Fast and Loose DSLs

(with Hedgehog)

Tim Humphries
@thumphriees
teh.id.au

Hello, etc.
My name's Tim. This talk is about applying **really naive** techniques to get a **pretty good** test suite for a programming language using Hedgehog. This makes it really easy to **trust** your language implementation, and to **extend** the language later.
It's a little bit **half-baked and anecdote-heavy,** so apologies in advance!

Simply-Typed λ

```
data Expr =
    EBool Bool
  | EInt Int
  | EString String
  | EVar Name
  | ELam Name Type Expr
  | EApp Expr Expr
```

Let’s build our language on top of something recognisable, like **the simply-typed lambda calculus**.
The STLC usually has three constructs: **variables, lambdas, and function application**.

Simply-Typed λ

```
data Expr =
    EBool Bool
  | EInt Int
  | EString String
  | EVar Name
  | ELam Name Type Expr
  | EApp Expr Expr
```

This isn’t all that useful on its own, so let’s extend it with some primitive types.
We’ve got **Ints, Bools and Strings.** If I had better attention to detail, I’d also have a bunch of primitive operations.
In my formulation here, function parameters come with **type annotations**.
Let’s see how Types are defined:

Simply-Typed λ

```
data Type =
    TBool
  | TInt
  | TString
  | TArrow Type Type
```

Types are pretty simple. **Bools, Ints, Strings, and Functions (Arrows)**

Simply-Typed λ

We want to build a useful tool around this data structure.
The first thing we need is a way to get information into and out of this format: **concrete syntax**.
We come up with this little **informal specification**.
In order to implement this informal specification, we write two functions: a **parser** and a **printer**.
These functions are really notoriously fiddly and difficult to get right! (Even with a formal specification - they're not often executable)

Simply-Typed λ

We want to build a useful tool around this data structure.
The first thing we need is a way to get information into and out of this format: **concrete syntax**.
We come up with this little **informal specification**.
In order to implement this informal specification, we write two functions: a **parser** and a **printer**.
These functions are really notoriously fiddly and difficult to get right! (Even with a formal specification - they’re not often executable)

Simply-Typed λ

```
55
55 + 22
```

We want to build a useful tool around this data structure.
The first thing we need is a way to get information into and out of this format: **concrete syntax**.
We come up with this little **informal specification**.
In order to implement this informal specification, we write two functions: a **parser** and a **printer**.
These functions are really notoriously fiddly and difficult to get right! (Even with a formal specification - they're not often executable)

Simply-Typed λ

```
55
55 + 22
(\z : Int -> z + z) 55
```

We want to build a useful tool around this data structure.
The first thing we need is a way to get information into and out of this format: **concrete syntax**.
We come up with this little **informal specification**.
In order to implement this informal specification, we write two functions: a **parser** and a **printer**.
These functions are really notoriously fiddly and difficult to get right! (Even with a formal specification - they're not often executable)

Simply-Typed λ

```
55
55 + 22
(\z : Int -> z + z) 55

--| Turn an ‘Expr’ into a ‘String’.
ppExpr :: Expr -> String
ppExpr = …
```

We want to build a useful tool around this data structure.
The first thing we need is a way to get information into and out of this format: **concrete syntax**.
We come up with this little **informal specification**.
In order to implement this informal specification, we write two functions: a **parser** and a **printer**.
These functions are really notoriously fiddly and difficult to get right! (Even with a formal specification - they’re not often executable)

Simply-Typed λ

```
55
55 + 22
(\z : Int -> z + z) 55

--| Turn an ‘Expr’ into a ‘String’.
ppExpr :: Expr -> String
ppExpr = …

--| Turn a ‘String’ into an ‘Expr’.
parse :: String -> Either ParseError Expr
parse fp = …
```

We want to build a useful tool around this data structure.
The first thing we need is a way to get information into and out of this format: **concrete syntax**.
We come up with this little **informal specification**.
In order to implement this informal specification, we write two functions: a **parser** and a **printer**.
These functions are really notoriously fiddly and difficult to get right! (Even with a formal specification - they’re not often executable)

Simply-Typed λ

We want to be sure our code kinda works. So we put on our Uncle Bob hats and start racking our brains.
We start out quite happy, coming up with the kind of functions that we can imagine writing immediately.
We then start to get a little bit paranoid. It starts to affect day-to-day life. You can’t trust anything anymore.

Simply-Typed λ

We want to be sure our code kinda works. So we put on our Uncle Bob hats and start racking our brains.
We start out quite happy, coming up with the kind of functions that we can imagine writing immediately.
We then start to get a little bit paranoid. It starts to affect day-to-day life. You can’t trust anything anymore.

Simply-Typed λ

```
55
55 + 22
```

We want to be sure our code kinda works. So we put on our Uncle Bob hats and start racking our brains.
We start out quite happy, coming up with the kind of functions that we can imagine writing immediately.
We then start to get a little bit paranoid. It starts to affect day-to-day life. You can’t trust anything anymore.

Simply-Typed λ

```
55
55 + 22
(\z : Int -> z + z) 55
```

We want to be sure our code kinda works. So we put on our Uncle Bob hats and start racking our brains.
We start out quite happy, coming up with the kind of functions that we can imagine writing immediately.
We then start to get a little bit paranoid. It starts to affect day-to-day life. You can’t trust anything anymore.

Simply-Typed λ

```
55
55 + 22
(\z : Int -> z + z) 55
(\x : ((Int -> Int) -> Int) -> x 55 “abc”)
```

We want to be sure our code kinda works. So we put on our Uncle Bob hats and start racking our brains.
We start out quite happy, coming up with the kind of functions that we can imagine writing immediately.
We then start to get a little bit paranoid. It starts to affect day-to-day life. You can’t trust anything anymore.

Simply-Typed λ

```
55
55 + 22
(\z : Int -> z + z) 55
(\x : ((Int -> Int) -> Int) -> x 55 “abc”)
(\x : String -> append x x) “hello”
```

We want to be sure our code kinda works. So we put on our Uncle Bob hats and start racking our brains.
We start out quite happy, coming up with the kind of functions that we can imagine writing immediately.
We then start to get a little bit paranoid. It starts to affect day-to-day life. You can’t trust anything anymore.

Simply-Typed λ

```
55
55 + 22
(\z : Int -> z + z) 55
(\x : ((Int -> Int) -> Int) -> x 55 “abc”)
(\x : String -> append x x) “hello”
(\y : Int ->
```

We want to be sure our code kinda works. So we put on our Uncle Bob hats and start racking our brains.
We start out quite happy, coming up with the kind of functions that we can imagine writing immediately.
We then start to get a little bit paranoid. It starts to affect day-to-day life. You can’t trust anything anymore.

Simply-Typed λ

```
55
55 + 22
(\z : Int -> z + z) 55
(\x : ((Int -> Int) -> Int) -> x 55 “abc”)
(\x : String -> append x x) “hello”
(\y : Int ->
  (\z -> Int ->
```

We want to be sure our code kinda works. So we put on our Uncle Bob hats and start racking our brains.
We start out quite happy, coming up with the kind of functions that we can imagine writing immediately.
We then start to get a little bit paranoid. It starts to affect day-to-day life. You can’t trust anything anymore.

Simply-Typed λ

```
55
55 + 22
(\z : Int -> z + z) 55
(\x : ((Int -> Int) -> Int) -> x 55 “abc”)
(\x : String -> append x x) “hello”
(\y : Int ->
  (\z -> Int ->
    (\x : (Int -> Int -> Int) -> (x y z))))
```

We want to be sure our code kinda works. So we put on our Uncle Bob hats and start racking our brains.
We start out quite happy, coming up with the kind of functions that we can imagine writing immediately.
We then start to get a little bit paranoid. It starts to affect day-to-day life. You can’t trust anything anymore.

Simply-Typed λ

```
55
55 + 22
(\z : Int -> z + z) 55
(\x : ((Int -> Int) -> Int) -> x 55 “abc”)
(\x : String -> append x x) “hello”
(\y : Int ->
  (\z -> Int ->
    (\x : (Int -> Int -> Int) -> (x y z))))
(\x : Bool ->
```

We want to be sure our code kinda works. So we put on our Uncle Bob hats and start racking our brains.
We start out quite happy, coming up with the kind of functions that we can imagine writing immediately.
We then start to get a little bit paranoid. It starts to affect day-to-day life. You can’t trust anything anymore.

Simply-Typed λ

```
55
55 + 22
(\z : Int -> z + z) 55
(\x : ((Int -> Int) -> Int) -> x 55 “abc”)
(\x : String -> append x x) “hello”
(\y : Int ->
  (\z -> Int ->
    (\x : (Int -> Int -> Int) -> (x y z))))
(\x : Bool ->
  ((\foo123 : Bool -> and x (not foo123))
```

We want to be sure our code kinda works. So we put on our Uncle Bob hats and start racking our brains.
We start out quite happy, coming up with the kind of functions that we can imagine writing immediately.
We then start to get a little bit paranoid. It starts to affect day-to-day life. You can’t trust anything anymore.

Simply-Typed λ

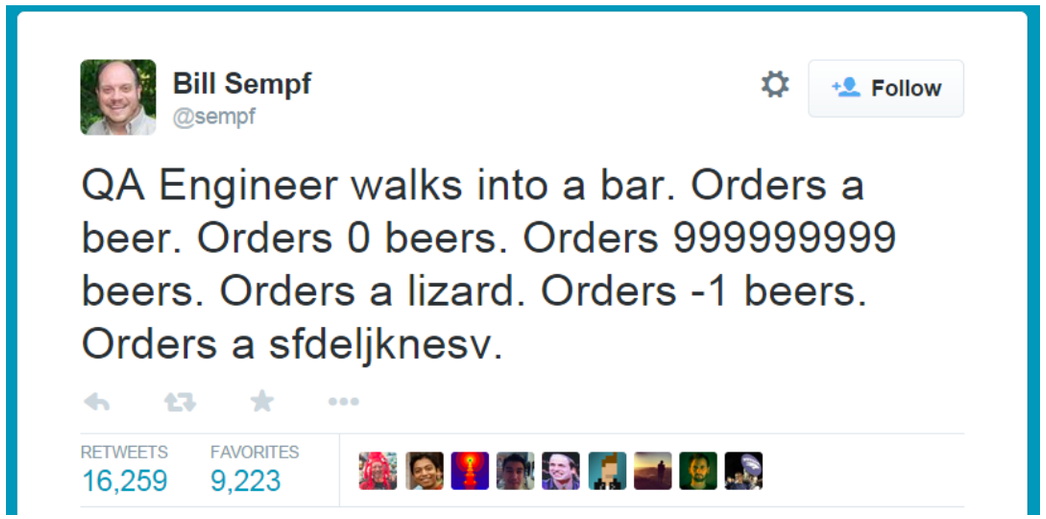
```
55
55 + 22
(\z : Int -> z + z) 55
(\x : ((Int -> Int) -> Int) -> x 55 “abc”)
(\x : String -> append x x) “hello”
(\y : Int ->
  (\z -> Int ->
    (\x : (Int -> Int -> Int) -> (x y z))))
(\x : Bool ->
  ((\foo123 : Bool -> and x (not foo123))
  False)
```

We want to be sure our code kinda works. So we put on our Uncle Bob hats and start racking our brains.
We start out quite happy, coming up with the kind of functions that we can imagine writing immediately.
We then start to get a little bit paranoid. It starts to affect day-to-day life. You can’t trust anything anymore.

