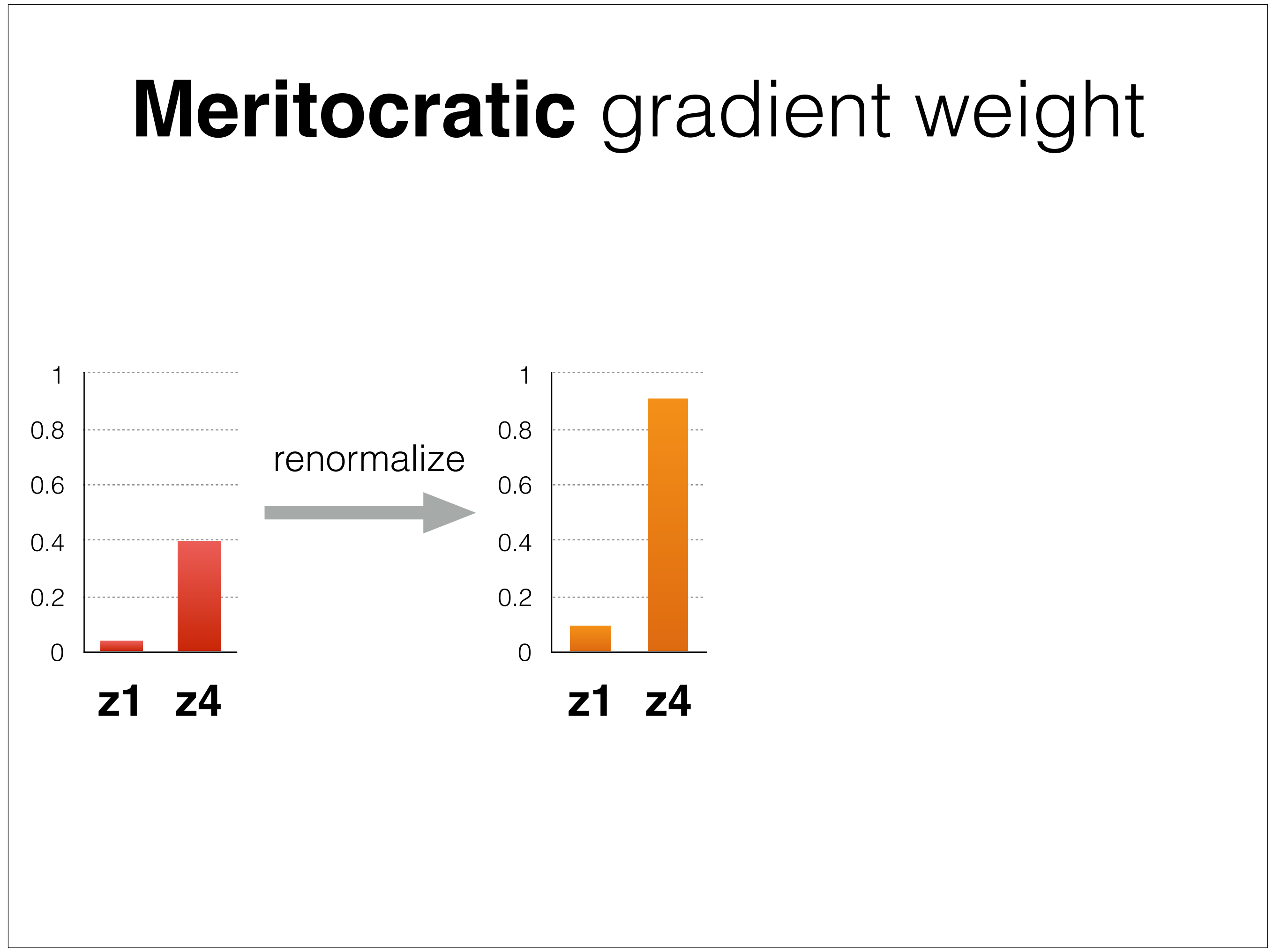


What we do is take the initial gradient weights
and renormalize them so that they sum to 1

Then we think of the gradient weight as a probability distribution, and raise the temperature of that distribution.
If you raise the temperature to infinity, you end up with equal weights.

Interestingly, the weights on the left correspond to the original RL gradient
while the renormalized weights correspond exactly to the MML gradient

We go one step further and adjust the temperature of that distribution, yielding what we call a meritocratic gradient.

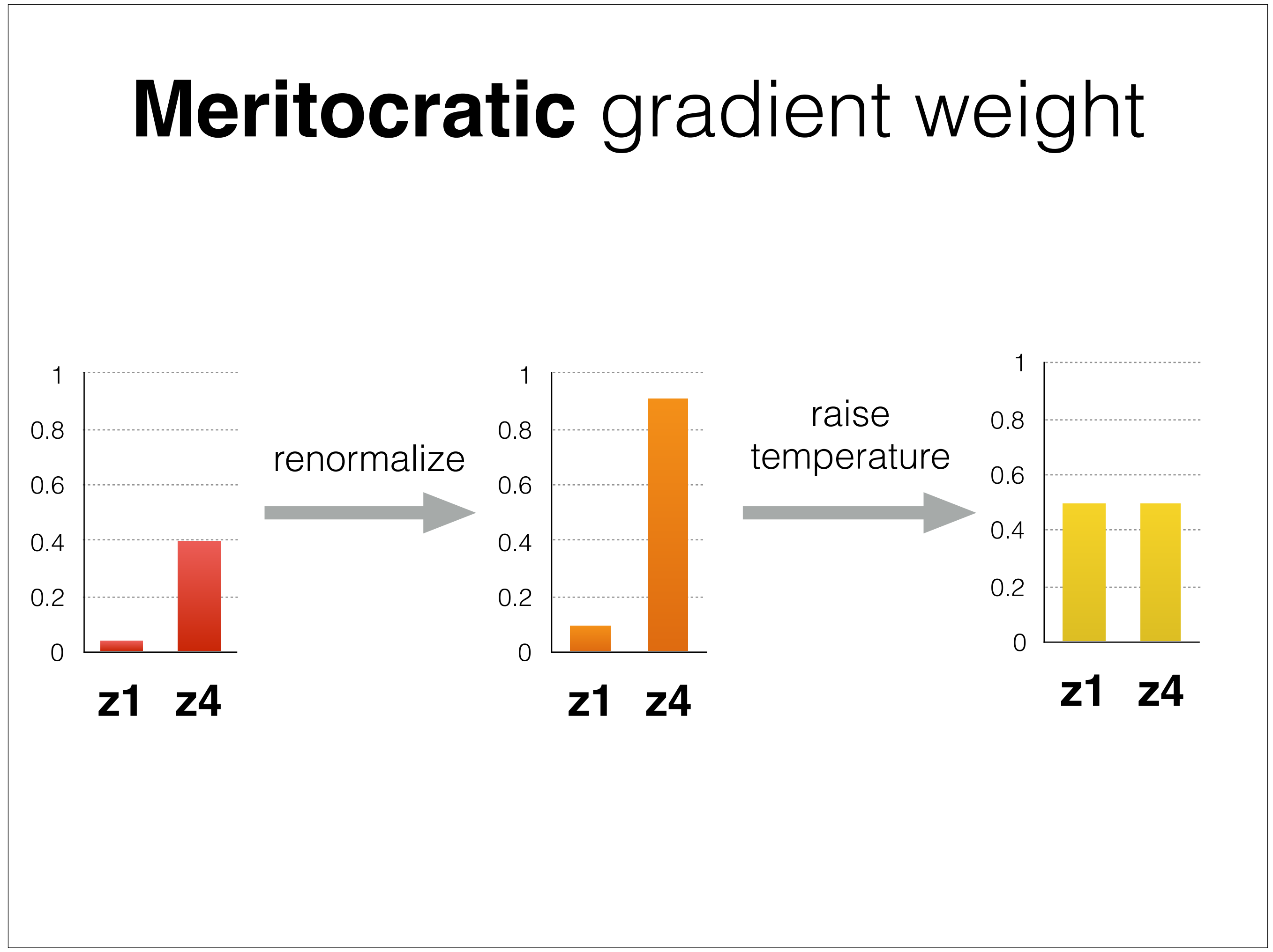


What we do is take the initial gradient weights
and renormalize them so that they sum to 1

Then we think of the gradient weight as a probability distribution, and raise the temperature of that distribution.
If you raise the temperature to infinity, you end up with equal weights.

Interestingly, the weights on the left correspond to the original RL gradient
while the renormalized weights correspond exactly to the MML gradient

We go one step further and adjust the temperature of that distribution, yielding what we call a meritocratic gradient.



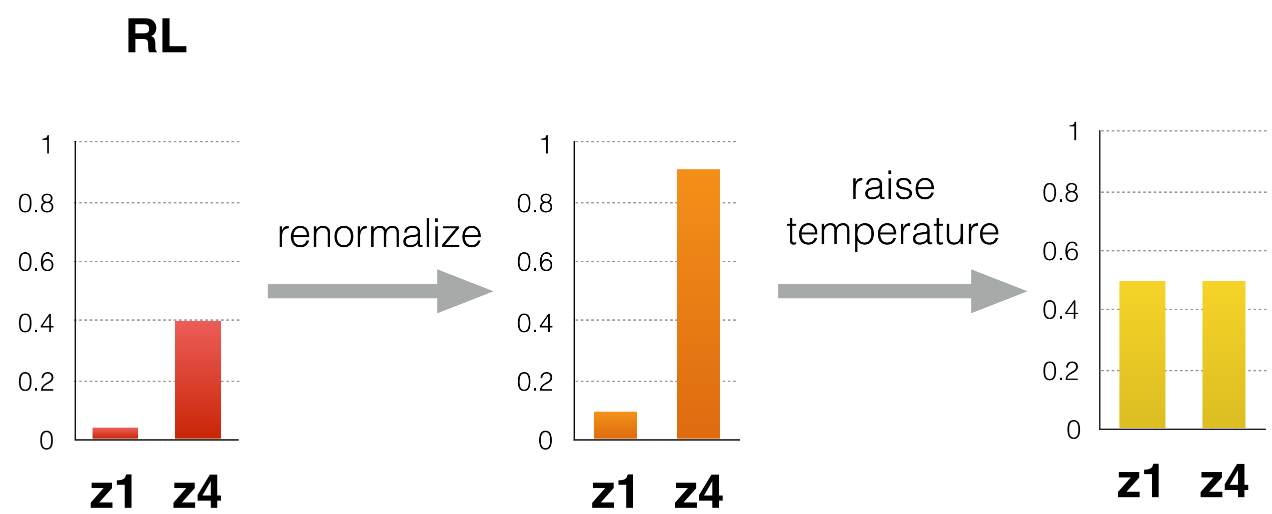
What we do is take the initial gradient weights
and renormalize them so that they sum to 1

Then we think of the gradient weight as a probability distribution, and raise the temperature of that distribution.
If you raise the temperature to infinity, you end up with equal weights.

Interestingly, the weights on the left correspond to the original RL gradient
while the renormalized weights correspond exactly to the MML gradient

We go one step further and adjust the temperature of that distribution, yielding what we call a meritocratic gradient.

