Tuning Random Generators

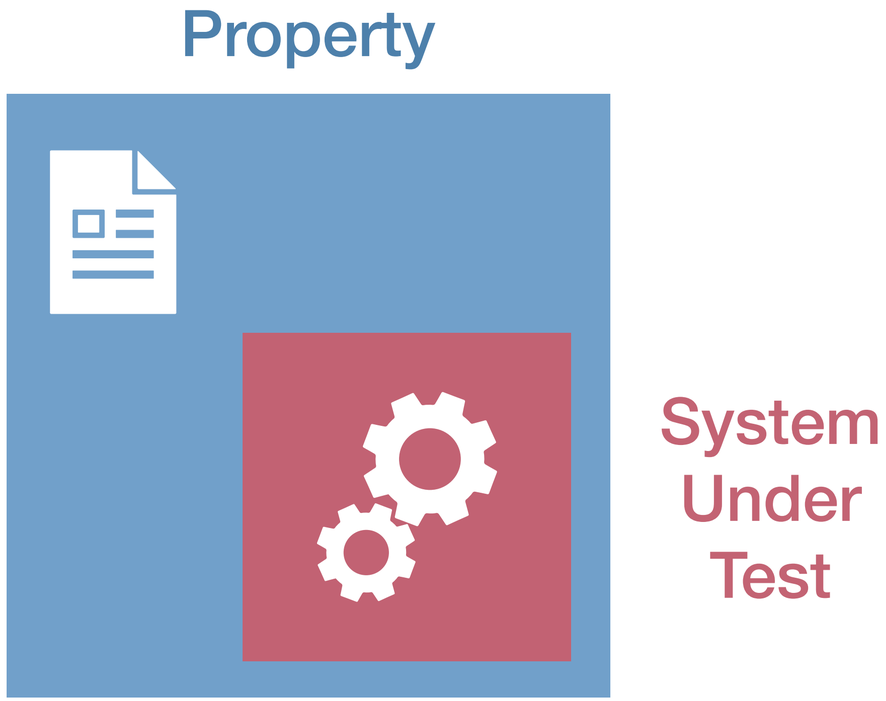
Property-Based Testing as Probabilistic Programming

Ryan Tjoa, Poorva Garg, Harrison Goldstein, Todd Millstein, Benjamin C. Pierce, Guy Van den Broeck

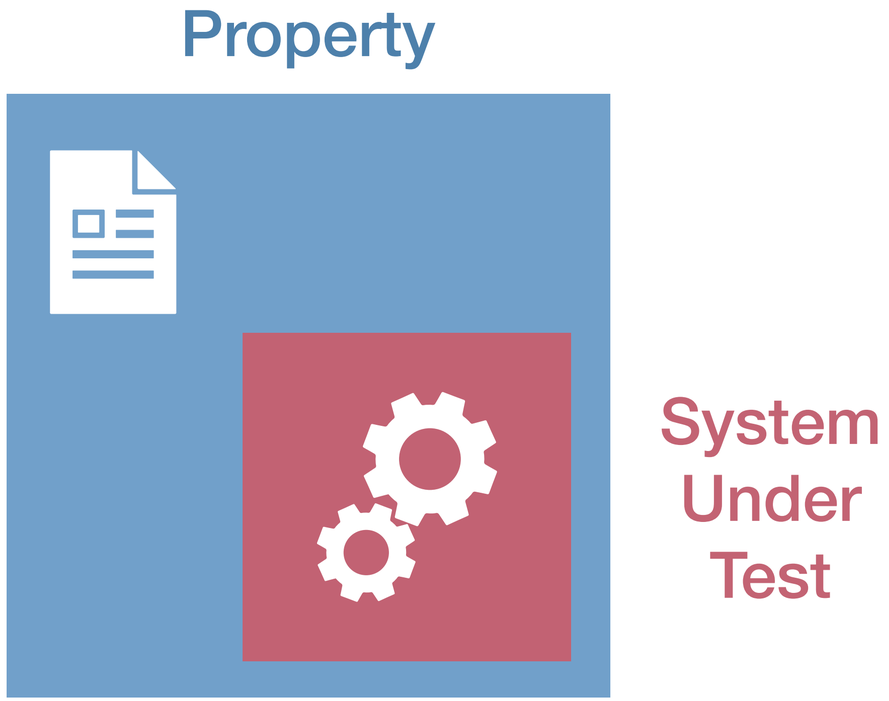
Property-based testing (PBT)

System
Under
Test

Property-based testing (PBT)

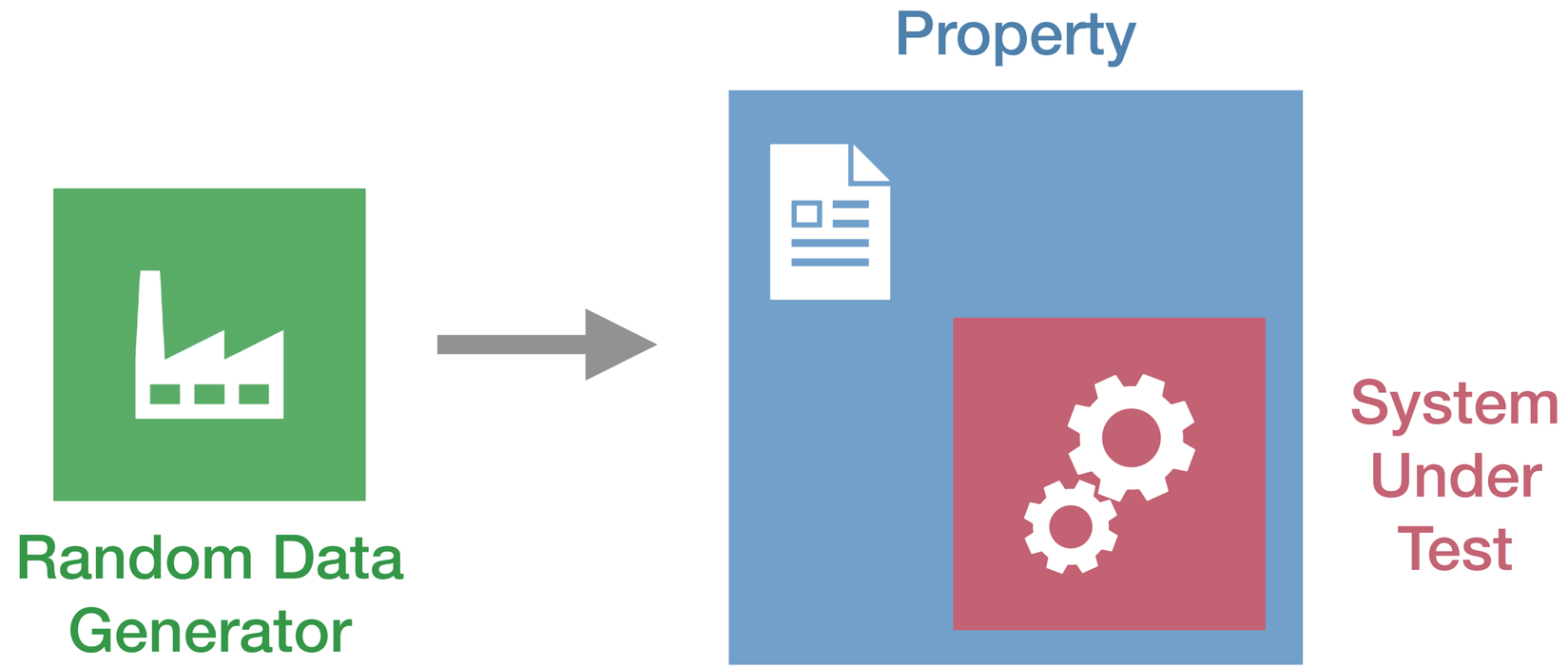


Property-based testing (PBT)




```
∀list. reverse(reverse(list)) = list
```

Property-based testing (PBT)




```
∀list. reverse(reverse(list)) = list
```

A generator and its distribution

```
let genList n =
  match n with
  | 0 -> []
  | S n' ->
    freq [
      1, [];
      1, let x = flip 0.5 in
         let xs = genList n' in
         x :: xs
    ]
```

A generator and its distribution

```
let genList n =
  match n with
  | 0 -> []
  | S n' ->
    freq [
      1, [];
      1, let x = flip 0.5 in
         let xs = genList n' in
         x :: xs
    ]
```

Uniformly choose from two branches

A generator and its distribution

```
let genList n =
  match n with
  | 0 -> []
  | S n' ->
    freq [
      1, [];
      1, let x = flip 0.5 in
         let xs = genList n' in
         x :: xs
    ]
```

50% true
50% false

Uniformly choose from two branches

A generator and its distribution

```
let genList n =
  match n with
  | 0 -> []
  | S n' ->
    freq [
      1, [];
      1, let x = flip 0.5 in
         let xs = genList n' in
         x :: xs
    ]
```

50% true
50% false

Uniformly choose from two branches

```
[]
[F]
[]
[]
[T;F;F]
[]
[T]
[]
[F]
[]
...
```

A generator and its distribution

```
let genList n =
  match n with
  | 0 -> []
  | S n' ->
    freq [
      1, [];
      1, let x = flip 0.5 in
         let xs = genList n' in
         x :: xs
    ]
```

Uniformly choose from two branches

50% true
50% false

⇝

```
[]
[F]
[]
[]
[T;F;F]
[]
[T]
[]
[F]
[]
...
```

A generator and its distribution

```
let genList n =
  match n with
  | 0 -> []
  | S n' ->
    freq [
      1, [];
      3, let x = flip 0.5 in
         let xs = genList n' in
         x :: xs
    ]
```

A generator and its distribution

```
let genList n =
  match n with
  | 0 -> []
  | S n' ->
    freq [
      1, [];
      3, let x = flip 0.5 in
         let xs = genList n' in
         x :: xs
    ]
```

```
[T;F;T;T]
[T;F;F]
[T;F;T]
[F;T;T;F;T;T;F;F;T;T]
[F]
[T]
[T;T;T;T;T;F;F;F;F]
[F;T;T;T]
[F;F;T;F]
[]
...
```

A generator and its distribution

```
let genList n =
  match n with
  | 0 -> []
  | S n' ->
    freq [
      1, [];
      3, let x = flip 0.5 in
         let xs = genList n' in
```

In general, it's difficult to reason how weights affect the distribution

```
[T;F;T;T]
[T;F;F]
[T;F;T]
[F;T;T;F;T;T;F;F;T;T]
[F]
[T]
[T;T;T;T;T;F;F;F;F]
[F;T;T;T]
[F;F;T;F]
[]
...
```

A generator and its distribution

```
let genList n =
  match n with
  | 0 -> []
  | S n' ->
    freq [
      1, [];
      3, let x = flip 0.5 in
         let xs = genList n' in
```

What distribution do we *want*?

```
[T;F;T;T]
[T;F;F]
[T;F;T]
[F;T;T;F;T;T;F;F;T;T]
[F]
[T]
[T;T;T;T;T;F;F;F;F]
[F;T;T;T]
[F;F;T;F]
[]
...
```

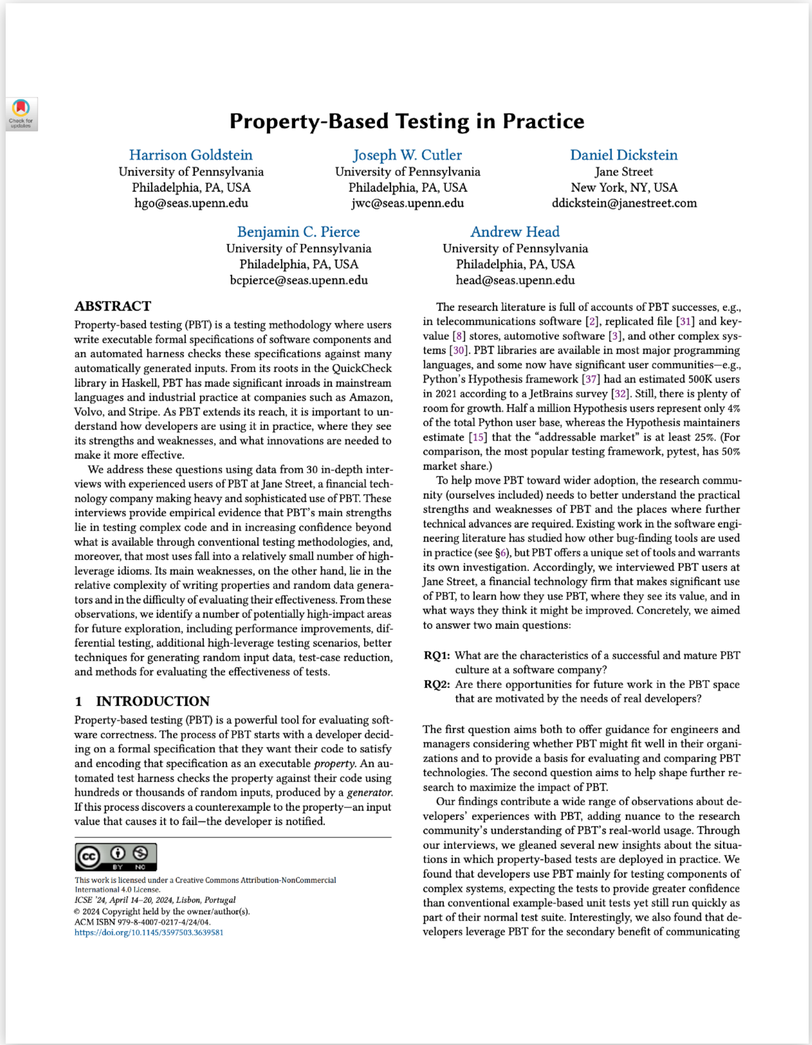
In general, it's difficult to reason how weights affect the distribution

Choosing probabilities is a “mental strain” that feels like it requires “[studying] probability and statistics.”

Property-Based Testing in Practice (ICSE 2024)

“The most difficult and unsatisfactory part of engineering a good random tester is setting the probabilities properly.”

John Regehr (Blog Post)



Property-Based Testing in Practice

Harrison Goldstein
University of Pennsylvania
Philadelphia, PA, USA
hgo@seas.upenn.edu

Joseph W. Cutler
University of Pennsylvania
Philadelphia, PA, USA
jwc@seas.upenn.edu

Daniel Dickstein
Jane Street
New York, NY, USA
ddickstein@janestreet.com

Benjamin C. Pierce
University of Pennsylvania
Philadelphia, PA, USA
bcpierce@seas.upenn.edu

Andrew Head
University of Pennsylvania
Philadelphia, PA, USA
head@seas.upenn.edu

ABSTRACT

Property-based testing (PBT) is a testing methodology where users write executable formal specifications of software components and an automated harness checks these specifications against many automatically generated inputs. From its roots in the QuickCheck library in Haskell, PBT has made significant inroads in mainstream languages and industrial practice at companies such as Amazon, Volvo, and Stripe. As PBT extends its reach, it is important to understand how developers are using it in practice, where they see its strengths and weaknesses, and what innovations are needed to make it more effective.