Simply-Typed λ

```
55
55 + 22
(\z : Int -> z + z) 55
(\x : ((Int -> Int) -> Int) -> x 55 “abc”)
(\x : String -> append x x) “hello”
(\y : Int ->
  (\z -> Int ->
    (\x : (Int -> Int -> Int) -> (x y z))))
(\x : Bool ->
  ((\foo123 : Bool -> and x (not foo123))
  False)
((\ſåd : String -> ßåd) √¬åd π®ød ß¬åd)
```

We want to be sure our code kinda works. So we put on our Uncle Bob hats and start racking our brains.
We start out quite happy, coming up with the kind of functions that we can imagine writing immediately.
We then start to get a little bit paranoid. It starts to affect day-to-day life. You can’t trust anything anymore.



We do our best to enumerate as many cases as we can. They start to get less and less realistic, and closer to the edges of the informal specification.
This takes days, and we don't feel a whole lot better at the end of it: **we only thought of things we thought of**.
What about those unknown unknowns?

Property-Based Testing

An enterprise List.reverse testing framework

We show up to FP-SYD every month, so we know all about property-based testing, a really fancy way to test that your lists reverse properly.
We know about testing functions that are supposed to be idempotent, like List.reverse.
We also know about testing functions that are supposed to be involutive, like Set toList and fromList.
Sometimes it's difficult to generalise from these common examples to things that are useful in the real-world.
I didn't become a heavy user of property-based testing for boring enterprisey problems until I worked on a team where that was the default. Testing IO-heavy code and fancy stateful processes really takes some practise.
However, it's **really easy** to jump from these to my **favourite property of all time for parsers and printers**:

Property-Based Testing

An enterprise List.reverse testing framework

```
List.reverse (List.reverse xs) === xs
```

We show up to FP-SYD every month, so we know all about property-based testing, a really fancy way to test that your lists reverse properly.
We know about testing functions that are supposed to be idempotent, like List.reverse.
We also know about testing functions that are supposed to be involutive, like Set toList and fromList.
Sometimes it's difficult to generalise from these common examples to things that are useful in the real-world.
I didn't become a heavy user of property-based testing for boring enterprisey problems until I worked on a team where that was the default. Testing IO-heavy code and fancy stateful processes really takes some practise.
However, it's **really easy** to jump from these to my **favourite property of all time for parsers and printers**:

Property-Based Testing

An enterprise List.reverse testing framework

```
List.reverse (List.reverse xs) === xs
```

```
Set.fromList (Set.toList xs) === xs
```

We show up to FP-SYD every month, so we know all about property-based testing, a really fancy way to test that your lists reverse properly.
We know about testing functions that are supposed to be idempotent, like List.reverse.
We also know about testing functions that are supposed to be involutive, like Set toList and fromList.
Sometimes it's difficult to generalise from these common examples to things that are useful in the real-world.
I didn't become a heavy user of property-based testing for boring enterprisey problems until I worked on a team where that was the default. Testing IO-heavy code and fancy stateful processes really takes some practise.
However, it's **really easy** to jump from these to my **favourite property of all time for parsers and printers**:

Round Tripping

```
parse (print ast) === pure ast
```

This leads me to the most effective property I know of, the **round trip property**.
It's just our standard **involutive** property.
It's especially good for figuring out if either of the two functions are lossy.
In our case, it also helps us understand that our functions are self-consistent: that they both implement the same language.

Hedgehog!

I added this slide about ten minutes ago, so hopefully it makes sense.
Everyone I spoke to when I walked in the room hadn’t heard of Hedgehog, so here’s a quick introduction.

There are official Haskell, Scala, F#, Purescript, and R versions. Haskell version has the most features.

Hedgehog!

(Like QuickCheck, but…)

I added this slide about ten minutes ago, so hopefully it makes sense.
Everyone I spoke to when I walked in the room hadn't heard of Hedgehog, so here's a quick introduction.

There are official Haskell, Scala, F#, Purescript, and R versions. Haskell version has the most features.

Hedgehog!

(Like QuickCheck, but…)

- Shrinking for free

I added this slide about ten minutes ago, so hopefully it makes sense.
Everyone I spoke to when I walked in the room hadn't heard of Hedgehog, so here's a quick introduction.

There are official Haskell, Scala, F#, Purescript, and R versions. Haskell version has the most features.

Hedgehog!

(Like QuickCheck, but…)

- Shrinking for free
  - Generators have invariants, and shrinks preserve them by construction

I added this slide about ten minutes ago, so hopefully it makes sense.
Everyone I spoke to when I walked in the room hadn't heard of Hedgehog, so here's a quick introduction.

There are official Haskell, Scala, F#, Purescript, and R versions. Haskell version has the most features.

Hedgehog!

(Like QuickCheck, but…)

- Shrinking for free
  - Generators have invariants, and shrinks preserve them by construction
- Modern Haskell interface

I added this slide about ten minutes ago, so hopefully it makes sense.
Everyone I spoke to when I walked in the room hadn't heard of Hedgehog, so here's a quick introduction.

There are official Haskell, Scala, F#, Purescript, and R versions. Haskell version has the most features.

Hedgehog!

(Like QuickCheck, but…)

- Shrinking for free
  - Generators have invariants, and shrinks preserve them by construction
- Modern Haskell interface
- Ergonomic improvements over QuickCheck

I added this slide about ten minutes ago, so hopefully it makes sense.
Everyone I spoke to when I walked in the room hadn't heard of Hedgehog, so here's a quick introduction.

There are official Haskell, Scala, F#, Purescript, and R versions. Haskell version has the most features.

Hedgehog!

(Like QuickCheck, but…)

- Shrinking for free
  - Generators have invariants, and shrinks preserve them by construction
- Modern Haskell interface
- Ergonomic improvements over QuickCheck
  - Fancy parallel test runner etc

I added this slide about ten minutes ago, so hopefully it makes sense.
Everyone I spoke to when I walked in the room hadn't heard of Hedgehog, so here's a quick introduction.

There are official Haskell, Scala, F#, Purescript, and R versions. Haskell version has the most features.

Hedgehog!

(Like QuickCheck, but…)

- Shrinking for free
  - Generators have invariants, and shrinks preserve them by construction
- Modern Haskell interface
- Ergonomic improvements over QuickCheck
  - Fancy parallel test runner etc
  - Precise control over the range of generated values

I added this slide about ten minutes ago, so hopefully it makes sense.
Everyone I spoke to when I walked in the room hadn't heard of Hedgehog, so here's a quick introduction.

There are official Haskell, Scala, F#, Purescript, and R versions. Haskell version has the most features.

Hedgehog!

(Like QuickCheck, but…)

- Shrinking for free
  - Generators have invariants, and shrinks preserve them by construction
- Modern Haskell interface
- Ergonomic improvements over QuickCheck
  - Fancy parallel test runner etc
  - Precise control over the range of generated values
  - No spurious typeclasses – value-level Haskell

I added this slide about ten minutes ago, so hopefully it makes sense.
Everyone I spoke to when I walked in the room hadn't heard of Hedgehog, so here's a quick introduction.

There are official Haskell, Scala, F#, Purescript, and R versions. Haskell version has the most features.

Hedgehog!

(Like QuickCheck, but…)

- Shrinking for free
  - Generators have invariants, and shrinks preserve them by construction
- Modern Haskell interface
- Ergonomic improvements over QuickCheck
  - Fancy parallel test runner etc
  - Precise control over the range of generated values
  - No spurious typeclasses – value-level Haskell
  - Doesn't hang your machine

I added this slide about ten minutes ago, so hopefully it makes sense.
Everyone I spoke to when I walked in the room hadn't heard of Hedgehog, so here's a quick introduction.

There are official Haskell, Scala, F#, Purescript, and R versions. Haskell version has the most features.

Hedgehog!

(Like QuickCheck, but…)

- Shrinking for free
  - Generators have invariants, and shrinks preserve them by construction
- Modern Haskell interface
- Ergonomic improvements over QuickCheck
  - Fancy parallel test runner etc
  - Precise control over the range of generated values
  - No spurious typeclasses – value-level Haskell
  - Doesn’t hang your machine
- State machine testing a la Erlang QuickCheck (new! amazing!)

I added this slide about ten minutes ago, so hopefully it makes sense.
Everyone I spoke to when I walked in the room hadn’t heard of Hedgehog, so here’s a quick introduction.

There are official Haskell, Scala, F#, Purescript, and R versions. Haskell version has the most features.

Hedgehog!

(Like QuickCheck, but…)

- Shrinking for free
  - Generators have invariants, and shrinks preserve them by construction
- Modern Haskell interface
- Ergonomic improvements over QuickCheck
  - Fancy parallel test runner etc
  - Precise control over the range of generated values
  - No spurious typeclasses – value-level Haskell
  - Doesn't hang your machine
- State machine testing a la Erlang QuickCheck (new! amazing!)
- See Jacob's FP-Syd / Lambda Jam / CUFP talk "Gens 'N Roses – Appetite for Reduction"

I added this slide about ten minutes ago, so hopefully it makes sense.
Everyone I spoke to when I walked in the room hadn't heard of Hedgehog, so here's a quick introduction.

There are official Haskell, Scala, F#, Purescript, and R versions. Haskell version has the most features.

Arbitrary Terms

So, let’s get our round-trip property going with Hedgehog.
To actually run this property, we need to be able to generate random input. We have few invariants right now, so any expression should do.
Hedgehog provides some shorthand for generating recursive data structures. This helps us write generators that terminate - if you’ve used QuickCheck heavily you’ll know why this is really useful.

Arbitrary Terms

```
genExpr :: Gen Expr
```

So, let’s get our round-trip property going with Hedgehog.
To actually run this property, we need to be able to generate random input. We have few invariants right now, so any expression should do.
Hedgehog provides some shorthand for generating recursive data structures. This helps us write generators that terminate - if you’ve used QuickCheck heavily you’ll know why this is really useful.

Arbitrary Terms

```
genExpr :: Gen Expr
genExpr =
```

So, let’s get our round-trip property going with Hedgehog.
To actually run this property, we need to be able to generate random input. We have few invariants right now, so any expression should do.
Hedgehog provides some shorthand for generating recursive data structures. This helps us write generators that terminate - if you’ve used QuickCheck heavily you’ll know why this is really useful.

Arbitrary Terms

```
genExpr :: Gen Expr
genExpr =
  Gen.recursive Gen.choice [
```

So, let’s get our round-trip property going with Hedgehog.
To actually run this property, we need to be able to generate random input. We have few invariants right now, so any expression should do.
Hedgehog provides some shorthand for generating recursive data structures. This helps us write generators that terminate - if you’ve used QuickCheck heavily you’ll know why this is really useful.

