Utrecht University

Generic enumerations: completely, fairly

Wouter Swierstra

Utrecht University

The enumeration problem

Problem: **Given the declaration of an algebraic data type, list all its inhabitants.**

The enumeration problem

Problem: Given the declaration of an algebraic data type, list all its inhabitants.

Enumerating the booleans may return a finite list:

```
True, False
```

The enumeration problem

Problem: Given the declaration of an algebraic data type, list all its inhabitants.

Enumerating the booleans may return a finite list:

```
True, False
```

Whereas enumerating binary trees should result in an infinite list:

```
Leaf, Node Leaf Leaf, Node (Node Leaf Leaf) Leaf, ...
```

The enumeration problem

Problem: Given the declaration of an algebraic data type, list all its inhabitants.

Enumerating the booleans may return a finite list:

```
True, False
```

Whereas enumerating binary trees should result in an infinite list:

```
Leaf, Node Leaf Leaf, Node (Node Leaf Leaf) Leaf, ...
```

Enumerating polymorphic or dependent data types requires a bit more work.

Why enumerate?

Property-based testing libraries, such as QuickCheck or SmallCheck in Haskell, try to falsify a given statement by passing (random) inputs to a function and observing its outputs.

For this to work, we need a way to generate values of (arbitrary) data types.

Some libraries (such as QuickCheck) generate random values; others *enumerate* all possible inputs up to some fixed size.

Why enumerate?

Property-based testing libraries, such as QuickCheck or SmallCheck in Haskell, try to falsify a given statement by passing (random) inputs to a function and observing its outputs.

For this to work, we need a way to generate values of (arbitrary) data types.

Some libraries (such as QuickCheck) generate random values; others *enumerate* all possible inputs up to some fixed size.

Can we define a data type generic enumeration algorithm?

What is the type of an enumeration?

To enumerate the elements of some data type amounts to listing its elements. A first approximation might be:

```
Enumerator a = List a
```

What is the type of an enumeration?

To enumerate the elements of some data type amounts to listing its elements. A first approximation might be:

```
Enumerator a = List a
```

However, recursive data types typically have infinitely many inhabitants. If we want to reason about our enumerators – the inhabitants obviously don't fit in a finite list.

What is the type of an enumeration?

We often model a datatype T as the (least) fixpoint of a functor:

$$\mu X \,.\, F\,X$$

What is the type of an enumeration?

We often model a datatype T as the (least) fixpoint of a functor:

$$\mu X \,.\, F X$$

- $F\,0$ corresponds to the non-recursive parts of this data type (where 0 is the empty type): e.g. the `Leaf` of a binary tree.

What is the type of an enumeration?

We often model a datatype T as the (least) fixpoint of a functor:

$$\mu X \,.\, F X$$

- $F\,0$ corresponds to the non-recursive parts of this data type (where 0 is the empty type): e.g. the `Leaf` of a binary tree.
- $F(F\,0)$ corresponds to the inhabitants that unroll a single layer of recursion: e.g. the tree `Node Leaf Leaf` (or just `Leaf`)

What is the type of an enumeration?

We often model a datatype T as the (least) fixpoint of a functor:

$$\mu X \, . \, F X$$

- $F\,0$ corresponds to the non-recursive parts of this data type (where 0 is the empty type): e.g. the `Leaf` of a binary tree.
- $F(F\,0)$ corresponds to the inhabitants that unroll a single layer of recursion: e.g. the tree `Node Leaf Leaf` (or just `Leaf`)
- $F(F(F\,0))$ corresponds to trees at most 'three constructors deep' e.g. `Node (Node Leaf Leaf) Leaf`, ...

What is the type of an enumeration?

We often model a datatype T as the (least) fixpoint of a functor:

$$\mu X \,.\, F X$$

- $F\,0$ corresponds to the non-recursive parts of this data type (where 0 is the empty type): e.g. the `Leaf` of a binary tree.
- $F(F\,0)$ corresponds to the inhabitants that unroll a single layer of recursion: e.g. the tree `Node Leaf Leaf` (or just `Leaf`)
- $F(F(F\,0))$ corresponds to trees at most ‘three constructors deep’ e.g. `Node (Node Leaf Leaf) Leaf`, ...

