# Generating Constrained Test Data using Datatype Generic Programming

Cas van der Rest

Utrecht University

# Contents

- Introduction
- Agda Formalization of 2 type universes
    - Regular Types
    - Indexed Descriptions
- Implementation in Haskell
- Conclusion

# Introduction

Suppose we have the following QuickCheck property

```
prop :: [Int] -> [Int] -> Property
prop xs ys = sorted xs && sorted ys ==> sorted (merge xs ys)
```

# Problem Statement

What happens when we test this property?

```
sorted xs && sorted ys ==> sorted (merge xs ys)
```

# Problem Statement

What happens when we test this property?

```
sorted xs && sorted ys ==> sorted (merge xs ys)
```

```
*** Gave up! Passed only 22 tests; 1000 discarded tests.
```

The vast majority of generated `xs` and `ys` fail the precondition!

# Problem Statement

What goes wrong here?

# Problem Statement

What goes wrong here?

- All the generated lists are random, thus **unsorted** with very high probability

# Problem Statement

What goes wrong here?

- All the generated lists are random, thus **unsorted** with very high probability
- A random list that happens to be sorted is **likely to be a very short list**

## Problem Statement

We could define a custom generator

```haskell
gen_sorted :: Gen [Int]
gen_sorted = arbitrary >>= return . diff
  where diff :: [Int] -> [Int]
        diff [] = []
        diff (x:xs) = x:map (+x) (diff xs)
```

# Problem Statement

We could define a custom generator

```
gen_sorted :: Gen [Int]
gen_sorted = arbitrary >>= return . diff
  where diff :: [Int] -> [Int]
        diff [] = []
        diff (x:xs) = x:map (+x) (diff xs)
```

But what if the we require data with more complex structure (i.e. well-formed programs)

# Approach

We can represent constrained data as an indexed family

# Approach

We can represent constrained data as an indexed family

```
data Sorted : List ℕ → Set where
    nil    : Sorted []
    single : ∀ {n} → Sorted [ n ]
    step   : ∀ {n m xs} → n ≤ m → Sorted (m ∷ xs)
                        → Sorted (n ∷ m ∷ xs)
```

`Sorted xs` is inhabited if and only if `xs` is a sorted list

# Approach

We can convert a value of type `Sorted xs` to a value of type `List ℕ`

```
toList : ∀ {xs} → Sorted xs → List ℕ
toList {xs} _ = xs
```

# Approach

We can convert a value of type `Sorted xs` to a value of type `List ℕ`

```
toList : ∀ {xs} → Sorted xs → List ℕ
toList {xs} _ = xs
```

If we can generate values of type `Sorted xs`, we can generate sorted lists!

## Approach

We can generalize this approach:

# Approach

We can generalize this approach:

1. Define a suitable indexed type to describe our constrained test data

# Approach

We can generalize this approach:

1. Define a suitable indexed type to describe our constrained test data

2. Generate inhabitants of this type

# Approach

We can generalize this approach:

1. Define a suitable indexed type to describe our constrained test data
2. Generate inhabitants of this type
3. Convert back to the original (non-indexed) datatype

# Approach

The core contributions of this thesis concern the second step

## Approach

The core contributions of this thesis concern the second step

More specifically: *how can we generically generate values of arbitrary indexed families?*

We tackle this question by looking at various type universes, and defining generators for them

# Type universes

Each type universe consists of the following elements:

1. A datatype `U` describing codes in the universe
2. A semantics `⟦_⟧ : U → Set` that maps codes to a type

# Type universes

Each type universe consists of the following elements:

1. A datatype `U` describing codes in the universe
2. A semantics `⟦_⟧ : U → Set` that maps codes to a type

Our goal is then to define `deriveGen : (u : U) → Gen ⟦ u ⟧`

# Generators

We use an abstract generator type `Gen a t i` that generates values of indexed types

# Generators

We use an abstract generator type **`Gen a t i`** that generates values of indexed types

```
data Gen {i : Set} : Set → (i → Set) → i → Set where
    Pure : ∀ {a t x} → a → Gen a t x
    Or   : ∀ {a t x} → Gen a t x → Gen a t x → Gen a t x
    ...
    μ    : ∀ {t} → (x : i) → Gen (t x) t x
    ...
    Call : ∀ {j t s x} → (y : j)
              → ((y' : j) → Gen (s y') s y') → Gen (s y) t x
```

