ICFP 2021 SRC

Ungenerators

Harrison Goldstein

University of Pennsylvania

Full PDF Version with References

Key Points

- Generators are probabalistic programs that produce random test inputs for scenarios like property-based testing.
- Ungenerators act as backward generators and recover the random choices that lead to a given test input.

Generator	Ungenerator
🎲 → a	a → Maybe 🎲

- Our framework allows users to define generators and ungenerators simultaneously, with minimal extra code. These bidirectional generators are extremely expressive.
- Ungenerators can be used to optimize generator distributions by extracting choices from existing inputs and informing future generator choices.

1. What are Ungenerators?

Generators operate like parsers: they transform randomness into structured data.

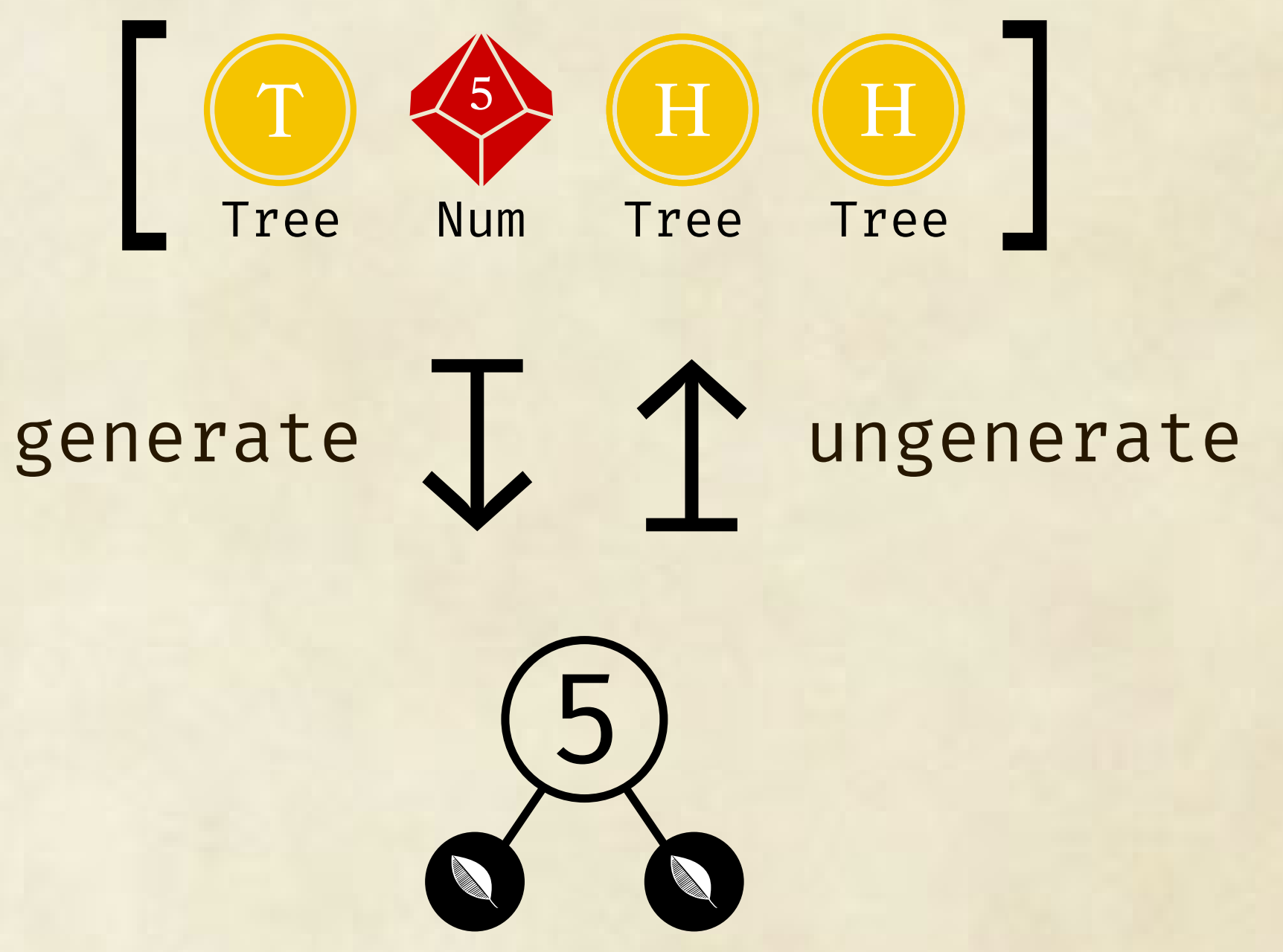
It is well known that bidirectional programming can be used to reverse a parser, resulting in a compatible pretty-printer (where `parse . print = id`).