We address these questions using data from 30 in-depth interviews with experienced users of PBT at Jane Street, a financial technology company making heavy and sophisticated use of PBT. These interviews provide empirical evidence that PBT's main strengths lie in testing complex code and in increasing confidence beyond what is available through conventional testing methodologies, and, moreover, that most uses fall into a relatively small number of high-leverage idioms. Its main weaknesses, on the other hand, lie in the relative complexity of writing properties and random data generators and in the difficulty of evaluating their effectiveness. From these observations, we identify a number of potentially high-impact areas for future exploration, including performance improvements, differential testing, additional high-leverage testing scenarios, better techniques for generating random input data, test-case reduction, and methods for evaluating the effectiveness of tests.

1 INTRODUCTION

Property-based testing (PBT) is a powerful tool for evaluating software correctness. The process of PBT starts with a developer deciding on a formal specification that they want their code to satisfy and encoding that specification as an executable *property*. An automated test harness checks the property against their code using hundreds or thousands of random inputs, produced by a *generator*. If this process discovers a counterexample to the property—an input value that causes it to fail—the developer is notified.

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3639581

The research literature is full of accounts of PBT successes, e.g., in telecommunications software [2], replicated file [31] and key-value [8] stores, automotive software [3], and other complex systems [30]. PBT libraries are available in most major programming languages, and some now have significant user communities—e.g., Python's Hypothesis framework [37] had an estimated 500K users in 2021 according to a JetBrains survey [32]. Still, there is plenty of room for growth. Half a million Hypothesis users represent only 4% of the total Python user base, whereas the Hypothesis maintainers estimate [15] that the "addressable market" is at least 25%. (For comparison, the most popular testing framework, pytest, has 50% market share.)

To help move PBT toward wider adoption, the research community (ourselves included) needs to better understand the practical strengths and weaknesses of PBT and the places where further technical advances are required. Existing work in the software engineering literature has studied how other bug-finding tools are used in practice (see §6), but PBT offers a unique set of tools and warrants its own investigation. Accordingly, we interviewed PBT users at Jane Street, a financial technology firm that makes significant use of PBT, to learn how they use PBT, where they see its value, and in what ways they think it might be improved. Concretely, we aimed to answer two main questions:

RQ1: What are the characteristics of a successful and mature PBT culture at a software company?
RQ2: Are there opportunities for future work in the PBT space that are motivated by the needs of real developers?

The first question aims both to offer guidance for engineers and managers considering whether PBT might fit well in their organizations and to provide a basis for evaluating and comparing PBT technologies. The second question aims to help shape further research to maximize the impact of PBT.

Our findings contribute a wide range of observations about developers' experiences with PBT, adding nuance to the research community's understanding of PBT's real-world usage. Through our interviews, we gleaned several new insights about the situations in which property-based tests are deployed in practice. We found that developers use PBT mainly for testing components of complex systems, expecting the tests to provide greater confidence than conventional example-based unit tests yet still run quickly as part of their normal test suite. Interestingly, we also found that developers leverage PBT for the secondary benefit of communicating

Problem:

Controlling test distributions is critical but difficult.

Problem:

Controlling test distributions is critical but difficult.

Goal:

Automatically tune generators for desirable distributions.

Problem:

Controlling test distributions is critical but difficult.

Goal:

Automatically tune generators for desirable distributions.

Insight:

View generators as probabilistic programs.

Generators are probabilistic programs

```
let genList n =
  match n with
  | 0 -> []
  | S n' ->
    freq [
      1,  [];
      1,  let x = flip 0.5 in
          let xs = genList n' in
          x :: xs
    ]
```

Generators are probabilistic programs

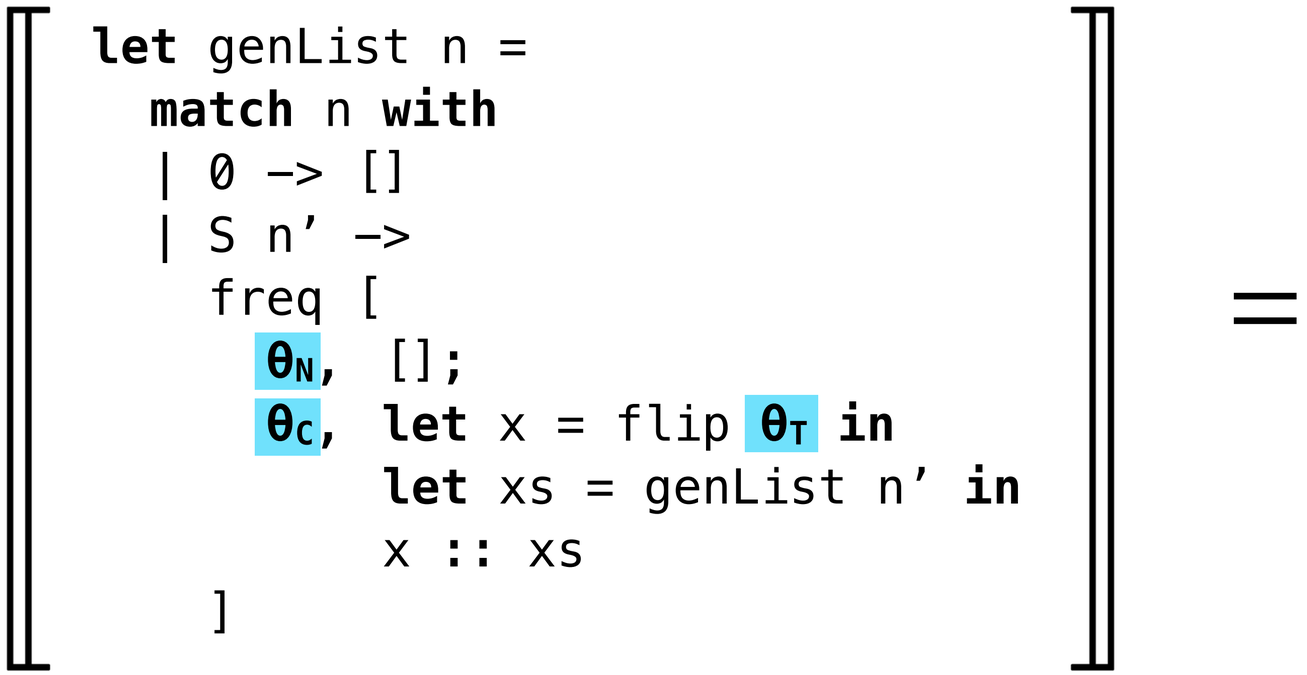
⟦
```
let genList n =
  match n with
  | 0 -> []
  | S n' ->
    freq [
      1,  [];
      1,  let x = flip 0.5 in
          let xs = genList n' in
          x :: xs
    ]
```
⟧

Generators are probabilistic programs

⟦
```
let genList n =
  match n with
  | 0 -> []
  | S n' ->
    freq [
      1,  [];
      1,  let x = flip 0.5 in
          let xs = genList n' in
          x :: xs
    ]
```
⟧ =

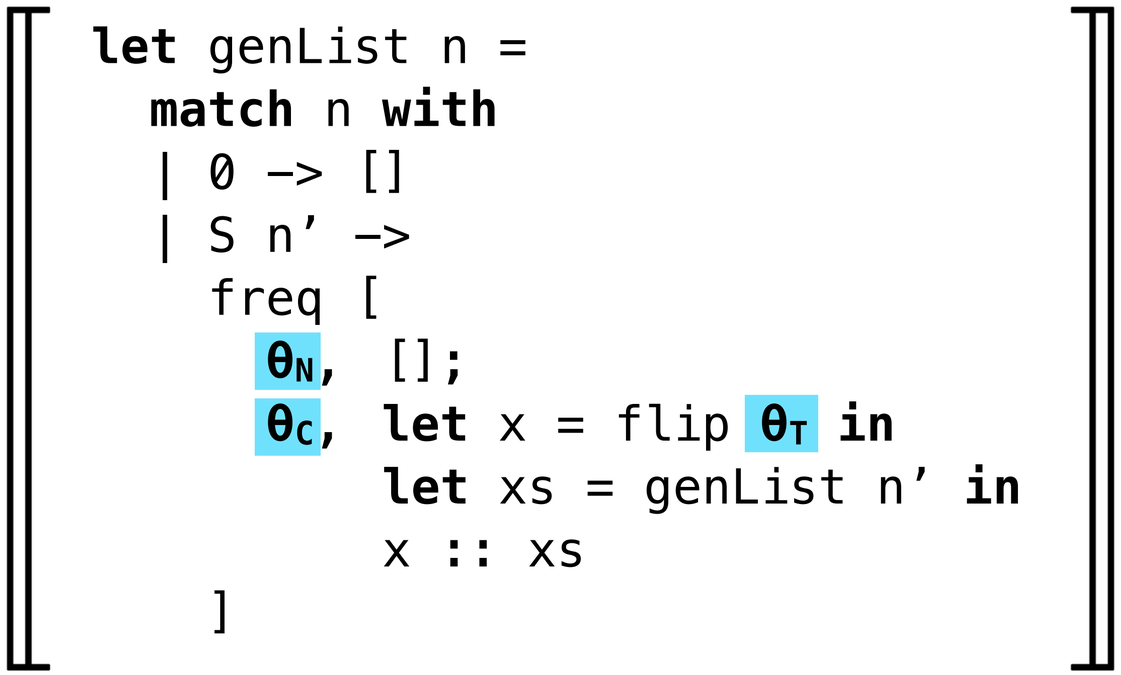
```
{
 [] ↦ 0.5
 [F] ↦ 0.125
 [T] ↦ 0.125
 [F;F] ↦ 0.03125
 ...
}
```

Generators are probabilistic programs



=

Generators are probabilistic programs



```
let genList n =
  match n with
  | 0 -> []
  | S n' ->
    freq [
      θN, [];
      θC, let x = flip θT in
          let xs = genList n' in
          x :: xs
    ]
```

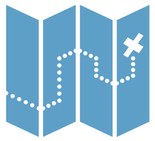
$$=$$

$$
\begin{aligned}
&\{\\
&\ [] \mapsto \theta_N/(\theta_N+\theta_C)\\
&\ [F] \mapsto (1-\theta_T)\theta_N\theta_C/(\theta_N+\theta_C)^2\\
&\ [T] \mapsto \theta_T\theta_N\theta_C/(\theta_N+\theta_C)^2\\
&\ [F;F] \mapsto (1-\theta_T)^2\theta_N\theta_C^2/(\theta_N+\theta_C)^3\\
&\ \ldots\\
&\}
\end{aligned}
$$

Generators are probabilistic programs

⟦
```
let genList n =
  match n with
  | 0 -> []
  | S n' ->
    freq [
      θN, [];
      θC, let x = flip θT in
          let xs = genList n' in
          x :: xs
    ]
```
⟧ =

$$
\begin{aligned}
\{ \\
&[] \mapsto \theta_N/(\theta_N+\theta_C) \\
&[F] \mapsto (1-\theta_T)\theta_N\theta_C/(\theta_N+\theta_C)^2 \\
&[T] \mapsto \theta_T\theta_N\theta_C/(\theta_N+\theta_C)^2 \\
&[F;F] \mapsto (1-\theta_T)^2\theta_N\theta_C^2/(\theta_N+\theta_C)^3 \\
&\ldots \\
\}
\end{aligned}
$$

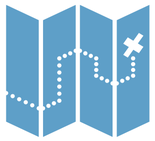


“Maximize `Pr([F;F])`”

Generators are probabilistic programs

⟦
```
let genList n =
  match n with
  | 0 -> []
  | S n' ->
    freq [
      θN, [];
      θC, let x = flip θT in
          let xs = genList n' in
          x :: xs
    ]
```
⟧ =

$$
\begin{aligned}
\{ \\
&[] \mapsto \theta_N/(\theta_N+\theta_C) \\
&[F] \mapsto (1-\theta_T)\theta_N\theta_C/(\theta_N+\theta_C)^2 \\
&[T] \mapsto \theta_T\theta_N\theta_C/(\theta_N+\theta_C)^2 \\
&[F;F] \mapsto (1-\theta_T)^2\theta_N\theta_C^2/(\theta_N+\theta_C)^3 \\
&\ldots \\
\}
\end{aligned}
$$



“Maximize `Pr([F;F])`”

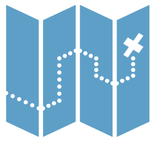
Generators are probabilistic programs

$$\llbracket$$

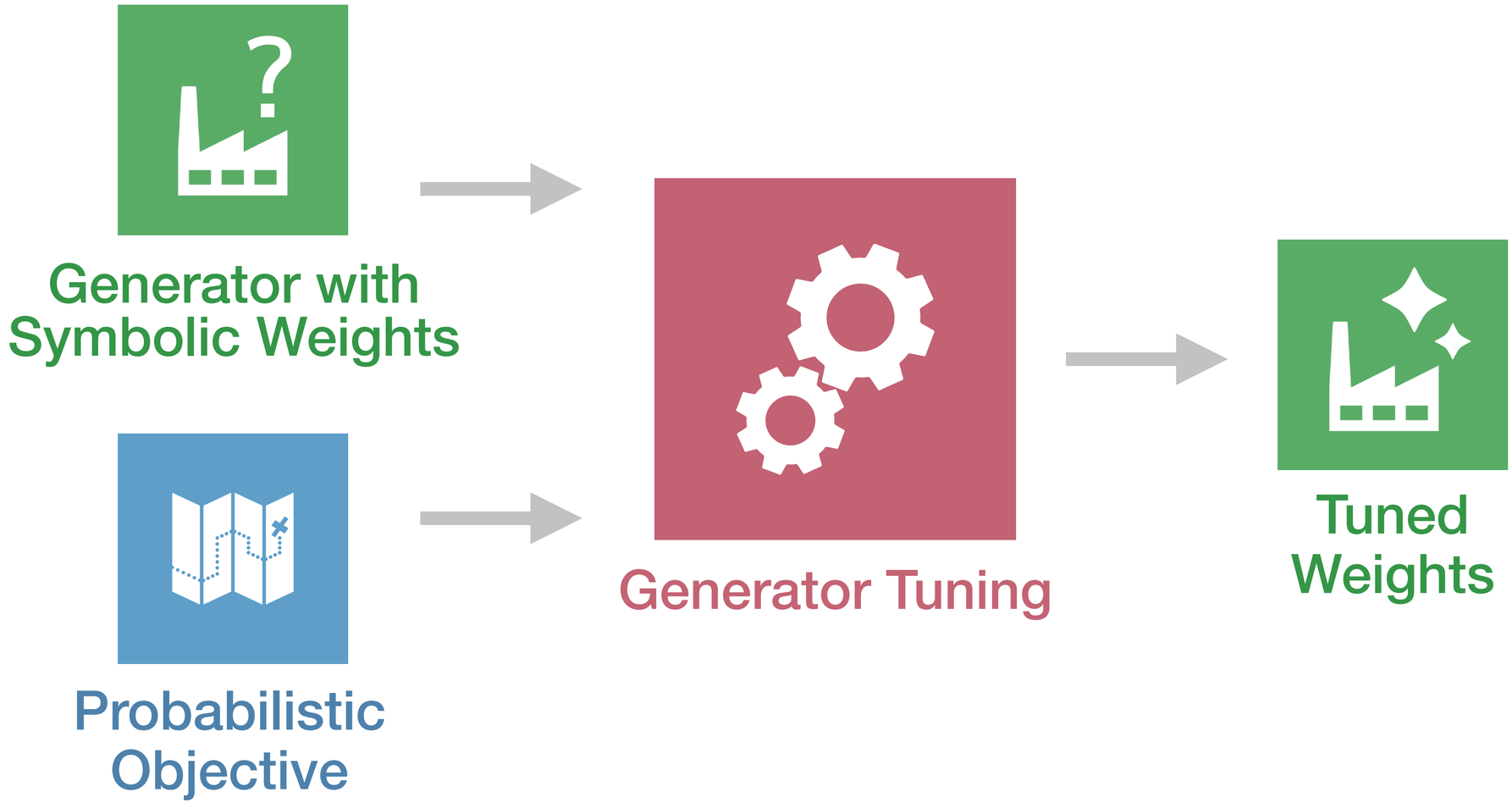
```
let genList n =
  match n with
  | 0 -> []
  | S n' ->
    freq [
      θN, [];
      θC, let x = flip θT in
          let xs = genList n' in
          x :: xs
    ]
```