# Generators

We use an abstract generator type **`Gen a t i`** that generates values of indexed types

```
data Gen {i : Set} : Set → (i → Set) → i → Set where
    Pure : ∀ {a t x} → a → Gen a t x
    Or   : ∀ {a t x} → Gen a t x → Gen a t x → Gen a t x
    ...
    μ    : ∀ {t} → (x : i) → Gen (t x) t x
    ...
    Call : ∀ {j t s x} → (y : j)
             → ((y' : j) → Gen (s y') s y') → Gen (s y) t x
```

**`Gen`** is a deep embedding of the functions exposed by the **`Applicative`**, **`Monad`** and **`Alternative`** typeclasses

# Generators

We can map **`Gen`** to any concrete generator type we require

# Generators

We can map **`Gen`** to any concrete generator type we require

```
enumerate : ∀ {A T} → Gen A T → Gen T T → ℕ → List A
enumerate g g' zero    = []
-- ...
enumerate μ g' (suc n) = enumerate g' g' n
```

We can map **Gen** to any concrete generator type we require

```
enumerate : ∀ {A T} → Gen A T → Gen T T → ℕ → List A
enumerate g g' zero    = []
-- ...
enumerate μ g' (suc n) = enumerate g' g' n
```

We use the enumerative mapping exclusively, but others are possible

# Generators

Why not skip **Gen** altogether?

```
nat : (ℕ → List ℕ) → ℕ → List ℕ
nat μ =  ⦇ zero  ⦈
      || ⦇ suc μ ⦈

fix : ∀ {A} → (A → A) → A
fix f = f (fix f)
```

# Generators

Why not skip **`Gen`** altogether?

```
nat : (ℕ → List ℕ) → ℕ → List ℕ
nat μ =  ⦇ zero  ⦈
      || ⦇ suc μ ⦈

fix : ∀ {A} → (A → A) → A
fix f = f (fix f)
```

`fix` is rejected by the termination checker, using **`Gen`** circumvents this issue

# Generators

In practice we will allmost never use the constructors of `Gen a t i`

## Generators

In practice we will allmost never use the constructors of `Gen a t i`

```
fin : (n : ℕ) → Gen (Fin n) Fin n
fin zero    = empty
fin (suc n) =  ⦇ zero       ⦈
            || ⦇ suc (μ n) ⦈
```

Or desugared:

```
fin : (n : ℕ) → Gen (Fin n) Fin n
fin zero    = empty
fin (suc n) =  pure zero
            || pure suc <*> (μ n)
```

# Generators

`Set` is isomorphic to the trivial function space `⊤ → Set`, so we can generate non-indexed types too

# Generators

`Set` is isomorphic to the trivial function space `⊤ → Set`, so we can generate non-indexed types too

In practice, this heavily pollutes the code, so we will be a bit liberal with notation

# Generators

`Set` is isomorphic to the trivial function space `⊤ → Set`, so we can generate non-indexed types too

In practice, this heavily pollutes the code, so we will be a bit liberal with notation

```
nat : Gen ℕ ℕ
nat =  ⦇ zero  ⦈
    || ⦇ suc μ ⦈
```

instead of

```
nat : Gen ℕ (λ { tt → ℕ }) tt
nat =  ⦇ zero        ⦈
    || ⦇ suc (μ tt) ⦈
```

# Generator completeness

We want to assert that generators behave correctly

# Generator completeness

We want to assert that generators behave correctly

We formulate the following completeness property for this:

```
Complete : ∀ {T} → Gen T T → Set
Complete gen = ∀ {x} → ∃[ n ] x ∈ enumerate gen gen n
```

A generator is complete if *all values of the type it produces at some point occur in the enumeration*

# Agda Formalization

# Universe definition

We start by looking at the universe of *Regular Types*

# Universe definition

We start by looking at the universe of *Regular Types*

Roughly, this corresponds to algebraic datatypes in Haskell 98

# Universe definition

We start by looking at the universe of *Regular Types*

Roughly, this corresponds to algebraic datatypes in Haskell 98

Examples of regular types include `Bool` and `List`

# Universe Definition

The universe includes unit types (`U`), empty types(`Z`), constant types (`K`) and recursive positions (`I`):

```
data Reg : Set where
  U I Z : Reg
  K     : Set → Reg
```

# Universe Definition

The universe includes unit types (`U`), empty types(`Z`), constant types (`K`) and recursive positions (`I`):

```
data Reg : Set where
  U I Z : Reg
  K     : Set → Reg
```

