Generic Enumerations: Completely, Fairly (Functional Pearl)

ANONYMOUS AUTHOR(S)

How can we enumerate the inhabitants of an algebraic data type? This pearl explores a *datatype generic* solution that works for a wide variety of types, including the *regular types* and *indexed families*. The enumerators presented here are provably *complete*—they will eventually produce every value—and *fair*—they avoid bias in the order of elements.

1 INTRODUCTION

To reduce the cost of formal verification, lightweight techniques—such as program testing—can help catch some errors early. *Property-based testing* is one approach to software testing that has been popularised by libraries such as QuickCheck [Claessen and Hughes 2000]. Property-based testing libraries try to find counterexamples that falsify a property that is expected to hold by passing automatically generated inputs to the programs being tested. If no counterexample can be found, the property may not hold in general—but in practice many 'obvious' errors in the code and its specification can be found in this fashion.

The central technology that underlies property-based testing libraries is the generation of suitable test values to serve as input to the programs being tested. This paper shows how to *enumerate* all the values inhabiting a given data type. This enumeration is *complete*—it is guaranteed to produce every inhabitant eventually—and *fair*—producing results in a balanced fashion. The enumerators we define here are *data type generic*, enumerating all the inhabitants of every *regular data type* and *indexed family*. Finally, although efficiency is not our primary concern, we show how we can exploit the recursive structure of our data types to avoid superfluous computation.

It is important to emphasise that this is not a new problem, or even a new idea. There is a substantial body of work on generating random data and enumerating data types—some of which we try to cover in the related work section of this paper; yet we feel the exposition here, showing how the enumeration of indexed data families is a natural generalisation of the enumeration of regular data types, is still worthwhile. We strive to keep the presentation simple and clean, while still formally verifying the key properties enumerations support.

About this paper. All definitions and proofs shown or mentioned in this paper have been formalised in Agda [Norell 2009], although we have taken some notational liberties to improve the presentation: we omit universally quantified implicits and universe levels. Although we use Agda in this paper to present our ideas, we believe that they are applicable in other proof assistants using dependent types, such as Coq [Coq Development Team 2020], Idris [Brady 2013], F★ [Swamy et al. 2016], or Lean [de Moura et al. 2015].

2 FAIR AND COMPLETE ENUMERATION

In this section, we will define the key types, combinators, and properties of enumerators that we will use throughout this paper. What does it mean to *enumerate* the inhabitants of a type A? The simplest definition might be some list of values of A:

```
Enumerator : Set → Set
Enumerator A = List A
```

2021. 2475-1421/2021/1-ART1 $15.00
https://doi.org/

Yet many recursive data types, such as trees or lists, have an *infinite* number of inhabitants. Hence a (finite) list will not suffice; we could use a (potentially infinite) 'co-list' instead, but instead we will choose a slightly different approach. The central type of this paper, Enumerator, is defined as follows:

```
Enumerator : Set → Set → Set
Enumerator A B = List A → List B
```

We define an *enumerator* as a function from lists to lists. Given a list of structurally 'smaller' ingredients of type A that we have already constructed, an enumerator builds a list containing 'larger' elements of type B. For the moment, however, we will not use argument list passed to an enumerator until consider the enumeration of recursive data types (Section 2.3).

2.1 Enumerator Combinators

The simplest enumerators are the empty enumerator (producing no elements) and singleton enumerators (producing exactly one element):

```
∅ : Enumerator A B
∅ = const []

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

Both these enumerators ignore their parameter and immediately return a list.

Furthermore, enumerators are functorial in their second argument; we can define the required operation (⟨$⟩) by mapping over the resulting list of values:

```
_⟨$⟩_ : (A → B) → Enumerator C A → Enumerator C B
f ⟨$⟩ e = map f ∘ e
```

Next, we would like to combine the elements produced by two enumerations using the following *choice* operator:

```
_⟨|⟩_ : (e₁ e₂ : Enumerator A B) → Enumerator A B
```

The obvious way to define this operation, is by appending the resulting lists:

```
(e₁ ⟨|⟩ e₂) as = (e₁ as) ++ (e₂ as)
```

At this point, however, it is worth thinking about the properties that we expect this combinator to satisfy. One important property is that each element produced by either e_1 or e_2 should also occur in $e_1 \langle|\rangle e_2$. To reason about the elements produced by our enumerators, we will use the _∈_ relation, capturing when an element occurs somewhere in a list:

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

It is easy to prove that the append operator on lists preserves this relation:

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

In practice, however, combining lists in this fashion is *biased*: all the elements of xs will appear in the resulting enumeration before the first element of ys. What property can we use to rule out this definition?

Fairness. To avoid this bias, we begin by defining an ordering on the elements of our enumerations. To do so, we begin by mapping each position in a list to its corresponding natural number:

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

Now we can compare two positions—not necessarily in the same list—by using the familiar ordering on their underlying natural numbers:

```
_≺_ : x ∈ xs → y ∈ ys → Set
p ≺ q = | p | < | q |
```

Now that we have on order on positions, we can return to our original problem: formulating and proving fairness of the choice operator. The inl and inr lemmas above prove that the **+** operation does not discard elements; constructively, however, we can also regard them as *functions* that compute where the elements of xs and ys will appear in the resulting list. Using our ordering on positions, we can now use the inl and inr lemmas to formulate the following fairness properties:

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

The **+** operator satisfies the first property, but not the second: the first element of ys will come after the last element of xs in xs **+** ys. For this reason, as the **+** operation does not respect the order of elements, we consider it to be *unfair*.

Fair choice. So what is a fair notion of choice operator? Unsurprisingly, the solution is to draw elements alternatingly from the two lists:

```
interleave : List A → List A → List A
interleave [] ys       = ys
interleave (x :: xs) ys = x :: interleave ys xs
```

In contrast to the list append function, interleave is *fair*. To establish this, we begin by showing that it does not discard elements:

```
interleave∈-left  : (xs ys : List a) → x ∈ xs → x ∈ interleave xs ys
interleave∈-right : (xs ys : List a) → y ∈ ys → y ∈ interleave xs ys
```

In contrast to appending lists, however, *interleaving* lists is *fair*, as witnessed by a pair of lemmas with the following types:

```
(p : x ∈ xs) (q : y ∈ ys) → p ≺ q → (interleave∈-left  xs ys p) ≺ (interleave∈-right xs ys q)
(p : x ∈ xs) (q : y ∈ ys) → p ≺ q → (interleave∈-right xs ys p) ≺ (interleave∈-left  xs ys q)
```

Using the interleave function, we can now define a fair choice operation on enumerators easily enough:

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

We can use the choice operation to enumerate types that have more than one constructor, such as the booleans:

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

2.2 Fairly applicative & fairly monadic

Next, we would like to show that these enumerators are applicative by defining the usual applicative ⊛ operator:

```
_⊛_ : Enumerator C (A → B) → Enumerator C A → Enumerator C B
```

The 'obvious' definition of ⊛ uses the concatMap on the underlying lists:

```
(e₁ ⊛ e₂) ts = concatMap (λ f → map f (e₂ ts)) (e₁ ts)
```

However, just as we saw for **+**, the concat function is not fair: it is biased towards elements that occur in earlier lists. Recall that the concat function has the following type:

```
concat : List (List A) → List A
```

Where our previous fairness properties established that the order of elements in a list are preserved, we now need to reason about (the order of) elements in lists-of-lists. To do so, we define the following relation:

```
data _∈∈_ (x : A) (xss : List (List A)) : Set where
  _,_ : {xs : List A} → xs ∈ xss → x ∈ xs → x ∈∈ xss
```

That is, to establish that `x : a` occurs in `xss : List (List a)` we need to find an `xs` such that `x ∈ xs` and `xs ∈ xss`. Using the comma as constructor enables us to write such proofs as `(p , q)`, reminiscent of the notation used to denote a point on the Cartesian plane.

The key idea behind our fair version of the applicative ⊛ operator is to traverse the list-of-lists in column major order: beginning with all the first elements of the inner lists, before continuing recursively. Or equivalently, we will transpose and then flatten the outer list. Doing so will respect the order of the elements in the inner lists, ensuring that we select elements for the list of enumerated values fairly.

Before defining the transpose function, we need an auxiliary definition:

```
zipCons : List A → List (List A) → List (List A)
zipCons []       yss        = yss
zipCons (x ∷ xs) []         = map [_] (x ∷ xs)
zipCons (x ∷ xs) (ys ∷ yss) = (x ∷ ys) ∷ zipCons xs yss
```

The call `zipCons xs xss` function adds the i-th element of `xs` to the head of the i-th list in `xss`. We need to take some care here to ensure that elements are not discarded if the length of the lists differ. The following two lemmas ensure that `zipCons` does not discard elements:

```
zipCons∈  : ∀ x ∈ xs → x ∈∈ zipCons xs xss
zipCons∈∈ : ∀ x ∈∈ xss → x ∈∈ zipCons xs xss
```

Using `zipCons`, we can now define the transpose function directly:

```
transpose : List (List A) → List (List A)
transpose []         = []
transpose (xs ∷ xss) = zipCons xs (transpose xss)
```

As you would expect, `transpose` also does not discard elements:

```
transpose∈∈ : x ∈∈ xss → x ∈∈ transpose xss
```

Finally, we can use `transpose` to define a fair merge operation that begins by listing the first elements of its constituent lists, followed by the second elements, and so forth:

```
merge : List (List A) → List A
merge xss = concat (transpose xss)
```

Once again, the merge operation preserves the elements:

```
merge∈∈ : x ∈∈ xss → x ∈ merge xss
merge∈∈ h = concat∈∈ (transpose∈∈ h)
```

Furthermore, we can prove that the merge operation is fair in the following sense:

```
merge-fair : (p₁ : xs ∈ xss) (p₂ : x ∈ xs) (q₁ : ys ∈ xss) (q₂ : y ∈ ys) →
  p₂ ≺ q₂ → merge∈∈ (p₁ , p₂) ≺ merge∈∈ (q₁ , q₂)
```

This proof of fairness relies on a characteristic property of transposition:

```
|transpose| : (p₁ : xs ∈ xss) (p₂ : x ∈ xs) → | p₂ | ≡ | fst (transpose∈∈ (p₁ , p₂)) |
```

In other words, the transpose function maps every element of the i-th column to a position in the i-th row. The merge-fair lemma follows immediately from this property.

Using this merge operation, we can now finally give a fair definition of the applicative ⊛ operator for our enumerators:

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

This allows us to write enumerators in the familiar applicative style. For example, we can compute the Cartesian product of elements generated by two enumerators as follows:

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

These enumerators are not only applicative, but also monadic:

```
_>>=_ : Enumerator C A → (A → Enumerator C B) → Enumerator C B
(e₁ >>= e₂) = λ cs → merge (map (λ x → e₂ x cs) (e₁ cs))
```

The monadic structure of enumerators is necessary to combine enumerators whose *type may depend* on a previously generated element. Once we attempt to enumerate indexed families, the monadic bind operation becomes essential. To illustrate this point, consider the following enumerator for *dependent pairs*, also known as Σ-types:

```
sigmas : Enumerator C A → ((x : A) → Enumerator C (B x)) → Enumerator C (Σ A B)
sigmas e f = e >>= λ x →
             f x >>= λ y →
             pure (x , y)
```

Since the type enumerated by f is *dependent* on its argument, the value generated for the first element of the pair, x, needs to be in scope to extract the corresponding enumerator. It is instructive to compare this enumerator with the one for pairs we saw previously: in the dependent case, the choice of the *value* for the first component influences the enumeration of the second component.

2.3 Recursive enumerators

How can we define an enumerator for a recursive type? This will be where we use the additional argument passed to each enumerator. Consider the following data type for binary trees:

```
data Tree : Set where
  Leaf : Tree
  Node : Tree → Tree → Tree
```

If we naively try to compute the list of trees of a given size, we might use the applicative instance for lists to write:

```
list-trees : Nat → List Tree
list-trees Zero     = []
list-trees (Succ n) = [ Leaf ] ++ Node ⟨$⟩ list-trees n ⊛ list-trees n
```

In this way, a call to list-trees n will compute a list of trees with depth at most n. There is, however, a problem with this definition: the two calls to trees n give rise to an exponentially slow function. Fortunately, there is a well-known solution: we can pass the result of the previous recursive as an argument to our enumerator, avoiding the superfluous recomputation. This is where our additional list argument in the definition of the enumerator type will finally be used.

Firstly, we can define the following trivial enumerator, rec, that simply returns its argument list:

```
rec : Enumerator A A
rec = λ as → as
```

We can now use almost all the combinators we have seen so far to define a 'recursive' enumerator for trees:

```
trees : Enumerator Tree Tree
trees = pure Leaf ⟨|⟩ Node ⟨$⟩ rec ⊛ rec
```

Note that this enumerator is not really recursive: it simply defines a function List A → List A. By iteratively applying this function to an initially empty list we can create lists of increasingly deep trees. More generally, we can define the enumerate function that produces a finite list of elements of type A from its argument enumerator:

```
enumerate : Enumerator A A → Nat → List A
enumerate e n = iterate n e []
  where
  iterate : Nat → (A → A) → A → A
  iterate Zero     f x = x
  iterate (Succ n) f x = f (iterate n f x)
```

Crucially, we avoid unnecessary recursive calls in this style, as we saw in the list-trees function. Here all the 'smaller' trees are passed as an argument to the trees function; the trees function itself describes a single step in the generation process, assembling larger trees from the subtrees in its argument list. Of course, there are other possible interpretations of an enumerator, such as producing an infinite stream of increasingly long lists. For the purpose of this paper, however, we will only concern ourselves with the enumerate function.

2.4 Enumerator completeness

The type of our enumerators does not guarantee anything about its behaviour. For example, the following enumerator for the booleans is type correct, but wrong:

```
boolsWrong : Enumerator Bool Bool
boolsWrong = ∅
```

To rule out such definitions, we identify the key property that our enumerators must satisfy: *completeness*. An enumerator is *complete* when every possible value of a type will eventually be generated. In the remainder of this section, we will make this precise.

We begin by defining the following Occurs relation:

```
data Occurs (x : A) (e : Enumerator A A) : Set where
  occurs : (n : Nat) → x ∈ enumerate e n → Occurs x e
```

When there is some natural number n such that x ∈ enumerate e n, we say that x *occurs* in the enumerator e. An enumerator e is complete when each x : A occurs in e:

```
Complete : Enumerator A A → Set
Complete e = ∀ x → Occurs x e
```

To prove an enumerator e is Complete amounts to showing that for every value x : A, we will eventually produce x in the list enumerator e n for large enough values of n.

To demonstrate how completeness proofs may help to weed out erroneous, but type-correct definitions, we consider the completeness proof for the simple enumerator of the booleans, bools, that we saw previously:

```
bools-complete : Complete bools
bools-complete false = occurs 1 (Here)
bools-complete true  = occurs 1 (There Here)
```

On the other hand, it is *not* possible to construct a proof that boolsWrong is a complete enumerator; *a fortiori*, we can prove that boolsWrong is not complete:

```
boolsWrong-not-complete : Complete boolsWrong → ⊥
```

In what follows, we will rarely write completeness proofs by hand, but rather define *generic* enumerators that are complete by construction.

3 GENERIC ENUMERATION OF REGULAR TYPES

In the previous section, we gave a handful of example of enumerator for booleans and trees. In this section, we show to generalise these results and write a generic enumerator for a collection simple algebraic data types; that is, we show how suitable enumerators can be *generated* by induction over the structure of such types.

To achieve this, we will reify a collection of types as values of some *universe* U : Set. A universe is accompanied by a semantics, ⟦_⟧, that interprets values in U as an Agda type. To define a generic enumerator (approximately) amounts to defining a function:

```
enumerate : (u : U) → Enumerator ⟦ u ⟧ ⟦ u ⟧
```

To illustrate the general approach, we start by defining enumerators for the *regular types* before moving on to a more complicated universe in the next section. Despite its simplicity, this universe is able to describe many familiar, simple algebraic data types.

3.1 Regular types

The universe of regular types contains the *empty type* (zero), *unit type* (one), *recursion* (var) and *type constants* (k), and is closed under *products* (⊗) and *coproducts* (⊕). We describe regular types as values of the description type Desc:[1]

```
data Desc (P : Set → Set) : Set where
  zero : Desc P
  one  : Desc P
  var  : Desc P
  k    : (S : Set) → {P S} → Desc P
  _⊗_  : (D₁ D₂ : Desc P) → Desc P
  _⊕_  : (D₁ D₂ : Desc P) → Desc P
```

[1] The Desc type, as presented here, is large as the constant constructor quantifies over all types. While we omit universe levels from the typeset version of this paper, it is easy to stratify this construction by only allowing constants drawn from some smaller universe U : Set.

This definition is mostly standard. Descriptions have an extra parameter, P : Set ⟶ Set, that describes what (if any) extra information needs to be recorded for the constants. In what follows, we will use this to require information about how to enumerate the inhabitants of type constants that appear in a description.

To write generic programs, we need to give an *interpretation* (or *semantics*) to descriptions. We define the semantics of descriptions as a functor Set ⟶ Set in the usual fashion:

```
⟦_⟧ : Desc P ⟶ (Set ⟶ Set)
⟦ zero    ⟧ X = ⊥
⟦ one     ⟧ X = ⊤
⟦ var     ⟧ X = X
⟦ k S     ⟧ X = S
⟦ D₁ ⊗ D₂ ⟧ X = ⟦ D₁ ⟧ X × ⟦ D₂ ⟧ X
⟦ D₁ ⊕ D₂ ⟧ X = ⟦ D₁ ⟧ X ⊎ ⟦ D₂ ⟧ X
```

This definition is entirely standard. By taking the fix-point of these functors, we can model simple recursive data types such trees and lists. The Fix data type ties the recursive knot:

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

By defining the Desc data type explicitly, allows us to define (generic) functions by pattern matching on the constructors of Desc.

Example: Lists. As an example, we consider how to encode the List type as a value of Desc:

```
data List (A : Set) : Set where
  []  : List A
  _::_ : A ⟶ List A ⟶ List A
```

We choose the description ListD : Set ⟶ Desc such that Fix (ListD A) is isomorphic to List A:

```
ListD : Set ⟶ Desc (λ _ ⟶ ⊤)
ListD A = one ⊕ (k A ⊗ var)
```

The description ListD consists of a *coproduct* (or *choice*) of either one (representing the empty list, []), or a pair consisting of a constant value of type A, and a recursive position (corresponding to ::). We can describe the singleton list 0 :: [], for example, as a value of type Fix (ListD Nat):

```
consZeroNil = In (inj₂ (0 , In (inj₁ tt)))
```

3.2 A Generic Enumerator For Regular Types

We are now ready to define a generic enumerator for regular types. Down the line, this means that we give a definition for an generic enumeration function, genumerate, with the following type:

```
genumerate : (D : Desc List) ⟶ Enumerator (Fix D) (Fix D)
```

Given any description D we will enumerate the recursive data types that can be built from this description. Note that we expect a description, D : Desc List, that already stores a (finite) list of all the constant types that occur in our descriptions.

We cannot define this genumerate function directly. In particular, because Desc is closed under products and coproducts, we need to *recurse* over the description as we define its enumerator. To do so, we must be careful to separate the description under consideration (D_1) from the description that describes the type of recursive positions (D_2):

```
enumerateD : ∀ (D₁ D₂ : Desc List) ⟶ Enumerator (Fix D₂) (⟦ D₁ ⟧ (Fix D₂))
```

This is a common pattern when defining such generic functions—passing two descriptions to a generic function: one representing the *top-level* description; whereas the other description is traversed recursively.

The definition of this enumerateD function is now immediate, using all the auxiliary functions defined in the previous section.

