Utrecht University

Generic Enumerators

Cas van der Rest, Wouter Swierstra, Manuel Chakravarty

Utrecht University

Introduction

Problem Statement

Conditional properties are common in property based testing

Problem Statement

Conditional properties are common in property based testing

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

Problem Statement

Conditional properties are common in property based testing

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

However, testing this property outright is problematic:

```
*> smallCheck 4 prop
Completed 21904 tests without failure.
But 18768 did not meet ==> condition.
```

Problem Statement

We could try our luck with a custom `Series` instance. However:

Problem Statement

We could try our luck with a custom `Series` instance. However:

- This can be very difficult depending on the specific constraints used
- We require a new instance everytime constraints change

Problem Statement

We could try our luck with a custom `Series` instance. However:

- This can be very difficult depending on the specific constraints used
- We require a new instance everytime constraints change

Is there a generic recipe for enumerators producing constrained test data?

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

Approach

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

Approach

We try to answer the following question: *how can we generically enumerate 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 enumerate 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 ⟧`

Enumerator completeness

We formulate the following completeness property for our enumerators:

```
Complete : ∀ {T} → (ℕ → List A) → Set
Complete enum = ∀ {x} → ∃[ n ] x ∈ enum n
```

Enumerator completeness

We formulate the following completeness property for our enumerators:

```
Complete : ∀ {T} → (ℕ → List A) → Set
Complete enum = ∀ {x} → ∃[ n ] x ∈ enum n
```

Although this property is relatively weak, it is a good sanity check.

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`, maps a value of type **Reg** to a value in `Set → Set`

Regular types - Semantics

The semantics, `⟦_⟧ : Reg → Set → Set`, maps 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
```

Regular types - Fixpoint operation

We use the following fixpoint operation:

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

Regular types - Deriving an enumerator

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

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

We use *do-notation* and *idiom* brackets to assemble enumerators, lifting the canonical `Monad` and `Applicative` instances for `List` to the function space `ℕ → List a`.

Regular types - Deriving an enumerator

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

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

The programmer somehow needs to provide an enumerator for constant types.

Regular types - Proving completeness

Basically, we do the following steps:

1. Prove the easy cases (unit types and empty types)
2. Prove that we combine products and coproducts in a completeness preserving way
3. Use the induction hypothesis to close the proof for recursive positions.

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
  In : ⟦ w ⟧ (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))
```

Indexed containers - Example

Let's consider vectors as an example

```
data Vec (A : Set) : ℕ → Set where
  nil  : Vec A 0
  cons : A → Vec A n → Vec A (suc n)
```

Indexed containers - Example

Let's consider vectors as an example

```
data Vec (A : Set) : ℕ → Set where
  nil  : Vec A 0
  cons : A → Vec A n → Vec A (suc n)

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

```
enumerate : ∀ {I : Set} → (S : Sig I)
          → (i : I) → ℕ → List (Fix S i)
enumerate (Op ◁ Ar | Ty) i = do
  op ← enumerate (Op i)
  ar ← coenumerate (Ar op) (Ar op)
    (λ ar → enumerate (Op ◁ Ar | Ty) (Ty ar))
  pure (In (op , ar x))
```

```
enumerate : ∀ {I : Set} → (S : Sig I)
          → (i : I) → ℕ → List (Fix S i)
enumerate (Op ◃ Ar | Ty) i = do
  op ← enumerate (Op i)
  ar ← coenumerate (Ar op) (Ar op)
    (λ ar → enumerate (Op ◃ Ar | Ty) (Ty ar))
  pure (In (op , ar x))
```

`coenumerate` enumerates function types

Indexed containers - Generic enumerator

```
enumerate : ∀ {I : Set} → (S : Sig I)
          → (i : I) → ℕ → List (Fix S i)
enumerate (Op ◁ Ar | Ty) i = do
  op ← enumerate (Op i)
  ar ← coenumerate (Ar op) (Ar op)
    (λ ar → enumerate (Op ◁ Ar | Ty) (Ty ar))
  pure (In (op , ar x))
```

`coenumerate` enumerates function types

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.

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 containers - Limitations

Not all indexed families can be described as an indexed container

Indexed containers - Limitations

Not all indexed families can be described as an indexed container

```
data STree (A : Set) : ℕ → Set where
  leaf : STree A 0
  node : ∀ {n m} → STree A n → A → STree A m
       → STree A (suc (n + m))
```

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 straightforward

```
⟦_⟧ : ∀ {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 straightforward

```
⟦_⟧ : ∀ {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`.

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

We associate the following fixpoint operation with this universe;

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

The enumerator type has the same structure as for regular types

```
enumerate : ∀ {I i φ} → (δ : IDesc I)
          → ℕ → List ⟦ δ ⟧ (Fix φ)
```

Indexed descriptions - Deriving an enumerator

The enumerator type has the same structure as for regular types

```
enumerate : ∀ {I i φ} → (δ : IDesc I)
          → ℕ → List ⟦ δ ⟧ (Fix φ)
```

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

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

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

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

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

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

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

How do we get `s`?

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

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

How do we get `s`?

We have the programmer supply an enumerator!

Indexed descriptions - Deriving an enumerator

We define a metadata structure:

Indexed descriptions - Deriving an enumerator

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 : Fin n → IDesc I}
        → ((fn : Fin 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 an enumerator

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 : Fin n → IDesc I}
        → ((fn : Fin 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 an enumerator

We parameterize `enumerate` over a metadata structure containing enumerators

Indexed descriptions - Deriving an enumerator

We parameterize `enumerate` over a metadata structure containing enumerators

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

In the case of `STree`, this means that we have to supply an enumerator that enumerates pairs of numbers and proofs that their sum is particular number

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

In the case of `STree`, this means that we have to supply an enumerator that enumerates pairs of numbers and proofs that their sum is particular number

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

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

In the case of `STree`, this means that we have to supply an enumerator that enumerates pairs of numbers and proofs that their sum is particular number

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

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

The user decides where the enumerator comes from

Indexed descriptions - Proving completeness

The completeness proof is roughly the same as the completeness proof for regular types

Indexed descriptions - Proving completeness

The completeness proof is roughly the same as the completeness proof for regular types

Additionally, we need to prove that our useage of monadic bind is also completeness preserving.

Conclusion

Summary

To summarize, we did the following:

1. Describe three type universes in Agda, and derive enumerators from codes in these universes
2. For two of these universes, prove that the enumerators derived from them are complete

Additionally, we have constructed a Haskell library that implements the generic enumerator for indexed descriptions

Questions?