Regular types are closed under the product and coproduct operations:

```
  _⊗_    : Reg → Reg → Reg
  _⊕_    : Reg → Reg → Reg
```

# Regular types - Universe Definition

For example: `Bool` is a choice between two nullary constructors:

```
data Bool : Set where
  true  : Bool
  false : Bool
```

# Regular types - Universe Definition

For example: `Bool` is a choice between two nullary constructors:

```
data Bool : Set where
  true  : Bool
  false : Bool
```

Hence, we can describe it as the coproduct of two unit types:

```
Bool : Reg
Bool = U ⊕ U
```

# Regular types - Semantics

The semantics, `⟦_⟧ : Reg → Set → Set`, map a value of type `Reg` to a value in `Set → Set`

# Regular types - Semantics

The semantics, `⟦_⟧ : Reg → Set → Set`, map a value of type **Reg** to a value in `Set → Set`

```
⟦_⟧ : Reg → Set → Set
⟦ Z       ⟧ r = ⊥
⟦ U       ⟧ r = ⊤
⟦ I       ⟧ r = r
⟦ K x     ⟧ r = x
⟦ c₁ ⊗ c₂ ⟧ r = ⟦ c₁ ⟧ r × ⟦ c₂ ⟧ r
⟦ c₁ ⊕ c₂ ⟧ r = ⟦ c₁ ⟧ r ⊎ ⟦ c₂ ⟧ r
```

`r` is the type of recursive positions!

# Regular types - Fixpoint operation

We use the following fixpoint operation:

```
data Fix (c : Reg) : Set where
  In : ⟦ c ⟧ (Fix c) → Fix c
```

# Regular types - Fixpoint operation

We use the following fixpoint operation:

```
data Fix (c : Reg) : Set where
  In : ⟦ c ⟧ (Fix c) → Fix c
```

`Fix (U ⊕ U)` is isomorphic to `Bool`.

# Regular types - Deriving a generator

We now aim to define generators from values in `Reg`

```
deriveGen : (c c' : Reg)
          → Gen (⟦ c ⟧ (Fix c')) (⟦ c' ⟧ (Fix c'))
```

We now aim to define generators from values in **`Reg`**

```
deriveGen : (c c' : Reg)
          → Gen (⟦ c ⟧ (Fix c')) (⟦ c' ⟧ (Fix c'))
```

Notice the difference between the type parameters of **`Gen`**!

## Regular types - Deriving a generator

```
deriveGen Z         c' =  empty
deriveGen U         c' =  ⦇ tt   ⦈
deriveGen I         c' =  ⦇ In μ ⦈
deriveGen (c₁ ⊗ c₂) c' =  ⦇ (deriveGen c₁ c')
                           , (deriveGen c₂ c') ⦈
deriveGen (c₁ ⊕ c₂) c' =  ⦇ inj₁ (deriveGen c₁ c') ⦈
                       || ⦇ inj₂ (deriveGen c₂ c') ⦈
```

```
deriveGen Z         c' =  empty
deriveGen U         c' =  ⦇ tt   ⦈
deriveGen I         c' =  ⦇ In μ ⦈
deriveGen (c₁ ⊗ c₂) c' =  ⦇ (deriveGen c₁ c')
                          , (deriveGen c₂ c') ⦈
deriveGen (c₁ ⊕ c₂) c' =  ⦇ inj₁ (deriveGen c₁ c') ⦈
                       || ⦇ inj₂ (deriveGen c₂ c') ⦈
```

What about **K** (constant types)?

# Regular types - Deriving a generator

The semantics of **K** is the type it carries.

We need the programmer's input to generate values of this type

# Regular types - Deriving a generator

The semantics of **K** is the type it carries.

We need the programmer's input to generate values of this type

How does the programmer supply the required generators?

# Regular types - Deriving a generator

We define a *metadata structure* that carries additional information about the types stored in a code

We define a *metadata structure* that carries additional information about the types stored in a code

```
data KInfo (P : Set → Set) : Reg → Set where
  Z~   : KInfo P Z
  U~   : KInfo P U
  I~   : KInfo P I
  _⊗~_ : ∀ {c₁ c₂} → KInfo P c₁
                   → KInfo P c₂ → KInfo P (c₁ ⊗ c₂)
  _⊕~_ : ∀ {c₁ c₂} → KInfo P c₁
                   → KInfo P c₂ → KInfo P (c₁ ⊕ c₂)
  K~   : ∀ {S} → P S → KInfo P (K S)
```