Meritocratic gradient weight

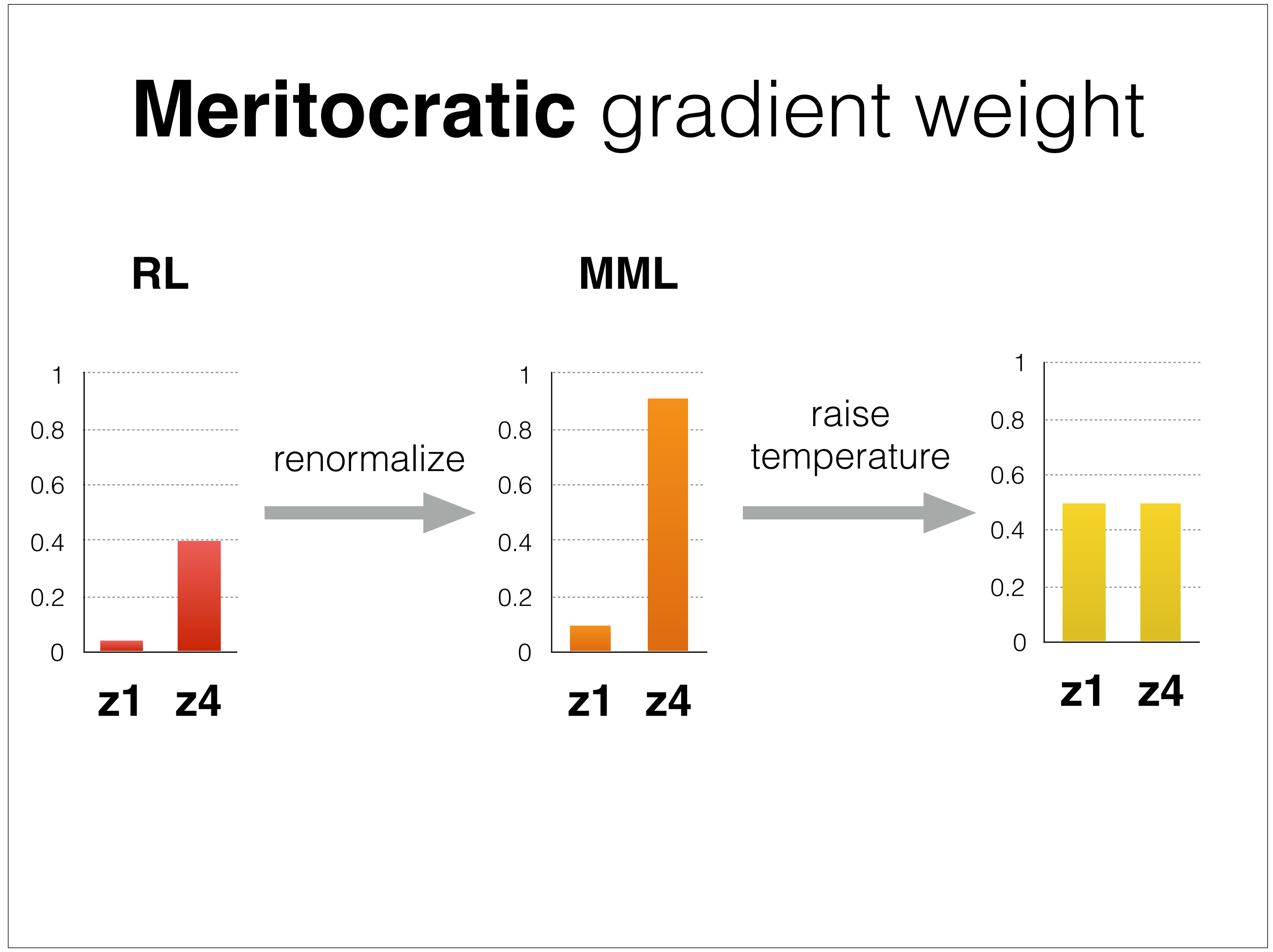


What we do is take the initial gradient weights
and renormalize them so that they sum to 1

Then we think of the gradient weight as a probability distribution, and raise the temperature of that distribution.
If you raise the temperature to infinity, you end up with equal weights.

Interestingly, the weights on the left correspond to the original RL gradient
while the renormalized weights correspond exactly to the MML gradient

We go one step further and adjust the temperature of that distribution, yielding what we call a meritocratic gradient.

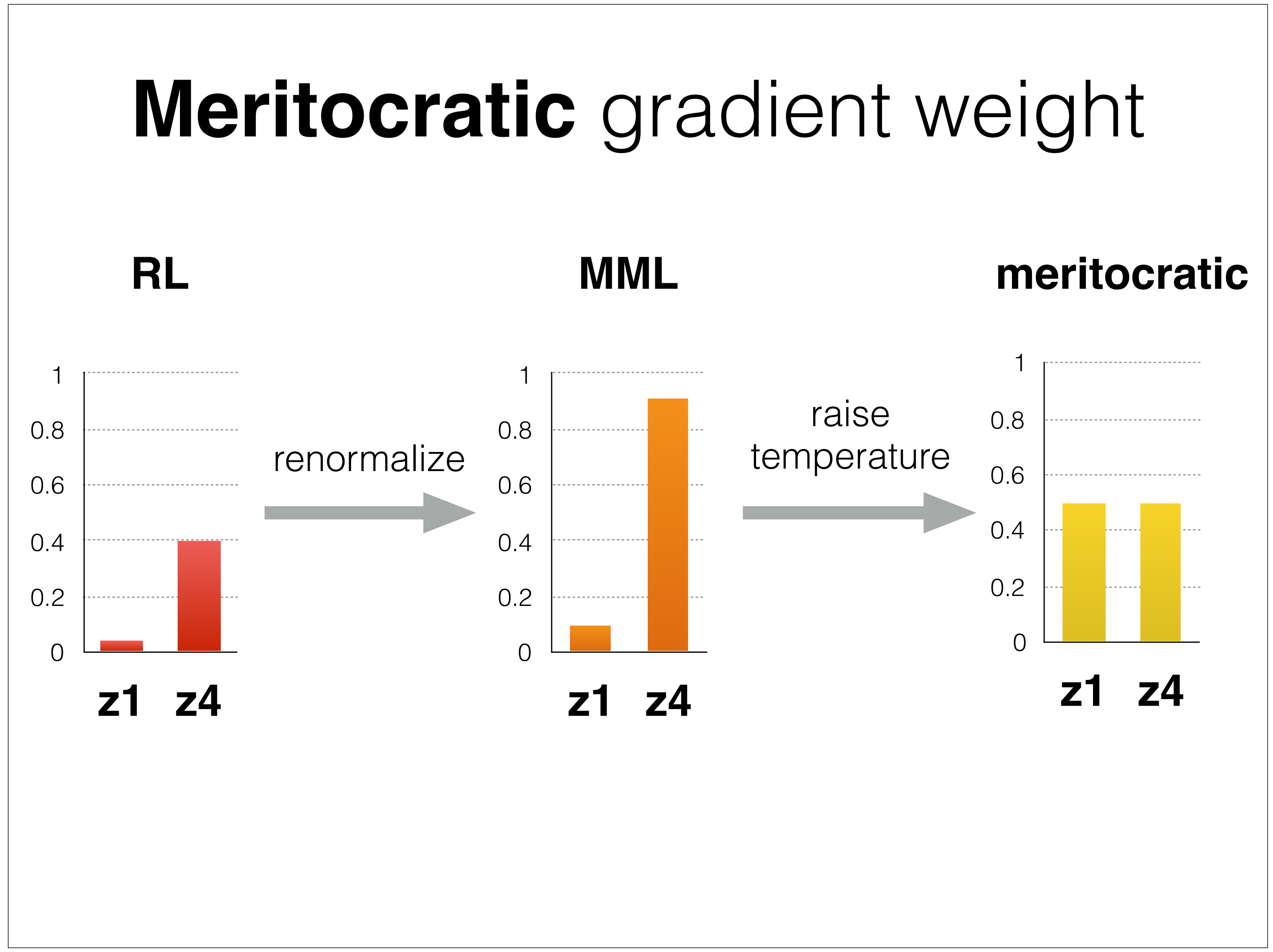


What we do is take the initial gradient weights
and renormalize them so that they sum to 1

Then we think of the gradient weight as a probability distribution, and raise the temperature of that distribution.
If you raise the temperature to infinity, you end up with equal weights.

Interestingly, the weights on the left correspond to the original RL gradient
while the renormalized weights correspond exactly to the MML gradient

We go one step further and adjust the temperature of that distribution, yielding what we call a meritocratic gradient.

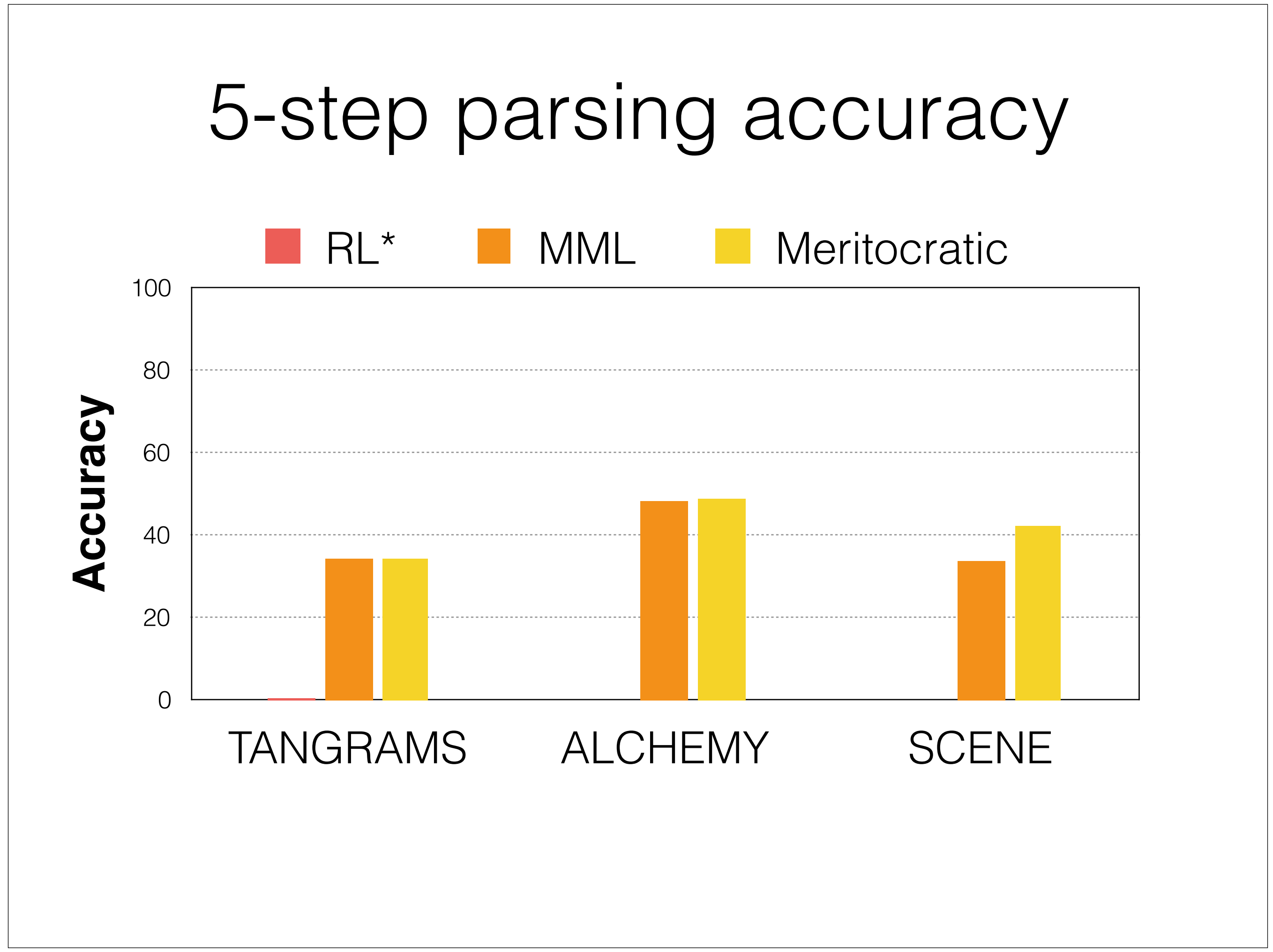


What we do is take the initial gradient weights
and renormalize them so that they sum to 1

Then we think of the gradient weight as a probability distribution, and raise the temperature of that distribution.
If you raise the temperature to infinity, you end up with equal weights.