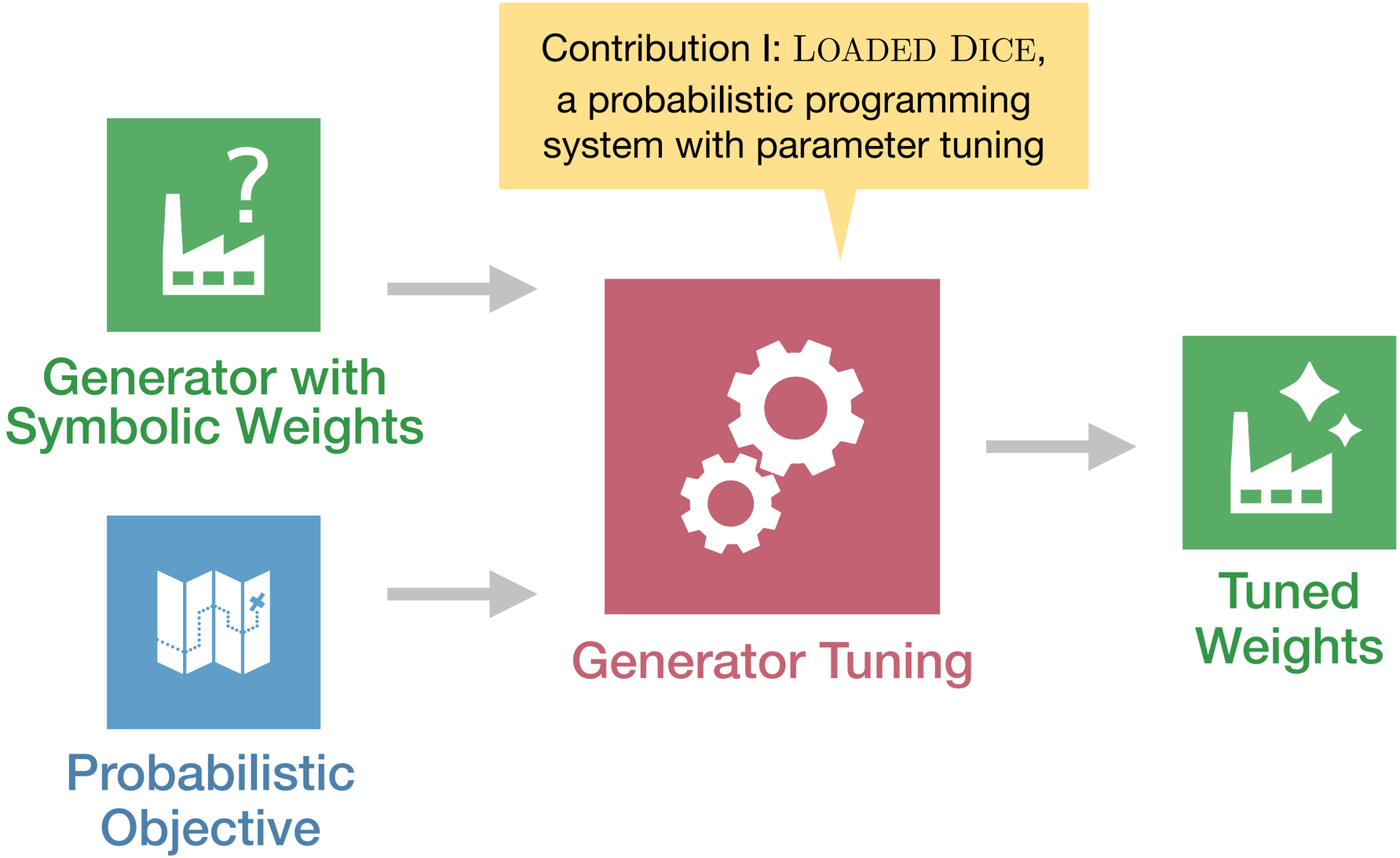
$$\rrbracket =$$

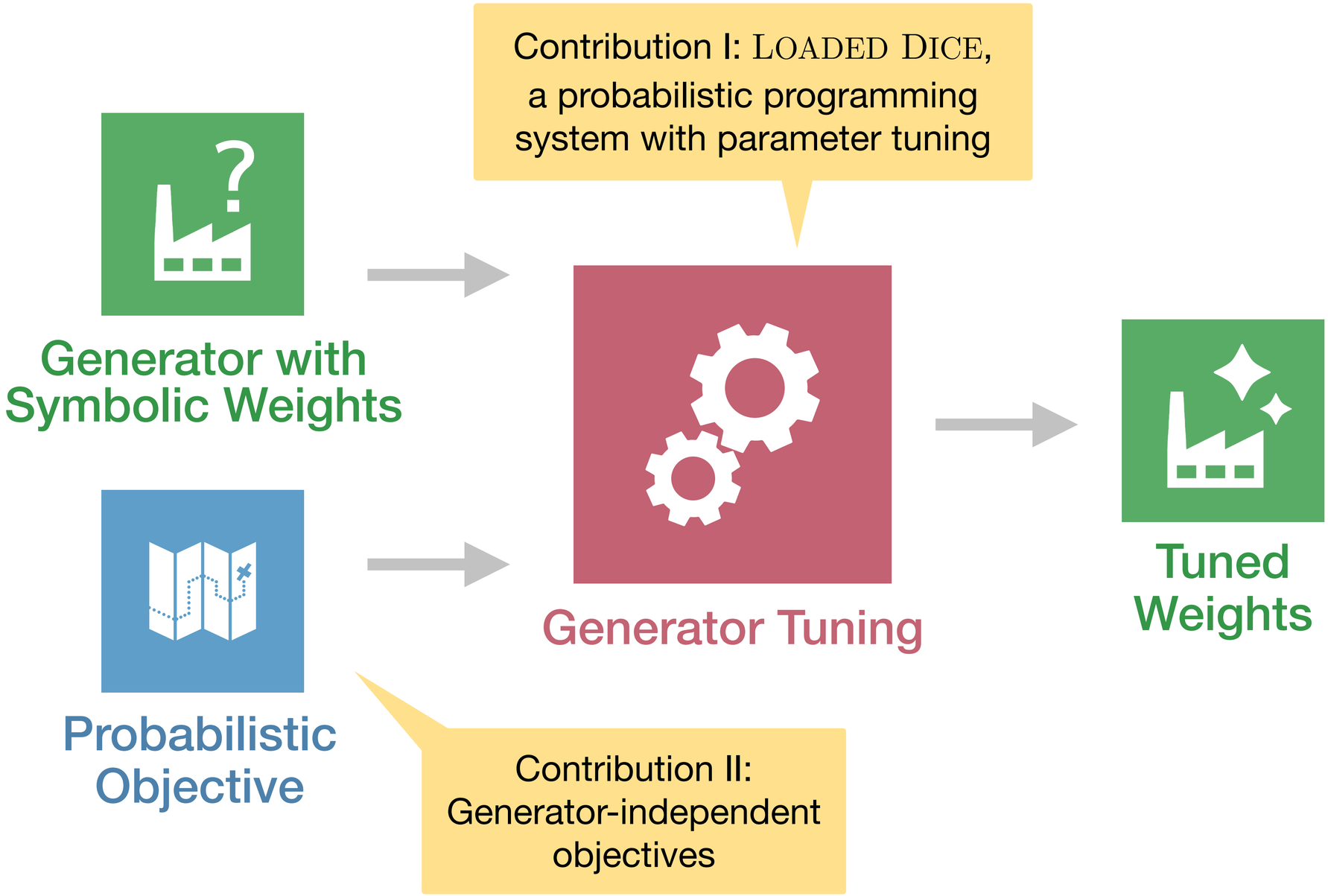
$$\{ [] \mapsto \theta_N/(\theta_N+\theta_C) \quad [F] \mapsto (1-\theta_T)\theta_N\theta_C/(\theta_N+\theta_C)^2 \quad [T] \mapsto \theta_T\theta_N\theta_C/(\theta_N+\theta_C)^2 \quad [F;F] \mapsto (1-\theta_T)^2\theta_N\theta_C^2/(\theta_N+\theta_C)^3 \quad \ldots \}$$