# Regular types - Defining a generator

We parameterise `deriveGen` over a metadata structure with type `KInfo Gen`

```
deriveGen : (c c' : Reg) → KInfo Gen c
          → Gen (⟦ c ⟧ (Fix c')) (⟦ c' ⟧ (Fix c'))
```

# Regular types - Defining a generator

We parameterise **`deriveGen`** over a metadata structure with type **`KInfo Gen`**

```
deriveGen : (c c' : Reg) → KInfo Gen c
          → Gen (⟦ c ⟧ (Fix c')) (⟦ c' ⟧ (Fix c'))
```

For constant types, **`deriveGen`** then simply invokes the supplied generator

```
deriveGen (K _) c' (K~ g) = Call g
```

# Regular types - Proving completeness

We prove the completeness of **`deriveGen`** by induction over the input code:

```
complete-thm : ∀ {c c' x} →
  ∃[ n ] (x ∈ enumerate (deriveGen c c') (deriveGen c' c') n)
```

## Regular types - Proving completeness

We prove the completeness of **`deriveGen`** by induction over the input code:

```
complete-thm : ∀ {c c' x} →
  ∃[ n ] (x ∈ enumerate (deriveGen c c') (deriveGen c' c') n)
```

The cases for **`U`** and **`Z`** are trivial

```
complete-thm {U} = 1 , here
complete-thm {Z} {c'} {()}
```

# Regular types - Proving completeness

For product and coproduct, we prove that we combine the derived generators in a completeness preserving manner

For product and coproduct, we prove that we combine the derived generators in a completeness preserving manner

This amounts to proving the following lemmas (in pseudocode):

```
Complete g₁ → Complete g₂
  → Complete ((⦇ inj₁ g₁ ⦈ || ⦇ inj₂ g₂ ⦈))

Complete g₁ → Complete g₂ → Complete ⦇ g₁ , g₂ ⦈
```

# Regular types - Proving completeness

Recursive positions (`I`) are slightly more tricky

```
complete-thm {I} {c'} {In x} with complete-thm {c'} {c'} {x}
... | prf = {!!}
```

# Regular types - Proving completeness

Recursive positions (`I`) are slightly more tricky

```
complete-thm {I} {c'} {In x} with complete-thm {c'} {c'} {x}
... | prf = {!!}
```

We **must** pattern match on `In x`, otherwise the recursive call is flagged by the termination checker

We complete this case by proving a lemma of the form:

```
Complete μ → Complete ⦇ In μ ⦈
```

# Regular types - Proving completeness

For constant types, we parameterize `complete-thm` over a metadata structure containing proofs

```
KInfo (λ S → Σ[ g ∈ Gen S S ] Complete g)
```

# Regular types - Proving completeness

For constant types, we parameterize `complete-thm` over a metadata structure containing proofs

```
KInfo (λ S → Σ[ g ∈ Gen S S ] Complete g)
```

We then return the proof stored in the metadata structure

# Indexed descriptions

# Indexed descriptions - Universe definition

The universe of indexed descriptions is largely derived from the universe of regular types

# Indexed descriptions - Universe definition

The universe of indexed descriptions is largely derived from the universe of regular types

```
data IDesc (I : Set) : Set where
  `1   : IDesc I
  `var : I → IDesc I
  _`×_ : IDesc I → IDesc I → IDesc I
```

These correspond to `U`, `I` and product in the universe of regular types

# Indexed descriptions - Universe definition

The regular coproduct is replaced with a generalized version:

```
`σ  : (n : ℕ) → (Fin n → IDesc I) → IDesc I
```

The regular coproduct is replaced with a generalized version:

```
`σ  : (n : ℕ) → (Fin n → IDesc I) → IDesc I
```

Constant types are replaced with dependent pairs:

```
`Σ : (S : Set) → (S → IDesc I) → IDesc I
```

# Indexed descriptions - Universe definition

The regular coproduct is replaced with a generalized version:

```
`σ  : (n : ℕ) → (Fin n → IDesc I) → IDesc I
```

Constant types are replaced with dependent pairs:

```
`Σ : (S : Set) → (S → IDesc I) → IDesc I
```

We denote the empty type with `'σ 0 λ()`

# Indexed descriptions - Semantics

The semantic of `'1`, `'var`, and `_'×_` are taken (almost) direrectly from the semantics of regular types

```
⟦_⟧ : ∀ {I} → IDesc I → (I → Set) → Set
⟦ `1        ⟧ r = ⊤
⟦ `var x    ⟧ r = r x
⟦ δ₁ `× δ₂ ⟧ r = ⟦ δ₁ ⟧ r × ⟦ δ₂ ⟧ r
```