We use the same technique to turn a generator into a compatible ungenerator.

Parser	Pretty-Printer
String → a	a → String
Generator	**Ungenerator**
🎲 → a	a → Maybe 🎲

The program **genTree** can be used as a generator for random binary trees, but it can also be used as an ungenerator. In the backward direction, we recover a list of generator choices that might produce a given tree.

```
genTree :: BiGen Tree Tree
genTree =
  select "Tree" [ return Leaf,
                  do
                    x ← comap getVal genNum
                    l ← comap getLeft genTree
                    r ← comap getRight genTree
                    return (Node l x r)
                ]
  where
    genNum = select "Num" [return x | x ← [1 .. 10]]
```



2. Implementation

Normal QuickCheck generators are built using a mondadic interface.

```
class Applicative m ⇒ Monad m where
  return :: a → m a
  (≫=) :: m a → (a → m b) → m b
```

The `return` function makes a trivial generator, and ≫= is used to sequence generators.

We follow *Xia, et al. 2019* and add support for bidirectional computation by requiring that our generators also be profunctors:

```
class Profunctor p where
  comap :: (c → Maybe b) → p b a → p c a
```

The `comap` operation is the secret implementing bidirectional computations compositionally — it allows the programmer to annotate correct behavior for the backward direction locally. Then the monad's bind can be used to compose these bidirectional programs together.

```
class (forall a. Monad (g a), Profunctor g) ⇒
      BiGen g where
  select :: String → [g a a] → g a a
```

The `select` operation completes the interface, providing a way to make random choices in the forward direction and recover random chocies in the backward direction.

3. Case Studies

We briefly present three case studies that show off what ungenerators can do. System F is evidence that bidirectional generators are expressive enough for interesting tasks; Inputs from Hell and RLCheck both show how ungenerators can be used to build on existing results in the testing literature.

So far we have only scratched the surface of what we think ungenerators are capable of.

System F

The monadic profunctor interface can express complex generators, including one for well-typed System F programs.

Generators for well-typed programs usually generate a type first and then generate an expression of that type:

```
genType
  ≫= genExpr
```

For the equivalent ungenerator, we use our type checker to recover the type in the backward direction.

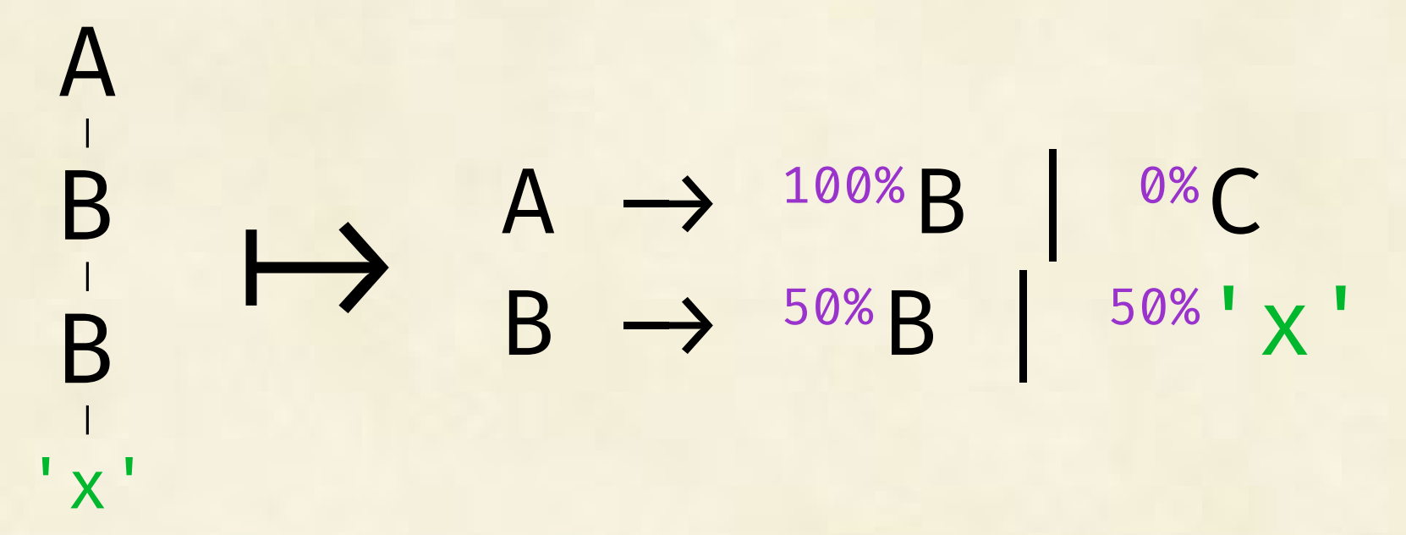
```
comap typeOf genType
  ≫= genExpr
```

While it does require some extra effort, this annotation is the only major change to the generator structure.