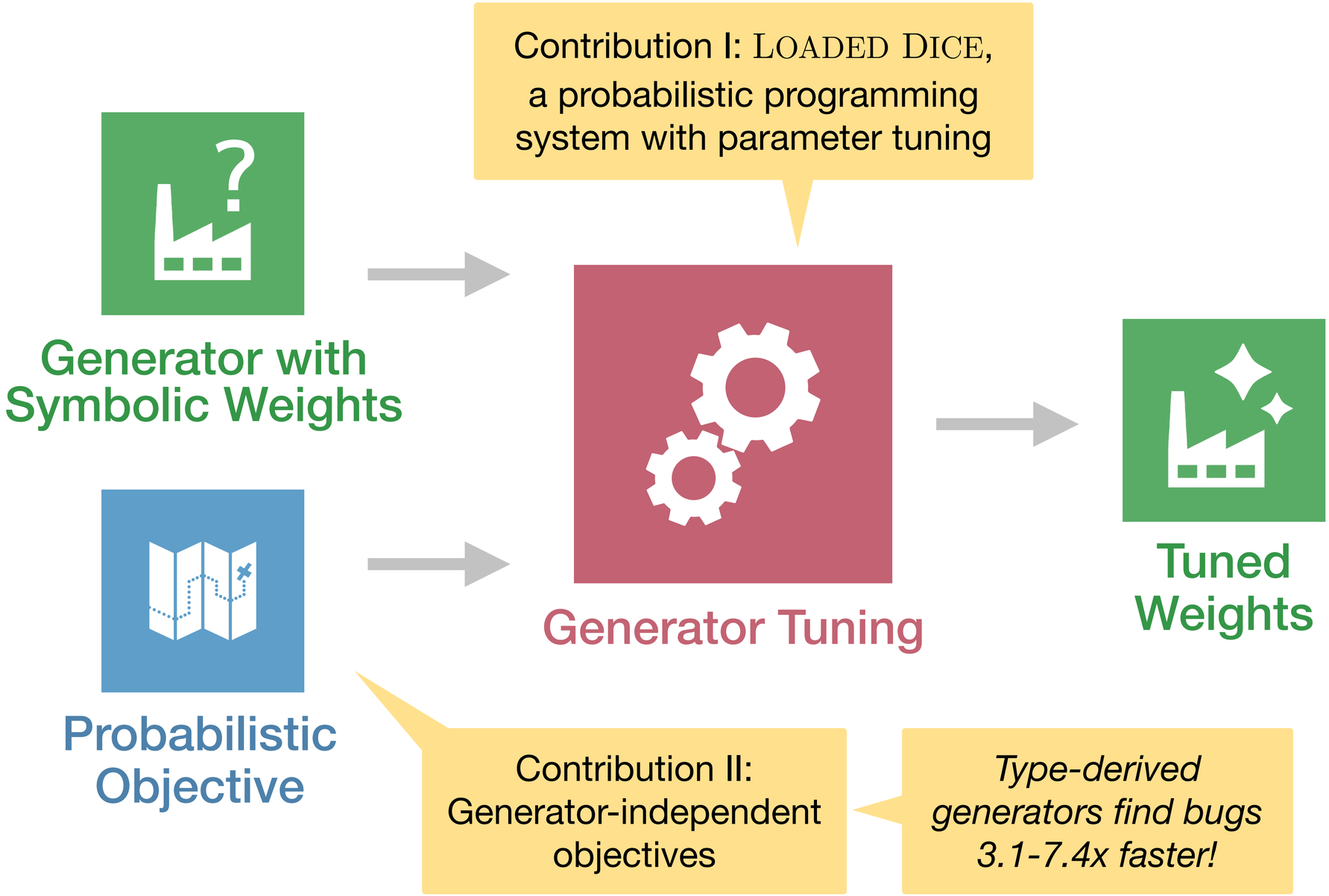


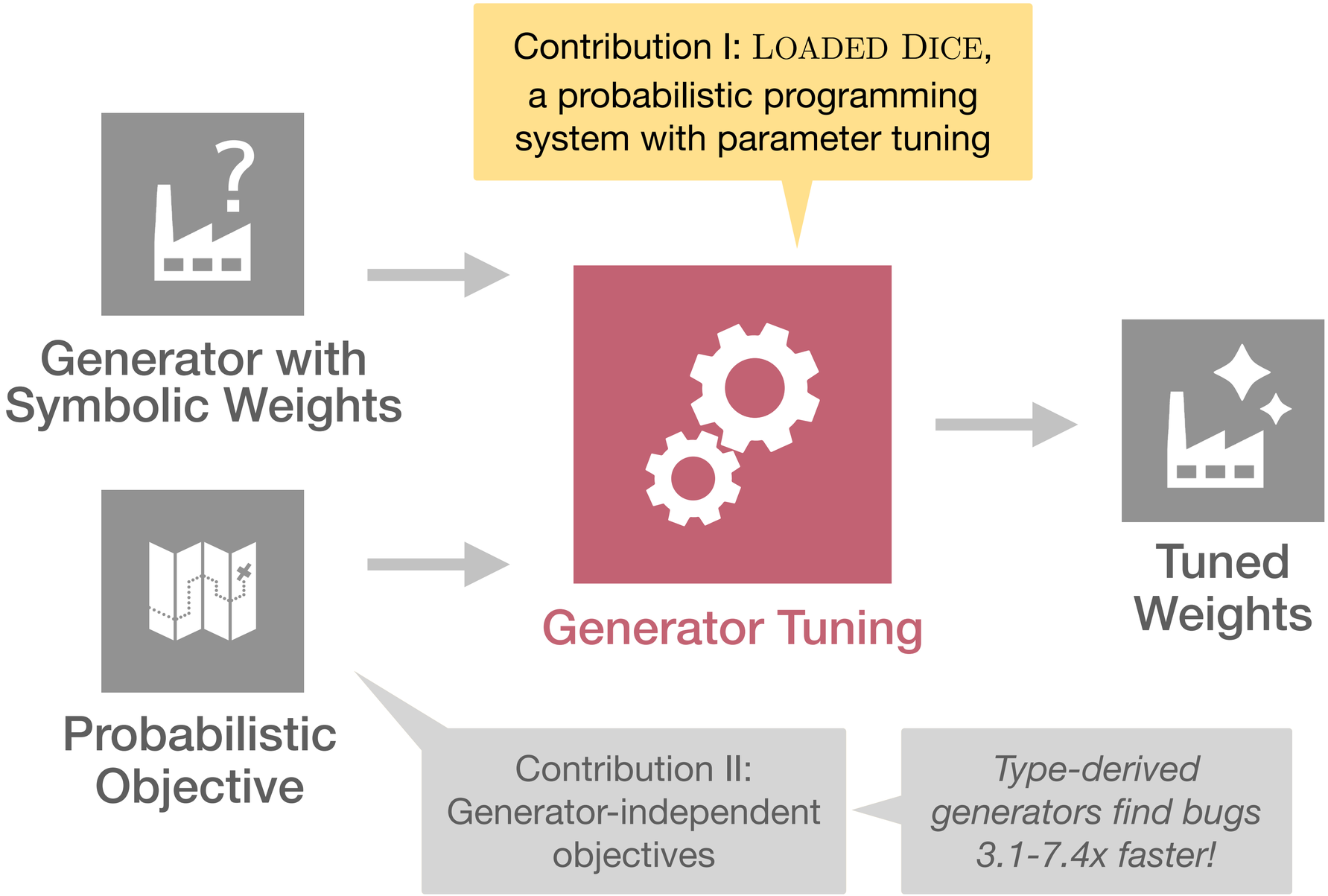
“Maximize Pr([F;F])” → $\theta_N = 1, \theta_C = 2, \theta_T = 0$


Generator with
Symbolic Weights
Probabilistic
Objective
Generator Tuning
Tuned
Weights


Contribution I: LOADED DICE,
a probabilistic programming
system with parameter tuning
Generator with
Symbolic Weights
Probabilistic
Objective
Generator Tuning
Tuned
Weights

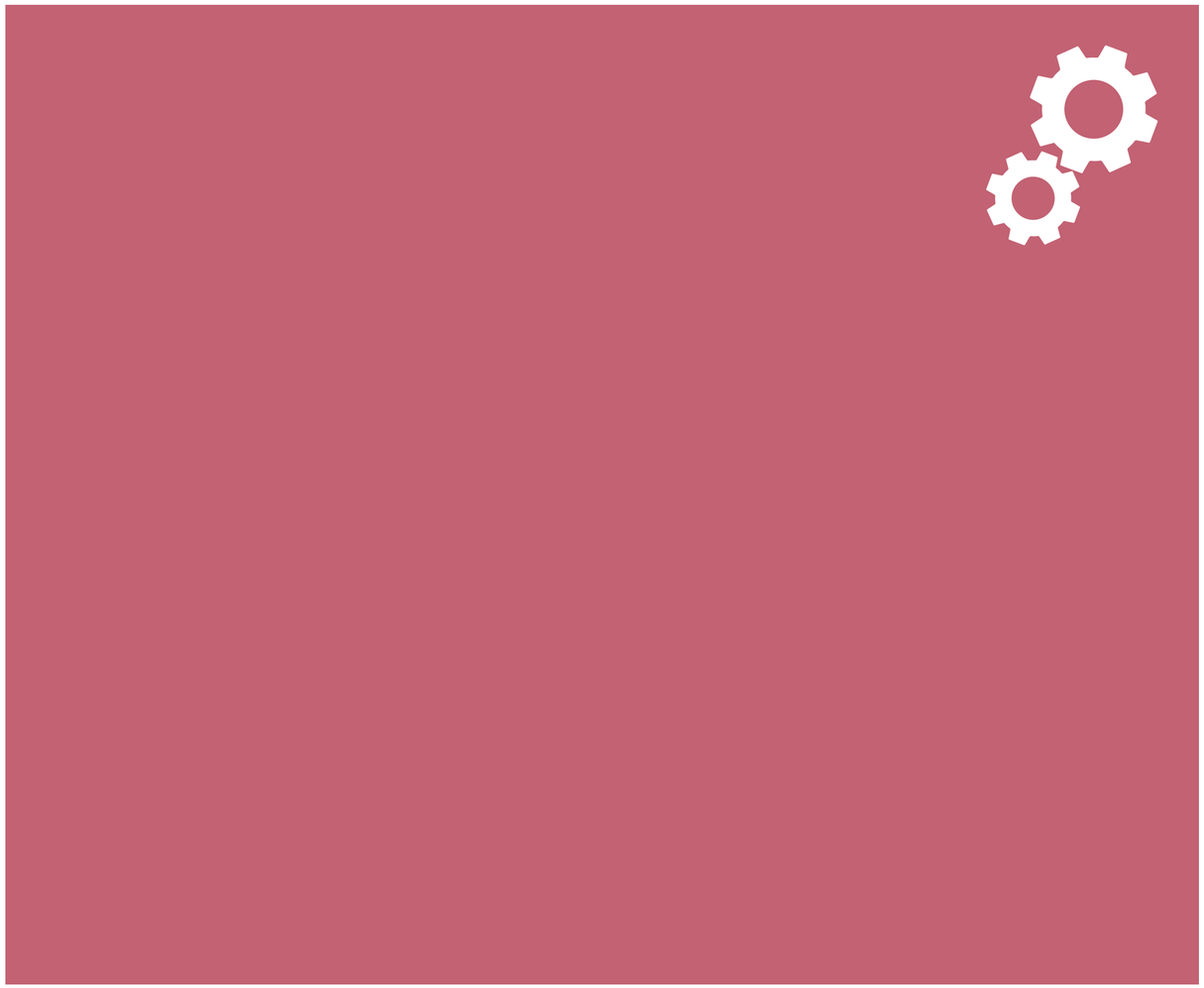

Contribution I: LOADED DICE,
a probabilistic programming
system with parameter tuning
Generator with
Symbolic Weights
Probabilistic
Objective
Generator Tuning
Tuned
Weights
Contribution II:
Generator-independent
objectives


Contribution I: LOADED DICE,
a probabilistic programming
system with parameter tuning
Generator with
Symbolic Weights
Probabilistic
Objective
Generator Tuning
Tuned
Weights
Contribution II:
Generator-independent
objectives
*Type-derived
generators find bugs
3.1-7.4x faster!*


Contribution I: LOADED DICE,
a probabilistic programming
system with parameter tuning
Generator with
Symbolic Weights
Probabilistic
Objective
Generator Tuning
Tuned
Weights
Contribution II:
Generator-independent
objectives
*Type-derived
generators find bugs
3.1-7.4x faster!*

Generator with Symbolic Weights

Probabilistic Objective



Generator Tuning

Tuned Weights

```
if flip(θ) then 2 else 3
```

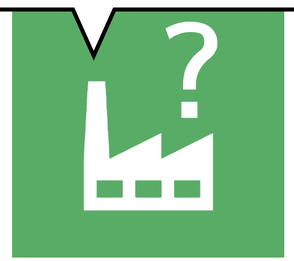
Generator with Symbolic Weights

Generator Tuning

Probabilistic Objective

Tuned Weights

```
if flip(θ) then 2 else 3
```



Generator with Symbolic Weights

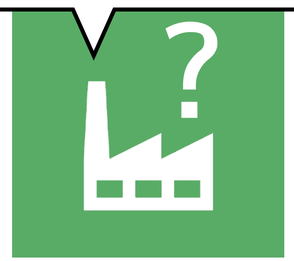
"Target distribution $\{2 \mapsto 1.0\}$"

Probabilistic Objective

Generator Tuning

Tuned Weights

```
if flip(θ) then 2 else 3
```



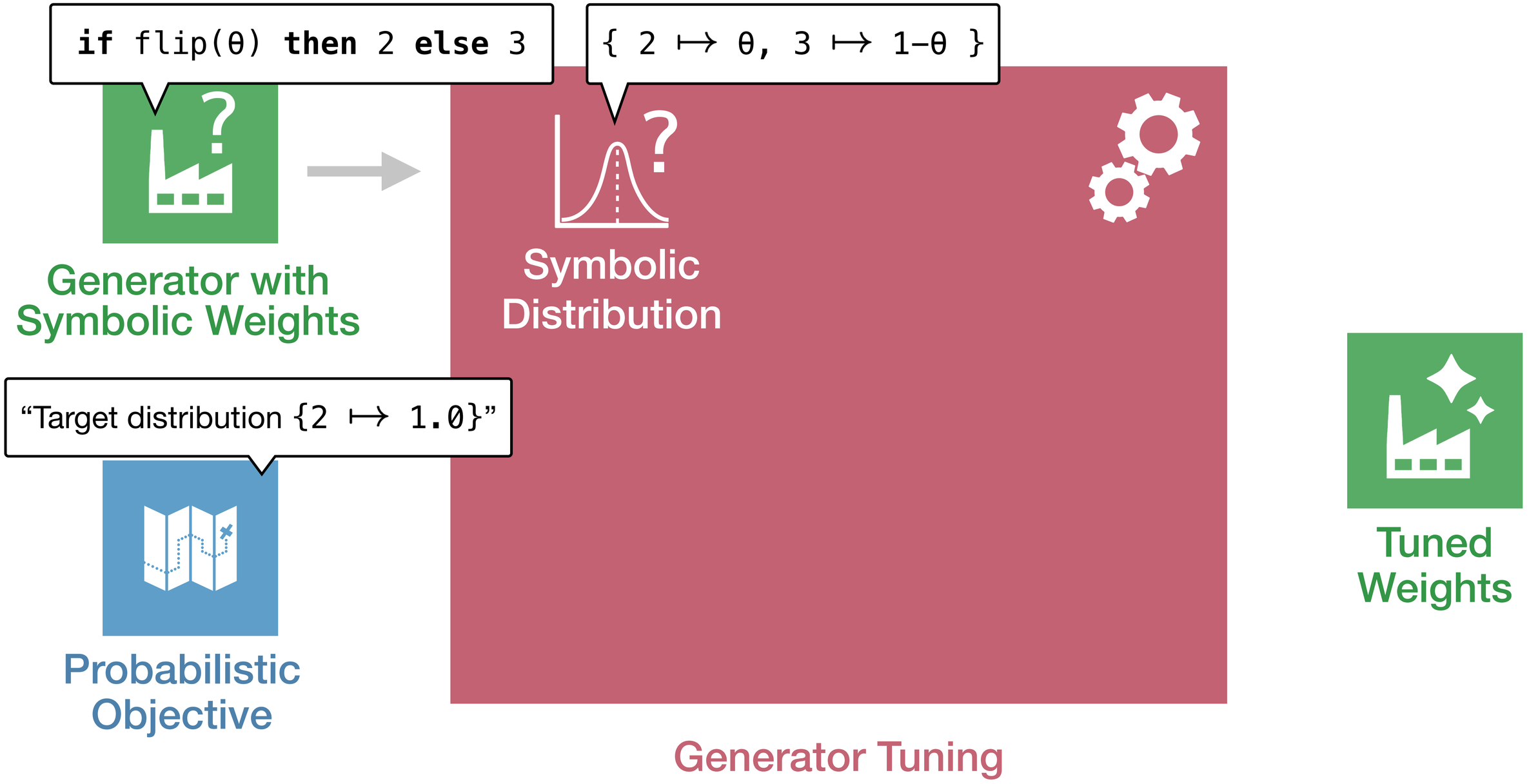
Generator with Symbolic Weights

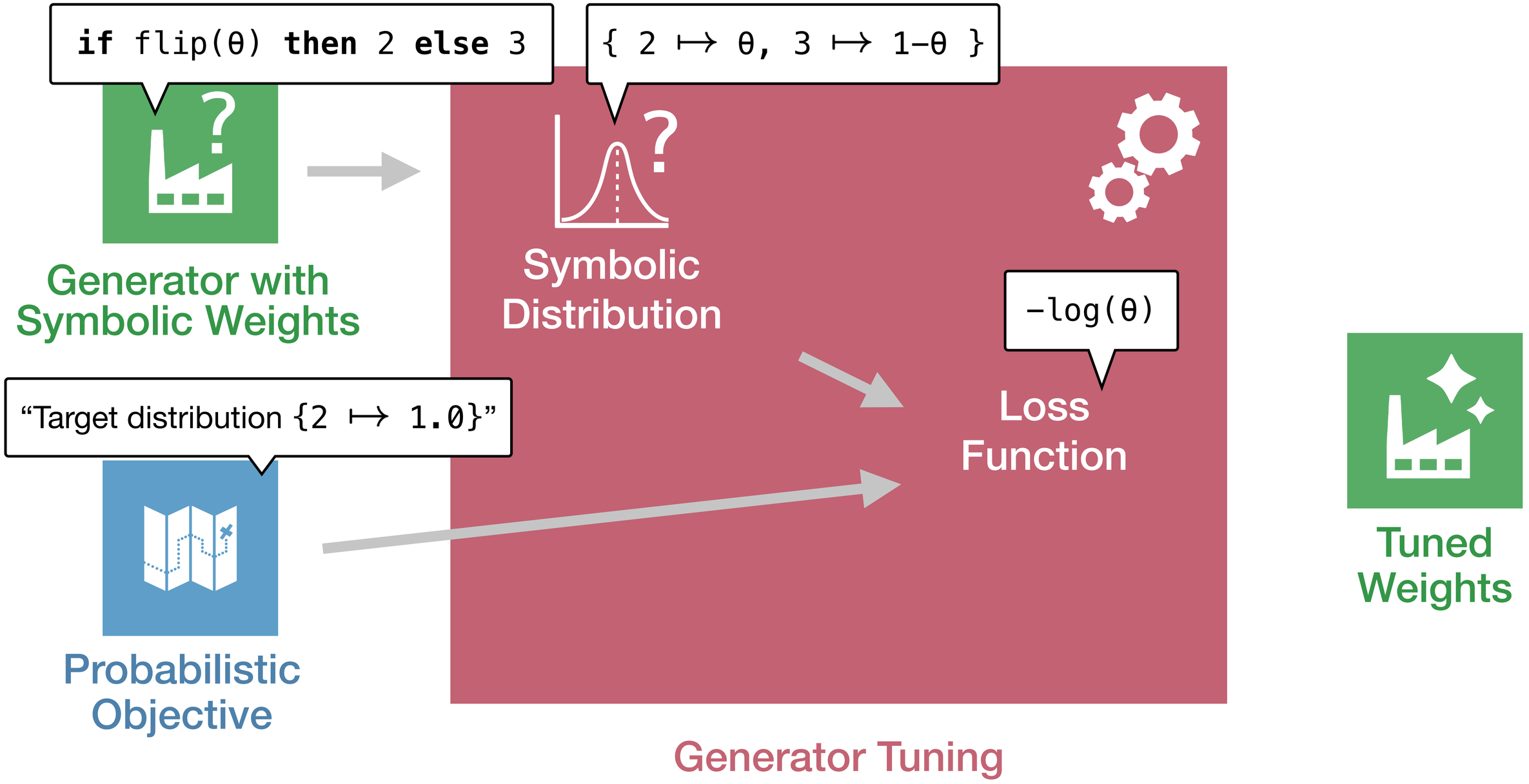
"Target distribution $\{2 \mapsto 1.0\}$"