Interestingly, the weights on the left correspond to the original RL gradient
while the renormalized weights correspond exactly to the MML gradient

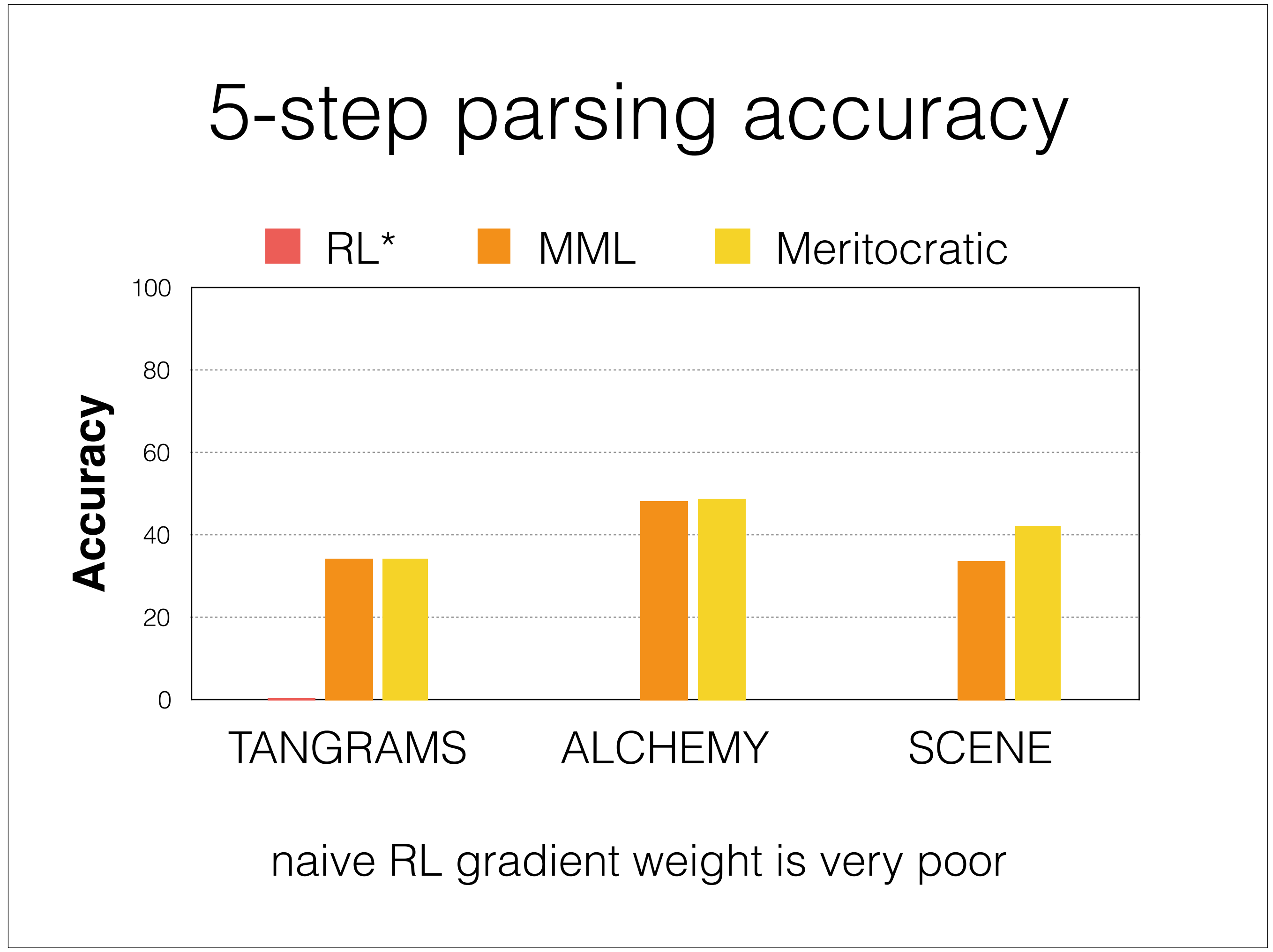
We go one step further and adjust the temperature of that distribution, yielding what we call a meritocratic gradient.



When we compare the impact of using these different gradient weights, we see some pretty interesting results.

Across different tasks, the meritocratic update is always better or as good as the MML update.

The red bars for RL are actually all near zero. We found that when you try to do the exact RL update, things really don't train that well, and a significant amount of epsilon greedy dithering was needed to actually make it work.

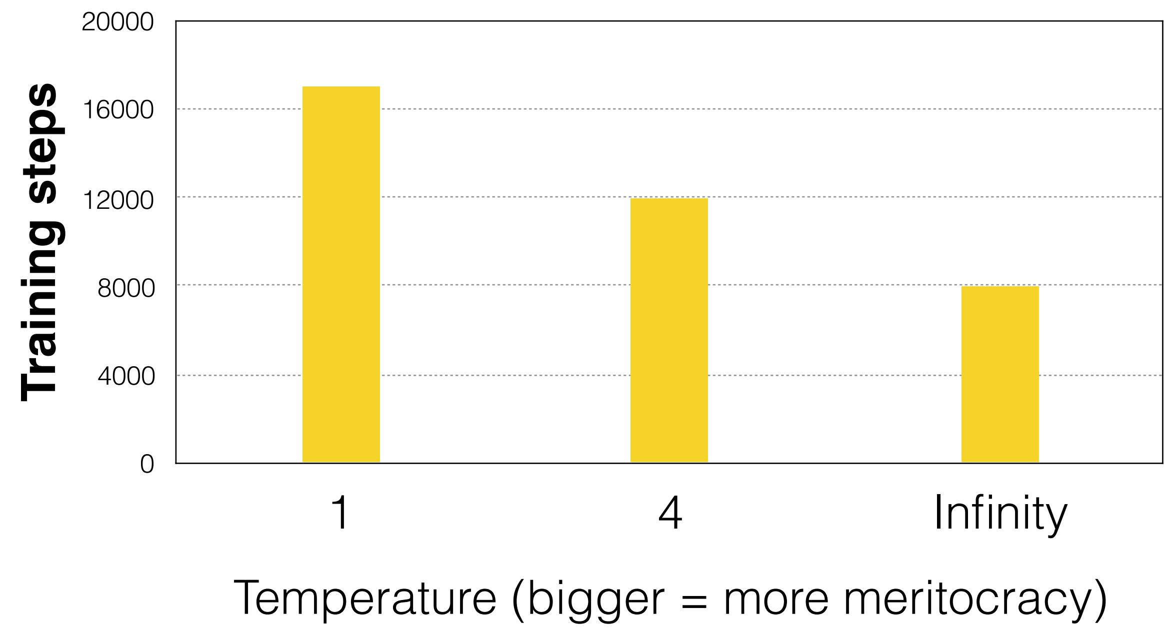


When we compare the impact of using these different gradient weights, we see some pretty interesting results.

Across different tasks, the meritocratic update is always better or as good as the MML update.

The red bars for RL are actually all near zero. We found that when you try to do the exact RL update, things really don't train that well, and a significant amount of epsilon greedy dithering was needed to actually make it work.

Training time



Another benefit we noticed was that more meritocracy leads to faster overall training speed.
Note that when temperature = 1, this is identical to MML.

$$g^{\mathrm{RL}} = \sum_{\mathbf{z}} p_\theta(\mathbf{z} \mid x)\, R(\mathbf{z})\, \nabla_\theta \log p_\theta(\mathbf{z} \mid x)$$

gradient weight

So, that closes our discussion of the gradient weight.

But one thing I didn't explain is how we handle the intractable sum over millions of programs.

$$g^{\mathrm{RL}} = \overbrace{\sum_{\mathbf{z}}}^{\textit{sum over all programs } (\textbf{\textit{millions}})} \underbrace{p_\theta\left(\mathbf{z} \mid x\right) R\left(\mathbf{z}\right)}_{\textbf{\textit{gradient weight}}} \nabla_\theta \log p_\theta\left(\mathbf{z} \mid x\right)$$

So, that closes our discussion of the gradient weight.

But one thing I didn't explain is how we handle the intractable sum over millions of programs.

SOLUTION

combining the best of RL and MML

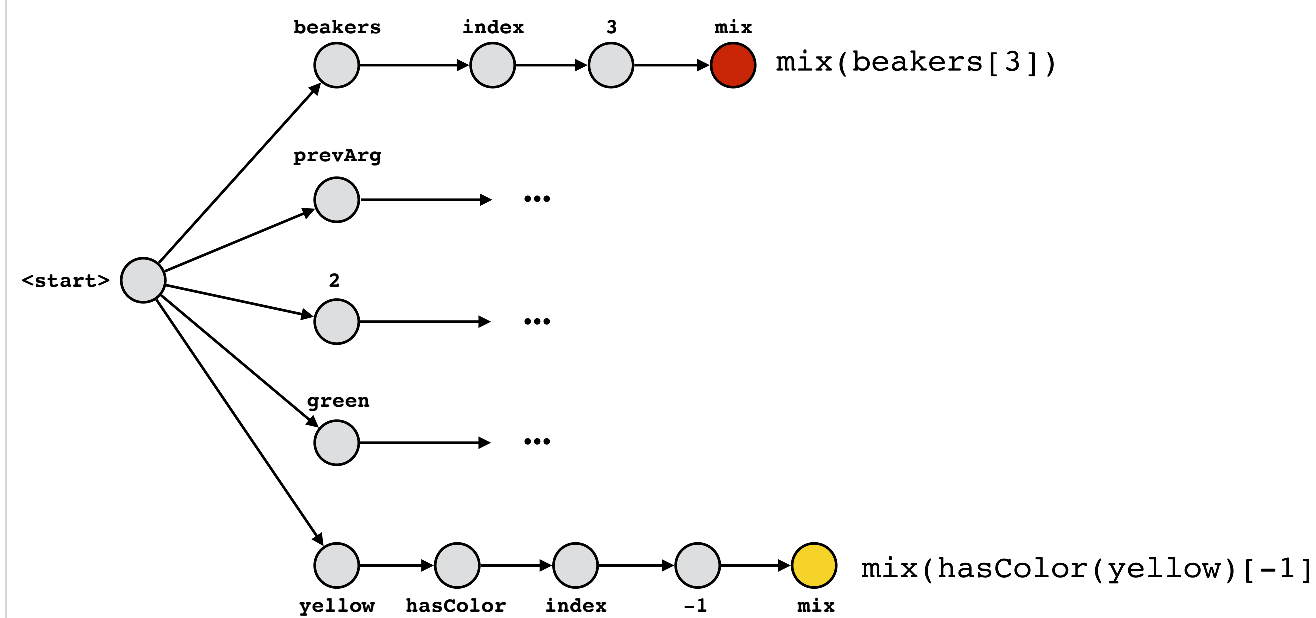
1. meritocratic gradient weights

2. randomized beam search

This brings us to our second contribution, randomized beam search.

REINFORCE

$$g^{\mathrm{RL}} = \sum_{\mathbf{z}} p_\theta(\mathbf{z} \mid x)\, R(\mathbf{z})\, \nabla_\theta \log p_\theta(\mathbf{z} \mid x)$$



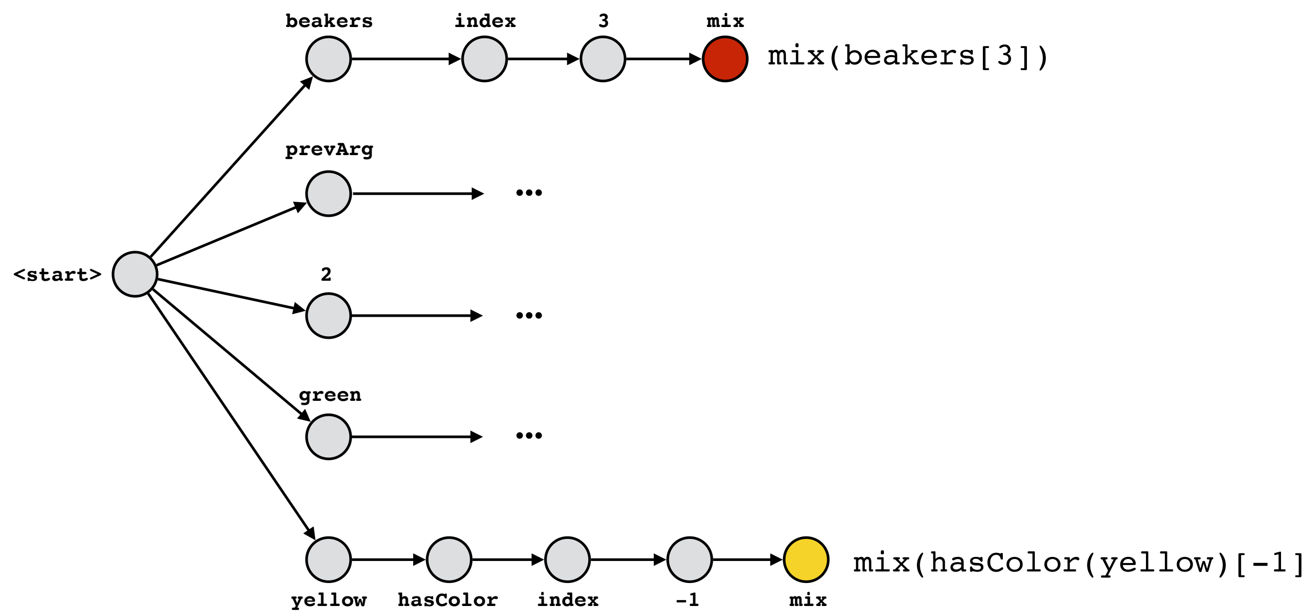
The standard approach in RL for approximating the intractable sum is to use sampling.

Rather than enumerating all programs, we just sample one program from the model policy, and perform the update.

The initial policy is quite bad, so at first, it is just randomly exploring.