Arbitrary Terms

```
genExpr :: Gen Expr
genExpr =
  Gen.recursive Gen.choice [
      -- Non-recursive generators
```

So, let’s get our round-trip property going with Hedgehog.
To actually run this property, we need to be able to generate random input. We have few invariants right now, so any expression should do.
Hedgehog provides some shorthand for generating recursive data structures. This helps us write generators that terminate - if you’ve used QuickCheck heavily you’ll know why this is really useful.

Arbitrary Terms

```
genExpr :: Gen Expr
genExpr =
  Gen.recursive Gen.choice [
      -- Non-recursive generators
      EBool   <$> Gen.bool
```

So, let’s get our round-trip property going with Hedgehog.
To actually run this property, we need to be able to generate random input. We have few invariants right now, so any expression should do.
Hedgehog provides some shorthand for generating recursive data structures. This helps us write generators that terminate - if you’ve used QuickCheck heavily you’ll know why this is really useful.

Arbitrary Terms

```
genExpr :: Gen Expr
genExpr =
  Gen.recursive Gen.choice [
      -- Non-recursive generators
      EBool   <$> Gen.bool
    , EInt    <$> Gen.enumBounded
```

So, let’s get our round-trip property going with Hedgehog.
To actually run this property, we need to be able to generate random input. We have few invariants right now, so any expression should do.
Hedgehog provides some shorthand for generating recursive data structures. This helps us write generators that terminate - if you’ve used QuickCheck heavily you’ll know why this is really useful.

Arbitrary Terms

```
genExpr :: Gen Expr
genExpr =
  Gen.recursive Gen.choice [
      -- Non-recursive generators
      EBool   <$> Gen.bool
    , EInt    <$> Gen.enumBounded
    , EString <$> unicodeString
```

So, let’s get our round-trip property going with Hedgehog.
To actually run this property, we need to be able to generate random input. We have few invariants right now, so any expression should do.
Hedgehog provides some shorthand for generating recursive data structures. This helps us write generators that terminate - if you’ve used QuickCheck heavily you’ll know why this is really useful.

Arbitrary Terms

```
genExpr :: Gen Expr
genExpr =
  Gen.recursive Gen.choice [
      -- Non-recursive generators
      EBool   <$> Gen.bool
    , EInt    <$> Gen.enumBounded
    , EString <$> unicodeString
    , EVar    <$> genName
```

So, let’s get our round-trip property going with Hedgehog.
To actually run this property, we need to be able to generate random input. We have few invariants right now, so any expression should do.
Hedgehog provides some shorthand for generating recursive data structures. This helps us write generators that terminate - if you’ve used QuickCheck heavily you’ll know why this is really useful.

Arbitrary Terms

```
genExpr :: Gen Expr
genExpr =
  Gen.recursive Gen.choice [
      -- Non-recursive generators
      EBool   <$> Gen.bool
    , EInt    <$> Gen.enumBounded
    , EString <$> unicodeString
    , EVar    <$> genName
    ] [
```

So, let’s get our round-trip property going with Hedgehog.
To actually run this property, we need to be able to generate random input. We have few invariants right now, so any expression should do.
Hedgehog provides some shorthand for generating recursive data structures. This helps us write generators that terminate - if you’ve used QuickCheck heavily you’ll know why this is really useful.

Arbitrary Terms

```
genExpr :: Gen Expr
genExpr =
  Gen.recursive Gen.choice [
      -- Non-recursive generators
      EBool   <$> Gen.bool
    , EInt    <$> Gen.enumBounded
    , EString <$> unicodeString
    , EVar    <$> genName
    ] [
      -- Recursive generators
```

So, let’s get our round-trip property going with Hedgehog.
To actually run this property, we need to be able to generate random input. We have few invariants right now, so any expression should do.
Hedgehog provides some shorthand for generating recursive data structures. This helps us write generators that terminate - if you’ve used QuickCheck heavily you’ll know why this is really useful.

Arbitrary Terms

```
genExpr :: Gen Expr
genExpr =
  Gen.recursive Gen.choice [
      -- Non-recursive generators
      EBool   <$> Gen.bool
    , EInt    <$> Gen.enumBounded
    , EString <$> unicodeString
    , EVar    <$> genName
    ] [
      -- Recursive generators
      EApp    <$> genExpr <*> genExpr
```

So, let’s get our round-trip property going with Hedgehog.
To actually run this property, we need to be able to generate random input. We have few invariants right now, so any expression should do.
Hedgehog provides some shorthand for generating recursive data structures. This helps us write generators that terminate - if you’ve used QuickCheck heavily you’ll know why this is really useful.

Arbitrary Terms

```haskell
genExpr :: Gen Expr
genExpr =
  Gen.recursive Gen.choice [
      -- Non-recursive generators
      EBool   <$> Gen.bool
    , EInt    <$> Gen.enumBounded
    , EString <$> unicodeString
    , EVar    <$> genName
    ] [
      -- Recursive generators
      EApp    <$> genExpr <*> genExpr
    , ELam    <$> genName <*> genType <*> genExpr
```

So, let’s get our round-trip property going with Hedgehog.
To actually run this property, we need to be able to generate random input. We have few invariants right now, so any expression should do.
Hedgehog provides some shorthand for generating recursive data structures. This helps us write generators that terminate - if you’ve used QuickCheck heavily you’ll know why this is really useful.

Arbitrary Terms

```
genExpr :: Gen Expr
genExpr =
  Gen.recursive Gen.choice [
      -- Non-recursive generators
      EBool   <$> Gen.bool
    , EInt    <$> Gen.enumBounded
    , EString <$> unicodeString
    , EVar    <$> genName
    ] [
      -- Recursive generators
      EApp    <$> genExpr <*> genExpr
    , ELam    <$> genName <*> genType <*> genExpr
    ]
```

So, let’s get our round-trip property going with Hedgehog.
To actually run this property, we need to be able to generate random input. We have few invariants right now, so any expression should do.
Hedgehog provides some shorthand for generating recursive data structures. This helps us write generators that terminate - if you’ve used QuickCheck heavily you’ll know why this is really useful.

Arbitrary Types

Arbitrary Types

```
genType :: Gen Type
```

Arbitrary Types

```
genType :: Gen Type
genType =
```

Arbitrary Types

```
genType :: Gen Type
genType =
  Gen.recursive Gen.choice [
```

Arbitrary Types

```
genType :: Gen Type
genType =
  Gen.recursive Gen.choice [
      -- Non-recursive generators
```

Arbitrary Types

```
genType :: Gen Type
genType =
  Gen.recursive Gen.choice [
      -- Non-recursive generators
      pure TBool
```

Arbitrary Types

```
genType :: Gen Type
genType =
  Gen.recursive Gen.choice [
      -- Non-recursive generators
      pure TBool
    , pure TInt
```

Arbitrary Types

```
genType :: Gen Type
genType =
  Gen.recursive Gen.choice [
      -- Non-recursive generators
      pure TBool
    , pure TInt
    , pure TString
```

Arbitrary Types

```
genType :: Gen Type
genType =
  Gen.recursive Gen.choice [
      -- Non-recursive generators
      pure TBool
    , pure TInt
    , pure TString
    ] [
```

Arbitrary Types

```
genType :: Gen Type
genType =
  Gen.recursive Gen.choice [
      -- Non-recursive generators
      pure TBool
    , pure TInt
    , pure TString
    ] [
      -- Recursive generators
```

Arbitrary Types

```
genType :: Gen Type
genType =
  Gen.recursive Gen.choice [
      -- Non-recursive generators
      pure TBool
    , pure TInt
    , pure TString
    ] [
      -- Recursive generators
     TArrow <$> genType <*> genType
```

Arbitrary Types

```
genType :: Gen Type
genType =
  Gen.recursive Gen.choice [
      -- Non-recursive generators
      pure TBool
    , pure TInt
    , pure TString
    ] [
      -- Recursive generators
     TArrow <$> genType <*> genType
    ]
```

Round Tripping

When it comes time to actually write the property, it's pretty straightforward: just generate your value and write it in verbatim.
Hedgehog's **tripping** function provides really nice shiny output when the property fails, so you should use that where possible. This improves the counterexample reporting quite a bit.

Round Tripping

```
-- | Test our parser and printer are aligned.
```

When it comes time to actually write the property, it’s pretty straightforward: just generate your value and write it in verbatim.
Hedgehog’s **tripping** function provides really nice shiny output when the property fails, so you should use that where possible. This improves the counterexample reporting quite a bit.

Round Tripping

```
-- | Test our parser and printer are aligned.
prop_expr_round_trip :: Property
```

When it comes time to actually write the property, it’s pretty straightforward: just generate your value and write it in verbatim.
Hedgehog’s **tripping** function provides really nice shiny output when the property fails, so you should use that where possible. This improves the counterexample reporting quite a bit.

Round Tripping

```haskell
-- | Test our parser and printer are aligned.
prop_expr_round_trip :: Property
prop_expr_round_trip =
```

When it comes time to actually write the property, it's pretty straightforward: just generate your value and write it in verbatim.
Hedgehog's **tripping** function provides really nice shiny output when the property fails, so you should use that where possible. This improves the counterexample reporting quite a bit.

Round Tripping

```
-- | Test our parser and printer are aligned.
prop_expr_round_trip :: Property
prop_expr_round_trip =
  property $ do
```

When it comes time to actually write the property, it's pretty straightforward: just generate your value and write it in verbatim.
Hedgehog's **tripping** function provides really nice shiny output when the property fails, so you should use that where possible. This improves the counterexample reporting quite a bit.

Round Tripping

```haskell
-- | Test our parser and printer are aligned.
prop_expr_round_trip :: Property
prop_expr_round_trip =
  property $ do
    expr <- forAll Arbitrary.genExpr
```

When it comes time to actually write the property, it’s pretty straightforward: just generate your value and write it in verbatim.
Hedgehog’s **tripping** function provides really nice shiny output when the property fails, so you should use that where possible. This improves the counterexample reporting quite a bit.

Round Tripping

```haskell
-- | Test our parser and printer are aligned.
prop_expr_round_trip :: Property
prop_expr_round_trip =
  property $ do
    expr <- forAll Arbitrary.genExpr
    STLC.parse (Pretty.ppExpr expr) === pure expr
```

When it comes time to actually write the property, it’s pretty straightforward: just generate your value and write it in verbatim.
Hedgehog’s **tripping** function provides really nice shiny output when the property fails, so you should use that where possible. This improves the counterexample reporting quite a bit.

Round Tripping

```haskell
-- | Test our parser and printer are aligned.
prop_expr_round_trip :: Property
prop_expr_round_trip =
  property $ do
    expr <- forAll Arbitrary.genExpr
    STLC.parse (Pretty.ppExpr expr) === pure expr
```

When it comes time to actually write the property, it's pretty straightforward: just generate your value and write it in verbatim.
Hedgehog's **tripping** function provides really nice shiny output when the property fails, so you should use that where possible. This improves the counterexample reporting quite a bit.