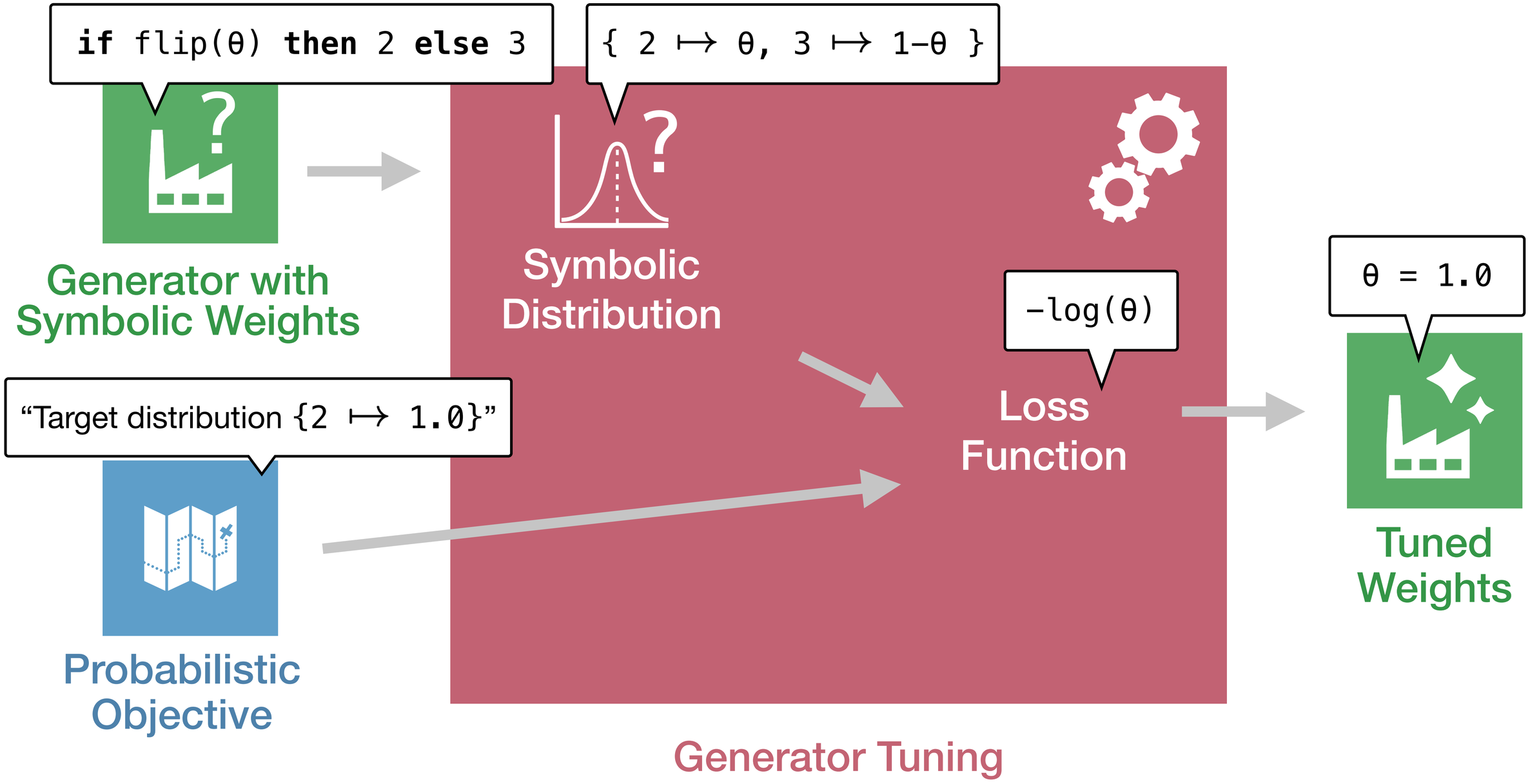
Probabilistic Objective

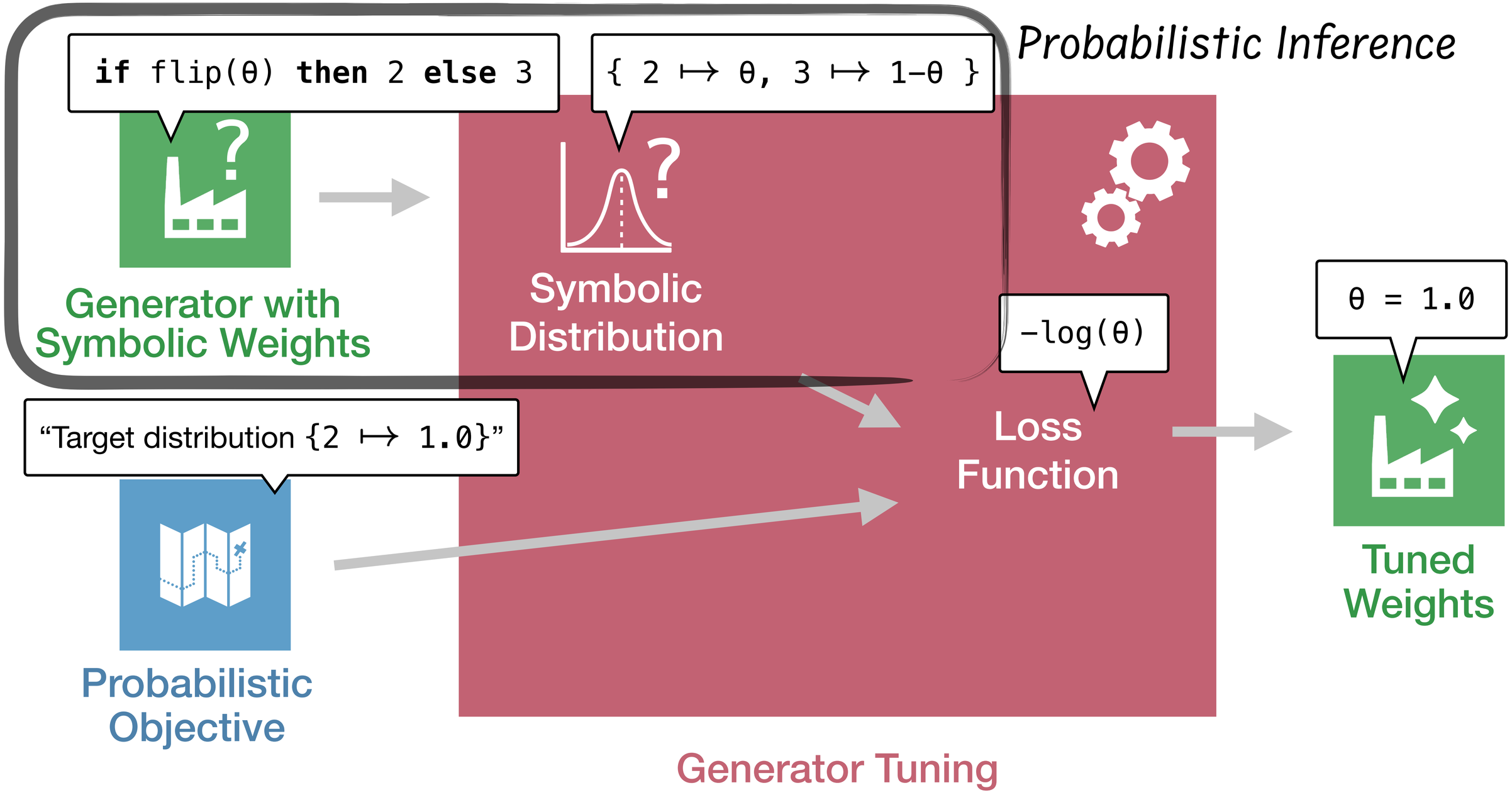
Generator Tuning

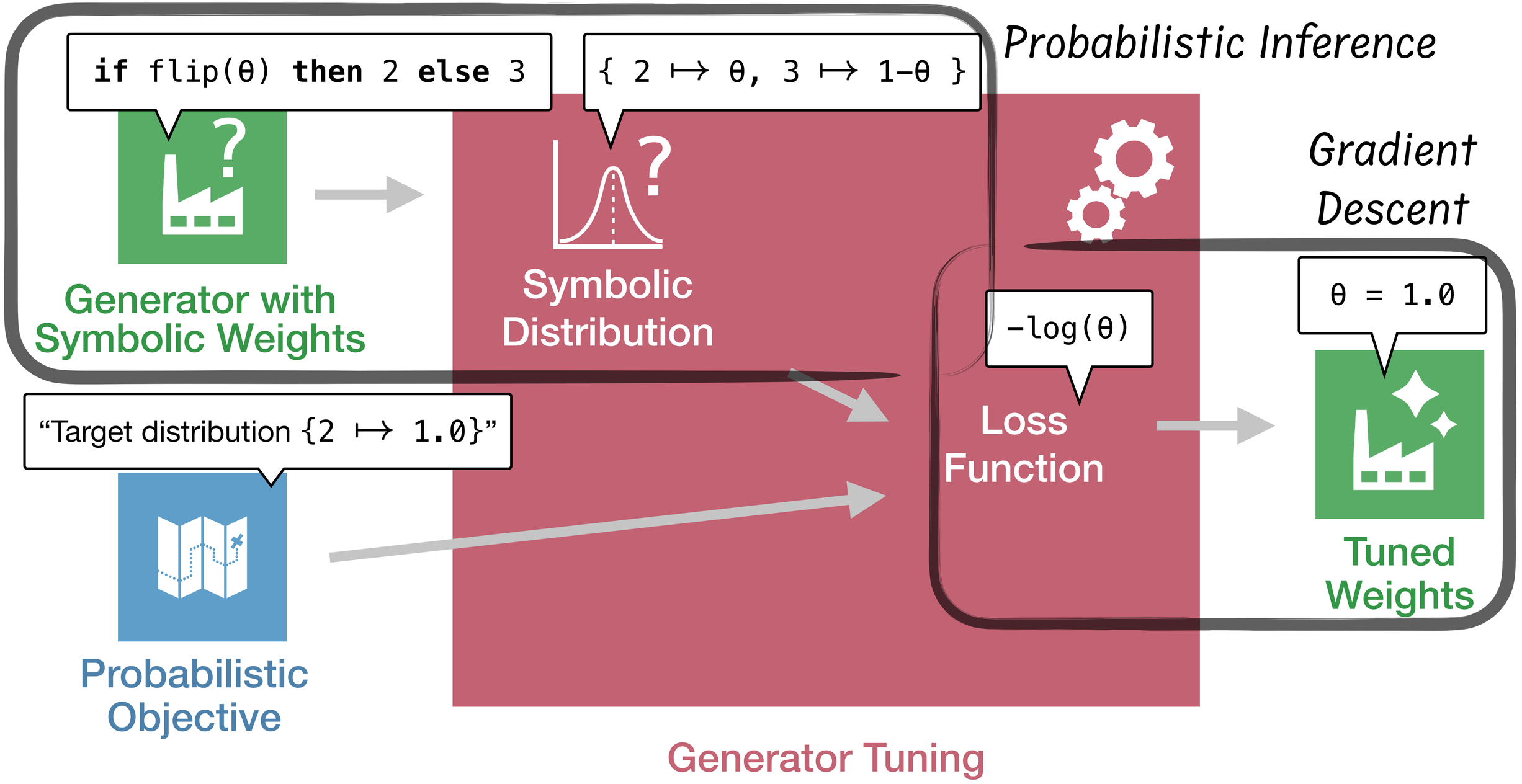
Tuned Weights


if flip(θ) then 2 else 3
{ 2 ↦ θ, 3 ↦ 1−θ }
Generator with Symbolic Weights
Symbolic Distribution
“Target distribution {2 ↦ 1.0}”
Probabilistic Objective
Generator Tuning
Tuned Weights


if flip(θ) then 2 else 3
{ 2 ↦ θ, 3 ↦ 1−θ }
Generator with Symbolic Weights
Symbolic Distribution
−log(θ)
Loss Function
“Target distribution {2 ↦ 1.0}”
Probabilistic Objective
Generator Tuning
Tuned Weights


if flip(θ) then 2 else 3
{ 2 ↦ θ, 3 ↦ 1−θ }
Generator with Symbolic Weights
Symbolic Distribution
−log(θ)
Loss Function
θ = 1.0
Tuned Weights
“Target distribution {2 ↦ 1.0}”
Probabilistic Objective
Generator Tuning


Probabilistic Inference
if flip(θ) then 2 else 3
{ 2 ↦ θ, 3 ↦ 1−θ }
Generator with Symbolic Weights
Symbolic Distribution
−log(θ)
θ = 1.0
“Target distribution {2 ↦ 1.0}”
Loss Function
Tuned Weights
Probabilistic Objective
Generator Tuning


if flip(θ) then 2 else 3
{ 2 ↦ θ, 3 ↦ 1−θ }
Probabilistic Inference
Gradient Descent
Generator with Symbolic Weights
Symbolic Distribution
−log(θ)
θ = 1.0
Loss Function
Tuned Weights
“Target distribution {2 ↦ 1.0}”
Probabilistic Objective
Generator Tuning