REINFORCE

$$g^{\mathrm{RL}} = \sum_{\mathbf{z}} p_\theta(\mathbf{z} \mid x) R(\mathbf{z}) \nabla_\theta \log p_\theta(\mathbf{z} \mid x)$$



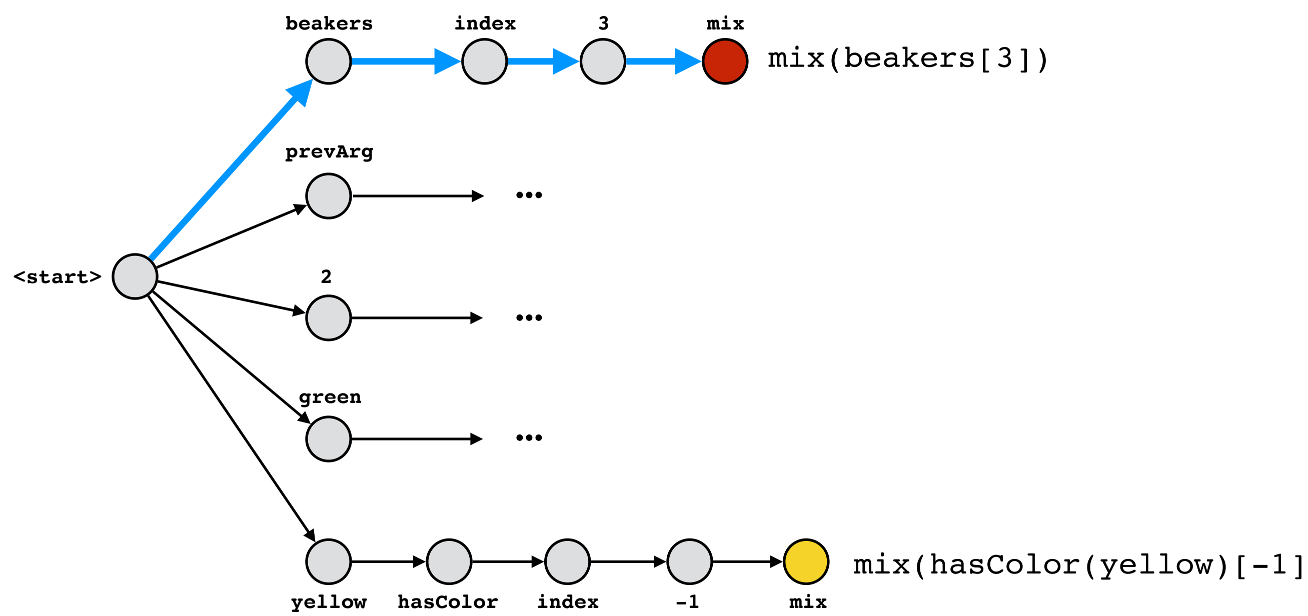
The standard approach in RL for approximating the intractable sum is to use sampling.

Rather than enumerating all programs, we just sample one program from the model policy, and perform the update.

The initial policy is quite bad, so at first, it is just randomly exploring.

REINFORCE

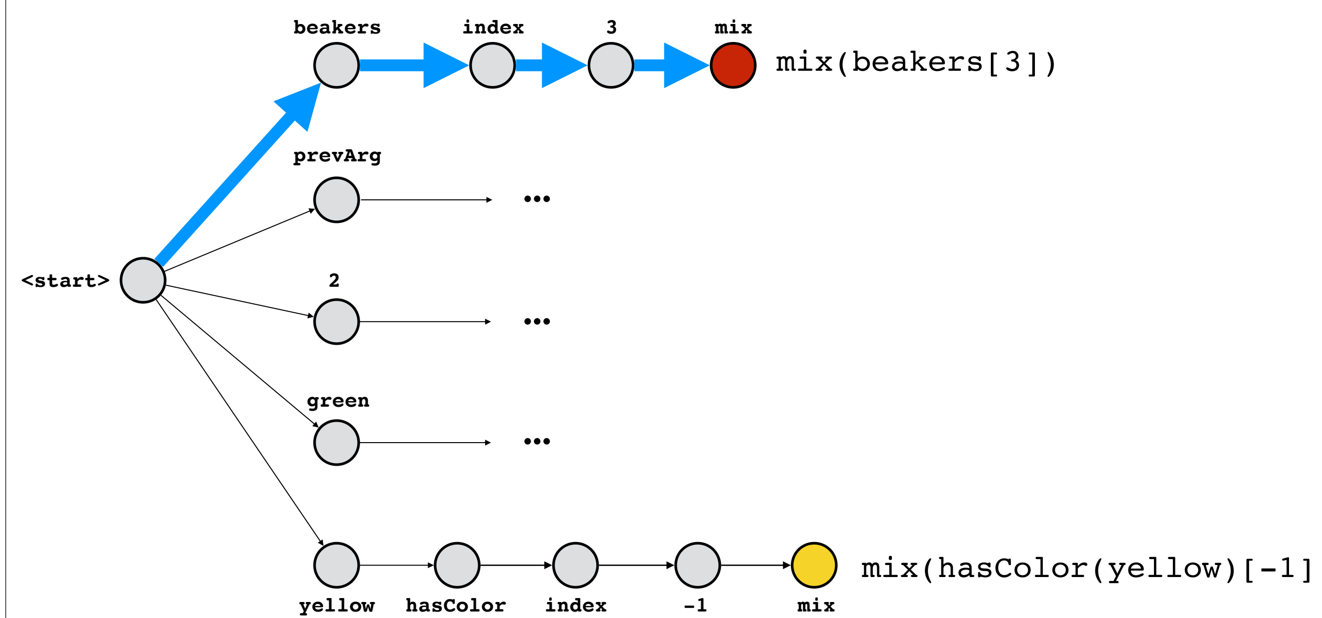
$$g^{\mathrm{RL}} = \sum_{\mathbf{z}} p_\theta(\mathbf{z} \mid x) R(\mathbf{z}) \nabla_\theta \log p_\theta(\mathbf{z} \mid x)$$



Once REINFORCE finds a program that earns reward, it upweights all actions along that path.

REINFORCE

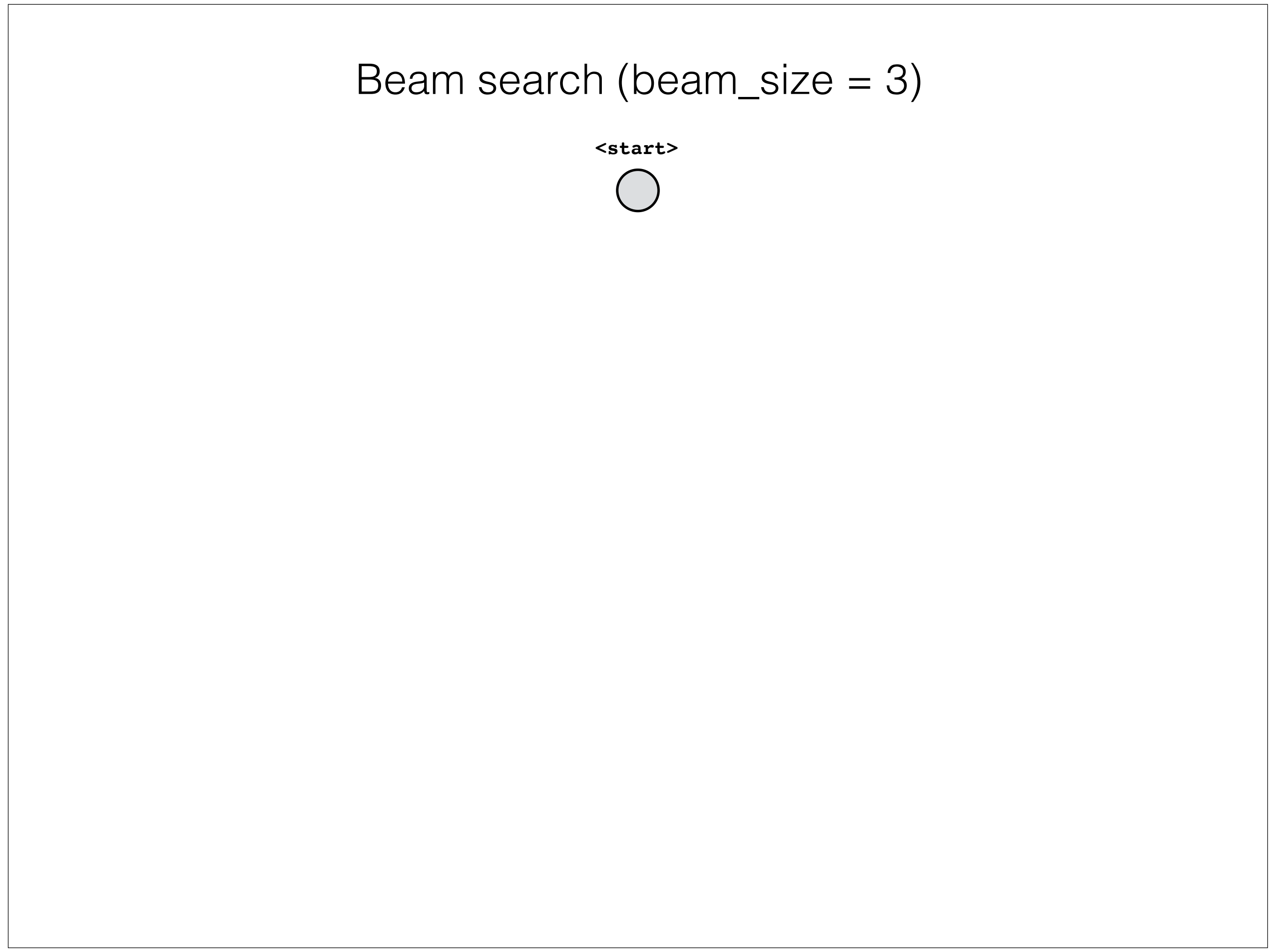
$$g^{\mathrm{RL}} = \sum_{\mathbf{z}} p_\theta(\mathbf{z} \mid x) R(\mathbf{z}) \nabla_\theta \log p_\theta(\mathbf{z} \mid x)$$



But if you upweight that path, then you are more likely to go there next time and upweight again, and this one path eventually steals all the probability from the alternatives.

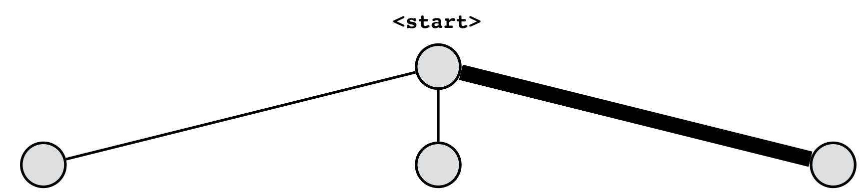
At this point, there is very low probability that REINFORCE will ever discover the correct program

This is a problem for meritocratic updates because we cannot renormalize over multiple programs if we only find one program.



Our solution is to borrow a standard idea from the MML literature, where beam search is very common.

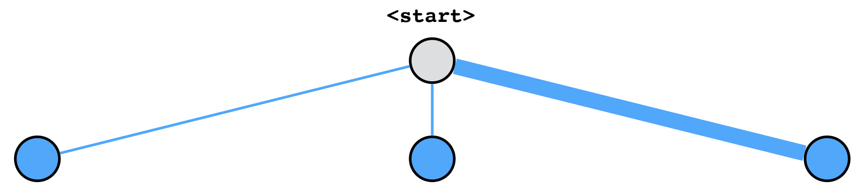
Beam search (beam_size = 3)