# Indexed descriptions - Semantics

The semantic of `'1`, `'var`, and `_'×_` are taken (almost) direrectly from the semantics of regular types

```
⟦_⟧ : ∀ {I} → IDesc I → (I → Set) → Set
⟦ `1       ⟧ r = ⊤
⟦ `var x   ⟧ r = r x
⟦ δ₁ `× δ₂ ⟧ r = ⟦ δ₁ ⟧ r × ⟦ δ₂ ⟧ r
```

Both sigma's are interpreted to a dependent pair:

```
⟦ `σ n T ⟧ r = Σ[ fn ∈ Fin n ] ⟦ T fn ⟧ r
⟦ `Σ S T ⟧ r = Σ[ s ∈ S ] ⟦ T s ⟧ r
```

# Indexed descriptions - Fixpoint

We can then describe an indexed type using a function of type `I → IDesc I`.

# Indexed descriptions - Fixpoint

We can then describe an indexed type using a function of type `I → IDesc I`.

The fixpoint operation associated with this universe is:

```
data Fix {I} (φ : I → IDesc I) (i : I) : Set where
  In : ⟦ φ i ⟧ (Fix φ) → Fix φ i
```

# Indexed descriptions - Example

Consider a datatype of trees indexed with their size:

```
data STree : ℕ → Set where
    leaf : STree zero
    node : ∀ {n m} → STree n → STree m
                   → STree (suc (n + m))
```

# Indexed descriptions - Example

Consider a datatype of trees indexed with their size:

```
data STree : ℕ → Set where
    leaf : STree zero
    node : ∀ {n m} → STree n → STree m
                   → STree (suc (n + m))
```

We can use the following indexed description to describe it

```
STree' : ℕ → IDesc ℕ
STree' zero    = `1
STree' (suc n) =
  `Σ (ℕ × ℕ) λ { ( m , k)
    → `Σ (m + k ≡ n) λ _ → `var m `× `var k }
```

# Indexed descriptions - Deriving a Generator

The generator has the same structure as for regular types

```
deriveGen : ∀ {I i} → (δ : IDesc I) → (φ : I → IDesc I)
          → Gen (⟦ δ ⟧I (Fix φ)) (λ i → ⟦ φ i ⟧I (Fix φ)) i
```

# Indexed descriptions - Deriving a Generator

The generator has the same structure as for regular types

```
deriveGen : ∀ {I i} → (δ : IDesc I) → (φ : I → IDesc I)
          → Gen (⟦ δ ⟧I (Fix φ)) (λ i → ⟦ φ i ⟧I (Fix φ)) i
```

The cases for `'1`, `'var` and `'×` are also (almost) the same

```
deriveGen `1          φ = ⦇ tt ⦈
deriveGen (`var x)    φ = ⦇ In (μ x) ⦈
deriveGen (δ₁ `× δ₂) φ = ⦇ (deriveGen δ₁ φ)
                           , (deriveGen δ₂ φ) ⦈
```

For the generalized coproduct, we again need to utilize the monadic structure of generators

For the generalized coproduct, we again need to utilize the monadic structure of generators

```
deriveGen (`σ n T) φ = do
  fn ← Call n genFin
  x  ← deriveGen (T fn) φ
  pure (fn , x)
```

`genFin n` generates values of type `Fin n`

The generalized coproduct is an instantiation of the dependent pair, so we reuse the definition

The generalized coproduct is an instantiation of the dependent pair, so we reuse the definition

```
deriveGen (`Σ S T) φ = do
  s ← {!!}
  x ← deriveGen (T s) φ (fm s)
  pure (s , x)
```

# Indexed descriptions - Deriving a Generator

The generalized coproduct is an instantiation of the dependent pair, so we reuse the definition

```
deriveGen (`Σ S T) φ = do
  s ← {!!}
  x ← deriveGen (T s) φ (fm s)
  pure (s , x)
```

How do we get `s`?

