Type-Based Synthesis of Sound and Complete Random Generators

A Presentation for the REPL 2024 REU

Written by **Lemuel De Los Santos**

Advised by Harry Goldstein and Benjamin Pierce

What are random generators?

Random generators are functions that produce random values of a given type.

What are random generators?

Random generators are functions that produce random values of a given type.

Program

```haskell
take5 :: [Char] -> [Char]
take5 = take 5 . filter (`elem` ['a'..'e'])
```

What are random generators?

Random generators are functions that produce random values of a given type.

Program

```haskell
take5 :: [Char] -> [Char]
take5 = take 5 . filter (`elem` ['a'..'e'])
```

Test

$$\forall\ s\ .\ \text{length}\ (\text{take5}\ s) = 5$$

What are random generators?

Random generators are functions that produce random values of a given type.

Program

```haskell
take5 :: [Char] -> [Char]
take5 = take 5 . filter (`elem` ['a'..'e'])
```

Test

$$\forall\ s\ .\ \text{length}\ (\text{take5}\ s) = 5$$

```haskell
test = \s -> length (take5 s) == 5
```

What are random generators?

Random generators are functions that produce random values of a given type.

Program

```haskell
take5 :: [Char] -> [Char]
take5 = take 5 . filter (`elem` ['a'..'e'])
```

Test

$$\forall\ s\ .\ \text{length}\ (\text{take5}\ s) = 5$$

```haskell
quickcheck (\s -> length (take5 s) == 5)
```

What are random generators?

Random generators are functions that produce random values of a given type.

Program

```
take5 :: [Char] -> [Char]
take5 = take 5 . filter (`elem` ['a'..'e'])
```

Test

$$\forall\ s\ .\ \text{length}\ (\text{take5}\ s) = 5$$

```
*A> quickcheck (\s -> length (take5 s) == 5)
```

What are random generators?

Random generators are functions that produce random values of a given type.

Program

```haskell
take5 :: [Char] -> [Char]
take5 = take 5 . filter (`elem` ['a'..'e'])
```

Test

$$\forall\ s\ .\ \text{length}\ (\text{take5}\ s) = 5$$

```
*A> quickcheck (\s -> length (take5 s) == 5)
Falsifiable, after 0 tests:
""
```

What are random generators?

Random generators are functions that produce random values of a given type.

Program

```
take5 :: [Char] -> [Char]
take5 = take 5 . filter (`elem` ['a'..'e'])
```

Test

$$\forall\ s\ .\ \text{length}\ (\text{take5}\ s) \leq 5$$

```
*A> quickcheck (\s -> length (take5 s) <= 5)
```

What are random generators?

Random generators are functions that produce random values of a given type.

Program

```haskell
take5 :: [Char] -> [Char]
take5 = take 5 . filter (`elem` ['a'..'e'])
```

Test

$$\forall\ s\ .\ \text{length}\ (\text{take5}\ s) \leq 5$$

```
*A> quickcheck (\s -> length (take5 s) <= 5)
OK, passed 100 tests.
```

What are random generators?

Random generators are functions that produce random values of a given type.

Program

```haskell
take5 :: [Char] -> [Char]
take5 = take 5 . filter (`elem` ['a'..'e'])
```

Test

$$\forall\ s\ .\ \text{length}\ (\text{take5}\ s) \leq 5$$

```
*A> quickcheck (\s -> length (take5 s) <= 5)
OK, passed 100 tests.
```

How does QuickCheck know what to generate?

QuickCheck provides a way to define a typeclass for generating random values of a given type.

How does QuickCheck know what to generate?

QuickCheck provides a way to define a typeclass for generating random values of a given type.

Arbitrary

```haskell
class Arbitrary a where
  arbitrary :: Gen a

instance Arbitrary Int where ...
instance Arbitrary Char where ...
instance Arbitrary a => Arbitrary [a] where ...
```

How does QuickCheck know what to generate?

QuickCheck provides a way to define a typeclass for generating random values of a given type.

Arbitrary

```haskell
class Arbitrary a where
  arbitrary :: Gen a

instance Arbitrary Int where ...
instance Arbitrary Char where ...
instance Arbitrary a => Arbitrary [a] where ...
```

Example

```haskell
f :: Int -> Int
f x = x + 3
```

How does QuickCheck know what to generate?

QuickCheck provides a way to define a typeclass for generating random values of a given type.

Arbitrary

```
class Arbitrary a where
  arbitrary :: Gen a

instance Arbitrary Int where ...
instance Arbitrary Char where ...
instance Arbitrary a => Arbitrary [a] where ...
```

Example

```
f :: Nat -> Nat
f x = x + 3
```

How does QuickCheck know what to generate?