Round Tripping

```haskell
-- | Test our parser and printer are aligned.
prop_expr_round_trip :: Property
prop_expr_round_trip =
  property $ do
    expr <- forAll Arbitrary.genExpr
    STLC.parse (Pretty.ppExpr expr) === pure expr
    tripping expr Pretty.ppExpr STLC.parse
```

When it comes time to actually write the property, it's pretty straightforward: just generate your value and write it in verbatim.
Hedgehog's **tripping** function provides really nice shiny output when the property fails, so you should use that where possible. This improves the counterexample reporting quite a bit.

Bugs Eaten

```
λ> check prop_expr_round_trip
  ✗ <interactive> failed after 1 test and 1 shrink.

    forAll0 =
      EBool False

    ━━━ Intermediate ━━━
    "False"
    ━━━ - Original / + Roundtrip ━━━
    - Right (EBool False)
    + Right (EVar Name { unName = "False" })

    This failure can be reproduced by running:
    > recheck (Size 0) (Seed 8657468963917336963 (-1842722639900274453)) <property>

False
```

The round trip immediately picks up a ton of subtle ambiguities.
In the screenshot, you'll see I'm making a classic parsing mistake - I'm confusing a built-in value (False) for a variable name.
If I'd prepared my talk properly I'd have a hundred of these. This property is **extremely effective** when interactively developing a parser / printer pair.

Golden Testing

We're now a little bit more confident that our parser and printer are consistently implementing the **same** language! We don't know **which** language.
If properties pass, we **never see our generated programs**. We still need some unit tests to pin the specification down.
The laziest technique I know of for this is called Golden Testing / Golden Master Testing. This means **checking in the input and expected output**.
This ensures we have visual examples of our program at work, and **hedges against accidental changes** in the future.

Golden Testing

(You still need unit tests)

We're now a little bit more confident that our parser and printer are consistently implementing the **same** language! We don't know **which** language.
If properties pass, we **never see our generated programs**. We still need some unit tests to pin the specification down.
The laziest technique I know of for this is called Golden Testing / Golden Master Testing. This means **checking in the input and expected output**.
This ensures we have visual examples of our program at work, and **hedges against accidental changes** in the future.

Golden Testing

(You still need unit tests)

- Come up with a decent (representative?) corpus somehow

We're now a little bit more confident that our parser and printer are consistently implementing the **same** language! We don't know **which** language.
If properties pass, we **never see our generated programs**. We still need some unit tests to pin the specification down.
The laziest technique I know of for this is called Golden Testing / Golden Master Testing. This means **checking in the input and expected output**.
This ensures we have visual examples of our program at work, and **hedges against accidental changes** in the future.

Golden Testing

(You still need unit tests)

- Come up with a decent (representative?) corpus somehow
  - By hand – not the worst thing in the world

We're now a little bit more confident that our parser and printer are consistently implementing the **same** language! We don't know **which** language.
If properties pass, we **never see our generated programs**. We still need some unit tests to pin the specification down.
The laziest technique I know of for this is called Golden Testing / Golden Master Testing. This means **checking in the input and expected output**.
This ensures we have visual examples of our program at work, and **hedges against accidental changes** in the future.

Golden Testing

(You still need unit tests)

- Come up with a decent (representative?) corpus somehow
  - By hand – not the worst thing in the world
  - Run your generator a bunch

We're now a little bit more confident that our parser and printer are consistently implementing the **same** language! We don't know **which** language.
If properties pass, we **never see our generated programs**. We still need some unit tests to pin the specification down.
The laziest technique I know of for this is called Golden Testing / Golden Master Testing. This means **checking in the input and expected output**.
This ensures we have visual examples of our program at work, and **hedges against accidental changes** in the future.

Golden Testing

(You still need unit tests)

- Come up with a decent (representative?) corpus somehow
  - By hand – not the worst thing in the world
  - Run your generator a bunch
  - Build it over time whenever your properties break!

We're now a little bit more confident that our parser and printer are consistently implementing the **same** language! We don't know **which** language.
If properties pass, we **never see our generated programs**. We still need some unit tests to pin the specification down.
The laziest technique I know of for this is called Golden Testing / Golden Master Testing. This means **checking in the input and expected output**.
This ensures we have visual examples of our program at work, and **hedges against accidental changes** in the future.

Golden Testing

(You still need unit tests)

- Come up with a decent (representative?) corpus somehow
  - By hand – not the worst thing in the world
  - Run your generator a bunch
  - Build it over time whenever your properties break!
- Check in the input and expected output (pretty-show!)

We're now a little bit more confident that our parser and printer are consistently implementing the **same** language! We don't know **which** language.
If properties pass, we **never see our generated programs**. We still need some unit tests to pin the specification down.
The laziest technique I know of for this is called Golden Testing / Golden Master Testing. This means **checking in the input and expected output**.
This ensures we have visual examples of our program at work, and **hedges against accidental changes** in the future.

Golden Testing

(You still need unit tests)

- Come up with a decent (representative?) corpus somehow
  - By hand – not the worst thing in the world
  - Run your generator a bunch
  - Build it over time whenever your properties break!
- Check in the input and expected output (pretty-show!)
- Test that they always line up

We're now a little bit more confident that our parser and printer are consistently implementing the **same** language! We don't know **which** language.
If properties pass, we **never see our generated programs**. We still need some unit tests to pin the specification down.
The laziest technique I know of for this is called Golden Testing / Golden Master Testing. This means **checking in the input and expected output**.
This ensures we have visual examples of our program at work, and **hedges against accidental changes** in the future.

Golden Testing

(You still need unit tests)

- Come up with a decent (representative?) corpus somehow
  - By hand – not the worst thing in the world
  - Run your generator a bunch
  - Build it over time whenever your properties break!
- Check in the input and expected output (pretty-show!)
- Test that they always line up
- Unexpected changes are visible and reviewable

We're now a little bit more confident that our parser and printer are consistently implementing the **same** language! We don't know **which** language.
If properties pass, we **never see our generated programs**. We still need some unit tests to pin the specification down.
The laziest technique I know of for this is called Golden Testing / Golden Master Testing. This means **checking in the input and expected output**.
This ensures we have visual examples of our program at work, and **hedges against accidental changes** in the future.

brace.in	Update regression tests
brace.out	Whitespace-sensitive HTML blocks
case.in	Support pattern matching on _
case.out	Support pattern matching on _
case_html.in	Update regression tests
case_html.out	Whitespace-sensitive HTML blocks
conid.in	Update regression tests
conid.out	Update regression tests
html.in	Update regression tests
html.out	Whitespace-sensitive HTML blocks
indent_hell.in	Update regression tests
indent_hell.out	Whitespace-sensitive HTML blocks
indent_tag.in	Update tests
indent_tag.out	Whitespace-sensitive HTML blocks

There's a library called **tasty-golden** for this, but I tend to roll it myself using QuickCheck or Hedgehog or whatever I'm using at the time.
Now it's a lot harder to break user code. We can see what language we're implementing.
We're insulated from one form of catastrophe.
This gives us breathing room to further extend the syntax in interesting ways.

Bugs Eaten

- Lexer: Holding Parsec wrong, int truncation
- Parser: precedence / ambiguity
- Printer: ambiguity
- Frontend: backward-incompatible changes

Indeed, this property is remarkably effective; I wish I had more examples prepared. It works really, stupidly well. Here are a bunch of things off the top of my head.

Type Checking

```
typeCheck :: Expr -> Either TypeError Type
typeCheck = …
```

We've now sorted out our user interface, we're pretty confident it works the way we intended.
Now we need to **validate** the user's input!
For our current language, that means **type checking**, which involves a type signature like **this**.
We still don't really want to write unit tests. It's **hard to anticipate** where the edge cases will lie.
We want **as much automatic coverage as possible.**

Type Checking

```
typeCheck :: Expr -> Either TypeError Type
typeCheck = …
```

So, let’s think about some useful properties.
We already have an arbitrary expression generator, so maybe we could build a property out of that?
Once we generate an arbitrary term, what can we assert about it? Not much!
(We could filter the things it generates, but that would be really inefficient - it’s best to constructively generate the things we actually want to test.)

Type Checking

```
typeCheck :: Expr -> Either TypeError Type
typeCheck = …
```

```
prop_typecheck :: Property
prop_typecheck =
  property $ do
    ex <- forAll Arbitrary.genExpr
    ???
```

So, let’s think about some useful properties.
We already have an arbitrary expression generator, so maybe we could build a property out of that?
Once we generate an arbitrary term, what can we assert about it? Not much!
(We could filter the things it generates, but that would be really inefficient - it’s best to constructively generate the things we actually want to test.)

Type Checking

```
typeCheck :: Expr -> Either TypeError Type
typeCheck = …
```

```
prop_typecheck :: Property
prop_typecheck =
  property $ do
    ~~ex <- forAll Arbitrary.genExpr~~
    ???
```

So, let’s think about some useful properties.
We already have an arbitrary expression generator, so maybe we could build a property out of that?
Once we generate an arbitrary term, what can we assert about it? Not much!
(We could filter the things it generates, but that would be really inefficient - it’s best to constructively generate the things we actually want to test.)

Type Checking

```
typeCheck :: Expr -> Either TypeError Type
typeCheck = …
```

So, let’s split the domain into two: well-typed expressions, and ill-typed expressions!
Suppose we have a generator for well-typed expressions, with this type signature.
If we generate a well-formed term with a known type, typechecking should produce that type!

Type Checking

```
typeCheck :: Expr -> Either TypeError Type
typeCheck = …

genWellTypedExpr :: Type -> Gen Expr
genWellTypedExpr = …
```

So, let’s split the domain into two: well-typed expressions, and ill-typed expressions!
Suppose we have a generator for well-typed expressions, with this type signature.
If we generate a well-formed term with a known type, typechecking should produce that type!

Type Checking

```
typeCheck :: Expr -> Either TypeError Type
typeCheck = …

genWellTypedExpr :: Type -> Gen Expr
genWellTypedExpr = …

prop_typecheck :: Property
```

So, let’s split the domain into two: well-typed expressions, and ill-typed expressions!
Suppose we have a generator for well-typed expressions, with this type signature.
If we generate a well-formed term with a known type, typechecking should produce that type!

Type Checking

```
typeCheck :: Expr -> Either TypeError Type
typeCheck = …

genWellTypedExpr :: Type -> Gen Expr
genWellTypedExpr = …

prop_typecheck :: Property
prop_typecheck =
```

So, let’s split the domain into two: well-typed expressions, and ill-typed expressions!
Suppose we have a generator for well-typed expressions, with this type signature.
If we generate a well-formed term with a known type, typechecking should produce that type!

Type Checking

```
typeCheck :: Expr -> Either TypeError Type
typeCheck = …

genWellTypedExpr :: Type -> Gen Expr
genWellTypedExpr = …

prop_typecheck :: Property
prop_typecheck =
  property $ do
```

So, let’s split the domain into two: well-typed expressions, and ill-typed expressions!
Suppose we have a generator for well-typed expressions, with this type signature.
If we generate a well-formed term with a known type, typechecking should produce that type!

Type Checking

```
typeCheck :: Expr -> Either TypeError Type
typeCheck = …

genWellTypedExpr :: Type -> Gen Expr
genWellTypedExpr = …

prop_typecheck :: Property
prop_typecheck =
  property $ do
    ty <- forAll Arbitrary.genType
```

So, let’s split the domain into two: well-typed expressions, and ill-typed expressions!
Suppose we have a generator for well-typed expressions, with this type signature.
If we generate a well-formed term with a known type, typechecking should produce that type!

