# Generic Enumerators

Cas van der Rest

Utrecht University

# Contents

# Introduction

# Problem Statement

Test data may have constraints

# Problem Statement

Test data may have constraints

```
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!

We could try our luck with 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)
```

## Problem Statement

We could try our luck with 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)
```

For more complex data, defining these generators is hard

# Approach

We can often represent constrained data as an indexed family

# Approach

We can often 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)
```

# Approach

We can often 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)
```

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

# Approach

We try to answer the following question: *how can we generically generate values of arbitrary indexed families?*

# Approach

We try to answer the following question: *how can we generically generate values of arbitrary indexed families?*

We tackle this question by looking at 3 increasingly complex type universes and defining generic enumerators from them.

# Approach

We try to answer the following question: *how can we generically generate values of arbitrary indexed families?*

We tackle this question by looking at 3 increasingly complex type universes and defining generic enumerators from them.

To simplify the problem a bit, we forget about sampling for now and only consider *enumerations*

# 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 a function **`enumerate : (u : U) → ℕ → List ⟦ u ⟧`**

# 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 a function `enumerate : (u : U) → ℕ → List ⟦ u ⟧`

# Enumerator completeness

We formulate the following completeness property for our enumerators:

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

# Enumerator completeness

We formulate the following completeness property for our enumerators:

```
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*

# Regular types

# 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 product and coproduct:

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

# 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 - Deriving a generator

We now aim to define an enumerator for all types that can be described by a code in **Reg**

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

We now aim to define an enumerator for all types that can be described by a code in **Reg**

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

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

# Regular types - Deriving a generator

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

## Regular types - Deriving a generator

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

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

# Regular types - Deriving an enumerator

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

# Regular types - Deriving an enumerator

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

This means that we need the programmer to somehow supply an enumerator for this type.

# Regular types - Deriving an enumerator

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

This means that we need the programmer to somehow supply an enumerator for this type.

We will come back to this later when considering indexed descriptions

# 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

# Regular types - Proving completeness

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

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 |)
```

Proofs for these lemma's follow readily from chosen instances of `Applicative` and `Alternative`

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

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

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

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

We then simply feed the induction hypothesis (`prf`) to this lemma to complete the proof.

## Regular types - Proving completeness

# Indexed containers

# Indexed containers - W-types

Indexed containers can be viewed as an extension to *W-types*

# Indexed containers - W-types

Indexed containers can be viewed as an extension to *W-types*

```
record WType : Set where
  constructor _~_
  field
    S : Set
    P : S → Set

⟦_⟧ : WType → Set → Set
⟦ S ~ P ⟧ r = Σ[ s ∈ S ] (P s → r)

data Fix (w : WType) : Set where
  In : ⟦ w ⟧sup (Fix w) → Fix w
```

## Indexed containers - Universe definition

We parameterize the *shape* and *position* over the index type, and add an typing discipline that describes the indices of recursive positions.

```
record Sig (I : Set) : Set where
  constructor _ ◁ _|_
  field
    Op : (i : I) → Set
    Ar : ∀ {i} → (Op i) → Set
    Ty : ∀ {i} {op : Op i} → Ar op → I

⟦_⟧ : ∀ {I} → Sig I → (I → Set) → I → Set
⟦ Op ◁ Ar | Ty ⟧ r i =
  Σ[ op ∈ Op i ] ((ar : Ar op) → r (Ty ar))

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

# Indexed containers - Example

Let's consider vectors as an example

Let's consider vectors as an example

```
Σ-vec a =
  let op-vec = (λ { zero → U ; (suc n) → K a})
      ar-vec = (λ {{zero} tt → Z ; {suc n} x → U})
      ty-vec = (λ {{suc n} {a} (In tt) → n})
  in op-vec ◁ ar-vec | ty-vec
```

# Indexed containers - Generic enumerator

"'agda deriveGen : □ {I : Set} □ (S : Sig I) □ (i : I) □ Gen (Fix S i) (Fix S i) deriveGen (Op □ Ar □ Ty) i = do op □ Call (genericGen (Op i)) ar □ Call (coenumerate (Ar op) (Ar op) (λ ar □ deriveGen (Op □ Ar □ Ty) (Ty ar))) pure (In (op , ar x)) . . .