QuickCheck provides a way to define a typeclass for generating random values of a given type.

Arbitrary

```
class Arbitrary a where
  arbitrary :: Gen a

instance Arbitrary Int where ...
instance Arbitrary Char where ...
instance Arbitrary a => Arbitrary [a] where ...
```

Example

```
f :: SortedList a -> Nat
```

How does QuickCheck know what to generate?

QuickCheck provides a way to define a typeclass for generating random values of a given type.

Arbitrary

```
class Arbitrary a where
  arbitrary :: Gen a

instance Arbitrary Int where ...
instance Arbitrary Char where ...
instance Arbitrary a => Arbitrary [a] where ...
```

Example

```
f :: BST a -> Nat
```

Type-Based Synthesis

Where the type of the program is given, and the task is to generate another program that has that type.

Type-Based Synthesis

Where the type of the program is given, and the task is to generate another program that has that type.

What is program synthesis?

Program synthesis is the task of automatically generating a program given a specification.

Type-Based Synthesis

Where the type of the program is given, and the task is to generate another program that has that type.

What is program synthesis?

Program synthesis is the task of automatically generating a program given a specification.

How can synthesis be type-based?

In type-based synthesis, the goal is to generate a program that satisfies a given type specification.

Introducing Synquid

Synquid synthesizes programs from refinement types.

Introducing Synquid

Synquid synthesizes programs from refinement types.

Refinement Types

$$\text{Nat} = \{\nu : \text{Int} \mid \nu \geq 0\}$$

Introducing Synquid

Synquid synthesizes programs from refinement types.

Refinement Types

$$\text{Nat} = \{\nu : \text{Int} \mid \nu \geq 0\}$$

The Language

```
type Nat = { Int | _v >= 0 }

zero :: { Nat | _v == 0 }
one :: { Nat | _v == 1 }

n :: Nat
n = ??
```

```
n = one
```

```
-----------------------------------------
-- Insertion into a binary search tree --
-----------------------------------------

-- Binary search tree:
-- note how the refinements on the Node constructor define
data BST a where
  Empty :: BST a
  Node  :: x: a -> l: BST { a | _v < x } -> r: BST { a | x

-- Size of a BST (termination metric)
termination measure size :: BST a -> { Int | _v >= 0 } wher
  Empty -> 0
  Node x l r -> size l + size r + 1

-- The set of all keys in a BST
measure keys :: BST a -> Set a where
  Empty -> []
  Node x l r -> keys l + keys r + [x]

leq :: x: a -> y: a -> { Bool | _v == (x <= y) }
neq :: x: a -> y: a -> { Bool | _v == (x != y) }

-- Our synthesis goal: a function that inserts a key into a
insert :: x: a -> t: BST a -> { BST a | keys _v == keys t +
insert = ??
```

```
insert = \x . \t .
    match t with
      Empty -> Node x Empty Empty
      Node x7 x8 x9 ->
        if (x <= x7) && (x7 <= x)
          then t
          else
            if x7 <= x
              then Node x7 x8 (insert x x9)
              else Node x7 (insert x x8) x9
```

Limitations

Limitations

$$\text{Pred} = \{\nu : \text{Int} \mid \nu \geq 0 \wedge \nu \leq 255\}$$

Limitations

$$\text{Pred} = \{\nu : \text{Int} \mid \nu \geq 0 \wedge \nu \leq 255\}$$

Generators (Sound and Complete)

$$\forall\ \nu\ .\ \nu \geq 0 \wedge \nu \leq 255$$

Limitations

$$\text{Pred} = \{\nu : \text{Int} \mid \nu \geq 0 \land \nu \leq 255\}$$

Generators (Sound and Complete)

$$\forall\ \nu\ .\ \nu \geq 0 \land \nu \leq 255$$

Synquid (Sound)

$$\exists\ \nu\ .\ \nu \geq 0 \land \nu \leq 255$$

Solution

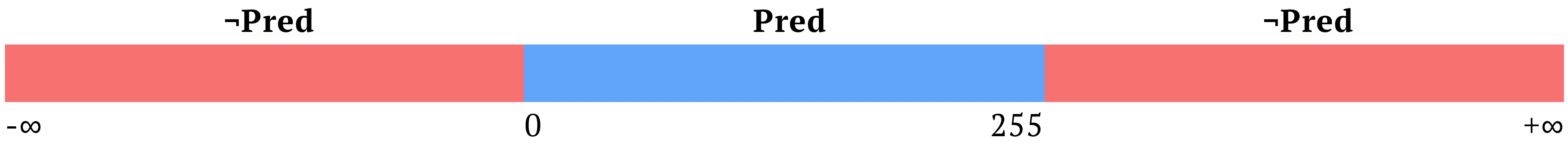
```
byte :: x: { Int | _v >= 0 && _v <= 255 } -> Int
```