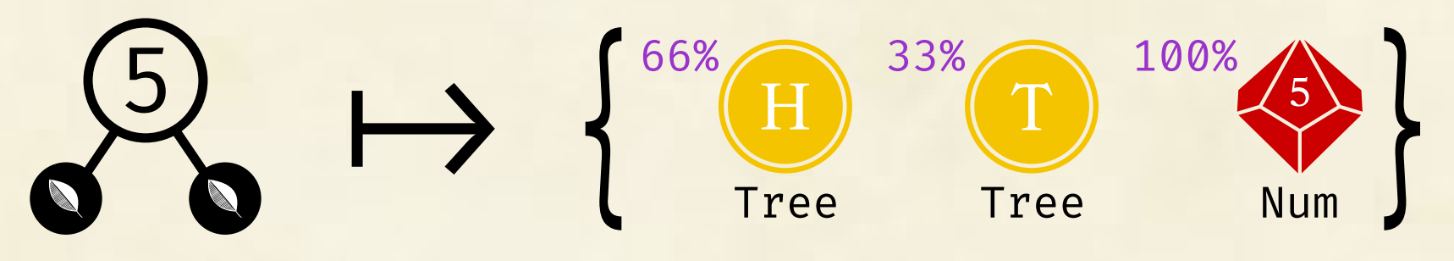
Inputs from Hell

Sormekun, et al. 2020 describes grammar-based generators that learn input distributions from input samples.

Given a sample parse tree, they construct a probabalistic grammar that produces similar trees by counting the non-terminal expansions in our sample and biasing the grammar accordingly:



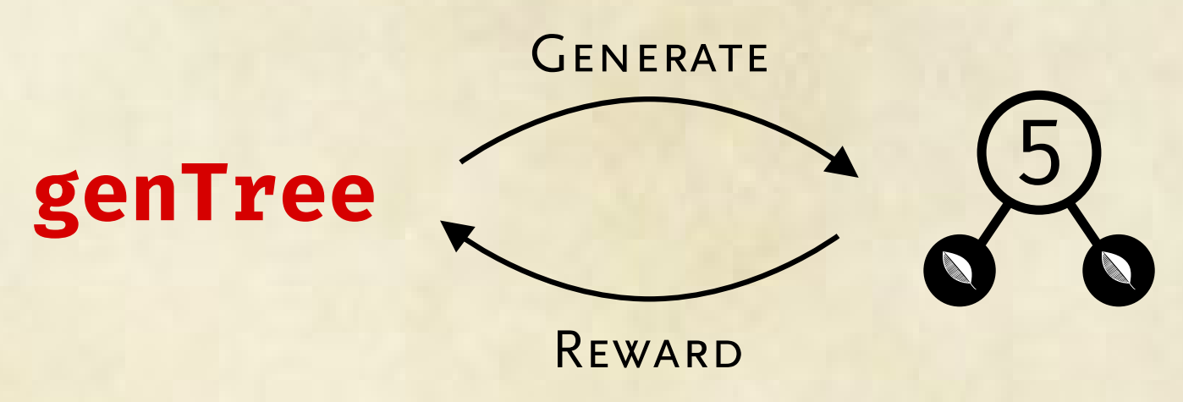
Ungenerators generalize this process for monadic generators.



RLCheck

Reddy, et al. 2020 presents a tool called RLCheck, which uses reinforcement learning to guide a generator to interesting inputs.

The standard version of RLCheck starts by making uniformly random choices; as generation progresses, those choices are refined to produce tests with desirable properties.



By reimplementing the RLCheck algorithm in terms of bidirectional generators, we are able to pre-train the algorithm and improve its initial choices.

References

Soremekun, Ezekiel, et al. "Inputs from Hell: Learning Input Distributions for Grammar-Based Test Generation." *IEEE Transactions on Software Engineering (2020).*

Xia, Li-yao, Dominic A. Orchard, and Meng Wang. "Composing bidirectional programs monadically." *Lecture Notes in Computer Science 11423 (2019): 147-175.*

Reddy, Sameer, et al. "Quickly generating diverse valid test inputs with reinforcement learning." *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020.