Type Checking

```
typeCheck :: Expr -> Either TypeError Type
typeCheck = …

genWellTypedExpr :: Type -> Gen Expr
genWellTypedExpr = …

prop_typecheck :: Property
prop_typecheck =
  property $ do
    ty <- forAll Arbitrary.genType
    ex <- forAll (genWellTypedExpr ty)
```

So, let’s split the domain into two: well-typed expressions, and ill-typed expressions!
Suppose we have a generator for well-typed expressions, with this type signature.
If we generate a well-formed term with a known type, typechecking should produce that type!

Type Checking

```
typeCheck :: Expr -> Either TypeError Type
typeCheck = …

genWellTypedExpr :: Type -> Gen Expr
genWellTypedExpr = …

prop_typecheck :: Property
prop_typecheck =
  property $ do
    ty <- forAll Arbitrary.genType
    ex <- forAll (genWellTypedExpr ty)
    typeCheck ex === pure ty
```

So, let’s split the domain into two: well-typed expressions, and ill-typed expressions!
Suppose we have a generator for well-typed expressions, with this type signature.
If we generate a well-formed term with a known type, typechecking should produce that type!

Type Checking

```
genWellTypedExpr :: Type -> Gen Expr
genWellTypedExpr = …
```

This might seem like a hard problem, but — and this is the whole point of my talk — doing it as naively as possible in Hedgehog leads to really good results!
STLC is really simple, so our initial generator is pretty straightforward. Let's break it down into parts first.
For most languages you can get away with a fairly straightforward structure: a way to **generate simple expressions**, a **search procedure** for expressions in scope, and a way to **generate larger recursive terms**.

Type Checking

```
genWellTypedExpr :: Type -> Gen Expr
genWellTypedExpr = …
```

```
genNewExpr        :: Type ->                Gen Expr
```

This might seem like a hard problem, but — and this is the whole point of my talk — doing it as naively as possible in Hedgehog leads to really good results!
STLC is really simple, so our initial generator is pretty straightforward. Let's break it down into parts first.
For most languages you can get away with a fairly straightforward structure: a way to **generate simple expressions**, a **search procedure** for expressions in scope, and a way to **generate larger recursive terms**.

Type Checking

```
genWellTypedExpr :: Type -> Gen Expr
genWellTypedExpr = …
```

```
genNewExpr       :: Type ->              Gen Expr
genKnownExpr     :: Type ->              Gen Expr
```

This might seem like a hard problem, but — and this is the whole point of my talk — doing it as naively as possible in Hedgehog leads to really good results!
STLC is really simple, so our initial generator is pretty straightforward. Let's break it down into parts first.
For most languages you can get away with a fairly straightforward structure: a way to **generate simple expressions**, a **search procedure** for expressions in scope, and a way to **generate larger recursive terms**.

Type Checking

```
genWellTypedExpr :: Type -> Gen Expr
genWellTypedExpr = …
```

```
genNewExpr       :: Type ->              Gen Expr
genKnownExpr     :: Type ->              Gen Expr
genRecursiveExpr :: Type ->              Gen Expr
```

This might seem like a hard problem, but — and this is the whole point of my talk — doing it as naively as possible in Hedgehog leads to really good results!
STLC is really simple, so our initial generator is pretty straightforward. Let's break it down into parts first.
For most languages you can get away with a fairly straightforward structure: a way to **generate simple expressions**, a **search procedure** for expressions in scope, and a way to **generate larger recursive terms**.

Type Checking

```
genWellTypedExpr :: Type -> Gen Expr
genWellTypedExpr =
```

```
genExpr'          :: Type ->          Gen Expr
genNewExpr        :: Type ->          Gen Expr
genKnownExpr      :: Type ->          Gen Expr
genRecursiveExpr  :: Type ->          Gen Expr
```

This might seem like a hard problem, but — and this is the whole point of my talk — doing it as naively as possible in Hedgehog leads to really good results!
STLC is really simple, so our initial generator is pretty straightforward. Let's break it down into parts first.
For most languages you can get away with a fairly straightforward structure: a way to **generate simple expressions**, a **search procedure** for expressions in scope, and a way to **generate larger recursive terms**.

Type Checking

```
genWellTypedExpr :: Type -> Gen Expr
genWellTypedExpr =
```

```
type Env = Map Type [Expr]

genExpr'          :: Type -> ReaderT Env Gen Expr
genNewExpr        :: Type -> ReaderT Env Gen Expr
genKnownExpr      :: Type -> ReaderT Env Gen Expr
genRecursiveExpr :: Type -> ReaderT Env Gen Expr
```

This might seem like a hard problem, but — and this is the whole point of my talk — doing it as naively as possible in Hedgehog leads to really good results!
STLC is really simple, so our initial generator is pretty straightforward. Let's break it down into parts first.
For most languages you can get away with a fairly straightforward structure: a way to **generate simple expressions**, a **search procedure** for expressions in scope, and a way to **generate larger recursive terms**.

Type Checking

```
genWellTypedExpr :: Type -> Gen Expr
genWellTypedExpr =
  flip runReaderT M.empty . genExpr'

type Env = Map Type [Expr]

genExpr'          :: Type -> ReaderT Env Gen Expr
genNewExpr        :: Type -> ReaderT Env Gen Expr
genKnownExpr      :: Type -> ReaderT Env Gen Expr
genRecursiveExpr :: Type -> ReaderT Env Gen Expr
```

This might seem like a hard problem, but — and this is the whole point of my talk — doing it as naively as possible in Hedgehog leads to really good results!
STLC is really simple, so our initial generator is pretty straightforward. Let's break it down into parts first.
For most languages you can get away with a fairly straightforward structure: a way to **generate simple expressions**, a **search procedure** for expressions in scope, and a way to **generate larger recursive terms**.

Simple Expressions

Generating simple expressions is easy, we've already done it for our Arbitrary generator.
This time, we just have to look at the type.
When generating a lambda, we have to put our bound variable into the Env - to make sure later generators can see it's in scope.

Simple Expressions

```
-- | Build a new simple expression from literals.
```

Generating simple expressions is easy, we've already done it for our Arbitrary generator.
This time, we just have to look at the type.
When generating a lambda, we have to put our bound variable into the Env - to make sure later generators can see it's in scope.

Simple Expressions

```haskell
-- | Build a new simple expression from literals.
genNewExpr :: Type -> ReaderT Env Gen Expr
```

Generating simple expressions is easy, we've already done it for our Arbitrary generator.
This time, we just have to look at the type.
When generating a lambda, we have to put our bound variable into the Env - to make sure later generators can see it's in scope.

Simple Expressions

```haskell
-- | Build a new simple expression from literals.
genNewExpr :: Type -> ReaderT Env Gen Expr
genNewExpr want = do
```

Generating simple expressions is easy, we've already done it for our Arbitrary generator.
This time, we just have to look at the type.
When generating a lambda, we have to put our bound variable into the Env - to make sure later generators can see it's in scope.

Simple Expressions

```haskell
-- | Build a new simple expression from literals.
genNewExpr :: Type -> ReaderT Env Gen Expr
genNewExpr want = do
  case want of
```

Generating simple expressions is easy, we've already done it for our Arbitrary generator.
This time, we just have to look at the type.
When generating a lambda, we have to put our bound variable into the Env - to make sure later generators can see it's in scope.

Simple Expressions

```haskell
-- | Build a new simple expression from literals.
genNewExpr :: Type -> ReaderT Env Gen Expr
genNewExpr want = do
  case want of
    TBool   -> EBool   <$> Gen.bool
```

Generating simple expressions is easy, we've already done it for our Arbitrary generator.
This time, we just have to look at the type.
When generating a lambda, we have to put our bound variable into the Env - to make sure later generators can see it's in scope.

Simple Expressions

```haskell
-- | Build a new simple expression from literals.
genNewExpr :: Type -> ReaderT Env Gen Expr
genNewExpr want = do
  case want of
    TBool   -> EBool   <$> Gen.bool
    TInt    -> EInt    <$> Gen.enumBounded
```

Generating simple expressions is easy, we've already done it for our Arbitrary generator.
This time, we just have to look at the type.
When generating a lambda, we have to put our bound variable into the Env - to make sure later generators can see it's in scope.

Simple Expressions

```haskell
-- | Build a new simple expression from literals.
genNewExpr :: Type -> ReaderT Env Gen Expr
genNewExpr want = do
  case want of
    TBool   -> EBool   <$> Gen.bool
    TInt    -> EInt    <$> Gen.enumBounded
    TString -> EString <$> genString
```

Generating simple expressions is easy, we've already done it for our Arbitrary generator.
This time, we just have to look at the type.
When generating a lambda, we have to put our bound variable into the Env - to make sure later generators can see it's in scope.

Simple Expressions

```haskell
-- | Build a new simple expression from literals.
genNewExpr :: Type -> ReaderT Env Gen Expr
genNewExpr want = do
  case want of
    TBool   -> EBool   <$> Gen.bool
    TInt    -> EInt    <$> Gen.enumBounded
    TString -> EString <$> genString
    TArrow t1 t2 -> do
```

Generating simple expressions is easy, we've already done it for our Arbitrary generator.
This time, we just have to look at the type.
When generating a lambda, we have to put our bound variable into the Env - to make sure later generators can see it's in scope.

Simple Expressions

```haskell
-- | Build a new simple expression from literals.
genNewExpr :: Type -> ReaderT Env Gen Expr
genNewExpr want = do
  case want of
    TBool   -> EBool   <$> Gen.bool
    TInt    -> EInt    <$> Gen.enumBounded
    TString -> EString <$> genString
    TArrow t1 t2 -> do
      x <- Arbitrary.genName
```

Generating simple expressions is easy, we've already done it for our Arbitrary generator.
This time, we just have to look at the type.
When generating a lambda, we have to put our bound variable into the Env - to make sure later generators can see it's in scope.

Simple Expressions

```haskell
-- | Build a new simple expression from literals.
genNewExpr :: Type -> ReaderT Env Gen Expr
genNewExpr want = do
  case want of
    TBool   -> EBool   <$> Gen.bool
    TInt    -> EInt    <$> Gen.enumBounded
    TString -> EString <$> genString
    TArrow t1 t2 -> do
      x <- Arbitrary.genName
      ELam x t1
```

Generating simple expressions is easy, we've already done it for our Arbitrary generator.
This time, we just have to look at the type.
When generating a lambda, we have to put our bound variable into the Env - to make sure later generators can see it's in scope.

Simple Expressions

```haskell
-- | Build a new simple expression from literals.
genNewExpr :: Type -> ReaderT Env Gen Expr
genNewExpr want = do
  case want of
    TBool   -> EBool   <$> Gen.bool
    TInt    -> EInt    <$> Gen.enumBounded
    TString -> EString <$> genString
    TArrow t1 t2 -> do
      x <- Arbitrary.genName
      ELam x t1
        <$> local (addToEnv (EVar x) t1) (genExpr' t2)
```

Generating simple expressions is easy, we've already done it for our Arbitrary generator.
This time, we just have to look at the type.
When generating a lambda, we have to put our bound variable into the Env - to make sure later generators can see it's in scope.