# Indexed descriptions - Deriving a Generator

We define a metadata structure:

We define a metadata structure:

```
data IDescM {I : Set} (P : Set → Set) : IDesc I → Set where
  `var~ : ∀ {i} → IDescM P (`var i)
  `Σ~   : ∀ {S T} → P S → ((s : S) → IDescM P (T s))
                        → IDescM P (`Σ S T)
  ...
```

The remaining constructors are handled similar to regular types

# Indexed descriptions - Deriving a Generator

We (again) parameterize **`deriveGen`** over a metadata structure containing generators

```
deriveGen (`Σ S T) φ (`Σ~ g mT) = do
  s ← Call g
  x ← deriveGen (T s) φ (mT s)
  pure (s , x)
```

# Indexed descriptions - Deriving a Generator

In the case of `STree`, this means that we have to supply a generator that generates pairs of numbers and proofs that their sum is particular number

```
+-inv : (n : ℕ) → Gen (Σ (ℕ × ℕ) λ { (k , m) → n ≡ k + m })
```

In the case of `STree`, this means that we have to supply a generator that generates pairs of numbers and proofs that their sum is particular number

```
+-inv : (n : ℕ) → Gen (Σ (ℕ × ℕ) λ { (k , m) → n ≡ k + m })
```

By using a metadata structure to generate for dependent pairs, we separate the hard parts of generation from the easy parts

# Indexed descriptions - Deriving a Generator

In the case of `STree`, this means that we have to supply a generator that generates pairs of numbers and proofs that their sum is particular number

```
+-inv : (n : ℕ) → Gen (Σ (ℕ × ℕ) λ { (k , m) → n ≡ k + m })
```

By using a metadata structure to generate for dependent pairs, we separate the hard parts of generation from the easy parts

A programmer can influence the generation process by supplying different generators

# Indexed descriptions - Proving completeness

We use the same proof structure as with regular types

```
complete-thm : ∀ {δ φ x i} →
  ∃[ n ] (x ∈ enumerate (deriveGen δ φ)
                        (λ y → deriveGen (φ y) φ) i n)
```

# Indexed descriptions - Proving completeness

We use the same proof structure as with regular types

```
complete-thm : ∀ {δ φ x i} →
  ∃[ n ] (x ∈ enumerate (deriveGen δ φ)
                       (λ y → deriveGen (φ y) φ) i n)
```

`enumerate` is slightly altered here to accommodate indexed generators

# Indexed descriptions - Proving completeness

The cases for `'1`, `'var` and `'×` follow naturally from the completeness proof for regular types

```
complete-thm {`1} {φ} {x}         = 1 , here
complete-thm {`var i} {φ} {In x}
    with complete-thm {φ i} {φ} {x}
... | prf = {!!}
complete-thm {δ₁ `× δ₂} {φ} {x} = {!!}
```

# Indexed descriptions - Proving completeness

The cases for `'1`, `'var` and `'×` follow naturally from the completeness proof for regular types

```
complete-thm {`1} {φ} {x}         = 1 , here
complete-thm {`var i} {φ} {In x}
    with complete-thm {φ i} {φ} {x}
... | prf = {!!}
complete-thm {δ₁ `× δ₂} {φ} {x} = {!!}
```

We require (again) additional lemmas of the form:

```
Complete g₁ → Complete g₂ → Complete ⦇ g₁ , g₂ ⦈

Complete μ → Complete ⦇ In μ ⦈
```

The generator for dependent pairs is constructed using a monadic bind

The generator for dependent pairs is constructed using a monadic bind

Hence, we need to prove an additional lemma about this operation

```
bind-thm :
  ∀ {g₁ g₂ A B} → Complete g₁ → ((x : A) → Complete (g₂ x))
  → Complete (g₁ >>= (λ x → g₂ x >>= λ y → pure x , y))
```

# Indexed descriptions - Proving completeness

To prove completeness for dependent pairs, we can simply invoke this lemma

```
complete-thm {`Σ S T} {φ} =
  bind-thm {!!} (λ x → deriveGen (T x) φ)
```

# Indexed descriptions - Proving completeness

To prove completeness for dependent pairs, we can simply invoke this lemma

```
complete-thm {`Σ S T} {φ} =
  bind-thm {!!} (λ x → deriveGen (T x) φ)
```

The first argument of **`bind-thm`** is a completeness proof for the user-supplied generator

So we have the user supply this proof

```
IDescM (λ S → Σ[ g ∈ Gen S S ] Complete g
```

# Indexed descriptions - Proving completeness

The generalized coproduct is just an instantiation of the dependent pair

# Indexed descriptions - Proving completeness

The generalized coproduct is just an instantiation of the dependent pair

So we can reuse the proof structure for dependent pairs to prove its completeness