Idea: We can exhaustively enumerate all the inhabitants by considering increasingly large finite approximations.

What is the type of an enumeration?

Consequently, we might consider the more general type for our enumerations:

```
Enumerator a = List a → List a
```

The intuition here is that, given a list of ‘smaller’ inhabitants we have already constructed, we should be able to produce a new list of ‘bigger’ values.

Each data type declaration gives rise to such an enumerator.

What is the type of an enumeration?

Consequently, we might consider the more general type for our enumerations:

```
Enumerator a = List a → List a
```

The intuition here is that, given a list of 'smaller' inhabitants we have already constructed, we should be able to produce a new list of 'bigger' values.

Each data type declaration gives rise to such an enumerator.

But it's useful to separate the co- and contravariant occurrences of a and define:

```
Enumerator : Set → Set → Set
Enumerator a b = List a → List b
```

When a and b coincide, we can iterate this function (starting with an empty list) to enumerate increasingly 'large' inhabitants.

Today's program(me)

- Construct a collection of enumerator combinators – what properties should they satisfy?
- Use these to define an enumeration of all *regular types*
- Sketch how this approach also works for *indexed functors*

We will use Agda to define, specify and verify our enumerators.

Atomic enumerations

Let's start with 0 and 1 – the basic building blocks of our enumerations:

```
∅        : Enumerator A B
∅        = const []

pure     : B → Enumerator A B
pure x   = const [ x ]
```

Combining enumerations

Next we may want to combine two enumerations somehow:

```
_⟨|⟩_ : Enumerator A B → Enumerator A B  → Enumerator A B
e₁ ⟨|⟩ e₂  = λ as → (e₁ as) ++ (e₂ as)
```

Combining enumerations

Next we may want to combine two enumerations somehow:

```
_⟨|⟩_ : Enumerator A B → Enumerator A B  → Enumerator A B
e₁ ⟨|⟩ e₂  = λ as → (e₁ as) ++ (e₂ as)
```

But different choices exist! What properties do we expect of this function?

Combining enumerations: specification

Obviously, we should not discard elements:

```
inl  :  x ∈ xs  →  x ∈ (xs ++ ys)
inr  :  y ∈ ys  →  y ∈ (xs ++ ys)
```

Combining enumerations: specification

Obviously, we should not discard elements:

```
inl  :  x ∈ xs  →  x ∈ (xs ++ ys)
inr  :  y ∈ ys  →  y ∈ (xs ++ ys)
```

But typically such enumeration combinators should alse be *fair* – in that they should not favour elements drawn from either of its arguments.

Question: How should formulate this notion of fairness?

Fairness

When considering completeness, we use the membership relation:

```
data _∈_ : A → List A → Set where
  Here   : x ∈ (x :: xs)
  There  : x ∈ xs → x ∈ (y :: xs)
```

Each such proof can readily be mapped to a natural number:

```
|_| : x ∈ xs → Nat
| Here |      = Zero
| There p |   = Succ | p |
```

This induces an ordering on membership proofs, written `p ≺ q`.

A **fair** enumeration respects this ordering.

Fairness

Remember that we proved the following completeness properties:

```
inl   : x ∈ xs  → x ∈ (xs ++ ys)
inr   : y ∈ ys  → y ∈ (xs ++ ys)
```

Read constructively, they map positions in the input list to positions in the output list.

Fairness

Remember that we proved the following completeness properties:

```
inl   : x ∈ xs  → x ∈ (xs ++ ys)
inr   : y ∈ ys  → y ∈ (xs ++ ys)
```

Read constructively, they map positions in the input list to positions in the output list.