Searching

Searching for known expressions is easy, as long as we've done the proper bookkeeping.
Just look up the type in the env, make a non-deterministic choice.
Otherwise, let the generator fail. Unlike QuickCheck, Hedgehog's Gen has an Alternative instance!

Searching

```
genKnownExpr :: Type -> ReaderT Env Gen Expr
genKnownExpr want = do
  env <- ask
  case findInEnv want env of
    Just exprs ->
      Gen.element exprs {- non-deterministic choice -}
    Nothing ->
      Gen.discard       {- fail via Alternative instance -}
```

Searching for known expressions is easy, as long as we've done the proper bookkeeping.
Just look up the type in the env, make a non-deterministic choice.
Otherwise, let the generator fail. Unlike QuickCheck, Hedgehog's Gen has an Alternative instance!

Recursive Expressions

Generating larger recursive expressions is also pretty straightforward!
We only have one recursive constructor right now, and it's for function application.
Proceed simply: Generate a function, and apply it to a value of the appropriate type.
Extend this function later as you add more features to the language.

Recursive Expressions

```
genRecursiveExpr :: Type -> ReaderT Env Gen Expr
```

Generating larger recursive expressions is also pretty straightforward!
We only have one recursive constructor right now, and it's for function application.
Proceed simply: Generate a function, and apply it to a value of the appropriate type.
Extend this function later as you add more features to the language.

Recursive Expressions

```
genRecursiveExpr :: Type -> ReaderT Env Gen Expr
genRecursiveExpr =
```

Generating larger recursive expressions is also pretty straightforward!
We only have one recursive constructor right now, and it's for function application.
Proceed simply: Generate a function, and apply it to a value of the appropriate type.
Extend this function later as you add more features to the language.

Recursive Expressions

```
genRecursiveExpr :: Type -> ReaderT Env Gen Expr
genRecursiveExpr =
  genApp
```

Generating larger recursive expressions is also pretty straightforward!
We only have one recursive constructor right now, and it's for function application.
Proceed simply: Generate a function, and apply it to a value of the appropriate type.
Extend this function later as you add more features to the language.

Recursive Expressions

```
genRecursiveExpr :: Type -> ReaderT Env Gen Expr
genRecursiveExpr =
  genApp
```

Generating larger recursive expressions is also pretty straightforward!
We only have one recursive constructor right now, and it's for function application.
Proceed simply: Generate a function, and apply it to a value of the appropriate type.
Extend this function later as you add more features to the language.

Recursive Expressions

```haskell
genRecursiveExpr :: Type -> ReaderT Env Gen Expr
genRecursiveExpr =
  genApp

-- | Generate a function application.
```

Generating larger recursive expressions is also pretty straightforward!
We only have one recursive constructor right now, and it's for function application.
Proceed simply: Generate a function, and apply it to a value of the appropriate type.
Extend this function later as you add more features to the language.

Recursive Expressions

```haskell
genRecursiveExpr :: Type -> ReaderT Env Gen Expr
genRecursiveExpr =
  genApp

-- | Generate a function application.
genApp :: Type -> ReaderT Env Gen Expr
```

Generating larger recursive expressions is also pretty straightforward!
We only have one recursive constructor right now, and it's for function application.
Proceed simply: Generate a function, and apply it to a value of the appropriate type.
Extend this function later as you add more features to the language.

Recursive Expressions

```haskell
genRecursiveExpr :: Type -> ReaderT Env Gen Expr
genRecursiveExpr =
  genApp

-- | Generate a function application.
genApp :: Type -> ReaderT Env Gen Expr
genApp want = do
```

Generating larger recursive expressions is also pretty straightforward!
We only have one recursive constructor right now, and it's for function application.
Proceed simply: Generate a function, and apply it to a value of the appropriate type.
Extend this function later as you add more features to the language.

Recursive Expressions

```haskell
genRecursiveExpr :: Type -> ReaderT Env Gen Expr
genRecursiveExpr =
  genApp

-- | Generate a function application.
genApp :: Type -> ReaderT Env Gen Expr
genApp want = do
  tg <- genType
```

Generating larger recursive expressions is also pretty straightforward!
We only have one recursive constructor right now, and it's for function application.
Proceed simply: Generate a function, and apply it to a value of the appropriate type.
Extend this function later as you add more features to the language.

Recursive Expressions

```haskell
genRecursiveExpr :: Type -> ReaderT Env Gen Expr
genRecursiveExpr =
  genApp

-- | Generate a function application.
genApp :: Type -> ReaderT Env Gen Expr
genApp want = do
  tg <- genType
  eg <- genExpr' tg
```

Generating larger recursive expressions is also pretty straightforward!
We only have one recursive constructor right now, and it's for function application.
Proceed simply: Generate a function, and apply it to a value of the appropriate type.
Extend this function later as you add more features to the language.

Recursive Expressions

```haskell
genRecursiveExpr :: Type -> ReaderT Env Gen Expr
genRecursiveExpr =
  genApp

-- | Generate a function application.
genApp :: Type -> ReaderT Env Gen Expr
genApp want = do
  tg <- genType
  eg <- genExpr' tg
  let tf = TArrow tg want
```

Generating larger recursive expressions is also pretty straightforward!
We only have one recursive constructor right now, and it's for function application.
Proceed simply: Generate a function, and apply it to a value of the appropriate type.
Extend this function later as you add more features to the language.

Recursive Expressions

```haskell
genRecursiveExpr :: Type -> ReaderT Env Gen Expr
genRecursiveExpr =
  genApp

-- | Generate a function application.
genApp :: Type -> ReaderT Env Gen Expr
genApp want = do
  tg <- genType
  eg <- genExpr' tg
  let tf = TArrow tg want
  ef <- genExpr' tf
```

Generating larger recursive expressions is also pretty straightforward!
We only have one recursive constructor right now, and it's for function application.
Proceed simply: Generate a function, and apply it to a value of the appropriate type.
Extend this function later as you add more features to the language.

Recursive Expressions

```haskell
genRecursiveExpr :: Type -> ReaderT Env Gen Expr
genRecursiveExpr =
  genApp

-- | Generate a function application.
genApp :: Type -> ReaderT Env Gen Expr
genApp want = do
  tg <- genType
  eg <- genExpr' tg
  let tf = TArrow tg want
  ef <- genExpr' tf
  pure (EApp ef eg)
```

Generating larger recursive expressions is also pretty straightforward!
We only have one recursive constructor right now, and it's for function application.
Proceed simply: Generate a function, and apply it to a value of the appropriate type.
Extend this function later as you add more features to the language.

Well-Typed Terms

```
genWellTypedExpr :: Type -> Gen Expr
genWellTypedExpr =
  flip runReaderT M.empty . genExpr'

genExpr' :: Type -> ReaderT Env Gen Expr
genExpr' want =
    Gen.recursive Gen.choice [
      -- Non-recursive generators
      genNewExpr want
    ] [
      -- Recursive generators
      genKnownExpr want <|> genRecursiveExpr want
    , genRecursiveExpr want
    ]
```

Now we just need to stitch it all together.
We proceed as before, using Hedgehog's recursive combinator to handle sizes for us.
Recall that genKnownExpr (our search procedure) could discard. Unless we provide an alternative generator, Hedgehog will give up and start again from the top. So, when it fails, we generate a recursive expression instead. Discards can be problematic.

Bugs Eaten

```
prop_typecheck :: Property
prop_typecheck =
  property $ do
    ty <- forAll Arbitrary.genType
    ex <- forAll (genWellTypedExpr ty)
    typeCheck ex === pure ty
```

When we run this property, we get really nice concise counterexamples **that are well typed, even after shrinking.** This is really nice. **The shrinking is free**.
The above expression is a Bool, but my initial typechecker had some flipped arguments when checking function types, leading to a spurious KindMismatch.

Bugs Eaten

```
λ> check prop_check_welltyped
  ✗ <interactive> failed after 3 tests and 5 shrinks.

    forAll0 =
      EApp
        (ELam Name { unName = "aa" } TBool (EBool False)) (EBool False)

    forAll1 =
      TBool

    Failed (- lhs =/= + rhs)
    - Left KindMismatch
    + Right TBool

    This failure can be reproduced by running:
    > recheck (Size 2) (Seed 4334636612275502147 9190120952592362849) <property>
```

```
prop_typecheck :: Property
prop_typecheck =
  property $ do
    ty <- forAll Arbitrary.genType
    ex <- forAll (genWellTypedExpr ty)
    typeCheck ex === pure ty
```

When we run this property, we get really nice concise counterexamples **that are well typed, even after shrinking.** This is really nice. **The shrinking is free**.
The above expression is a Bool, but my initial typechecker had some flipped arguments when checking function types, leading to a spurious KindMismatch.

Bugs Eaten

```
λ> check prop_check_welltyped
  ✗ <interactive> failed after 3 tests and 5 shrinks.

    forAll0 =
      EApp
        (ELam Name { unName = "aa" } TBool (EBool False)) (EBool False)

    forAll1 =
      TBool

    Failed (- lhs =/= + rhs)
    - Left KindMismatch
    + Right TBool

    This failure can be reproduced by running:
    > recheck (Size 2) (Seed 4334636612275502147 9190120952592362849) <property>
```

Well-typed shrink!

```
prop_typecheck :: Property
prop_typecheck =
  property $ do
    ty <- forAll Arbitrary.genType
    ex <- forAll (genWellTypedExpr ty)
    typeCheck ex === pure ty
```

When we run this property, we get really nice concise counterexamples **that are well typed, even after shrinking.** This is really nice. **The shrinking is free**.
The above expression is a Bool, but my initial typechecker had some flipped arguments when checking function types, leading to a spurious KindMismatch.

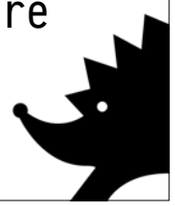
Type Checking

```
typeCheck :: Expr -> Either TypeError Type
genIllTypedExpr :: Gen Expr

prop_typecheck_fails :: Property
prop_typecheck_fails =
  property $ do
    ex <- forAll genIllTypedExpr
    ???
```

That’s our positive property sorted. We split the domain in two earlier; what about the other half?
Suppose we had a generator for ill-typed terms. What kind of properties could we write with it?

Type Checking

```
typeCheck :: Expr -> Either TypeError Type
genIllTypedExpr :: Gen Expr

prop_typecheck_fails :: Property
prop_typecheck_fails =
  property $ do
    ex <- forAll genIllTypedExpr
    case Check.typeCheck ex of
      Left  _  -> success
      Right ty -> annotateShow ty *> failure
```



That’s our positive property sorted. We split the domain in two earlier; what about the other half?
Suppose we had a generator for ill-typed terms. What kind of properties could we write with it?
A pretty straightforward one comes to mind: an ill-typed term should not pass typechecking.
This is pretty straightforward to express, though Hedgehog doesn’t have primitives for expected failures.