`coenumerate` enumeretes all functions from arity to recursive generator.

# Indexed containers - Generic enumerator

‘‘‘agda deriveGen : □ {I : Set} □ (S : Sig I) □ (i : I) □ Gen (Fix S i) (Fix S i) deriveGen (Op □ Ar □ Ty) i = do op □ Call (genericGen (Op i)) ar □ Call (coenumerate (Ar op) (Ar op) (λ ar □ deriveGen (Op □ Ar □ Ty) (Ty ar))) pure (In (op , ar x)) . . .

`coenumerate` enumeretes all functions from arity to recursive generator.

If we restrict operations and arities to regular types, we can define `coenumerate` generically.

## Indexed containers - Proving completeness

Unfortunately, we have not been able to prove completeness for this enumerator.

# Indexed containers - Proving completeness

Unfortunately, we have not been able to prove completeness for this enumerator.

We need to pattern match on the value `x` quantified over in the completeness property in order to guarantee termination

# Indexed containers - Proving completeness

Unfortunately, we have not been able to prove completeness for this enumerator.

We need to pattern match on the value `x` quantified over in the completeness property in order to guarantee termination

In the case of indexed containers, part of this value `x` is a function, so we cannot perform this pattern match.

# 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
```

# 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
```

# 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 describe indexed families with a function `I → IDesc I`.

# Indexed descriptions - Fixpoint

We describe indexed families with a function `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 - Deriving a Generator

The generator type 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 type 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) ⦈
```

# Indexed descriptions - Deriving a Generator

For the generalized coproduct, we utilize monadic structure of generators

# Indexed descriptions - Deriving a Generator

For the generalized coproduct, we utilize 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`

# Indexed descriptions - Deriving a Generator

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

# 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)
```

# 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:

# Indexed descriptions - Deriving a Generator

We define a metadata structure:

```
data IDescM (P : Set □ → Set) : IDesc I → Set where
    `var~ : ∀ {i : I} → IDescM P (`var i)
    `1~ : IDescM P `1
    _`×~_ : ∀ {d₁ d₂ : IDesc □ I} → IDescM P d₁
          → IDescM P d₂ → IDescM P (d₁ `× d₂)
    `σ~ : ∀ {n : ℕ} {T : Sl (lift n) → IDesc □ I}
        → ((fn : Sl (lift n)) → IDescM P (T fn))
        → IDescM P (`σ n T)
    `Σ~ : ∀ {S : Set □} {T : S → IDesc □ I} → P S
        → ((s : S) → IDescM P (T s))
        → IDescM P (`Σ S T)
```

## Indexed descriptions - Deriving a Generator

We define a metadata structure:

```
data IDescM (P : Set ▯ → Set) : IDesc I → Set where
    `var~ : ∀ {i : I} → IDescM P (`var i)
    `1~ : IDescM P `1
    _`×~_ : ∀ {d₁ d₂ : IDesc ▯ I} → IDescM P d₁
          → IDescM P d₂ → IDescM P (d₁ `× d₂)
    `σ~ : ∀ {n : ℕ} {T : Sl (lift n) → IDesc ▯ I}
        → ((fn : Sl (lift n)) → IDescM P (T fn))
        → IDescM P (`σ n T)
    `Σ~ : ∀ {S : Set ▯} {T : S → IDesc ▯ I} → P S
        → ((s : S) → IDescM P (T s))
        → IDescM P (`Σ S T)
```

Essentially, this is a *singleton type* for descriptions, carrying extra information for the first components of dependent pairs.

# Indexed descriptions - Deriving a Generator

We parameterize `deriveGen` over a metadata structure containing generators

We 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 })
```

## 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

# 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 proof is mostly the same as for regular types, however the generator for dependent pairs is constructed using a monadic bind

# Indexed descriptions - Proving completeness

The proof is mostly the same as for regular types, however 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 using a metadata structure.

```
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

# 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?