We can use these to formulate the property that `inl` and `inr` respect the ordering:

```
(p : x ∈ xs) (q : y ∈ ys)  → p ≺ q  →  inl p ≺ inr q
(p : x ∈ xs) (q : y ∈ ys)  → p ≺ q  →  inr p ≺ inl q
```

Note: that p and q need not refer to elements the same list.

The list append function satisfies the first property, but not the second.

A fair combination

The usual `interleave` function does satisfy these two properties.

As a result, we define the combination of enumerators in terms of interleaving:

```
_⟨|⟩_ : (e₁ e₂ : Enumerator A B) → Enumerator A B
e₁ ⟨|⟩ e₂ = λ as → interleave (e₁ as) (e₂ as)
```

And we can write (obviously trivial) enumerators:

```
bools : Enumerator Bool Bool
bools = pure true ⟨|⟩ pure false
```

But we'll need more than just choice...

Applicative enumerators

One useful combinator is the 'applicative star':

```
_⊛_          :  Enumerator C (A → B) → Enumerator C A → Enumerator C B
(e₁ ⊛ e₂) = λ cs → concat (map (λ f → map f (e2 cs)) (e1 cs))
```

But this is defined by mapping and concatenating results—this is not fair!

Applicative enumerators

One useful combinator is the 'applicative star':

```
_⊛_         :  Enumerator C (A → B) → Enumerator C A → Enumerator C B
(e₁ ⊛ e₂) = λ cs → concat (map (λ f → map f (e2 cs)) (e1 cs))
```

But this is defined by mapping and concatenating results—this is not fair!

A fairer definition flattens the *transposed* values:

```
_⊛_         :  Enumerator C (A → B) → Enumerator C A → Enumerator C B
e1 ⊛ e2 = λ cs → merge (map (λ f → map f (e2 cs)) (e1 cs))
  where
  merge = concat . transpose
```

We can still show this definition respects the ordering on positions – only now we have to talk about elements of a list-of-lists.

Cartesian products

We can use the applicative star to compute the cartesian product of elements drawn from two enumerators:

```
pairs       : Enumerator C A → Enumerator C B → Enumerator C (A × B)
pairs e₁ e₂  = pure _,_ ⊛ e₁ ⊛ e₂
```

Recursion

Now the hardest problem is—unsurprisingly–handling recursion.

Suppose we have the following Haskell data type for binary trees:

```
data Tree = Leaf | Node Tree Tree
```

If we naively try to compute the list of all trees up to a given depth, we might write:

```
trees : Nat → [Tree]
trees 0     = []
trees (n+1) = [ Leaf ] ++ [Node l r | l <- trees n, r <- trees n]
```

Recursion

Now the hardest problem is—unsurprisingly–handling recursion.

Suppose we have the following Haskell data type for binary trees:

```
data Tree = Leaf | Node Tree Tree
```

If we naively try to compute the list of all trees up to a given depth, we might write:

```
trees : Nat → [Tree]
trees 0     = []
trees (n+1) = [ Leaf ] ++ [Node l r | l <- trees n, r <- trees n]
```

But this is **very inefficient**!

In the same way the ‘naive’ Fibonacci definition fails to share recursive calls.

Better recursion

Recall that our enumerators have the following type:

```
Enumerator a b = List a → List b
```

In the special case where `a` and `b` coincide, we can refer to all the previously generated elements:

```
rec : Enumerator a a
rec = id
```

Better recursion

Recall that our enumerators have the following type:

```
Enumerator a b = List a → List b
```

In the special case where `a` and `b` coincide, we can refer to all the previously generated elements:

```
rec : Enumerator a a
rec = id
```

We can now write a more efficient enumerator, that recycles the previously enumerated trees:

```
trees = pure Leaf ⟨|⟩ pure Node ⊛ rec ⊛ rec
```

Producing values

We can iteratively apply an enumerator to an initially empty list:

```
enumerate : Enumerator a a → Nat → List a
enumerate e n = iterate n e []
```

Or produce a stream of infinite values. Or count the number of finite binary trees of a given size.