Ill-Typed Terms

Think carefully about the ways to produce type errors in this simple language.
There's only one way: **apply a function** to an argument with the wrong type.
This is pretty easy to generate! Just produce a lambda, make sure its type annotation doesn't line up with its argument.
There's one other way - we could just generate free variables. Should be doing that here, just ran out of time.

Ill-Typed Terms

```
-- | Generates an ill-typed function application.
```

Think carefully about the ways to produce type errors in this simple language.
There's only one way: **apply a function** to an argument with the wrong type.
This is pretty easy to generate! Just produce a lambda, make sure its type annotation doesn't line up with its argument.
There's one other way - we could just generate free variables. Should be doing that here, just ran out of time.

Ill-Typed Terms

```
-- | Generates an ill-typed function application.
genApp :: Gen Expr
```

Think carefully about the ways to produce type errors in this simple language.
There's only one way: **apply a function** to an argument with the wrong type.
This is pretty easy to generate! Just produce a lambda, make sure its type annotation doesn't line up with its argument.
There's one other way - we could just generate free variables. Should be doing that here, just ran out of time.

Ill-Typed Terms

```
-- | Generates an ill-typed function application.
genApp :: Gen Expr
genApp = do
```

Think carefully about the ways to produce type errors in this simple language.
There's only one way: **apply a function** to an argument with the wrong type.
This is pretty easy to generate! Just produce a lambda, make sure its type annotation doesn't line up with its argument.
There's one other way - we could just generate free variables. Should be doing that here, just ran out of time.

Ill-Typed Terms

```
-- | Generates an ill-typed function application.
genApp :: Gen Expr
genApp = do
  t1 <- Arbitrary.genType
```

Think carefully about the ways to produce type errors in this simple language.
There's only one way: **apply a function** to an argument with the wrong type.
This is pretty easy to generate! Just produce a lambda, make sure its type annotation doesn't line up with its argument.
There's one other way - we could just generate free variables. Should be doing that here, just ran out of time.

Ill-Typed Terms

```
-- | Generates an ill-typed function application.
genApp :: Gen Expr
genApp = do
  t1 <- Arbitrary.genType
  t2 <- Arbitrary.genType
```

Think carefully about the ways to produce type errors in this simple language.
There's only one way: **apply a function** to an argument with the wrong type.
This is pretty easy to generate! Just produce a lambda, make sure its type annotation doesn't line up with its argument.
There's one other way - we could just generate free variables. Should be doing that here, just ran out of time.

Ill-Typed Terms

```
-- | Generates an ill-typed function application.
genApp :: Gen Expr
genApp = do
  t1 <- Arbitrary.genType
  t2 <- Arbitrary.genType
  t3 <- Arbitrary.genType
```

Think carefully about the ways to produce type errors in this simple language.
There's only one way: **apply a function** to an argument with the wrong type.
This is pretty easy to generate! Just produce a lambda, make sure its type annotation doesn't line up with its argument.
There's one other way - we could just generate free variables. Should be doing that here, just ran out of time.

Ill-Typed Terms

```haskell
-- | Generates an ill-typed function application.
genApp :: Gen Expr
genApp = do
  t1 <- Arbitrary.genType
  t2 <- Arbitrary.genType
  t3 <- Arbitrary.genType
  guard (t1 /= t2) {- discard via Alternative -}
```

Think carefully about the ways to produce type errors in this simple language.
There's only one way: **apply a function** to an argument with the wrong type.
This is pretty easy to generate! Just produce a lambda, make sure its type annotation doesn't line up with its argument.
There's one other way - we could just generate free variables. Should be doing that here, just ran out of time.

Ill-Typed Terms

```
-- | Generates an ill-typed function application.
genApp :: Gen Expr
genApp = do
  t1 <- Arbitrary.genType
  t2 <- Arbitrary.genType
  t3 <- Arbitrary.genType
  guard (t1 /= t2) {- discard via Alternative -}
  f <- WellTyped.genExpr t3
```

Think carefully about the ways to produce type errors in this simple language.
There's only one way: **apply a function** to an argument with the wrong type.
This is pretty easy to generate! Just produce a lambda, make sure its type annotation doesn't line up with its argument.
There's one other way - we could just generate free variables. Should be doing that here, just ran out of time.

Ill-Typed Terms

```
-- | Generates an ill-typed function application.
genApp :: Gen Expr
genApp = do
  t1 <- Arbitrary.genType
  t2 <- Arbitrary.genType
  t3 <- Arbitrary.genType
  guard (t1 /= t2) {- discard via Alternative -}
  f <- WellTyped.genExpr t3
  g <- WellTyped.genExpr t2
```

Think carefully about the ways to produce type errors in this simple language.
There's only one way: **apply a function** to an argument with the wrong type.
This is pretty easy to generate! Just produce a lambda, make sure its type annotation doesn't line up with its argument.
There's one other way - we could just generate free variables. Should be doing that here, just ran out of time.

Ill-Typed Terms

```
-- | Generates an ill-typed function application.
genApp :: Gen Expr
genApp = do
  t1 <- Arbitrary.genType
  t2 <- Arbitrary.genType
  t3 <- Arbitrary.genType
  guard (t1 /= t2) {- discard via Alternative -}
  f <- WellTyped.genExpr t3
  g <- WellTyped.genExpr t2
  x <- Arbitrary.genName
```

Think carefully about the ways to produce type errors in this simple language.
There's only one way: **apply a function** to an argument with the wrong type.
This is pretty easy to generate! Just produce a lambda, make sure its type annotation doesn't line up with its argument.
There's one other way - we could just generate free variables. Should be doing that here, just ran out of time.

Ill-Typed Terms

```haskell
-- | Generates an ill-typed function application.
genApp :: Gen Expr
genApp = do
  t1 <- Arbitrary.genType
  t2 <- Arbitrary.genType
  t3 <- Arbitrary.genType
  guard (t1 /= t2) {- discard via Alternative -}
  f <- WellTyped.genExpr t3
  g <- WellTyped.genExpr t2
  x <- Arbitrary.genName
  pure $
```

Think carefully about the ways to produce type errors in this simple language.
There's only one way: **apply a function** to an argument with the wrong type.
This is pretty easy to generate! Just produce a lambda, make sure its type annotation doesn't line up with its argument.
There's one other way - we could just generate free variables. Should be doing that here, just ran out of time.

Ill-Typed Terms

```haskell
-- | Generates an ill-typed function application.
genApp :: Gen Expr
genApp = do
  t1 <- Arbitrary.genType
  t2 <- Arbitrary.genType
  t3 <- Arbitrary.genType
  guard (t1 /= t2) {- discard via Alternative -}
  f <- WellTyped.genExpr t3
  g <- WellTyped.genExpr t2
  x <- Arbitrary.genName
  pure $
    EApp (ELam x t1 f) g
```

Think carefully about the ways to produce type errors in this simple language.
There's only one way: **apply a function** to an argument with the wrong type.
This is pretty easy to generate! Just produce a lambda, make sure its type annotation doesn't line up with its argument.
There's one other way - we could just generate free variables. Should be doing that here, just ran out of time.

Ill-Typed Terms

This isn’t great coverage, though - the type error is right at the top of the expression.
We can improve this generator a bit by **growing the expression outward a little.**

Ill-Typed Terms

```
-- Grow a larger expression around our busted one.
```

This isn’t great coverage, though - the type error is right at the top of the expression.
We can improve this generator a bit by **growing the expression outward a little.**

Ill-Typed Terms

```
-- Grow a larger expression around our busted one.
growExpr :: Expr -> Gen Expr
```

This isn’t great coverage, though - the type error is right at the top of the expression.
We can improve this generator a bit by **growing the expression outward a little.**

Ill-Typed Terms

```
-- Grow a larger expression around our busted one.
growExpr :: Expr -> Gen Expr
growExpr badExpr =
```

This isn’t great coverage, though - the type error is right at the top of the expression.
We can improve this generator a bit by **growing the expression outward a little.**

Ill-Typed Terms

```
-- Grow a larger expression around our busted one.
growExpr :: Expr -> Gen Expr
growExpr badExpr =
  Gen.recursive Gen.choice [
```

This isn’t great coverage, though - the type error is right at the top of the expression.
We can improve this generator a bit by **growing the expression outward a little.**

Ill-Typed Terms

```
-- Grow a larger expression around our busted one.
growExpr :: Expr -> Gen Expr
growExpr badExpr =
  Gen.recursive Gen.choice [
      pure badExpr
```

This isn’t great coverage, though - the type error is right at the top of the expression.
We can improve this generator a bit by **growing the expression outward a little.**

Ill-Typed Terms

```
-- Grow a larger expression around our busted one.
growExpr :: Expr -> Gen Expr
growExpr badExpr =
  Gen.recursive Gen.choice [
      pure badExpr
    ] [
```

This isn’t great coverage, though - the type error is right at the top of the expression.
We can improve this generator a bit by **growing the expression outward a little.**

Ill-Typed Terms

```
-- Grow a larger expression around our busted one.
growExpr :: Expr -> Gen Expr
growExpr badExpr =
  Gen.recursive Gen.choice [
      pure badExpr
    ] [
      -- Grow an app around the error
```

This isn’t great coverage, though - the type error is right at the top of the expression.
We can improve this generator a bit by **growing the expression outward a little.**

Ill-Typed Terms

```
-- Grow a larger expression around our busted one.
growExpr :: Expr -> Gen Expr
growExpr badExpr =
  Gen.recursive Gen.choice [
      pure badExpr
    ] [
      -- Grow an app around the error
      do tg <- Arbitrary.genType
```

This isn’t great coverage, though - the type error is right at the top of the expression.
We can improve this generator a bit by **growing the expression outward a little.**

Ill-Typed Terms

```
-- Grow a larger expression around our busted one.
growExpr :: Expr -> Gen Expr
growExpr badExpr =
  Gen.recursive Gen.choice [
      pure badExpr
    ] [
      -- Grow an app around the error
      do tg <- Arbitrary.genType
         tf <- Arbitrary.genType
```

This isn’t great coverage, though - the type error is right at the top of the expression.
We can improve this generator a bit by **growing the expression outward a little.**

Ill-Typed Terms

```
-- Grow a larger expression around our busted one.
growExpr :: Expr -> Gen Expr
growExpr badExpr =
  Gen.recursive Gen.choice [
      pure badExpr
    ] [
      -- Grow an app around the error
      do tg <- Arbitrary.genType
         tf <- Arbitrary.genType
         let ta = TArrow tg tf
```

This isn’t great coverage, though - the type error is right at the top of the expression.
We can improve this generator a bit by **growing the expression outward a little.**

Ill-Typed Terms

```
-- Grow a larger expression around our busted one.
growExpr :: Expr -> Gen Expr
growExpr badExpr =
  Gen.recursive Gen.choice [
      pure badExpr
    ] [
      -- Grow an app around the error
      do tg <- Arbitrary.genType
         tf <- Arbitrary.genType
         let ta = TArrow tg tf
         ea <- WellTyped.genExpr ta
```

This isn’t great coverage, though - the type error is right at the top of the expression.
We can improve this generator a bit by **growing the expression outward a little.**