$$\text{Pred} = \{\nu : \text{Int} \mid \nu \geq 0 \land \nu \leq 255\}$$

Solution

```
byte :: x: { Int | _v >= 0 && _v <= 255 } -> Int
```

$$\text{Pred} = \{\nu : \text{Int} \mid \nu \geq 0 \wedge \nu \leq 255\}$$



Solution

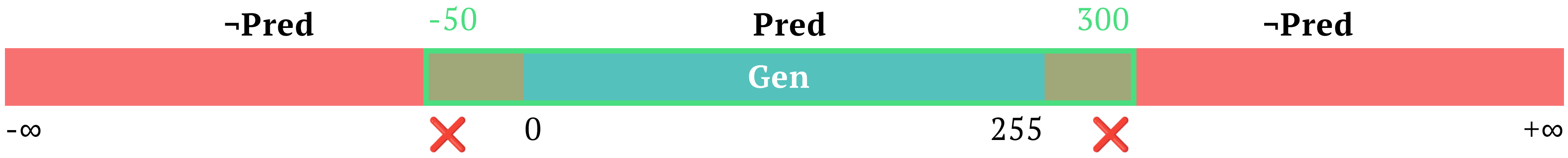
```
byte :: x: { Int | _v >= 0 && _v <= 255 } -> Int
```

$$\text{Pred} = \{\nu : \text{Int} \mid \nu \geq 0 \land \nu \leq 255\}$$

Solution

```
byte :: x: { Int | _v >= 0 && _v <= 255 } -> Int
```

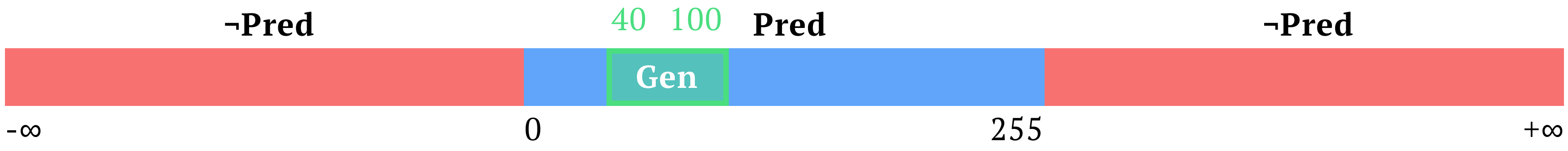
$$\text{Pred} = \{\nu : \text{Int} \mid \nu \geq 0 \land \nu \leq 255\}$$



Solution

```
byte :: x: { Int | _v >= 0 && _v <= 255 } -> Int
```

$$\text{Pred} = \{\nu : \text{Int} \mid \nu \geq 0 \land \nu \leq 255\}$$

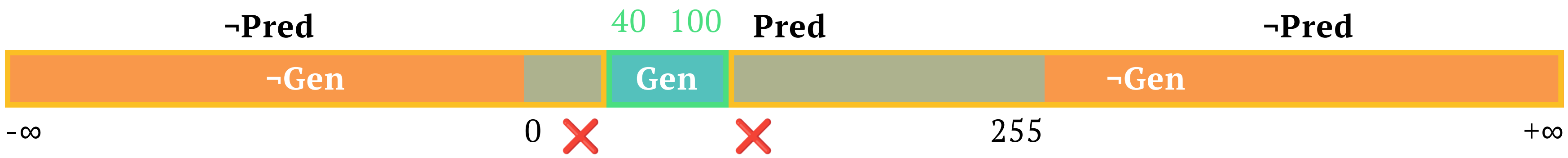


Solution

```
byte :: x: { Int | _v >= 0 && _v <= 255 } -> Int
<byte.x> :: <x.not>: { Int | _v < 0 || _v > 255 }
```