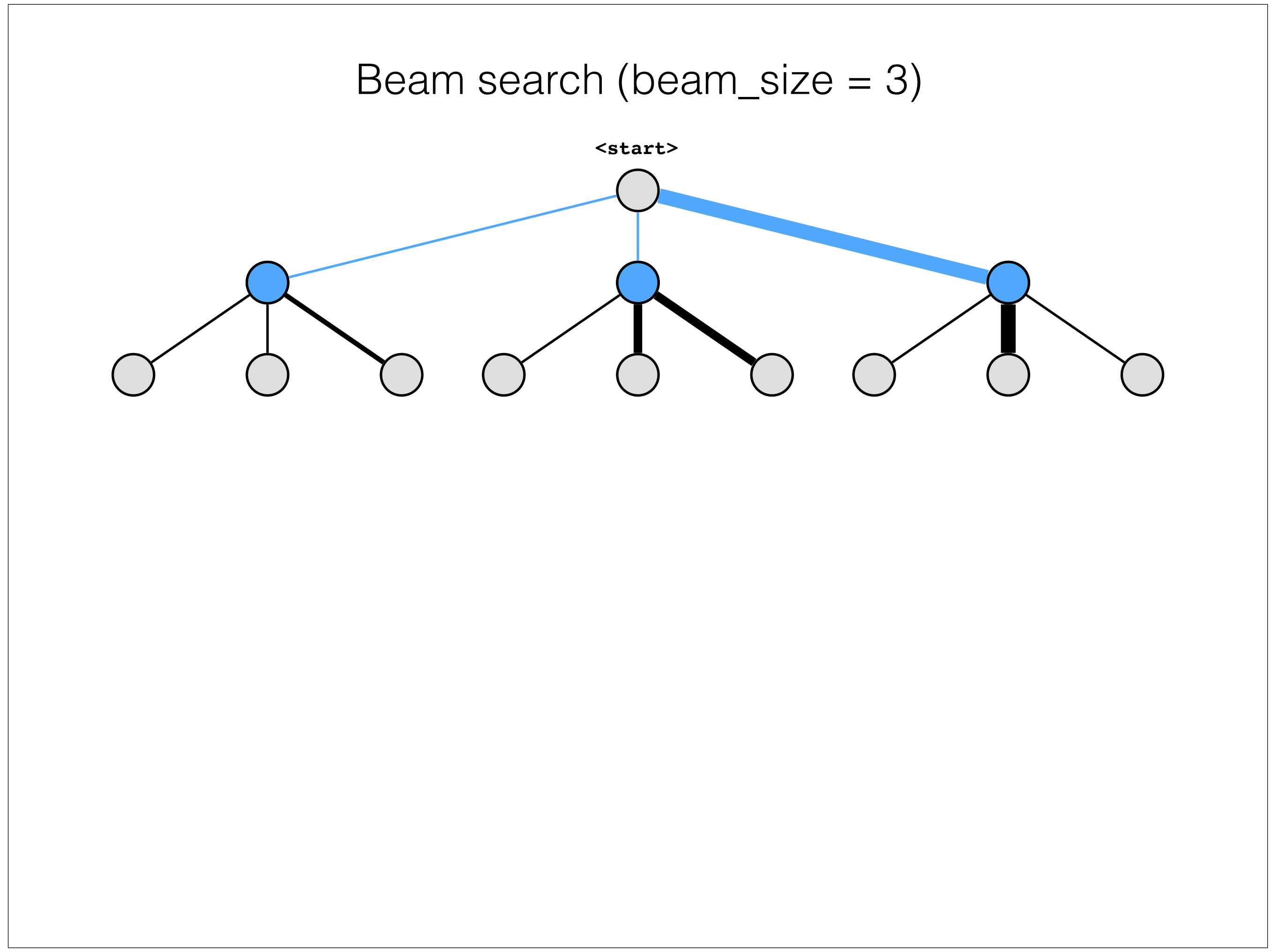


In beam search, rather than taking one path, we try to take multiple paths simultaneously.

Here, even though one path has much higher probability, we will take all three, because we have a beam size of 3.

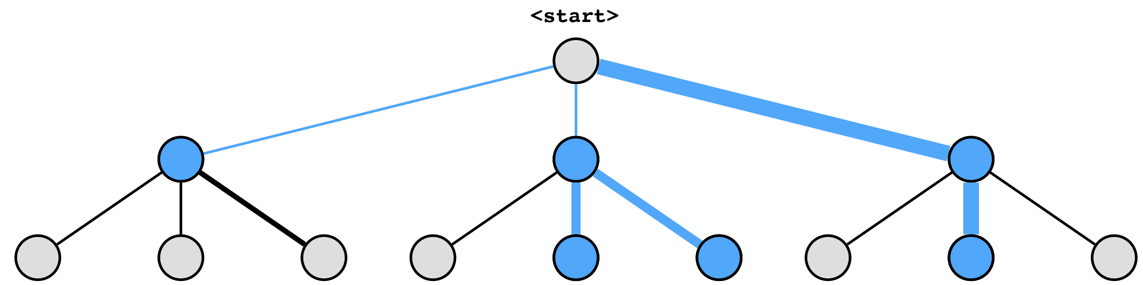
Beam search (beam_size = 3)



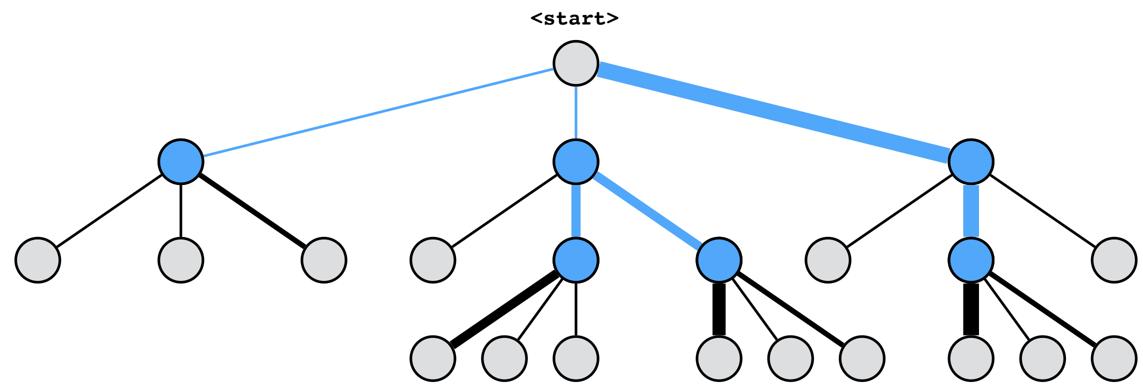


then we look at our next options, and again take the top 3

Beam search (beam_size = 3)

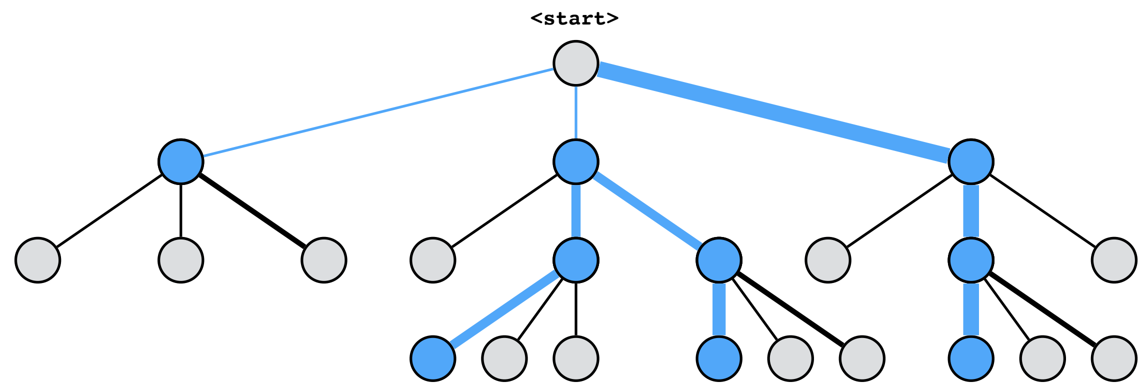


Beam search (beam_size = 3)



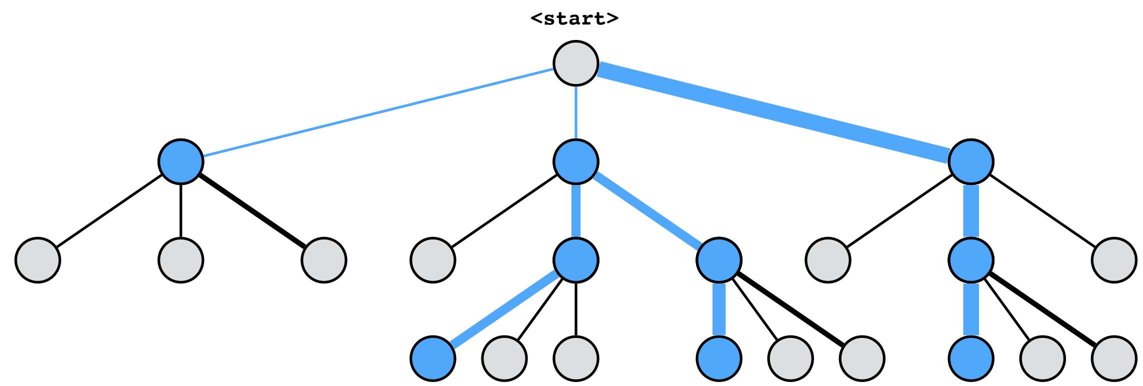
and so on

Beam search (beam_size = 3)



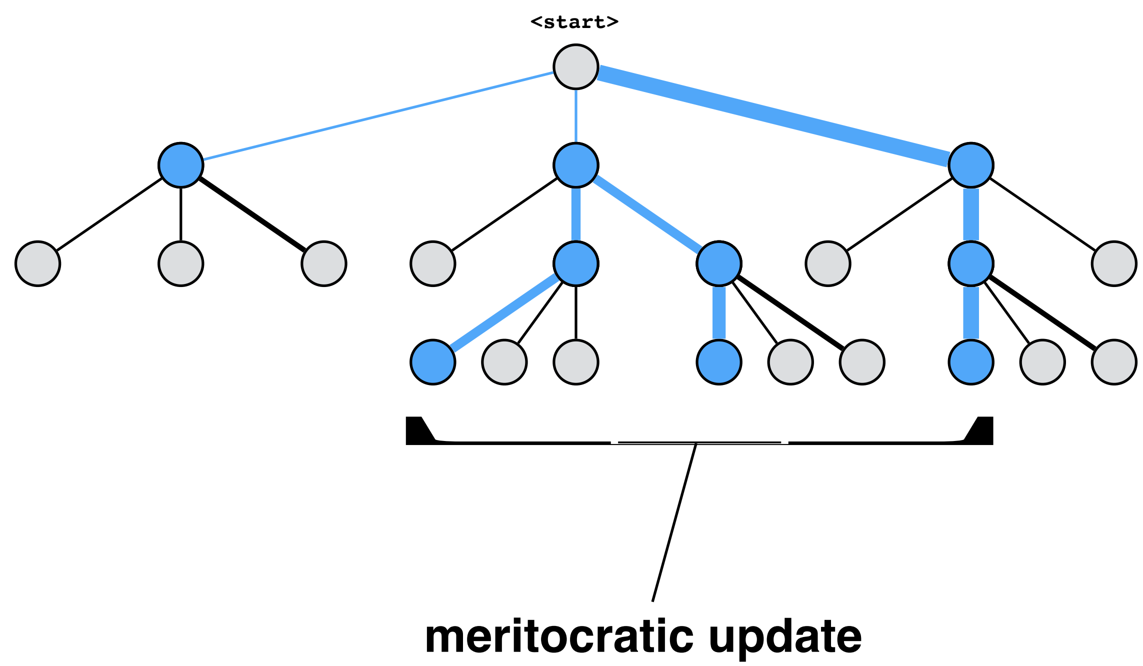
and so forth

Beam search (beam_size = 3)

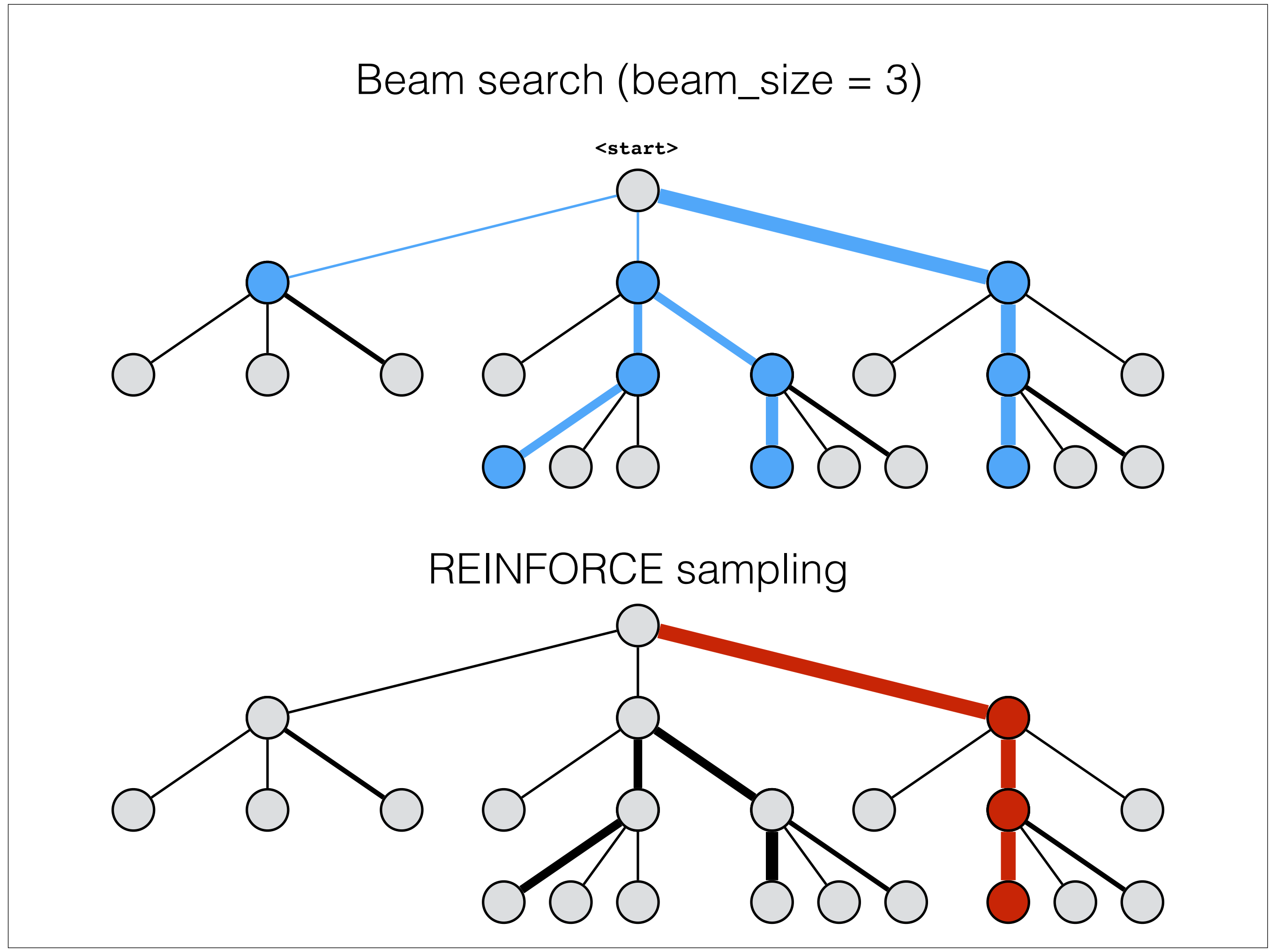


Now that we have multiple programs, we can normalize over them and do our meritocratic update