Scaling generator tuning

Scaling generator tuning

Problem 1: Naïvely computing gradients requires enumerating all execution paths

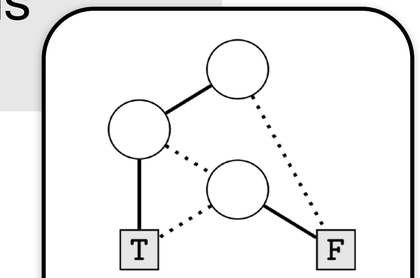
Scaling generator tuning

Problem 1: Naïvely computing gradients requires enumerating all execution paths

Solution: SOTA technique for discrete probabilistic *inference* is differentiable, so it can compute gradients as well

Scaling generator tuning

Problem 1: Naïvely computing gradients requires enumerating all execution paths

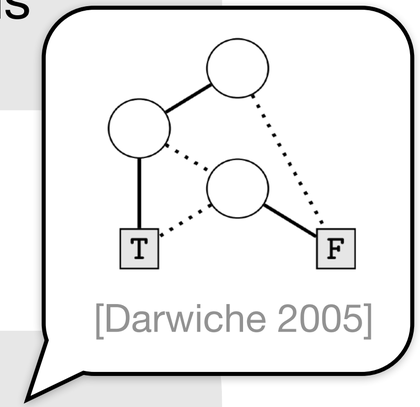


[Darwiche 2005]

Solution: SOTA technique for discrete probabilistic *inference* is differentiable, so it can compute gradients as well

Scaling generator tuning

Problem 1: Naïvely computing gradients requires enumerating all execution paths



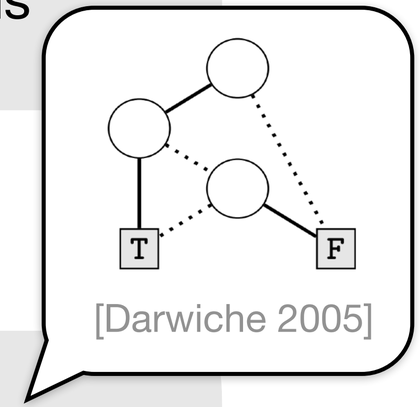
[Darwiche 2005]

Solution: SOTA technique for discrete probabilistic *inference* is differentiable, so it can compute gradients as well

Problem 2: Some objectives (e.g. entropy) enumerate a generator's distribution

Scaling generator tuning

Problem 1: Naïvely computing gradients requires enumerating all execution paths

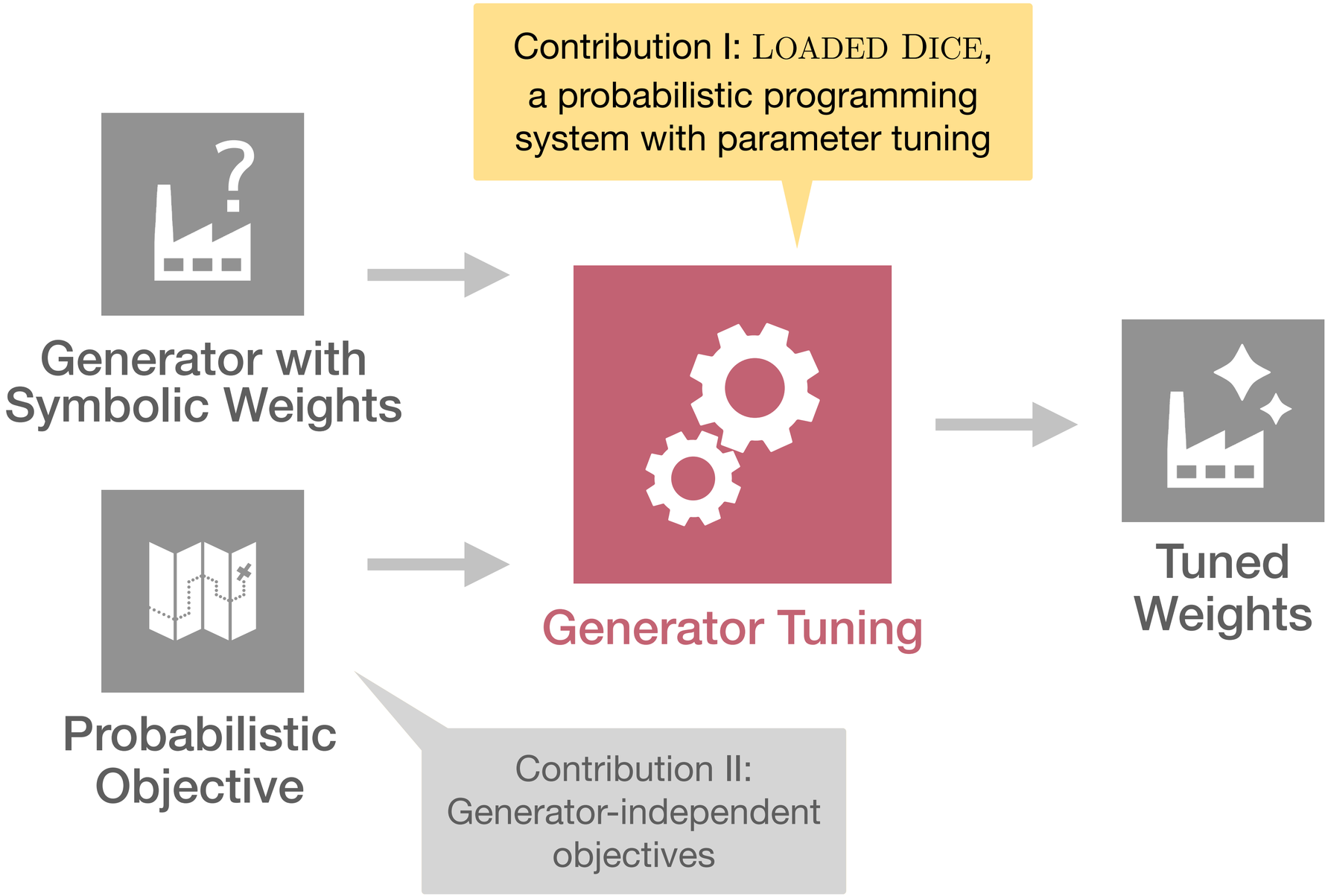


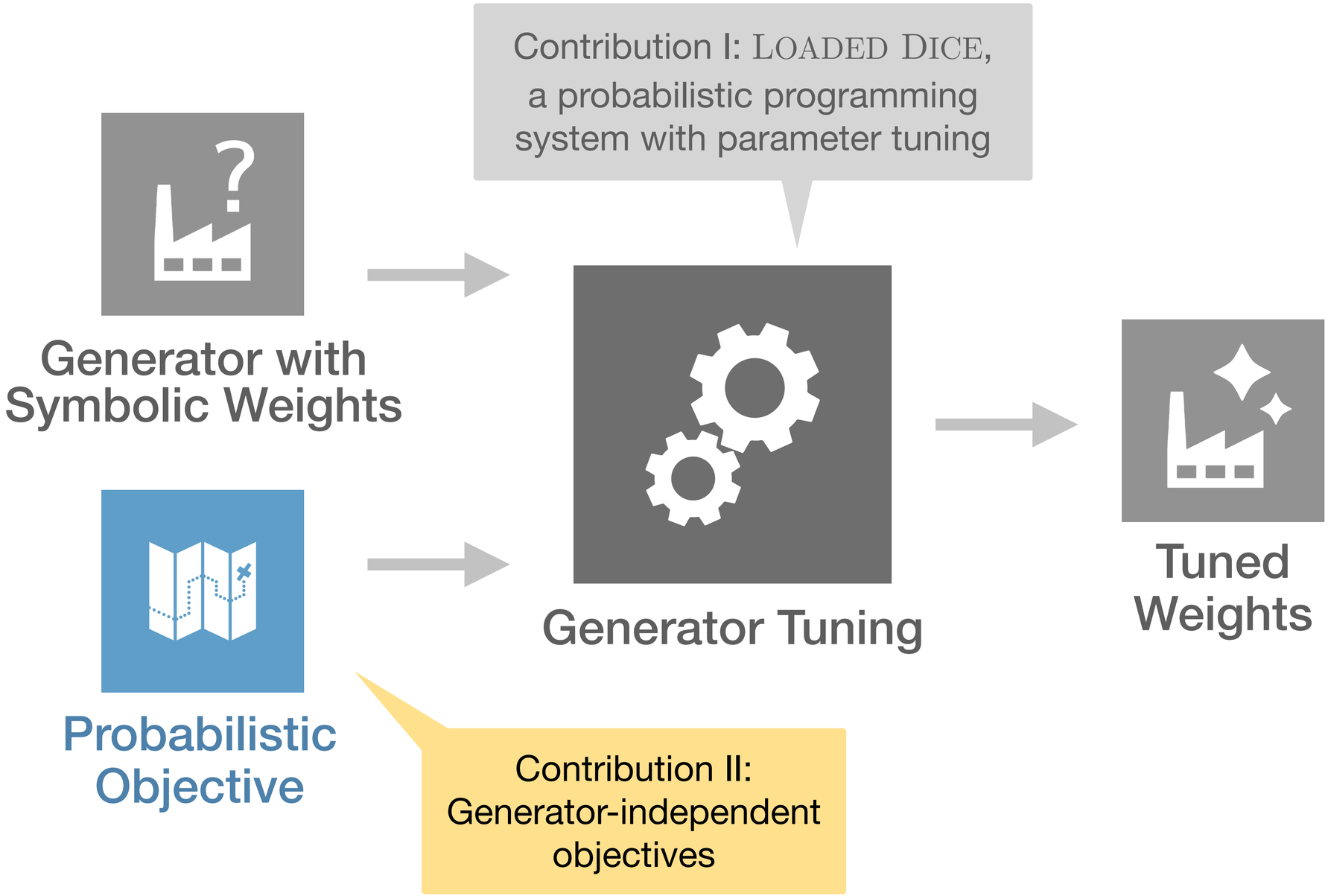
[Darwiche 2005]

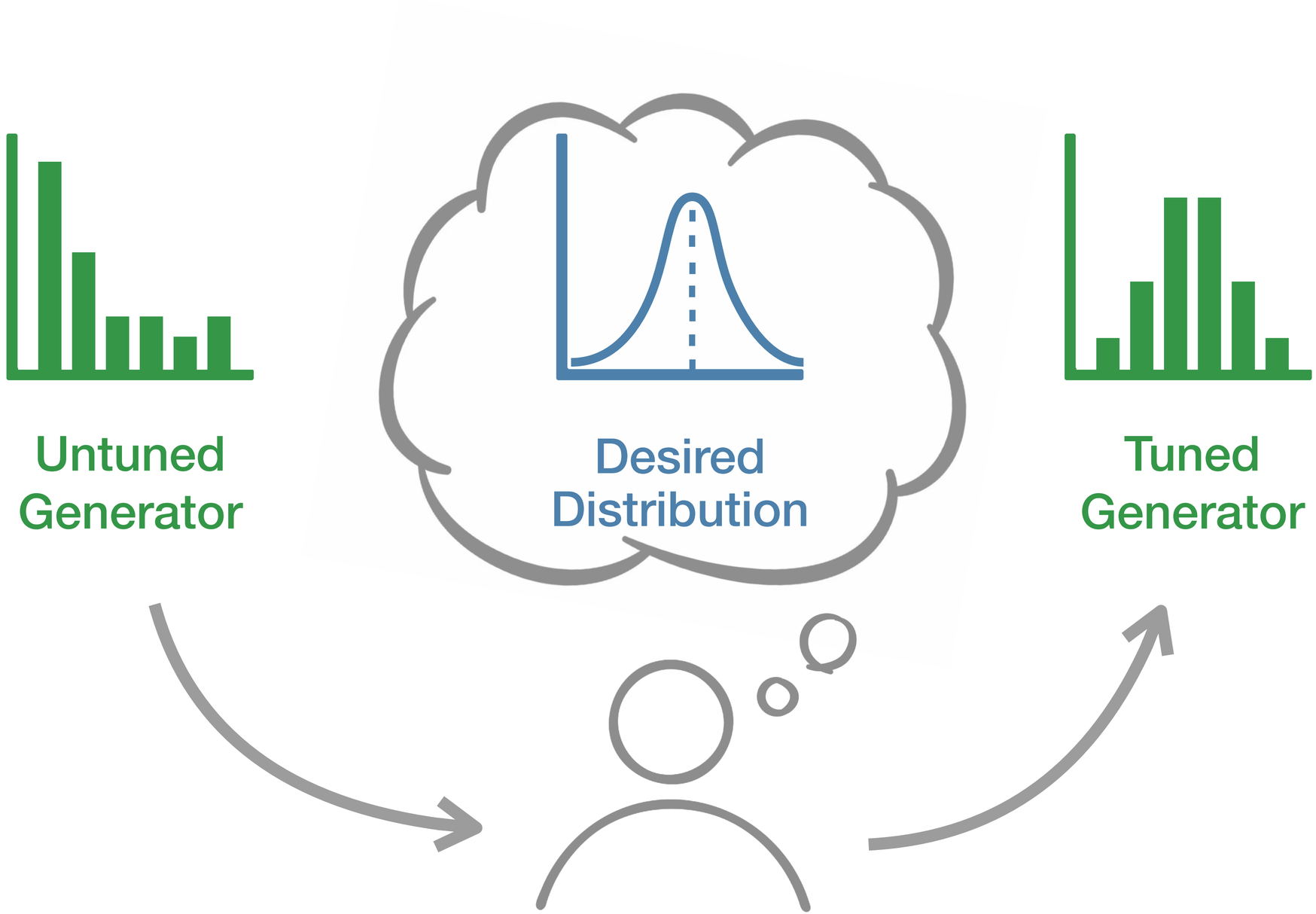
Solution: SOTA technique for discrete probabilistic *inference* is differentiable, so it can compute gradients as well