# Implementation in Haskell

# General Approach

We make a couple of changes compared to the Agda development:

# General Approach

We make a couple of changes compared to the Agda development:

- The `IDesc` type gets an extra parameter `a`, the type that a description describes

# General Approach

We make a couple of changes compared to the Agda development:

- The `IDesc` type gets an extra parameter `a`, the type that a description describes
- We represent the generalized coproduct as vector instead of a function

# General Approach

We make a couple of changes compared to the Agda development:

- The `IDesc` type gets an extra parameter `a`, the type that a description describes
- We represent the generalized coproduct as vector instead of a function
- We use shallow recursion, meaning that the semantics of recursive position is the associated type `a`

This means we have no fixpoint combinator!

# Universe definition

This all results in the following universe definition

```
data IDesc (a :: *) (i :: *) where
  One     :: IDesc a i
  Var     :: i -> IDesc a i
  (:*:)   :: IDesc a i -> IDesc a i -> IDesc a i
  (:+>)   :: SNat n -> Vec (IDesc a i) n -> IDesc a i
  Sigma   :: Proxy s -> IDesc a (s -> i) -> IDesc a i
```

# Representing dependent pairs

We choose a more restrictive form of the `'Σ` combinator, only allowing recursive positions to depend on its first element

# Representing dependent pairs

We choose a more restrictive form of the `'Σ` combinator, only allowing recursive positions to depend on its first element

Hence we can change the function `s -> IDesc a i` to a single description `IDesc a (s -> i)`.

# Representing dependent pairs

We choose a more restrictive form of the `'Σ` combinator, only allowing recursive positions to depend on its first element

Hence we can change the function `s -> IDesc a i` to a single description `IDesc a (s -> i)`.

Since we use shallow recursion, the semantics of this description is independent of the value of type `s`.

We describe the semantics in a type family:

```
type family Sem (d :: IDesc a i) :: *
type instance Sem One            = ()
type instance Sem (Var _ )       = a
type instance Sem (dl :*: dr)    = (Sem dl , Sem dr)
type instance Sem (Sigma p fd) = (UnProxy p , Sem fd)
```

We describe the semantics in a type family:

```
type family Sem (d :: IDesc a i) :: *
type instance Sem One          = ()
type instance Sem (Var _ )     = a
type instance Sem (dl :*: dr)  = (Sem dl , Sem dr)
type instance Sem (Sigma p fd) = (UnProxy p , Sem fd)
```

The semantics of the generalized coproduct is just a sum type of all the possible choices

We need a way to express the dependency between the input description, and the type of generated elements

# Deriving generators

We need a way to express the dependency between the input description, and the type of generated elements

In Agda, we can simply write a $\Pi$ type: `(d : IDesc I) → Gen ⟦ c ⟧`

# Deriving generators

We need a way to express the dependency between the input description, and the type of generated elements

In Agda, we can simply write a $\Pi$ type: `(d : IDesc I) → Gen ⟦ c ⟧`

In Haskell, we need *Singleton types* to do this

# Deriving generators

A singleton type is indexed by some other type, and has exactly one inhabitant for every inhabitant of that type

# Deriving generators

A singleton type is indexed by some other type, and has exactly one inhabitant for every inhabitant of that type

```
data SNat (n :: Nat) :: * where
  SZero :: SNat Zero
  SSuc  :: SNat n -> SNat (Suc n)
```

A singleton type is indexed by some other type, and has exactly one inhabitant for every inhabitant of that type

```
data SNat (n :: Nat) :: * where
  SZero :: SNat Zero
  SSuc  :: SNat n -> SNat (Suc n)
```

```
inc :: SNat n -> SNat (Suc n)
inc n = SSuc n
```

`inc` *must* return the successor of its argument, otherwise the typechecker complains!

We define such a singleton type for **`IDesc a i`** as well:

```
data SingIDesc (d :: IDesc a i) where
  SOne   :: SingIDesc One
  SVar   :: forall (i' :: i) . i -> SingIDesc (Var i')
  -- etcetera
  SSigma :: SingIDesc d -> Gen s s
         -> SingIDesc (Sigma ('Proxy :: Proxy s) d)
```

We define such a singleton type for `IDesc a i` as well:

```
data SingIDesc (d :: IDesc a i) where
  SOne   :: SingIDesc One
  SVar   :: forall (i' :: i) . i -> SingIDesc (Var i')
  -- etcetera
  SSigma :: SingIDesc d -> Gen s s
         -> SingIDesc (Sigma ('Proxy :: Proxy s) d)
```

`SingIDesc` simultaneously acts as a metadata structure, carrying generators for dependent pairs!