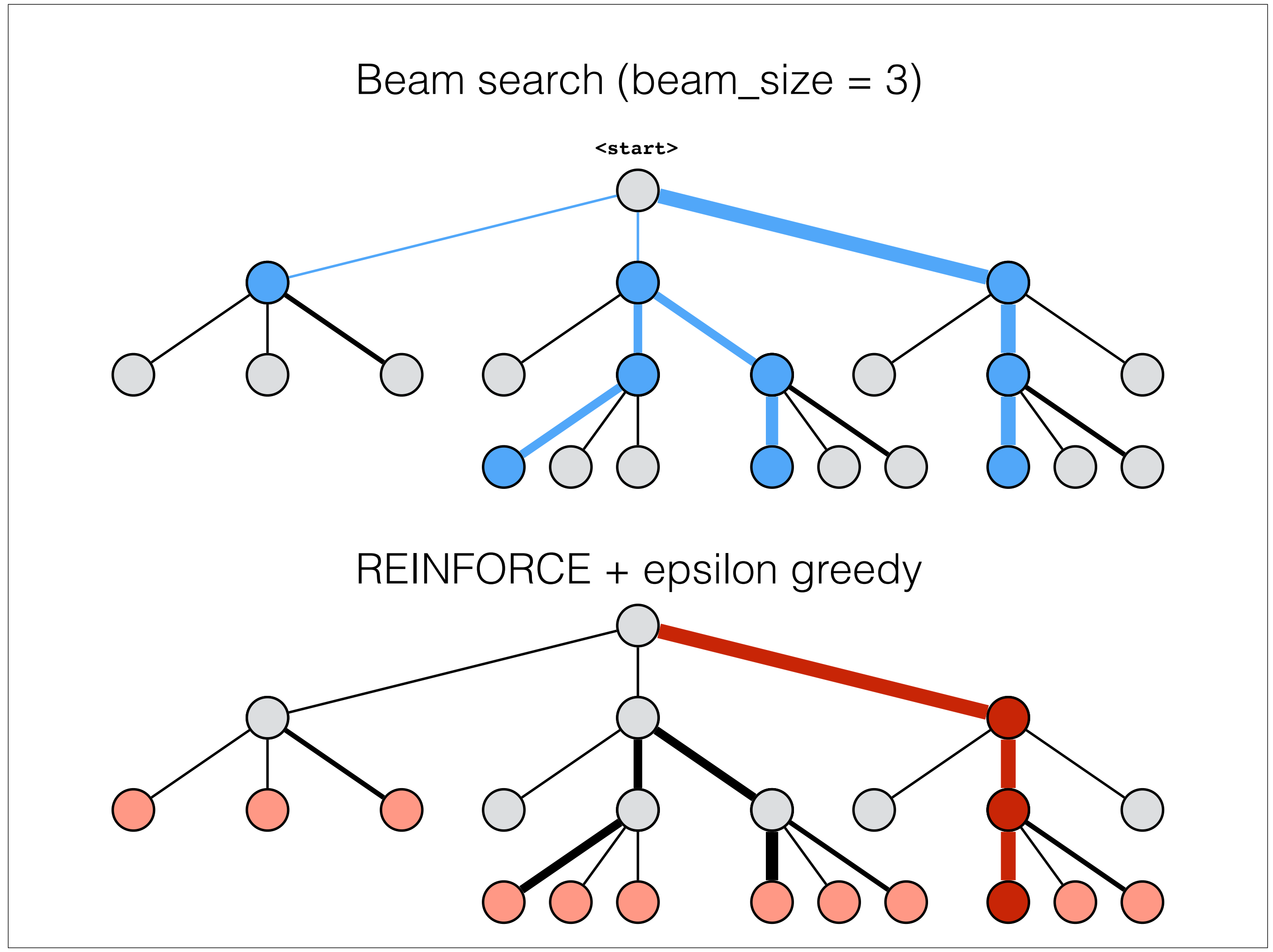
Beam search (beam_size = 3)



Now that we have multiple programs, we can normalize over them and do our meritocratic update

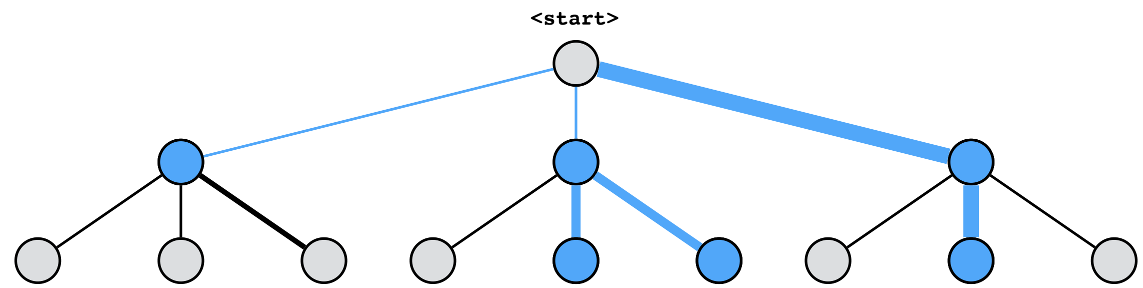


Also, compared to REINFORCE, this guarantees coverage of at least 3 programs, even if the model has concentrated its probability on one.



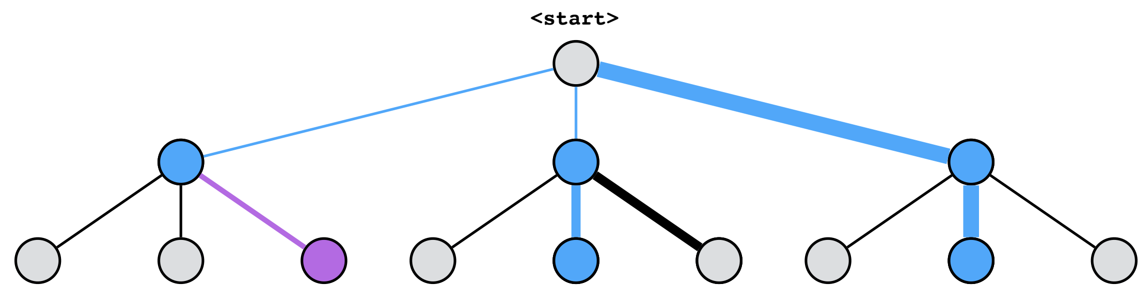
On the other hand, typical implementations of REINFORCE usually get a big boost in performance from being epsilon greedy
That is, epsilon of the time, they ignore the policy and choose a random action.

Randomized beam search



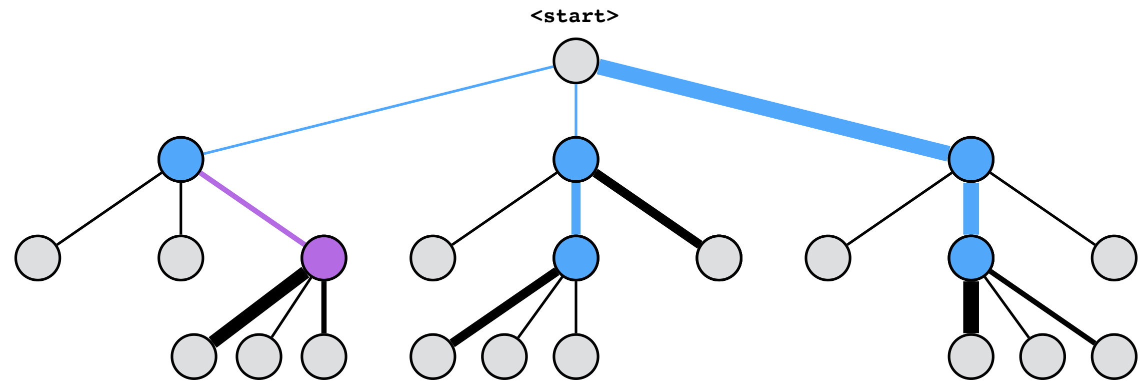
So we decided to add this trick to beam search.

Randomized beam search

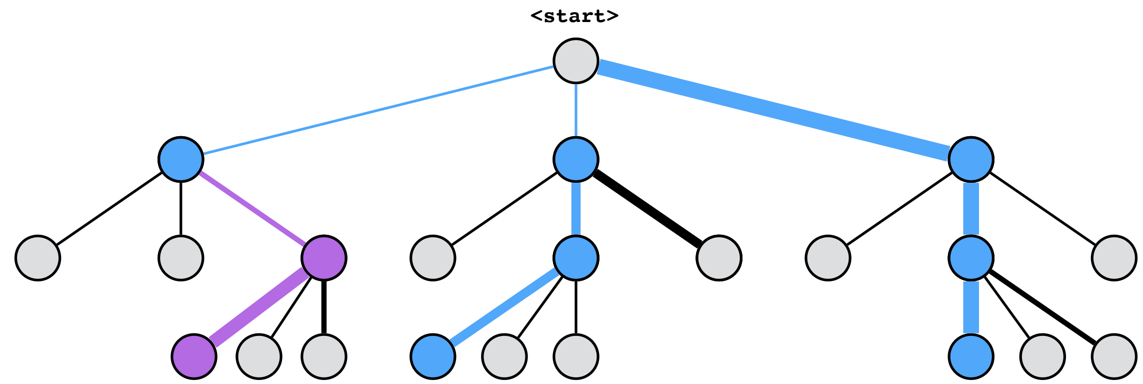


At each time step, we randomly replace a few elements on the beam with random choices.