Ill-Typed Terms

```
-- Grow a larger expression around our busted one.
growExpr :: Expr -> Gen Expr
growExpr badExpr =
  Gen.recursive Gen.choice [
      pure badExpr
    ] [
      -- Grow an app around the error
      do tg <- Arbitrary.genType
         tf <- Arbitrary.genType
         let ta = TArrow tg tf
         ea <- WellTyped.genExpr ta
         pure (EApp ea badExpr)
```

This isn’t great coverage, though - the type error is right at the top of the expression.
We can improve this generator a bit by **growing the expression outward a little.**

Ill-Typed Terms

```
-- Grow a larger expression around our busted one.
growExpr :: Expr -> Gen Expr
growExpr badExpr =
  Gen.recursive Gen.choice [
      pure badExpr
    ] [
      -- Grow an app around the error
      do tg <- Arbitrary.genType
         tf <- Arbitrary.genType
         let ta = TArrow tg tf
         ea <- WellTyped.genExpr ta
         pure (EApp ea badExpr)
    ]
```

This isn’t great coverage, though - the type error is right at the top of the expression.
We can improve this generator a bit by **growing the expression outward a little.**

Eval

We can write some nice properties for Eval without writing any new generators.
Eval should only run on validated programs, so our well-typed generator is fairly useful.
Using only these generators, we can assert that our eval function

Eval

```
prop_eval_idempotent :: Property
```

We can write some nice properties for Eval without writing any new generators.
Eval should only run on validated programs, so our well-typed generator is fairly useful.
Using only these generators, we can assert that our eval function

Eval

```
prop_eval_idempotent :: Property
prop_eval_idempotent =
```

We can write some nice properties for Eval without writing any new generators.
Eval should only run on validated programs, so our well-typed generator is fairly useful.
Using only these generators, we can assert that our eval function

Eval

```
prop_eval_idempotent :: Property
prop_eval_idempotent =
  property $ do
```

We can write some nice properties for Eval without writing any new generators.
Eval should only run on validated programs, so our well-typed generator is fairly useful.
Using only these generators, we can assert that our eval function

Eval

```
prop_eval_idempotent :: Property
prop_eval_idempotent =
  property $ do
    ty <- forAll Arbitrary.genType
```

We can write some nice properties for Eval without writing any new generators.
Eval should only run on validated programs, so our well-typed generator is fairly useful.
Using only these generators, we can assert that our eval function

Eval

```
prop_eval_idempotent :: Property
prop_eval_idempotent =
  property $ do
    ty <- forAll Arbitrary.genType
    ex <- forAll (WellTyped.genExpr ty)
```

We can write some nice properties for Eval without writing any new generators.
Eval should only run on validated programs, so our well-typed generator is fairly useful.
Using only these generators, we can assert that our eval function

Eval

```haskell
prop_eval_idempotent :: Property
prop_eval_idempotent =
  property $ do
    ty <- forAll Arbitrary.genType
    ex <- forAll (WellTyped.genExpr ty)
    eval (eval ex) === eval ex
```

We can write some nice properties for Eval without writing any new generators.
Eval should only run on validated programs, so our well-typed generator is fairly useful.
Using only these generators, we can assert that our eval function

Eval

```haskell
prop_eval_idempotent :: Property
prop_eval_idempotent =
  property $ do
    ty <- forAll Arbitrary.genType
    ex <- forAll (WellTyped.genExpr ty)
    eval (eval ex) === eval ex
```

We can write some nice properties for Eval without writing any new generators.
Eval should only run on validated programs, so our well-typed generator is fairly useful.
Using only these generators, we can assert that our eval function

Eval

```
prop_eval_idempotent :: Property
prop_eval_idempotent =
  property $ do
    ty <- forAll Arbitrary.genType
    ex <- forAll (WellTyped.genExpr ty)
    eval (eval ex) === eval ex


prop_type_safety :: Property
```

We can write some nice properties for Eval without writing any new generators.
Eval should only run on validated programs, so our well-typed generator is fairly useful.
Using only these generators, we can assert that our eval function

Eval

```haskell
prop_eval_idempotent :: Property
prop_eval_idempotent =
  property $ do
    ty <- forAll Arbitrary.genType
    ex <- forAll (WellTyped.genExpr ty)
    eval (eval ex) === eval ex


prop_type_safety :: Property
prop_type_safety =
```

We can write some nice properties for Eval without writing any new generators.
Eval should only run on validated programs, so our well-typed generator is fairly useful.
Using only these generators, we can assert that our eval function

Eval

```
prop_eval_idempotent :: Property
prop_eval_idempotent =
  property $ do
    ty <- forAll Arbitrary.genType
    ex <- forAll (WellTyped.genExpr ty)
    eval (eval ex) === eval ex


prop_type_safety :: Property
prop_type_safety =
  property $ do
```

We can write some nice properties for Eval without writing any new generators.
Eval should only run on validated programs, so our well-typed generator is fairly useful.
Using only these generators, we can assert that our eval function

Eval

```haskell
prop_eval_idempotent :: Property
prop_eval_idempotent =
  property $ do
    ty <- forAll Arbitrary.genType
    ex <- forAll (WellTyped.genExpr ty)
    eval (eval ex) === eval ex


prop_type_safety :: Property
prop_type_safety =
  property $ do
    ty <- forAll Arbitrary.genType
```

We can write some nice properties for Eval without writing any new generators.
Eval should only run on validated programs, so our well-typed generator is fairly useful.
Using only these generators, we can assert that our eval function

Eval

```haskell
prop_eval_idempotent :: Property
prop_eval_idempotent =
  property $ do
    ty <- forAll Arbitrary.genType
    ex <- forAll (WellTyped.genExpr ty)
    eval (eval ex) === eval ex


prop_type_safety :: Property
prop_type_safety =
  property $ do
    ty <- forAll Arbitrary.genType
    ex <- forAll (WellTyped.genExpr ty)
```

We can write some nice properties for Eval without writing any new generators.
Eval should only run on validated programs, so our well-typed generator is fairly useful.
Using only these generators, we can assert that our eval function

Eval

```
prop_eval_idempotent :: Property
prop_eval_idempotent =
  property $ do
    ty <- forAll Arbitrary.genType
    ex <- forAll (WellTyped.genExpr ty)
    eval (eval ex) === eval ex


prop_type_safety :: Property
prop_type_safety =
  property $ do
    ty <- forAll Arbitrary.genType
    ex <- forAll (WellTyped.genExpr ty)
    Check.typeCheck (eval ex) === pure ty
```

We can write some nice properties for Eval without writing any new generators.
Eval should only run on validated programs, so our well-typed generator is fairly useful.
Using only these generators, we can assert that our eval function

Growing the Language

Unfortunately this section is really half-baked, I ran out of time, sorry!
The real power of this set of techniques becomes self-evident when you try to extend the language.
We've decomposed it into half a dozen functions, and most extensions just require **updating them in the right spots**.
We can add substantial features without breaking backward-compatibility or breaking old features.

Growing the Language

- Extension: Let bindings
    - Naive generation will lead to non-termination!
    - Hack: Just ensure your call graph forms a DAG. This can be done constructively. Generate names and types first, then put them into scope one at a time as you generate terms
    - Bugs eaten: capture during substitution!

Unfortunately this section is really half-baked, I ran out of time, sorry!
The real power of this set of techniques becomes self-evident when you try to extend the language.
We've decomposed it into half a dozen functions, and most extensions just require **updating them in the right spots**.
We can add substantial features without breaking backward-compatibility or breaking old features.

Growing the Language

- Extension: Let bindings
    - Naive generation will lead to non-termination!
    - Hack: Just ensure your call graph forms a DAG. This can be done constructively. Generate names and types first, then put them into scope one at a time as you generate terms
    - Bugs eaten: capture during substitution!
- Extension: Sum types
    - Bookkeeping: inject a ton of pattern matches into the scope
    - Generation: Can always construct. Done

Unfortunately this section is really half-baked, I ran out of time, sorry!
The real power of this set of techniques becomes self-evident when you try to extend the language.
We've decomposed it into half a dozen functions, and most extensions just require **updating them in the right spots**.
We can add substantial features without breaking backward-compatibility or breaking old features.

Growing the Language

Unfortunately this section is really half-baked, I ran out of time, sorry!

Growing the Language

- Extension: Type inference
    - Delete type parameters from lambdas sometimes
    - Pretty straightforward until you have polymorphic types – even then, you'll be ok – change your Env and search procedure
    - Generate fairly concrete types to start with and then try to extend

Unfortunately this section is really half-baked, I ran out of time, sorry!

Growing the Language

- Extension: Type inference
    - Delete type parameters from lambdas sometimes
    - Pretty straightforward until you have polymorphic types – even then, you'll be ok – change your Env and search procedure
    - Generate fairly concrete types to start with and then try to extend
- Extension: layout-sensitive syntax
    - Instead of a pretty-printer, write a Gen Doc
    - Every time there's a formatting choice, use Gen.choice. Done

Unfortunately this section is really half-baked, I ran out of time, sorry!

Hedgehog is Nice

- We didn’t have to care about sizes
    - … and our recursive generators never diverged
- We wrote generators that could fail
    - … and they just worked
- We didn’t write a single shrink function
    - … and the automatic ones were *good enough*
    - … shrinking a well-typed term led to a well-typed term!
- Hedgehog itself didn’t eat exceptions or diverge
    - … QuickCheck has some bad habits
- We had fine-grained control over the length of lists, range of ints, etc
- We ran our tests in parallel

OK, so I’m sorry for leaving out the most interesting part of this talk - I’ll try to write a blog post about it soon (once I finish porting the code from Jack to Hedgehog)!
The search procedures we wrote to produce terms and types were really quite straightforward! We could probably have implemented them in QuickCheck. What does Hedgehog bring to the table?
First of all, we didn’t write much code.

This is Already Useful

- Naive use of Hedgehog produces decent results, even for quite complicated languages
- Each problem decomposes into smaller problems
- Automatic shrinking – refactor without fear
- Real compiler engineers could probably take Hedgehog much, much further!
- Techniques are broadly applicable – language problems are everywhere!
    - Reading and writing config files
    - Generating random data from a random schema

Even without all my fancy extensions, this naive formulation is really useful!

Questions

- Is this still property-based testing?
- Could we generate a term alongside its normal form? (stronger Eval property)
- How do we extend to deal with fancy types?
- How do we generate more interesting recursive programs that terminate?
- Testing our tests – what's coverage like?

- We're sorta drifting towards model checking or something like that?
- It would be nice to generate the normal form of a term alongside its un-normalised representation. This would give us a stronger eval property. If you had precise semantics, maybe this would look a bit like running them backwards?
- It would be nice to handle fancy types properly. I always generate concrete types. This is because I write boring DSLs, not general-purpose programming languages.
- My extremely hacky formulation to handle let bindings means we aren't testing recursive (terminating) programs. I suspect you could constructively generate recursive terminating programs, if you were a brain genius; I feel like it might look a bit like running a termination checker backwards (monotonic sizes etc)???
- It's hard to be sure your generators are producing good coverage. If you're not careful all your expressions will be `const`.

It'd be nice to be able to write assertions about the probability distribution etc.