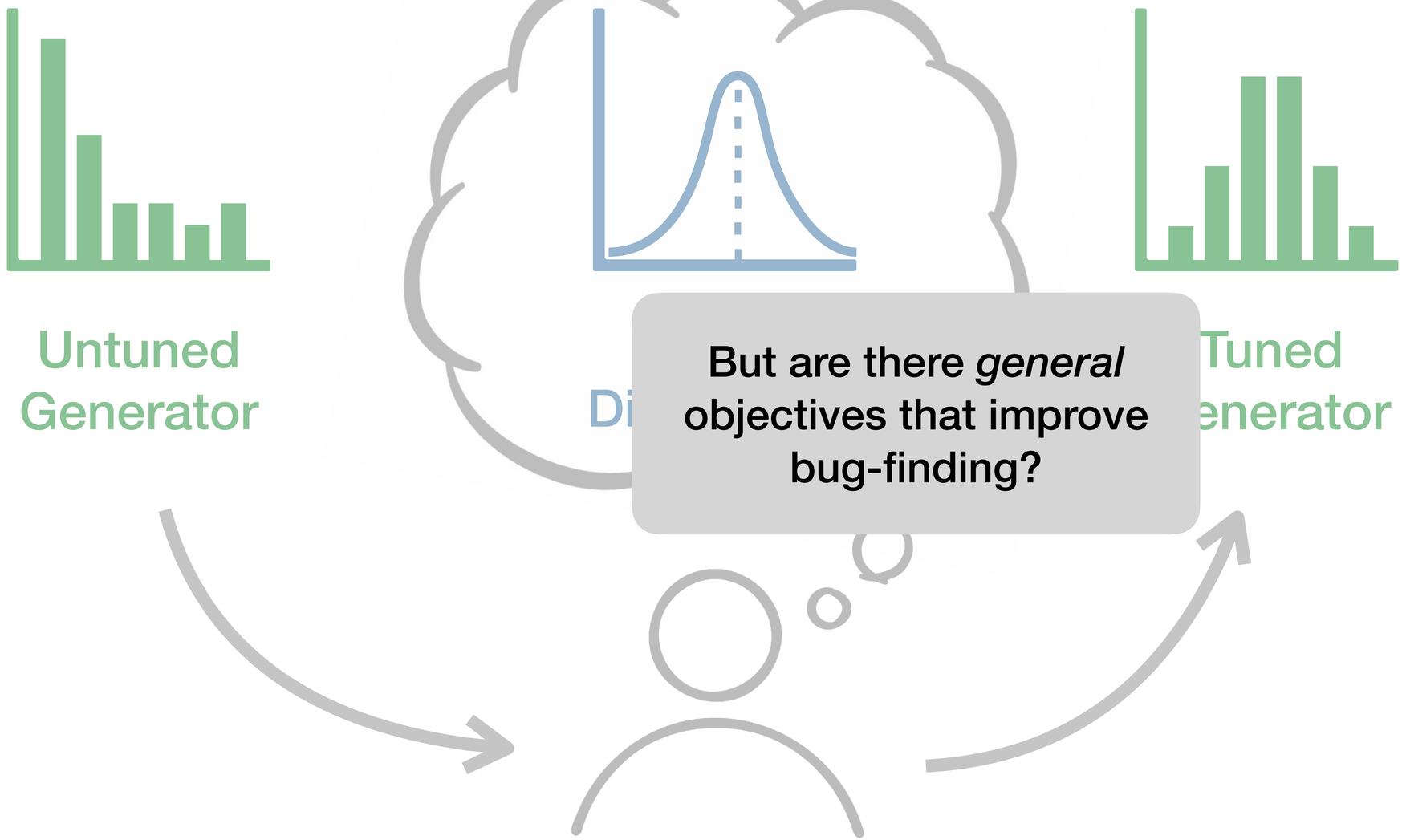
Problem 2: Some objectives (e.g. entropy) enumerate a generator's distribution

Solution: *Approximate* those gradients by sampling (via gradient estimators, similar to REINFORCE)


Contribution I: LOADED DICE,
a probabilistic programming
system with parameter tuning
Generator with
Symbolic Weights
Probabilistic
Objective
Generator Tuning
Tuned
Weights
Contribution II:
Generator-independent
objectives


Contribution I: LOADED DICE,
a probabilistic programming
system with parameter tuning
Generator with
Symbolic Weights
Probabilistic
Objective
Generator Tuning
Tuned
Weights
Contribution II:
Generator-independent
objectives


Untuned
Generator
Desired
Distribution
Tuned
Generator


Untuned
Generator
Di
But are there *general* objectives that improve bug-finding?
Tuned
enerator

Validity and **diversity** are desirable but conflict.

Satisfy property preconditions

Validity and **diversity** are desirable but conflict.

```
∀tree k v. is_rbt(tree) → is_rbt(insert(tree,k,v))
```

Satisfy property preconditions

Validity and **diversity** are desirable but conflict.

```
∀tree k v. is_rbt(tree) → is_rbt(insert(tree,k,v))
```

Satisfy property preconditions

Validity and **diversity** are desirable but conflict.

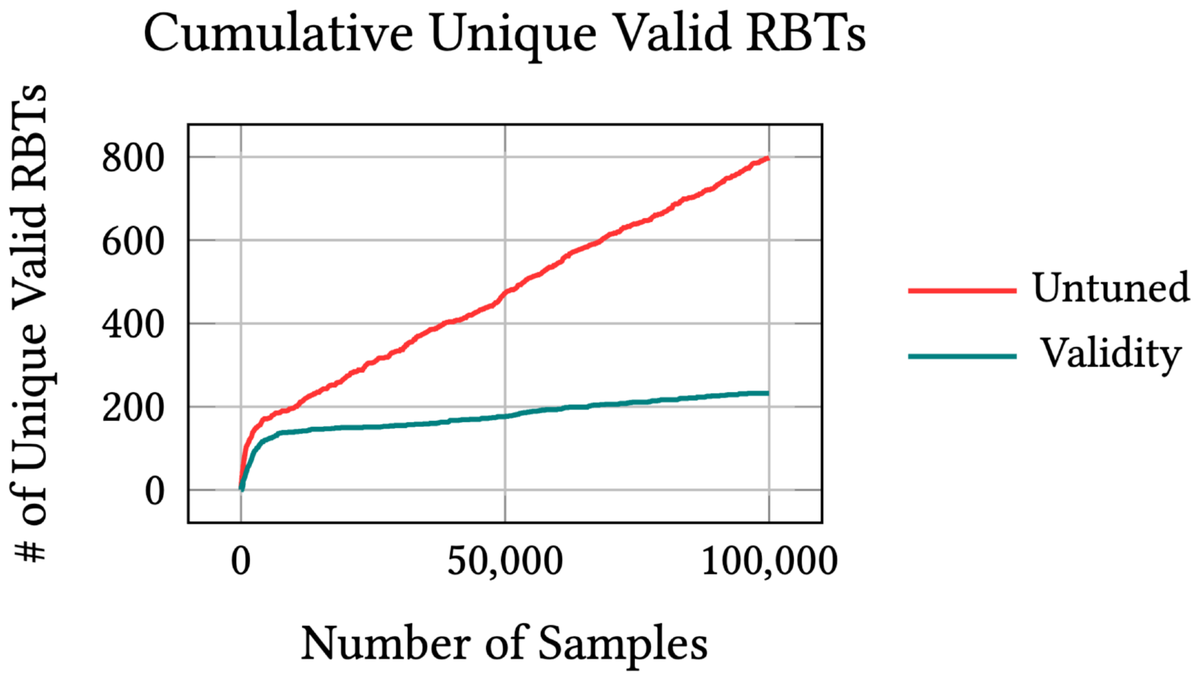
Incentivizes trivial samples!

```
∀tree k v. is_rbt(tree) → is_rbt(insert(tree,k,v))
```

Satisfy property preconditions

Validity and **diversity** are desirable but conflict.

Incentivizes trivial samples!



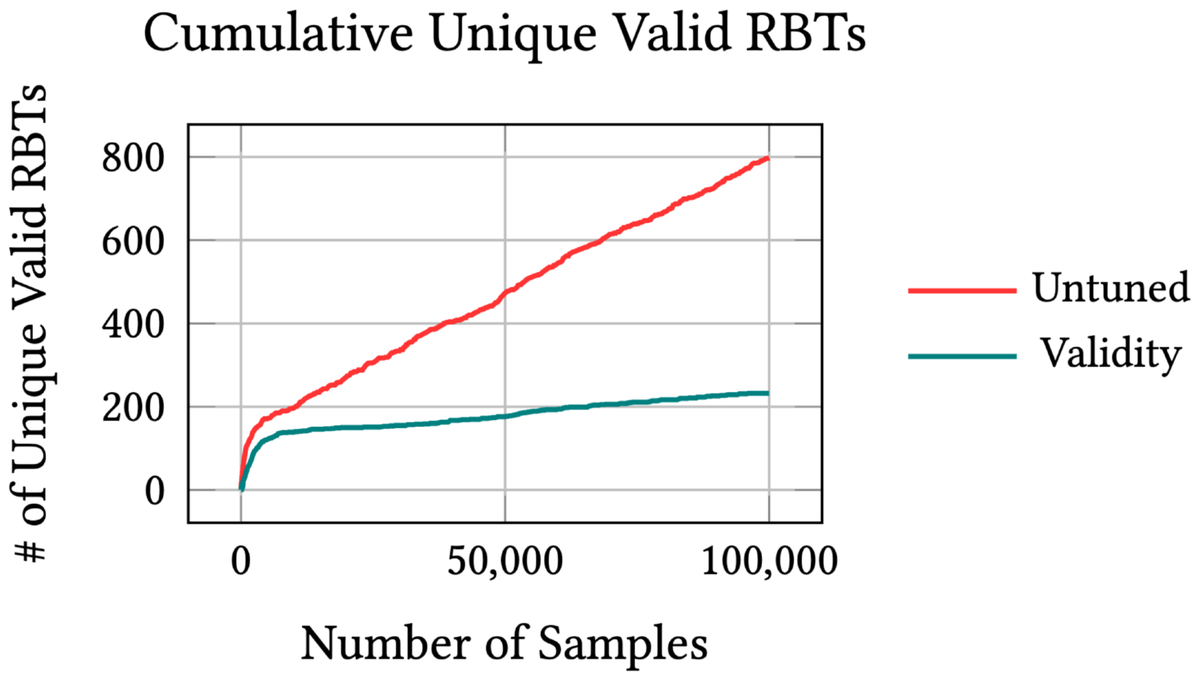
```
∀tree k v. is_rbt(tree) → is_rbt(insert(tree,k,v))
```

Satisfy property preconditions

Exercise more execution paths

Validity and **diversity** are desirable but conflict.

Incentivizes trivial samples!



```
∀tree k v. is_rbt(tree) → is_rbt(insert(tree,k,v))
```

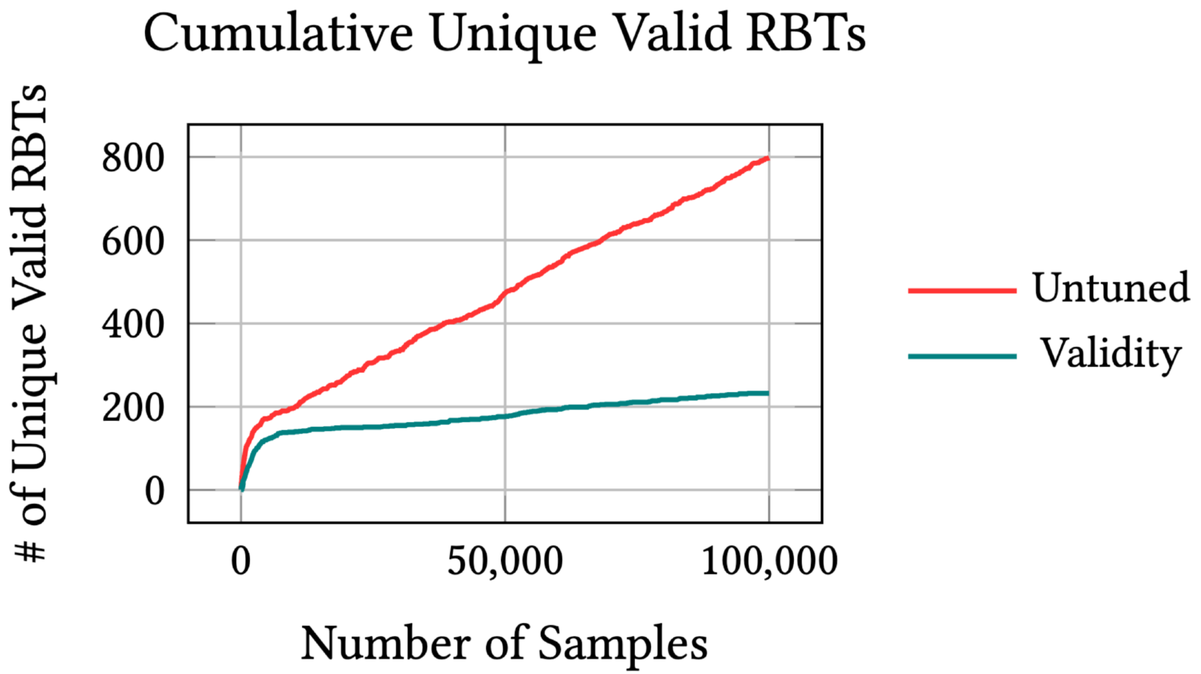
Satisfy property preconditions

Exercise more execution paths

Validity and **diversity** are desirable but conflict.

Incentivizes trivial samples!

Incentivizes large, invalid samples!



```
∀tree k v. is_rbt(tree) → is_rbt(insert(tree,k,v))
```

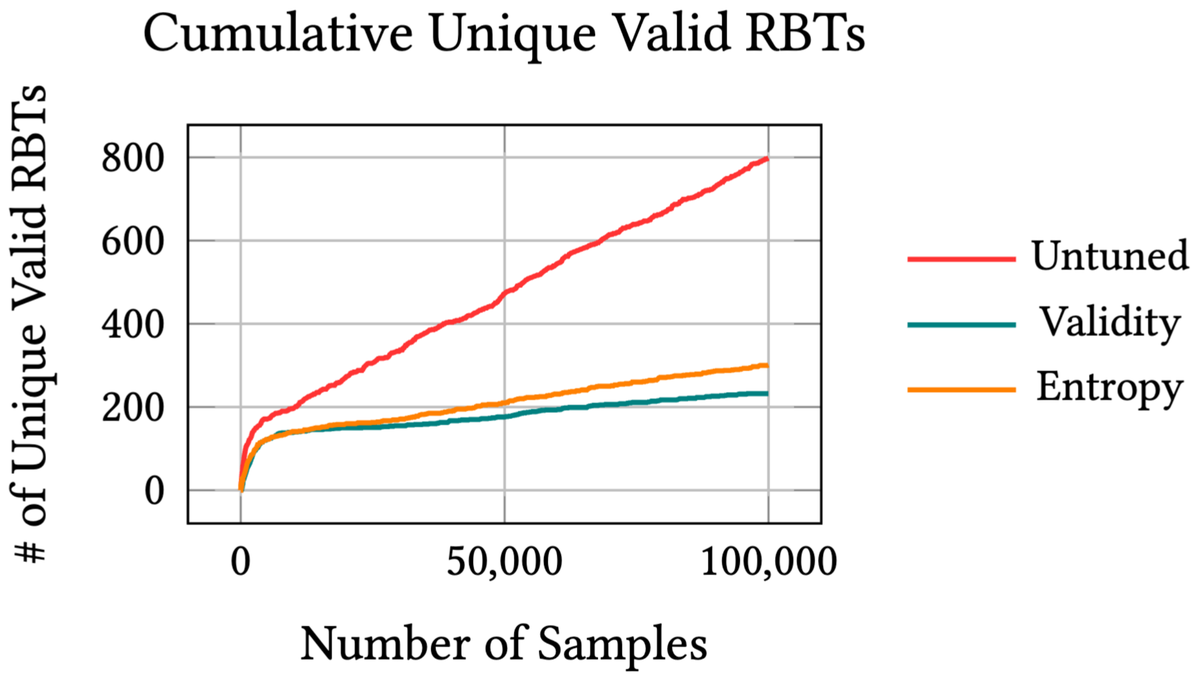
Satisfy property preconditions

Exercise more execution paths

Validity and **diversity** are desirable but conflict.