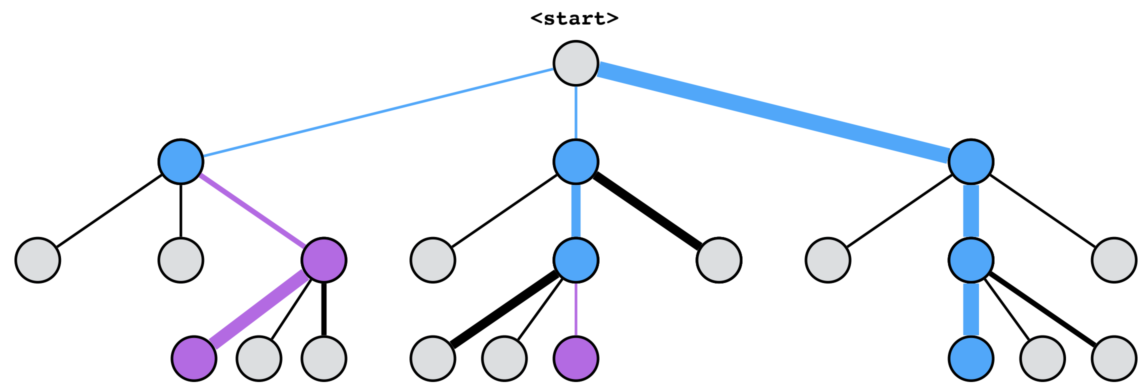
Randomized beam search



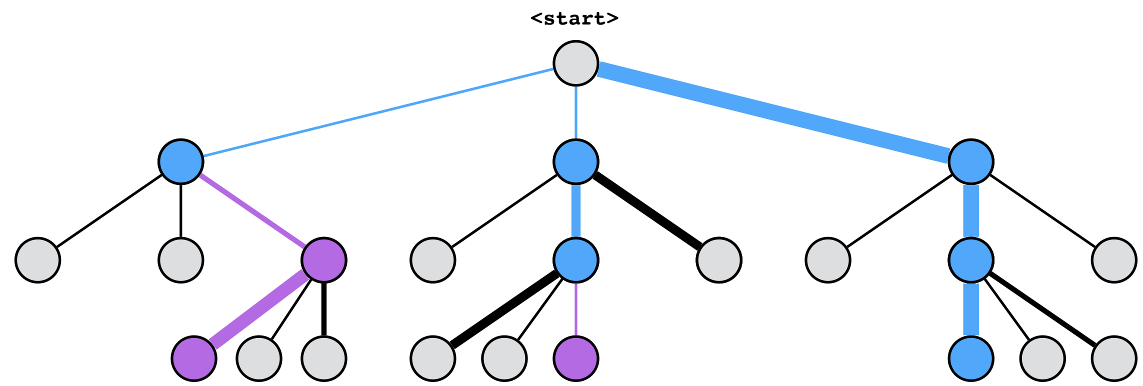
Randomized beam search



Randomized beam search

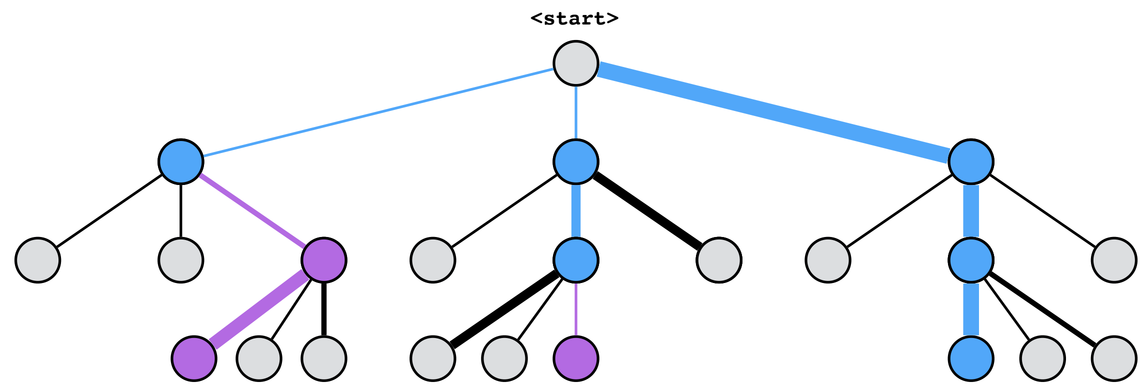


Randomized beam search



Purple nodes indicate where we chose randomly.

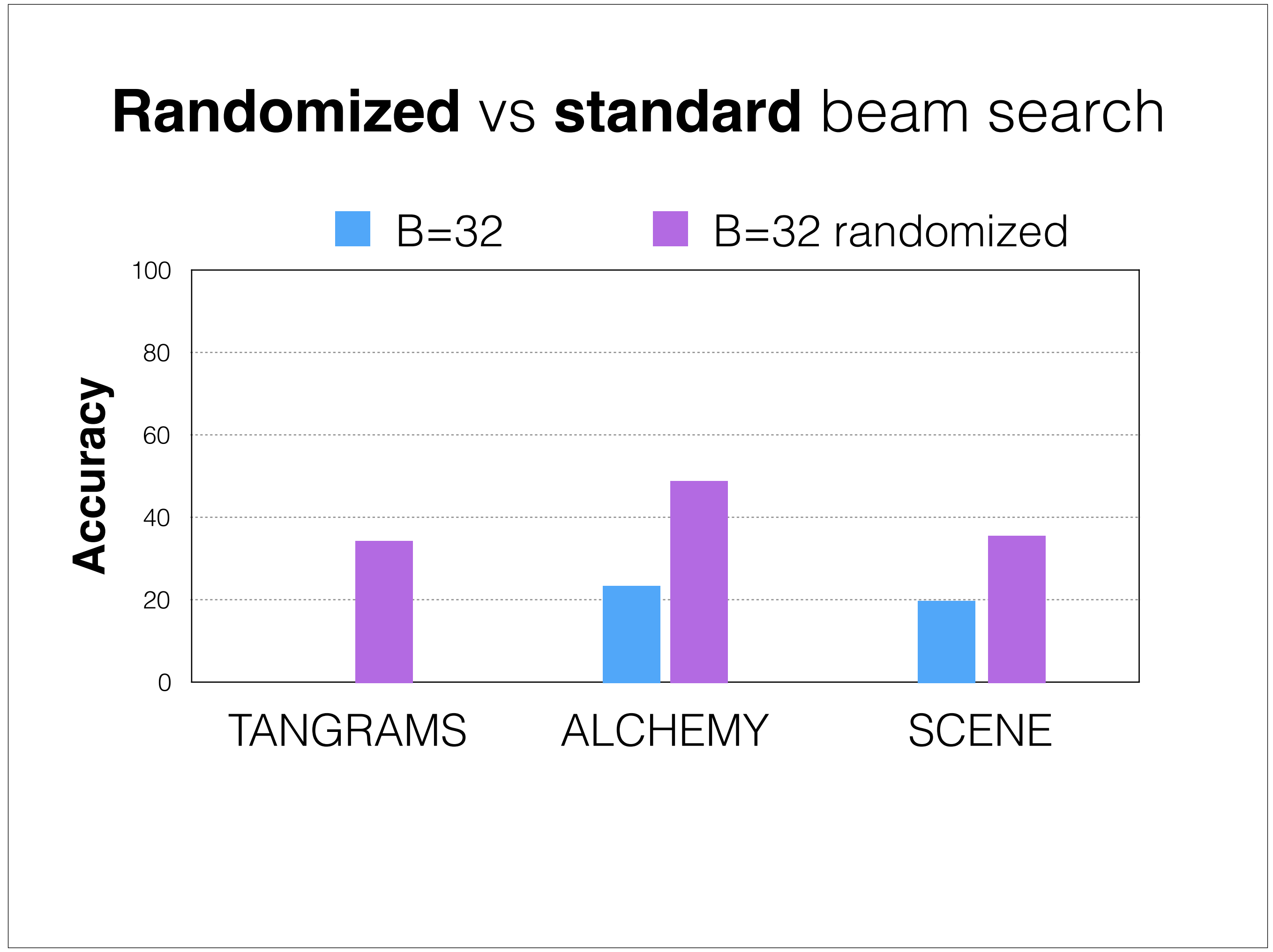
Randomized beam search



Extensions: diverse beam search

Gimpel+ 2013, Li+ 2015, Cho 2016, Vijayakumar+ 2016

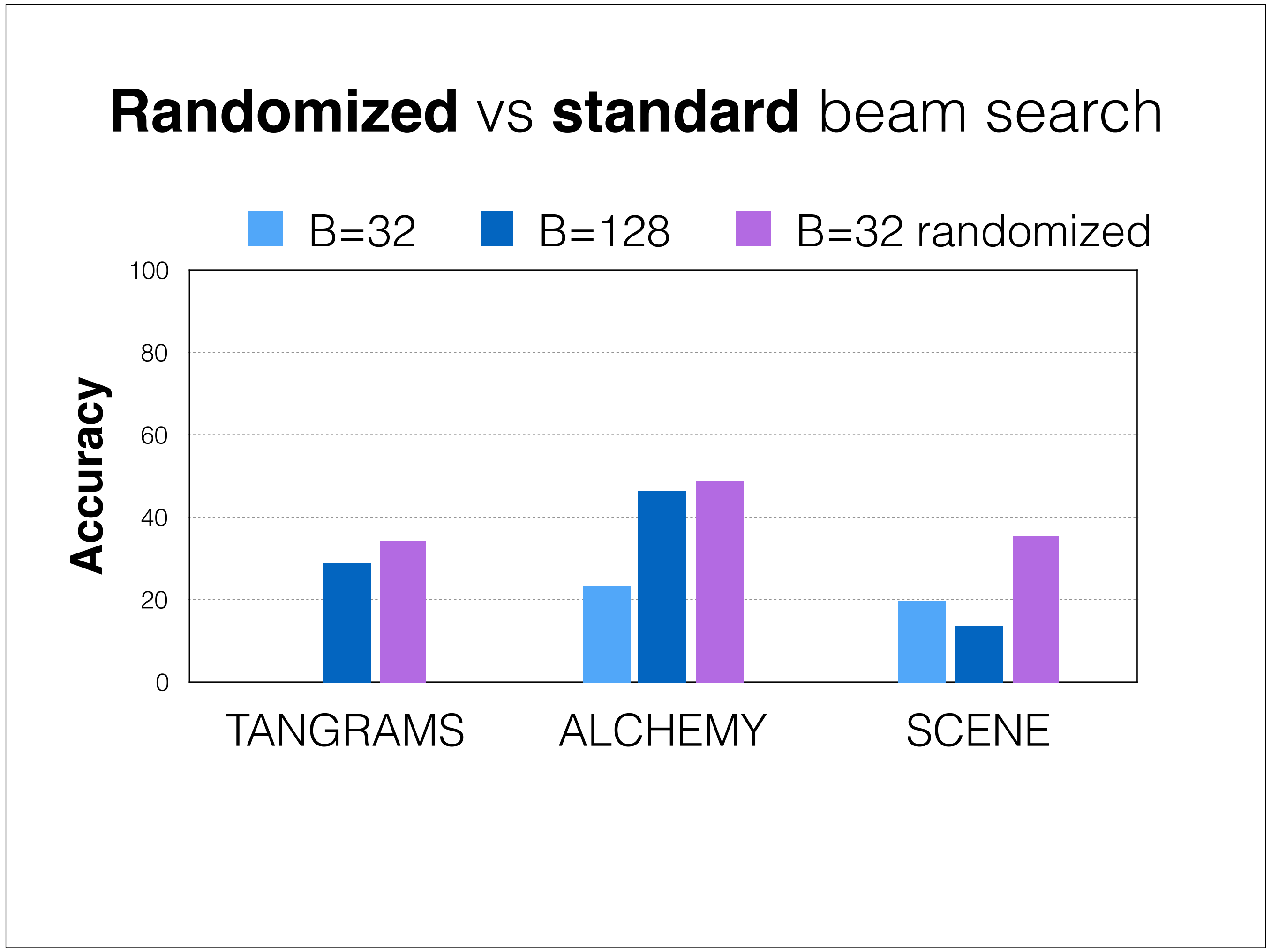
This is a very basic way of adding diversity to beam search, and you could certainly extend this to more sophisticated approaches.



But interestingly, we found that this very simple trick really boosts the performance beam search.

Across all domains, we see that randomized beam search performs much better than its classic counterpart.

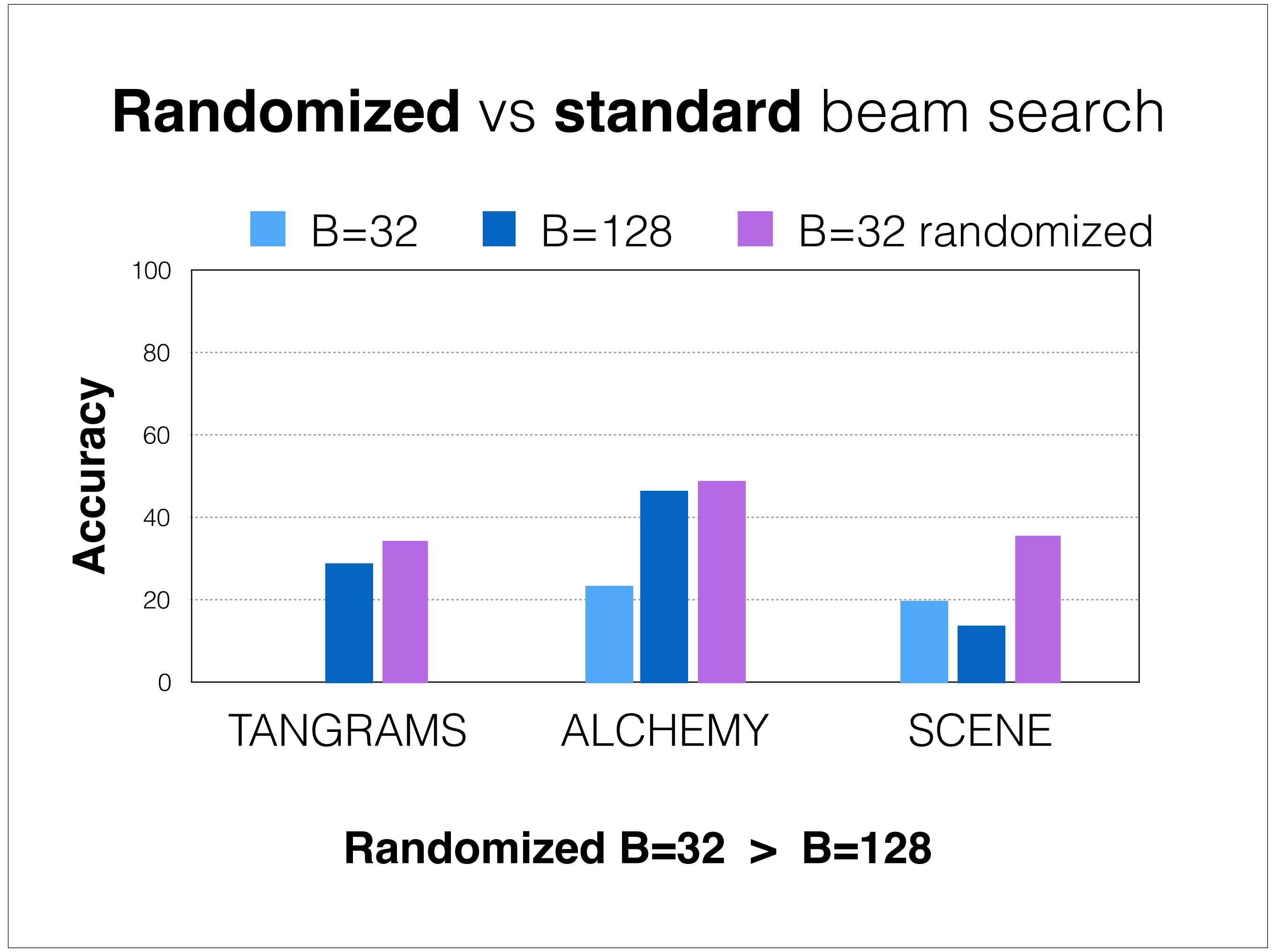
In fact, we found that just randomizing the beam is actually better than increasing the beam size by 4x.
So, it's interesting to note that this trick isn't more commonly used.