# Deriving generators

With `SingIdesc`, we can write `deriveGen`:

```
deriveGen :: SingIDesc d -> Gen (Sem d)
```

For the definition, we follow the Agda implementation.

# Using `deriveGen`

What do we need to use deriveGen?

1. A type level description `Desc :: i -> IDesc a i`
2. A singleton instance `desc :: Sing i -> SingIDesc (Desc i)`
3. A conversion function `to :: Sing i -> Sem (Desc i) -> a`

# Example - well typed expressions

Consider an expression type:

```
data Expr = AddE Expr Expr
          | LEQ Expr Expr
          | ValN Nat
          | ValB Bool
```

We'd like to generate well typed expressions (with `Type = TNat | TBool`)

# Example - well typed expressions

This comes down to generating values of the following GADT:

```
data Expr (t :: Type) :: * where
  AddE :: Expr TNat -> Expr TNat -> Expr TNat
  LEQ  :: Expr TNat -> Expr TNat -> Expr TBool
  ValN :: Nat -> Expr TNat
  ValB :: Bool -> Expr TBool
```

## Example - well typed expressions

We describe this GADT with the following type family ::

```
type family ExprDesc (t :: Type) :: IDesc Expr Type
type instance ExprDesc TNat =
  S2 :+> (    Var TNat :*: Var TNat
          ::: Sigma ('Proxy :: Proxy Nat) One
          ::: VNil )
type instance ExprDesc TBool =
  S2 :+> (    Var TNat :*: Var TNat
          ::: Sigma ('Proxy :: Proxy Bool) One
          ::: VNil )
```

# Example - well typed expressions

We describe this GADT with the following type family ::

```
type family ExprDesc (t :: Type) :: IDesc Expr Type
type instance ExprDesc TNat =
  S2 :+> (    Var TNat :*: Var TNat
          ::: Sigma ('Proxy :: Proxy Nat) One
          ::: VNil )
type instance ExprDesc TBool =
  S2 :+> (    Var TNat :*: Var TNat
          ::: Sigma ('Proxy :: Proxy Bool) One
          ::: VNil )
```

And an associated singleton instance: `exprDesc :: Sing t -> SingIDesc (ExprDesc t)`

## Example - well typed expressions

Converting back to `Expr` is then easy:

```
toExpr :: Sing t -> Interpret (ExprDesc t) -> Expr
toExpr STNat  (Left (e1 , e2)) = AddE e1 e2
toExpr STNat  (Right (n , ()))  = ValN n
toExpr STBool (Left (e1 , e2)) = LEQ e1 e2
toExpr STBool (Right (b , ()))  = ValB b
```

The definition of `toExpr` is mostly guided by Haskell's type system

# Example - well typed expressions

We can now generate well-typed expressions:

```
exprGen :: Sing t -> Gen Expr
exprGen t = toExpr <$> deriveGen (exprDesc t)
```

The elements produced by `exprGen` will all be well-typed expressions.

# Example - well typed expressions

We can now generate well-typed expressions:

```haskell
exprGen :: Sing t -> Gen Expr
exprGen t = toExpr <$> deriveGen (exprDesc t)
```

The elements produced by **`exprGen`** will all be well-typed expressions.

We can use **`deriveGen`** to generate test data with much richer structure – such as well-typed lambda terms.

# Conclusion

# Summary

To summarize, we did the following:

1. Describe three type universes in Agda, and derive generators from codes in these universes (only two of these discussed here)
2. For two of these universes, prove that the generators derived from them are complete
3. Implement our development for indexed descriptions in Haskell

# Conclusion

We have shown, as a proof of concept, that we can generate arbitrary indexed families

# Conclusion

We have shown, as a proof of concept, that we can generate arbitrary indexed families

Of course, this requires that the programmer inputs suitable generators

# Conclusion

We have shown, as a proof of concept, that we can generate arbitrary indexed families

Of course, this requires that the programmer inputs suitable generators

With this technique, it is (at least) possible to generate relatively simple well-formed data, such as typed expressions or lambda terms

# Future work

Possible avenues for future work include:

1. Considering more involved examples, such as polymorphic lambda terms
2. Integration with existing testing frameworks
3. Applying memoization techniques to the derived generators

Questions?