Generic enumerators

The enumerator for trees closely follows the data type declaration:

```
data Tree = Leaf | Node Tree Tree
```

```
trees = pure Leaf 〈|〉 pure Node ⊛ rec ⊛ rec
```

This is no coincidence – we can define a *datatype generic enumeration algorithm*:

- we define a uniform represention for a family of data types;
- define an algorithm over this representation type.

Generic programming

In Agda, we can write such generic programs by defining a *universe*:

```
data Desc : Set where
  zero one var : Desc
  _⊗_ _⊕_ : Desc → Desc → Desc
```

These descriptions correspond to the regular types: the empty type (`zero`), unit type (`one`), recursion (`var`), products (⊗) and coproducts (⊕).

Generic programming

In Agda, we can write such generic programs by defining a *universe*:

```
data Desc : Set where
  zero one var : Desc
  _⊗_ _⊕_ : Desc → Desc → Desc
```

These descriptions correspond to the regular types: the empty type (`zero`), unit type (`one`), recursion (`var`), products (⊗) and coproducts (⊕).

It is straightforward to map each such description to its corresponding functor:

```
  ⟦_⟧ : Desc → (Set → Set)
```

And finally, we tie the recursive know, computing the fixpoint of such functors:

```
  data Fix (D : Desc) : Set where
    In : ⟦ D ⟧ (Fix D) → Fix D
```

Generic enumerators

The generic enumerator is (almost) simple enough to fit on a single slide:

```
genumerate : (D : Desc) → Enumerator (Fix D) (⟦ D ⟧ (Fix D))
genumerate zero                   = ∅
genumerate one                    = pure unit
genumerate var                    = rec
genumerate (D₁ ⊕ D₂)              = (pure inj₁ ⊛ genumerate D₁)
                                      ⟨|⟩ (pure inj₂ ⊛ genumerate D₂)
genumerate (D₁ ⊗ D₂)              = pairs (genumerate D₁) (genumerate D₂)
```

This is reassuringly simple – but is it correct?

Complete enumeration

We call an enumerator `e : Enumerator a a` *complete* if it eventually produces each possible value. More formally:

```
Complete : (e : Enumerator a a) → Set
Complete e = ∀ (x : a) → ∃ n (x ∈ enumerate e n)
```

Is this *generic* enumerator complete?

Completeness proof - sketch

Given `x : Fix D` we can compute the *depth* of `x` – this is the obvious candidate for `n` (the number of iterations we apply the enumerating function).

But this proof requires **strong induction** – we need the completeness of all smaller depths, for instance, when handling the case for products.

```
∀ (D : Desc) (x : Fix D) (n : Nat) → depth x ≤ n → x ∈ genumerate D n
```

Enumerating dependent types

Somewhat surprisingly, defining a generic enumerator for dependent types (or more precisely, indexed families) follows the same pattern and is not much harder:

- define a universe closed under zero, one, recursion, coproducts, products and sigma types (dependent products);
- map descriptions to indexed functors `(I → Set) → Set`;
- define a generic enumeration function that unfolds one level of recursion - note that our type for indexed enumerators changes:

  `((i : I) → List (A i)) → List B`

  If `I` is a (regular) algebraic data type, we can memoise such functions using a generic trie.
- iterate this function to produce a list of `A i` for a given index `i`.
- prove completeness by computing the (generic) depth and using strong induction.

Discussion

- **Bad news:** When enumerating dependent types – such as the well-typed lambda terms – you may need to 'invent' indices. We can do this (assuming we know how to enumerate values of the index set) – but it's not very efficient.
- **Good news:** On the other hand, enumerating 'index-first' dependent types (where the value of the index determines the constructors) is no harder than enumerating the regular types.
- And at least this generic definition makes precise *where* such choices arise – and allows different heuristics to traverse the search space.

Discussion

- Fairness is a property of our combinators; completeness is a property of our enumerators.
- There’s a huge body of related work on LeanCheck, QuickCheck, QuickChick, SmallCheck, FEAT and many others – none are quite this simple.