But interestingly, we found that this very simple trick really boosts the performance beam search.

Across all domains, we see that randomized beam search performs much better than its classic counterpart.

In fact, we found that just randomizing the beam is actually better than increasing the beam size by 4x.
So, it's interesting to note that this trick isn't more commonly used.



But interestingly, we found that this very simple trick really boosts the performance beam search.

Across all domains, we see that randomized beam search performs much better than its classic counterpart.

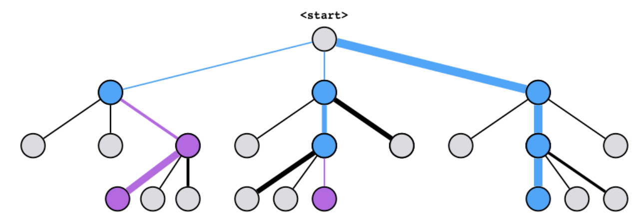
In fact, we found that just randomizing the beam is actually better than increasing the beam size by 4x.
So, it's interesting to note that this trick isn't more commonly used.

SOLUTION: RandoMer

Our final proposal is to combine the randomized beam search with the meritocratic updates, which we're going to call RandoMer.

SOLUTION: RandoMer

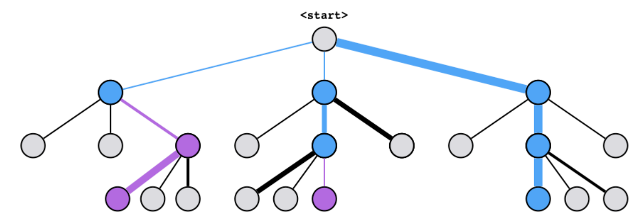
RANDOmized beam search



Our final proposal is to combine the randomized beam search with the meritocratic updates, which we're going to call RandoMer.

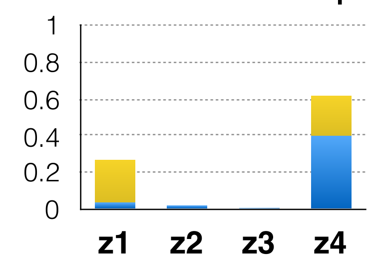
SOLUTION: RandoMer

RANDOmized beam search



+

MERitocratic updates

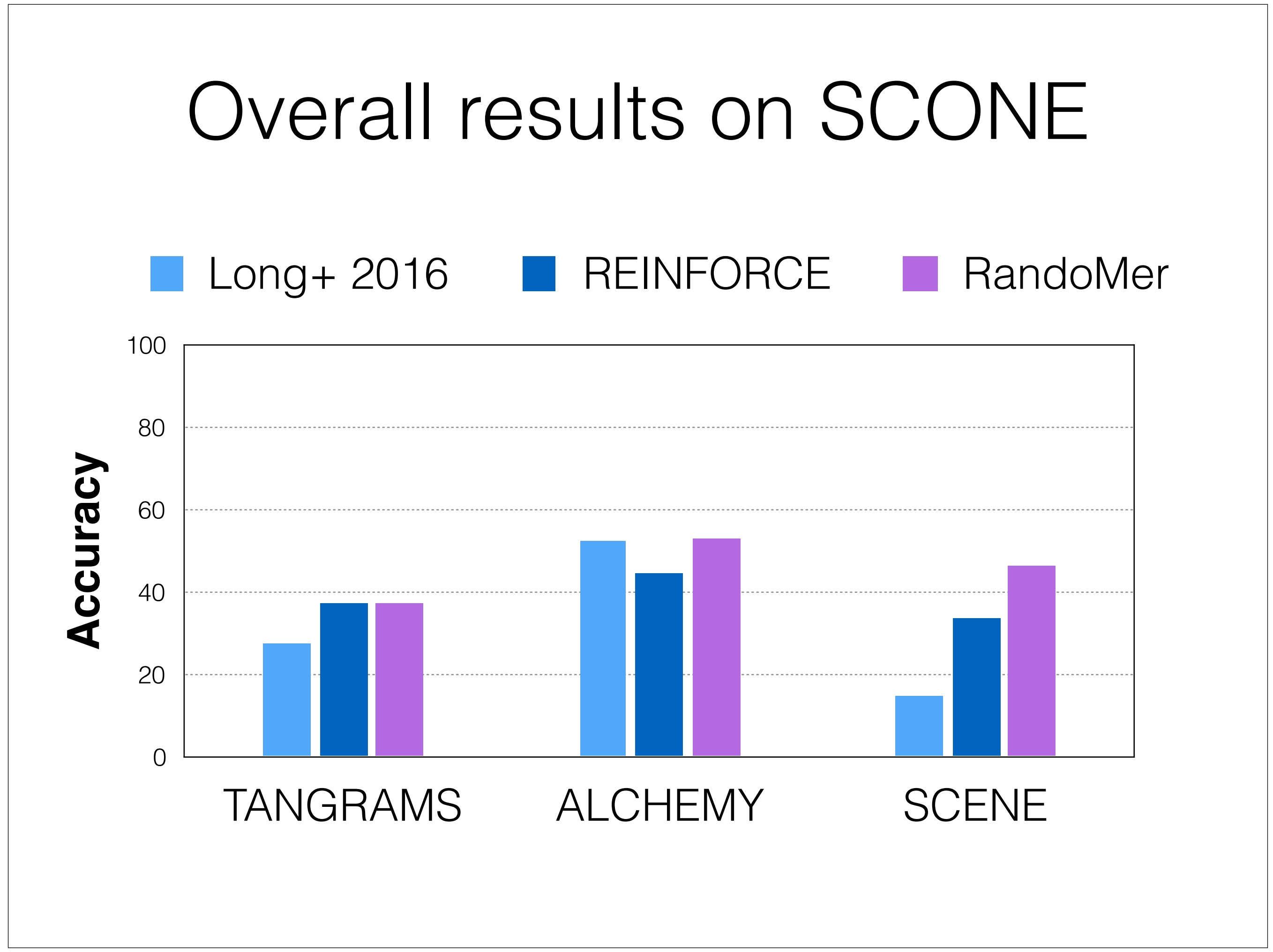


Our final proposal is to combine the randomized beam search with the meritocratic updates, which we're going to call RandoMer.

Overall results on SCONE

And the result is that we do consistently better across all of the tasks in SCONE.

The MML results come from previous work, which used a log-linear model with manually crafted features.
The REINFORCE and RandoMer use the exact same neural architecture, but are just trained differently.



And the result is that we do consistently better across all of the tasks in SCONE.

The MML results come from previous work, which used a log-linear model with manually crafted features.
The REINFORCE and RandoMer use the exact same neural architecture, but are just trained differently.

Conclusion

(slides will be at kelvinguu.com)

So, to wrap things up:

In this talk we presented the task of parsing language into programs

We introduced this idea of spurious programs and superstitious behavior, which is quite common in weak supervision

And, as far as I can tell, we are the first to really propose a solution to this problem, in the form of more balanced exploration and gradients

Conclusion

TASK: translate language into programs, learning from weak supervision.

`input`

```
x = "then into the first"
```

`output`

```
y = pour(prevArg(2), beakers[1])
```

(slides will be at kelvinguu.com)

So, to wrap things up:

In this talk we presented the task of parsing language into programs

We introduced this idea of spurious programs and superstitious behavior, which is quite common in weak supervision

And, as far as I can tell, we are the first to really propose a solution to this problem, in the form of more balanced exploration and gradients

Conclusion

TASK: translate language into programs, learning from weak supervision.

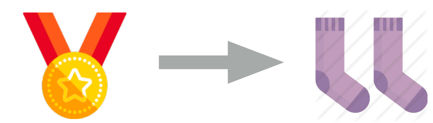
input

```
x = "then into the first"
```

output

```
y = pour(prevArg(2), beakers[1])
```

PROBLEM: spurious programs and superstitious behavior.



(slides will be at kelvinguu.com)

So, to wrap things up:

In this talk we presented the task of parsing language into programs

We introduced this idea of spurious programs and superstitious behavior, which is quite common in weak supervision

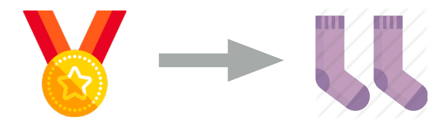
And, as far as I can tell, we are the first to really propose a solution to this problem, in the form of more balanced exploration and gradients

Conclusion

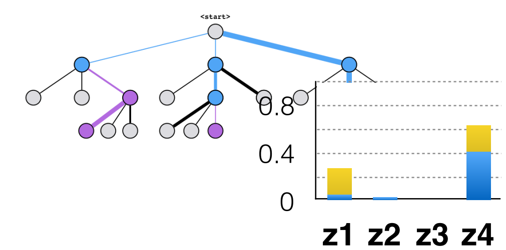
TASK: translate language into programs, learning from weak supervision.

input
```
x = "then into the first"
```
output
```
y = pour(prevArg(2), beakers[1])
```

PROBLEM: spurious programs and superstitious behavior.



SOLUTION:
combine RL + MML = RandoMer to combat spurious programs



(slides will be at kelvinguu.com)

So, to wrap things up:

In this talk we presented the task of parsing language into programs

We introduced this idea of spurious programs and superstitious behavior, which is quite common in weak supervision

And, as far as I can tell, we are the first to really propose a solution to this problem, in the form of more balanced exploration and gradients

Extras

Extras

- **NOT about mixing supervised learning and RL**

Extras

- **NOT about mixing supervised learning and RL**
    - Ranzato+ 2015, Bengio+ 2015, Norouzi+ 2016

Extras

- **NOT about mixing supervised learning and RL**
  - Ranzato+ 2015, Bengio+ 2015, Norouzi+ 2016
  - In contrast, our approach <u>does not</u> rely on any fully supervised training data.

Extras

- **NOT about mixing supervised learning and RL**
  - Ranzato+ 2015, Bengio+ 2015, Norouzi+ 2016
  - In contrast, our approach does not rely on any fully supervised training data.
- **RL vs MML**

Extras

- **NOT about mixing supervised learning and RL**
  - Ranzato+ 2015, Bengio+ 2015, Norouzi+ 2016
  - In contrast, our approach does not rely on any fully supervised training data.
- **RL vs MML**
  - Rich literature on **reinforcement learning** (REINFORCE, actor-critic, etc.)

Extras

- **NOT about mixing supervised learning and RL**
  - Ranzato+ 2015, Bengio+ 2015, Norouzi+ 2016
  - In contrast, our approach <u>does not</u> rely on any fully supervised training data.
- **RL vs MML**
  - Rich literature on **reinforcement learning** (REINFORCE, actor-critic, etc.)
  - Rich literature on learning **latent variable models** (EM, method of moments, variational inference)

Extras

- **NOT about mixing supervised learning and RL**
  - Ranzato+ 2015, Bengio+ 2015, Norouzi+ 2016
  - In contrast, our approach does not rely on any fully supervised training data.
- **RL vs MML**
  - Rich literature on **reinforcement learning** (REINFORCE, actor-critic, etc.)
  - Rich literature on learning **latent variable models** (EM, method of moments, variational inference)
  - Many more interesting connections to explore