$$\text{Pred} = \{\nu : \text{Int} \mid \nu \geq 0 \land \nu \leq 255\}$$

$$\neg\text{Pred} = \{\nu : \text{Int} \mid \nu < 0 \lor \nu > 255\}$$

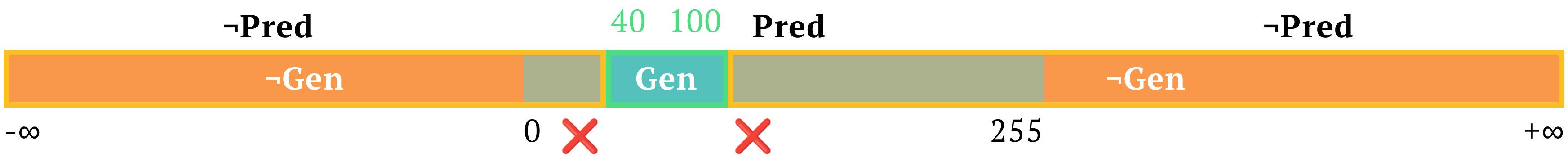


Solution

```
byte :: x: { Int | _v >= 0 && _v <= 255 } -> Int
<byte.x> :: <x.not>: { Int | _v < 0 || _v > 255 }
```

$$\text{Pred} = \{\nu : \text{Int} \mid \nu \geq 0 \land \nu \leq 255\}$$

$$\neg\text{Pred} = \{\nu : \text{Int} \mid \nu < 0 \lor \nu > 255\}$$



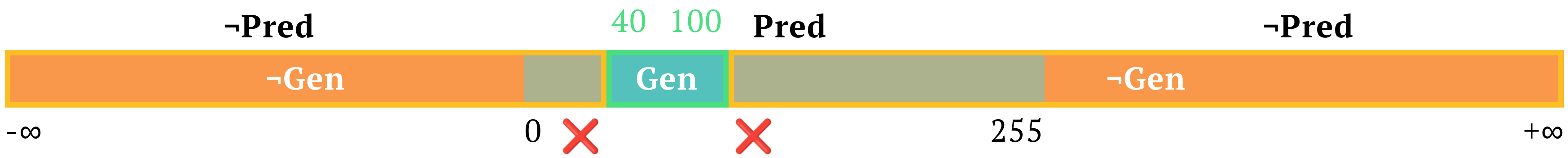
$$\begin{aligned} Gen &\implies Pred \\ \neg Gen &\implies \neg Pred \\ Pred &\implies Gen \\ \hline \therefore \quad Gen &\equiv Pred \end{aligned}$$

Solution

```
byte :: x: { Int | _v >= 0 && _v <= 255 } -> Int
<byte.x> :: <x.not>: { Int | _v < 0 || _v > 255 }
```

$$\text{Pred} = \{\nu : \text{Int} \mid \nu \geq 0 \land \nu \leq 255\}$$

$$\neg\text{Pred} = \{\nu : \text{Int} \mid \nu < 0 \lor \nu > 255\}$$



$$
\begin{aligned}
Gen &\implies Pred \\
\neg Gen &\implies \neg Pred \\
Pred &\implies Gen \\
\hline
\therefore \quad Gen &\equiv Pred
\end{aligned}
$$

Solution

```
byte :: x: { Int | _v >= 0 && _v <= 255 } -> Int
<byte.x> :: <x.not>: { Int | _v < 0 || _v > 255 }
```

$$\text{Pred} = \{\nu : \text{Int} \mid \nu \geq 0 \wedge \nu \leq 255\}$$

$$\neg\text{Pred} = \{\nu : \text{Int} \mid \nu < 0 \vee \nu > 255\}$$

$$\begin{aligned} Gen &\implies Pred \\ \neg Gen &\implies \neg Pred \\ Pred &\implies Gen \\ \hline \therefore \quad Gen &\equiv Pred \end{aligned}$$

```
pair :: x: Int -> y: { Int | _v == x + 3 } -> Int
pair_y_not :: x: Int -> y: { Int | _v == x + 3 } -> y_not:

gen_pair :: Int
gen_pair =
  let x = ?? in
  let x_value = g x in

  let y = ?? in
  let y_value = g y in
  let _ = pair_y_not x_value y_value (g_not y) in

  pair x_value y_value
```

```
gen_pair = let x = Eq int in
           let x_value = g x in
           let y = Eq (plus x_value three)
             in
           let y_value = g y in
           let _ = pair_y_not x_value
                     y_value (g_not y) in
           pair x_value y_value
```

- We were able to make Synquid synthesize a sound and complete generators given a refinement type:
  - `byte :: x: { Int | (_v >= 0 && _v <= 255) } -> Int`
  - `pair :: x: { Int | _v <= 3 } -> y: { Int | _v == x + 3 } -> Int`
  - `range :: x: {Int | _v >= -10 && !(_v < -10) } -> y: { Int | _v >= x } -> z: { Int | _v >= y && _v <= 10 } -> Int`

fin.