Incentivizes trivial samples!

Incentivizes large, invalid samples!



```
∀tree k v. is_rbt(tree) → is_rbt(insert(tree,k,v))
```

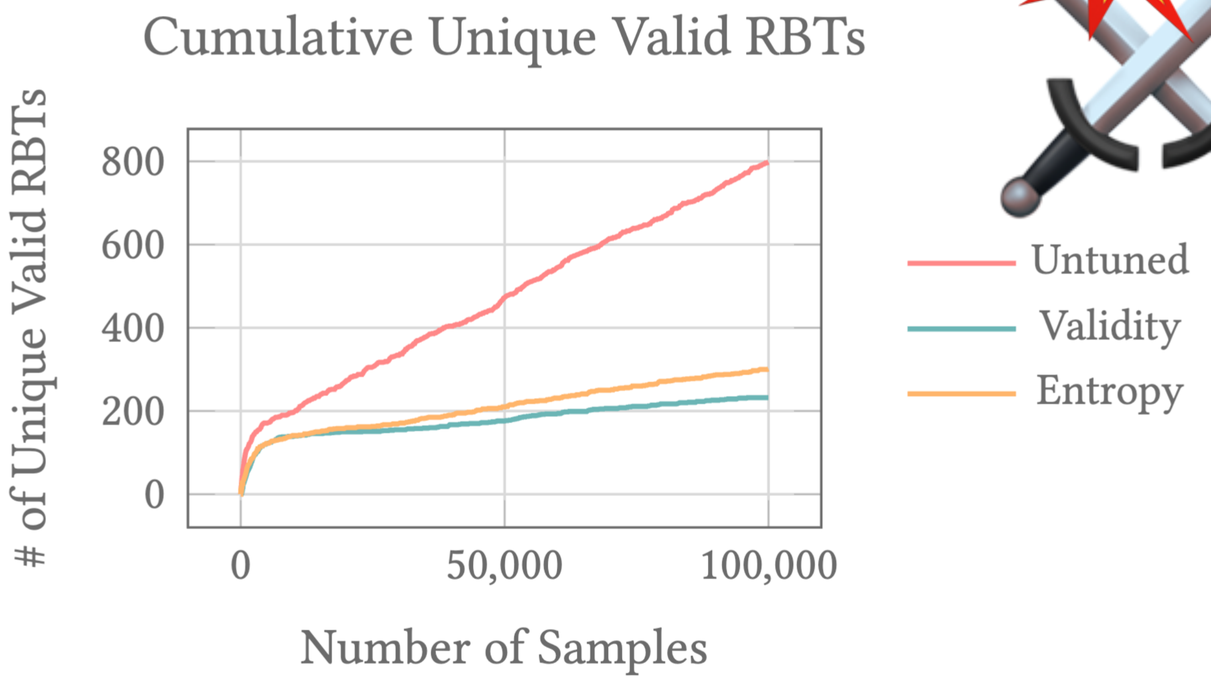
Satisfy property preconditions

Exercise more execution paths

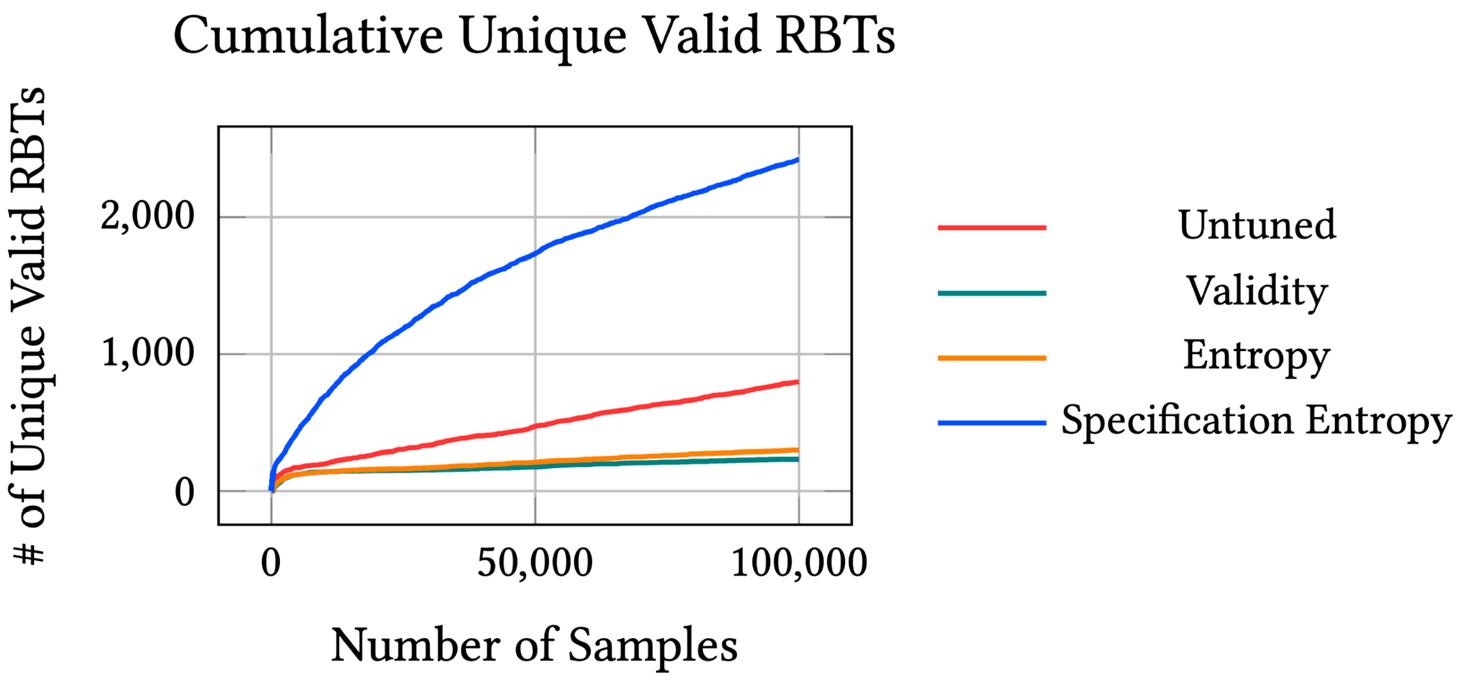
Validity and **diversity** are desirable but conflict.

Incentivizes trivial samples!

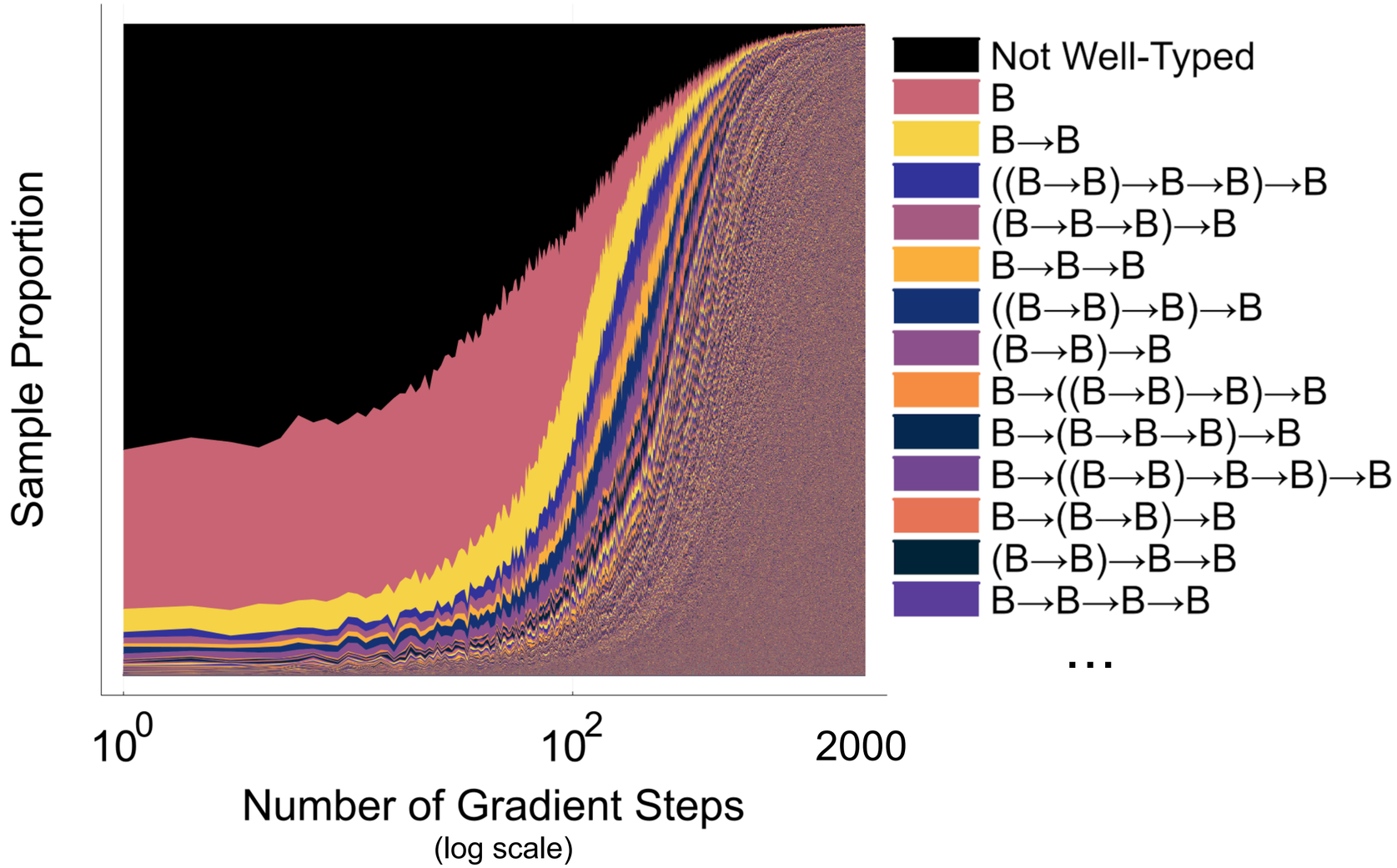
Incentivizes large, invalid samples!



“Specification entropy”: Diversity *within* the valid subspace of samples.

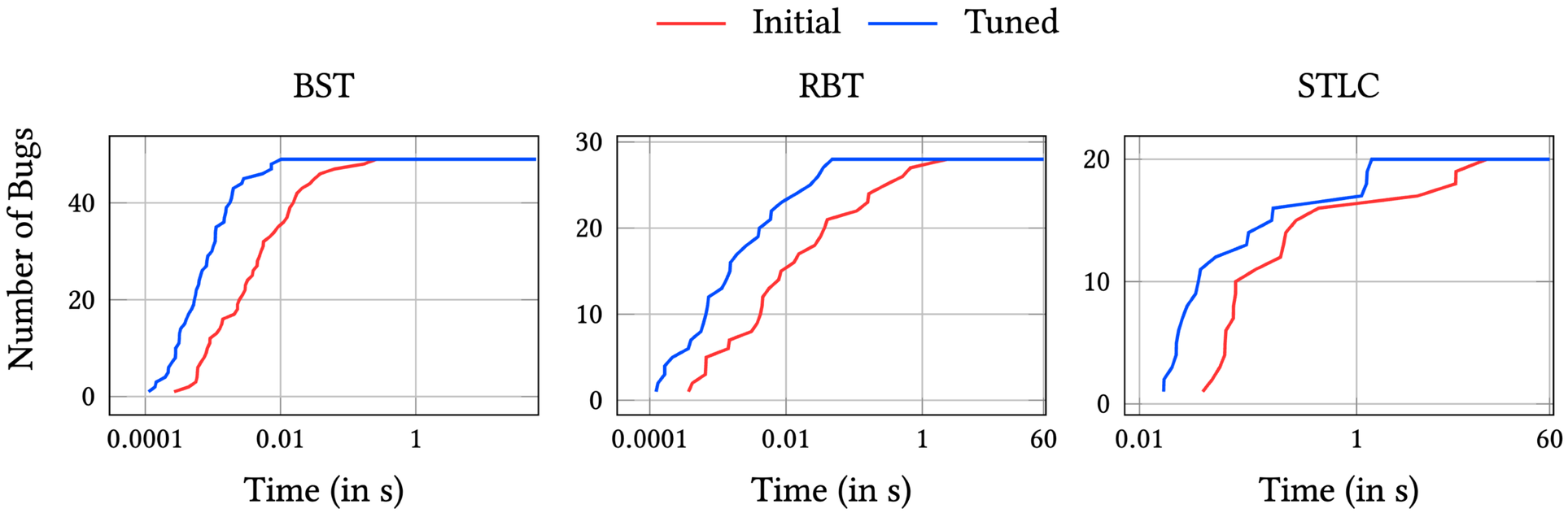


Tuning for STLC terms of diverse types



Evaluation on ETNA PBT benchmarks [Shi et al. 2023]

Tuning type-based generators for spec. entropy: **3.1-7.4x faster bug-finding**.



Conclusion: Automate tuning to specify distributions *declaratively* and generate *diverse and valid* test cases.

Loaded Dice: https://github.com/Tractables/Alea.jl/tree/loaded-dice

More in the paper!

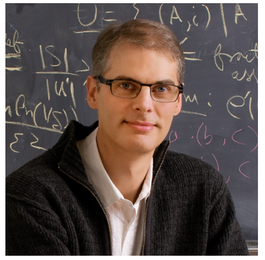
- Derive “tunable” generators from type definitions.
- A gradient estimator for specification entropy.
- Tuning a backtracking well-typed STLC generator.

Ryan Tjoa

Poorva Garg

Harry Goldstein

Todd Millstein



Benjamin Pierce

Guy Van den Broeck