```
enumerateD : ∀ (D : Desc List) {D' : Desc List} → Enumerator (Fix D') (⟦ D ⟧ (Fix D'))
enumerateD zero        = ∅
enumerateD one         = pure tt
enumerateD (k A {as})  = const as
enumerateD var         = rec
enumerateD (D₁ ⊕ D₂)   = (inj₁ <$> enumerateD D₁) <|> (inj₂ <$> enumerateD D₂)
enumerateD (D₁ ⊗ D₂)   = pairs (enumerateD D₁) (enumerateD D₂)
```

For the sake of completeness, we briefly go through the individual cases one by one. As there are no inhabitants of the empty type, we simply return the empty list in the case for zero. Similarly, there is a single inhabitant of the unit type. In the case for one we therefore return the singleton list with the value tt. When we encounter a constant type A, we have an implicit argument as : List A. We can simply return this list of values, ignoring the list of subtrees we receive as an additional argument.

This leaves the three most interesting cases: recursive positions, coproducts and products. When we encounter a recursive position designated by the var constructor, we return the list of 'smaller' values that we are passed as an argument. This is similar to how we generated subtrees for the Node constructor in enumerator for binary trees in the previous section. In the case for coproducts, $D_1 \oplus D_2$, we make two recursive calls on both D_1 and D_2, map the injections inj_1 and inj_2 over these results, and interleave the resulting values. Finally, in the case for products takes a Cartesian product of the two recursive calls. The pairs function that computes this Cartesian product is defined in Section 2.

Using the enumerateD function, we can now write our generic enumerator as follows:

```
genumerate : (D : Desc List) → Enumerator (Fix D) (Fix D)
genumerate D = λ ts → map In (enumerateD D ts)
```

This function simply calls the enumerateD function with the description D. This will result in a list of values of type ⟦ D ⟧ (Fix D); mapping the In constructor over this list of values produces the desired List (Fix D).

Example: enumerating lists. To illustrate our generic enumerator in action, we can revisit the description of lists we saw previously. We begin by defining the following description for lists of a given type A:

```
ListD : {A : Set} → List A → Desc List
ListD {A} as = one ⊕ (k A {as} ⊗ var)
```

The ListD function requires an argument as : List A, enumerating the elements of A. We can use this description to enumerate all lists up to some length as follows:

```
lists : {A : Set} → (xs : List A) → Nat → List (Fix (ListD xs))
lists xs = enumerate (genumerate (ListD xs))
```

For example, the following expression enumerates all lists consisting of at most three constructors, containing the characters 'a' and 'b':

```
lists ('a' ∷ ('b' ∷ [])) 3
```

This example illustrates most of the constructors of our Desc type. In particular, we can use the enumerators for constant types to generate primitive values such as characters, that have no associated data type declaration.

Completeness. We now briefly sketch the *completeness* proof, establishing that our generic enumerators will eventually produce every possible value.

To prove our generic enumerators are complete, amounts to showing that for all x : Fix D there is an n : Nat such that x ∈ enumerate (genumerate D) n. It should not come as a surprise that the required number n corresponds to the number of times we need to unroll the fixed-point to produce x. We refer to this number as the depth of a given tree; it can be readily computed as follows:

```
mutual
  depthD : (D : Desc P) → {D' : Desc P} → ⟦ D ⟧ (Fix D') → Nat
  depthD zero       _       = 0
  depthD one        _       = 0
  depthD (k _)      _       = 0
  depthD var        x       = depth _ x
  depthD (D₁ ⊕ D₂) (inj₁ x) = depthD D₁ x
  depthD (D₁ ⊕ D₂) (inj₂ y) = depthD D₂ y
  depthD (D₁ ⊗ D₂) (x , y)  = max (depthD D₁ x) (depthD D₂ y)

  depth : (D : Desc P) → Fix D → Nat
  depth D (In x) = Succ (depthD D x)
```

To prove that some x : Fix D is indeed in the corresponding enumerator requires some thought. We need a careful recursive argument: in particular, the depth of a pair returns the *maximum* depth of its elements. As a result, we need to use *strong induction* to show that our generic enumerator is complete, i.e. we can formulate and prove the following property:

```
completeD : (D : Desc List) (x : ⟦ D ⟧ (Fix D')) (xs : List (Fix D')) →
  ((y : (Fix D')) → depth y ≤ depthR D x → y ∈ xs) →
  x ∈ enumerateD D xs
```

Informally, this property states that x is guaranteed to occur in the the generic enumerator built from the list of values xs, provided each subtree y that x may contain already occurs in xs. The proof itself follows from the key property of our enumerator combinators that we showed in Section 2: they never discard elements.

Next, we can define the corresponding top-level proof that calls the completeD lemma, while passing itself recursively to prove the completeness of any recursive calls:

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

Finally, we can use this lemma to establish that all our generic enumerators are complete:

```
genumerateComplete : (D : Desc List) → Complete (genumerate D)
```

We have chosen to ignore constant types in this proof sketch. To complete the proof, we need to extend the completeD lemma with a further assumption that the lists of elements associated with the constant types that occur in D exhaustively enumerate all possible constants. Nonetheless, the proof terms for complete and completeD are fairly straightforward—once these definitions are fixed—spanning about twenty lines of proof and using a handful of auxiliary lemmas.

4 GENERIC ENUMERATORS FOR INDEXED FAMILIES

While regular types are fairly straightforward to enumerate, the enumeration of *indexed* types is more of a challenge. To tackle this problem, we need to shift from our universe of regular types to one capable of describing indexed data types.

4.1 Universe Definition

The universe of *indexed descriptions* describes a wide collection of indexed data types. We closely follow the exposition by Dagand [Dagand 2013], but similar constructions are ubiquitous in generic programming with indexed families [Benke et al. 2003; Chapman et al. 2010; Dagand and McBride 2012]. Where previously we constructed the codes regular types directly as a value in Desc P, we need to generalise this to handle indexed families of types. To do so, we introduce the following type constructor:

```
Func : (Set → Set) → Set → Set
Func P I = I → IDesc P I
```

The type I corresponds to the index set. For example, vectors are indexed by a natural number. To describe such indexed families, we define a function Func P I that computes the indexed description for each possible value i : I.

The type of codes, IDesc, is similar to the codes for regular types that we saw previously:

```
data IDesc (P : Set → Set) (I : Set) : Set where
  zero : IDesc P I
  one  : IDesc P I
  var  : (i : I) → IDesc P I
  _⊕_  : (D₁ D₂ : IDesc P I) → IDesc P I
  _⊗_  : (D₁ D₂ : IDesc P I) → IDesc P I
  ‘Σ   : (S : Set) → ⟦ P S ⟧ →
         (S → IDesc P I) → IDesc P I
```

The IDesc data type has constructors for the empty type (zero), unit type (one), the recursive positions (var) and is closed under products (⊗) and coproducts (⊕). Note that the recursive positions now contain further index information: the var constructor takes a value i : I as its argument. We use this value to designate the index associated with each recursive position. Finally, indexed descriptions are closed under *dependent products* (‘Σ), consisting of a constant type S and a description depending on S. We again include an extra parameter P : Set → Set to allow for extra information to be stored about the constant type stored in the first component of a dependent pair. We shall see examples of these indexed description shortly, but first we need to assign them semantics.

The associated semantics, ⟦_⟧, interprets a code with index type I to a function (I → Set) → Set. The argument function, I → Set, is used to assign semantics to the recursive positions:

```
⟦_⟧ : IDesc P I → (I → Set) → Set
⟦ one     ⟧ X = ⊤
⟦ zero    ⟧ X = ⊥
⟦ var i   ⟧ X = X i
⟦ D₁ ⊗ D₂ ⟧ X = ⟦ D₁ ⟧ X × ⟦ D₂ ⟧ X
⟦ D₁ ⊕ D₂ ⟧ X = ⟦ D₁ ⟧ X ⊎ ⟦ D₂ ⟧ X
⟦ ‘Σ S f   ⟧ X = Σ S λ s → ⟦ f s ⟧ X
```

Finally, we use the data type Fix to tie the recursive knot and take the least fix-point of indexed descriptions:

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

Example: Vectors. As an example, we consider the familiar example of a dependent type, namely vectors:

```
data Vec (A : Set) : Nat → Set where
  []  : Vec A Zero
  _::_ : A → Vec A n → Vec A (Succ n)
```

A value of type Vec A n is only inhabited by lists of length n. We can describe Vec as follows:

```
VecF : Set → Func (λ _ → ⊤) Nat
VecF _ Zero      = one
VecF A (Succ n) = `Σ A (λ _ → var n)
```

We choose the indexed description VecF carefully such that Fix (VecF A) n is isomorphic to Vec A n. Rather than modelling the choice between the constructors [] and :: as a coproduct, we use the fact that there is only one constructor of Vec available for each constructor of the index, returning one (corresponding to []) if the length is Zero, and a pair consisting of a value of type A and a recursive position with index n (corresponding to ::) if the index is of the form Succ n. Of course, the lack of coproducts in this description is specific to vectors: each index is associated with a single constructor.

4.2 A generic enumerator for indexed types

The generic enumerator for regular types is straightforward, once we defined the types and combinators for defining enumerators. In this section, we show how it can be extended to an enumerator for the *indexed descriptions*.

First, we revisit our type of enumerators. Rather than pass in a list of 'smaller values' as we did previously, we need to account for the additional index information. In particular, we are no longer passed a single list, but rather a function that maps each index i : I to a list of smaller values:

```
IEnumerator : {I : Set} → (I → Set) → Set → Set
IEnumerator {I} A B = ((i : I) → List (A i)) → List B
```

While we could also let the result type B depend on I, we will refrain from doing so—we will not need this additional generality. The semantics of our indexed descriptions ⟦_⟧ simply returns a Set—hence it suffices to generate a simple list of values. Note that these indexed enumerators are strictly more general than the simple Enumerator type from the introduction. Instantiating the index set to the unit type, yields a type that is isomorphic to the original enumerators defined in Section 2. Throughout the remainder of this section, we will use the familiar combinators for writing enumerators—even if we should strictly speaking provide alternative versions with the same definition, but a more general (indexed) type.

The generic enumerator once again consists of two parts: the first pattern matches on its argument description; the second is used to recurse back to the top-level description being enumerated. The first part, ienumerateD, produces a list of values of type ⟦ D ⟧ (Fix φ), given a description D and interpretation of the recursive positions, φ. The definition is reassuringly familiar, as most cases follow the same structure as we saw for the regular types.

```
ienumerateD : (desc : IDesc List I) → IEnumerator (Fix φ) (⟦ desc ⟧ (Fix φ))
ienumerateD zero        = ∅
ienumerateD one         = pure [ tt ]
ienumerateD (D₁ ⊗ D₂)   = pairs (ienumerateD D₁) (ienumerateD D₂)
ienumerateD (D₁ ⊗ D₂)   = (inj₁ ⟨$⟩ enumerateD D₁) ⟨|⟩ (inj₂ ⟨$⟩ enumerateD D₂)
ienumerateD (var i)     = λ rec → rec i
ienumerateD (`Σ S {e} f) = e ≫= λ s →
                           ienumerateD (f s) ≫= λ x →
                           return (s , x)
```

The first four cases should be familiar: the empty type, the unit type, products and coproducts were all covered previously. When we encounter a recursive subtree, var i, we once again use the list of smaller values we are passed. Rather than return the list directly, as we did for regular types, we return the list of values *at index i*. Finally, in the case for dependent pairs, `Σ, we use the (implicit) enumerator, e, stored in the constructor to produce a value of type S; the second component, is then produced using a recursive call to the enumeratorD function using f s as the new description to enumerate.

The top-level generic igenumerate invokes ienumerateD, instantiating the indexed description with φ i:

```
igenumerate : ∀ φ i → IEnumerator (Fix φ) (Fix φ i)
igenumerate φ i = In ⟨$⟩ ienumerateD (φ i) φ
```

Finally, we adapt the previous definition of our enumerate function to iteratively apply our enumerators a fixed number of times:

```
ienumerate : ((i : I) → IEnumerator A (A i)) → (i : I) → Nat → List (A i)
ienumerate f i Zero     = []
ienumerate f i (Succ n) = f i (λ i → ienumerate f i n)
```

Proving completeness

One pleasant property of our development is that many definitions and proofs on the universe of regular types can be easily extended to indexed families. Just as we defined the depth function on regular types, the idepth function counts the number of times the (indexed) functor must be unrolled to produce a given value:

```
idepthD : (D : IDesc P I) → ⟦ D ⟧ (Fix φ) → Nat
idepth  : ∀ Fix φ i → Nat
```

With these definitions in place, we can once again proceed to define the key lemma, icompleteD, by induction on the indexed description D:

```
icompleteD : (D : IDesc List I) (x : ⟦ D ⟧ (Fix φ)) (xsi : (i : I) → List (Fix φ i)) →
  (∀ i → (y : Fix φ i) → idepth y < idepthD D x → y ∈ xsi i) →
  x ∈ ienumerateD D xsi
```

The general structure of this proof is the same as we saw for the regular universe, completeD. There are a few differences worth pointing out. Instead of receiving a list of 'smaller' values that have previously been constructed, we are passed a function xsi : (i : I) → List (Fix φ i), that computes a list of values for each possible index. The stronger induction hypothesis in the penultimate argument guarantees that any y : Fix φ i will appear in the list associated with the index i. Each cases of this proof closely follows its regular counterpart. The base case for one is trivial; the cases for products and coproducts relies on the completeness of the pairs and interleave

combinators; in the case for recursive subtrees, we use our induction hypothesis. If D is a dependent pair, however, the proof is slightly more challenging. Recall that ‘Σ correspond to dependent products—we can mimic the completeness proof for pairs, though we have to prove new auxiliary lemmas that establish well-behavedness of the sigmas combinator. To prove completeness, we do require completeness of the enumerator for the set S that is used by the dependent product—just as we did for constants in the universe of regular types.

Finally, we can provide a suitable top-level completeness statement. The type of this statement is daunting at first, but captures the same style of recursion as we saw for the complete lemma in the previous section:

```
icomplete : ∀ (φ : I → IDesc List I) (i : I) (x : Fix φ i) (n : Nat) →
  idepth x ≤ n → x ∈ ienumerate (igenumerate φ) n i
```

Its proof is analogous: pattern matching on the In constructor and calling the icompleteD lemma sketched above. Once again, this proof sketch does not explicitly mention the constant types that appear in an indexed description, such as the type S that occurs in the ‘Σ constructor. To handle these, we require an additional argument explicitly assuming that the lists stored for these values are also complete.

5 CASE STUDY: ENUMERATING LAMBDA TERMS

In this section, we show how the enumerators we have defined can be applied to enumerate well-typed lambda terms. While this problem is well-studied [Fetscher et al. 2015; Palka et al. 2011; Tarau 2015; Yakushev and Jeuring 2009], our goal is not to write a novel or particularly efficient enumerator, but rather to illustrate how such an enumerator follows automatically from the definitions we have seen so far.

Let us first take a look at how to encode well-typed λ-terms as an indexed data type. Terms are indexed by a *context* and a *type* (Type). Types can be either a function (⟹) or the base type (ι):

```
data Type : Set where
  _⟹_ : Type → Type → Type
  ι : Type

Ctx = List Type
```

We can then write the following well-known intrinsically-typed abstract syntax type for the simply-typed λ-calculus:

```
data Term : Ctx → Type → Set where
  Var : σ ∈ Γ → Term Γ σ
  Abs : Term (σ ∷ Γ) τ → Term Γ (σ ⟹ τ)
  App : Term Γ (σ ⟹ τ) → Term Γ σ → Term Γ τ
```

The definition of Term is carefully chosen so that it is only inhabited by well-formed terms. Terms are not only indexed by their type, but also by a context. To reflect this in the indexed description we will build for Term, we simply uncurry its type declaration, choosing the product type Ctx × Type the index of our descriptions.

5.1 A description of well typed terms

To enumerate these well-typed terms, we need to represent them using the (indexed) descriptions we have defined previously. We will write descriptions for the each of the constructors of Term individually, before combining them into a single indexed description.

We start with an enumerator for membership proofs, which are used by the Var constructor. We could use our generic enumerator for indexed families—but we will begin by showing how

to define such an indexed enumerator by hand using a decision procedure type-eq? : (σ τ : Type) → Dec (σ ≡ τ), that determines when two types are equal:

```
elems : (Γ : Ctx) (σ : Type) → Enumerator A (σ ∈ Γ)
elems []       σ = ∅
elems (τ :: Γ) σ with type-eq? σ τ
... | yes refl = pure Here ⟨|⟩ There ⟨$⟩ elems Γ σ
... | no neq   = There ⟨$⟩ elems Γ σ
```

Essentially, this function finds all the occurrences of a type σ in the context Γ.

Additionally, we will need to enumerate the types of the simply typed lambda calculus. To see this, consider the App constructor of the Term data type. The type of the functions argument, σ in our definition, is (implicitly) bound—to use the App constructor to generate new terms, we will need to choose a suitable value for σ. To define this enumerator, we will the generic enumerator for regular types. To do so, we need to define a description corresponding to the types of our simply typed lambda calculus:

```
TypeD : Desc List
TypeD = one ⊕ (var ⊗ var)
```

This is the description as we saw for binary trees in Section 2. To get our hands on an enumerator for Type data type, we need to do a bit more work. Firstly, we observe that for any pair of isomorphic types A and B, we can convert an enumerator of on A to one on B:

```
≃enum : A ≃ B → Enumerator A A → Enumerator B B
```

Using this, we can insight we can convert the generic enumerator TypeD to an enumerator for Type, rather than Fix TypeD:

```
types : Enumerator Type Type
types = ≃enum type-iso (genumerate TypeD)
```

Here type-iso witnesses the isomorphism, Fix TypeD ≃ Type, between Type and its description as a regular type. In this style, we can recover enumerators for user-defined data types from the generic enumerators that are derived automatically.

We are now ready to define an indexed description for Term, starting with the Var constructor. Variables require a proof that some type σ can be found in the context Γ, so we need an enumerator that enumerates these proofs for any combination of context and type. Using elems, we can write the description corresponding to the Var constructor using a dependent pair. Since we do not need to store any additional information, the second component of the Σ is simply the unit type, one:

```
varD : Ctx → Type → IDesc (λ A → Enumerator A A) (Ctx × Type)
varD Γ σ = 'Σ (σ ∈ Γ) {elems Γ σ} λ _ → one
```

This example shows how to combine a hand-written enumerator (elems) in one that follows from the structure of our types. The choice of the additional information P stored in the IDesc description here is an enumerator of type Enumerator A A, for each type A that appears as the first component of a 'Σ constructor.

Next, we consider the constructor Abs. It produces a term of type Term Γ (σ ⟹ τ); its argument, the function body, has type Term (σ :: Γ) τ. Given a context Γ, domain σ and codomain τ, we simply recurse with the right choice of indices:

```
absD : Ctx → Type → IDesc (λ A → Enumerator A A) (Ctx × Type)
absD Γ (σ ⟹ τ) = var (σ :: Γ , τ)
absD Γ ι = zero
```

In the case when the desired type index is the base type, however, we cannot use the Abs constructor. Hence we return (the description of) the empty type $\bot$.

Function application presents an additional challenge. The constructor App takes two recursive argument, one of the form Term Γ (σ ⟹ τ) and one of the form Term Γ σ, producing a term with type τ. The choice of argument type σ, however, is not determined by the index type of the resulting application (τ). Since we define descriptions by induction over the index, we need to choose the argument type σ first, using a dependent pair. Once we have chosen σ, the other components of the application follow from two recursive calls to our enumerator:

```
appD : Ctx → Type → IDesc (λ A → Enumerator A A) (Ctx × Type)
appD Γ τ = 'Σ Type {types} (λ σ → var (Γ , (σ ⟹ τ)) ⊗ var (Γ , τ))
```

Just as we included the elems enumerator in the constructor for variables, here we use the types enumerator to enumerate the elements of the first component of our Σ-type. From these pieces, we create a description for well typed λ-terms as follows:

```
TermF : Func (λ A → Enumerator A A) (Ctx × Type)
TermF (Γ , τ) = varD Γ τ ⊕ appD Γ τ ⊕ absD Γ τ
```

We define TermF as a coproduct of three descriptions; the description associated with each individual constructor that we defined previously is passed the context Γ and index type τ.

By structuring the description this way, we use the fact that TermF *computes* an appropriate description from every index. For this reason it is not necessary to introduce explicitly equalities to describe the decomposition of the index type when invoking absD. In general we can derive such an "equality-free" descriptions for any indexed type where only constructors or variables occur in the index. For dependent types where the indices may contain arbitrary (non-injective) functions, however, this is no longer the case.

This generic enumerator, however, is not particularly efficient. The key inefficiency arises from the types enumerator that produces values of type Type. Fortunately, however, we can experiment freely with alternative enumerators, for instance by starting with listing types that appear in the context—which are more readily inhabited using the Var constructor. There is no magic here: generating well-typed lambda terms remains a hard problem. Crucially, however, the generic definitions make clear which parts of the generation can be done mechanically and which parts require further creativity. In particular, it is the enumerators stored in the 'Σ that can be used to steer the enumeration procedure.

6 DISCUSSION

6.1 Related work

There is a large body of related work on property-based testing, data type generic programming, and data type enumeration.

The original work on QuickCheck [Claessen and Hughes 2000] has generated a great deal of research in the area of property-based testing. The test data generation that we propose here, however, is not random, but more inspired by similar libraries based on exhaustive enumeration of values such as SmallCheck [Runciman et al. 2008]. Both SmallCheck and QuickCheck have been ported to numerous different languages since their original publication.

More recently, ideas from such property-based testing tools have started to emerge in the context of proof assistants. Previous work by Dybjer et al. [2005] and Haiyan [2007] explores the uniform random generation of indexed families in Agda. Work by Bulwahn [2012a,b] shows how to enumerate the inhabitants of a syntactic subset of Isabelle. A more recent notable example, is the work on QuickChick [Coq Development Team 2020; Dénès et al. 2014; Paraskevopoulou et al.

2015] that ports QuickCheck to the Coq proof assistant. The uniform generation of indexed families is not at all easy. As a result, the random generation algorithm used in QuickChick is more complex than the straightforward enumerators presented here. Each of these papers identifies a notion of completeness (sometimes referred to as surjectivity) and fairness (or uniformity in the case of random distributions). In a way, one of the key insights this paper provides is the *simplicity* of the definitions, showing that enumerating indexed families is only slightly more difficult than regular types. In a sense, the approach we take here is similar in spirit to LeanCheck [Braquehais 2017], that strives to define enumerators using a minimal set of combinators.

Similarly, there is a large body of work on randomly generating constrained data [Claessen et al. 2015] and inductive families [Lampropoulos et al. 2018]. Yakushev and Jeuring [2009] consider a similar problem in the context of Haskell, showing how GADTs can be enumerated by representing them using *spine views* [Hinze et al. 2006], extended with a form of existential quantification. They demonstrate that their approach is powerful enough to enumerate well-typed lambda terms. Their approach is, however, restricted to those invariants that can be expressed using a GADT, rather than the indexed families covered in this paper.

6.2 Further work

Performance. When writing these enumerators, we have not focused on performance. Repeatedly appending lists can quickly become a performance bottleneck. This might, in part, be possible to address using the difference list representation during enumeration [Hughes 1986]—but there are plenty of other opportunities for optimisations, such as fusing the repeated map operations over intermediate lists. Finally, Duregård et al. [2012] have shown how caching the intermediate sizes of the enumerated sub-terms can drastically improve performance when arbitrarily sampling from the enumeration. It would be interesting to attempt to extend their techniques to the (indexed) data types studied here, where we may be able to show how another iteration of our generic enumerator grows the (indexed) list of values generated in a predictable fashion.

Fairness. The 'column major' traversal of a list of enumerated values is certainly less biased than their concatenation. Although we use the merge operation to define hand-written enumerators in applicative style, the generic enumerators only rely on the enumerators for pairs and sigma types. Yet for these specific combinators, one might imagine a 'diagonal' traversal of the enumerated values gives a more even spread. New et al. [2017] give a more thorough treatment of fairness, especially aimed at the fair enumeration of (potentially) *infinite* lists. In our setting, however, we restrict ourselves to finite approximations of infinite lists, which makes things a bit simpler. We use dependent types extensively in this presentation: although our fairness definition relies on the comparison of natural numbers under the hood, we need to prove the completeness lemmas to even formulate the desired fairness properties. Furthermore, we can avoid some spurious cases by only ever comparing valid positions in a list, $x \in xs$, as opposed to any pair of natural numbers.

Automation. As our case study shows, there is still quite some overhead involved in manually writing the descriptions corresponding to a user-defined data type. Using Agda's reflection and metaprogramming facilities [Van Der Walt and Swierstra 2012], it should be possible to automate the derivation descriptions for data types, and their isomorphism converting between the two representations. By also using Agda's instance search [Devriese and Piessens 2011], we can then automatically generate enumerators for user-defined data types.

Specification discovery and tactics. A surprising application of property-based testing is the automatic generation of specifications. *QuickSpec* [Claessen et al. 2010] is one such tool that, based on QuickCheck. Given a set of functions, QuickSpec automatically generates collection of candidate

equalities. This collection of equations is then iteratively refined by checking them against randomly generated inputs produced by QuickCheck, and removing those equations that are falsified. The HipSpec tool [Claessen et al. 2012] takes these ideas one step further, by automatically proving the generated equalities.

Given these enumerators of indexed families, however, we can do even better. Tools such as QuickSpec only ever find *equalities* between terms—but oftentimes, we are more interested in proving that some inductive relation is inhabited. For example, given an insert function and isSorted predicate, one might imagine generating the following statement:

```
∀ x xs → isSorted xs → isSorted (insert x xs)
```

Testing such suitable candidate theorems requires the ability to generate arbitrary indexed families, which QuickSpec cannot do. One potential application area of these results is the automatic generation and testing of such statements.

Another potential application of these enumerators is in *proof automation*. Given a proof goal encoded as an indexed description, we try to generate an inhabitant by calling our enumerator. One might imagine extending this idea further, allowing the user to provide certain hypotheses that may be used in the enumeration. In this way, we can write our own version of Coq's `constructor` tactic that can be easily configured to restrict the search depth, constructors used, or hypotheses available.

Conclusion

We have shown how both regular data types and indexed families can be enumerated generically. We have sketched the proof of completeness for both these generic enumerators, guaranteeing that they eventually produce every possible inhabitant of every type; the enumerators we write in this style use combinators that we have shown to be *fair*. Using these definitions, we have shown how to write an enumerator for the terms of the simply-typed λ-calculus. Many of the pieces presented here have already been studied elsewhere, but we feel that the uniform presentation of our generic enumerators and the simplicity of our definitions and proofs, provides value beyond the sum of its parts.

REFERENCES

Marcin Benke, Peter Dybjer, and Patrik Jansson. 2003. Universes for Generic Programs and Proofs in Dependent Type Theory. *Nord. J. Comput.* 10, 4 (2003), 265–289.

Edwin C. Brady. 2013. Idris: general purpose programming with dependent types. In *Proceedings of the 7th Workshop on Programming languages meets program verification, PLPV 2013, Rome, Italy, January 22, 2013*, Matthew Might, David Van Horn, Andreas Abel, and Tim Sheard (Eds.). ACM, 1–2. https://doi.org/10.1145/2428116.2428118

Rudy Matela Braquehais. 2017. *Tools for discovery, refinement and generalization of functional properties by enumerative testing.* Ph.D. Dissertation. University of York, UK. http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.731590

Lukas Bulwahn. 2012a. The new quickcheck for Isabelle. In *International Conference on Certified Programs and Proofs*. Springer, 92–108.

Lukas Bulwahn. 2012b. Smart testing of functional programs in Isabelle. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 153–167.

James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. 2010. The gentle art of levitation. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, Paul Hudak and Stephanie Weirich (Eds.). ACM, 3–14. https://doi.org/10.1145/1863543.1863547

Koen Claessen, Jonas Duregård, and Michal H. Palka. 2015. Generating constrained random data with uniform distribution. *J. Funct. Program.* 25 (2015). https://doi.org/10.1017/S0956796815000143

Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, Martin Odersky and Philip Wadler (Eds.). ACM, 268–279. https://doi.org/10.1145/351240.351266

Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. 2012. HipSpec: Automating Inductive Proofs of Program Properties.. In *ATx/WInG@ IJCAR*. 16–25.

Koen Claessen, Nicholas Smallbone, and John Hughes. 2010. QuickSpec: Guessing Formal Specifications Using Testing. In *Tests and Proofs - 4th International Conference, TAP@TOOLS 2010, Málaga, Spain, July 1-2, 2010. Proceedings (Lecture Notes in Computer Science)*, Gordon Fraser and Angelo Gargantini (Eds.), Vol. 6143. Springer, 6–21. https://doi.org/10.1007/978-3-642-13977-2_3

Coq Development Team. 2020. *The Coq Proof Assistant Reference Manual*. Available at https://coq.inria.fr/doc/.

Pierre-Évariste Dagand. 2013. *A cosmology of datatypes : reusability and dependent types*. Ph.D. Dissertation. University of Strathclyde, Glasgow, UK. http://oleg.lib.strath.ac.uk/R/?func=dbin-jump-full&object_id=22713

Pierre-Évariste Dagand and Conor McBride. 2012. Transporting functions across ornaments. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, Peter Thiemann and Robby Bruce Findler (Eds.). ACM, 103–114. https://doi.org/10.1145/2364527.2364544

Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings (Lecture Notes in Computer Science)*, Amy P. Felty and Aart Middeldorp (Eds.), Vol. 9195. Springer, 378–388. https://doi.org/10.1007/978-3-319-21401-6_26

Maxime Dénès, , Cătălin Hriţcu, Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C Pierce. 2014. QuickChick: Property-based testing for Coq. In *The Coq Workshop*.

Dominique Devriese and Frank Piessens. 2011. On the bright side of type classes: instance arguments in Agda. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy (Eds.). ACM, 143–155. https://doi.org/10.1145/2034773.2034796

Jonas Duregård, Patrik Jansson, and Meng Wang. 2012. Feat: functional enumeration of algebraic types. In *Proceedings of the 5th ACM SIGPLAN Symposium on Haskell, Haskell 2012, Copenhagen, Denmark, 13 September 2012*, Janis Voigtländer (Ed.). ACM, 61–72. https://doi.org/10.1145/2364506.2364515

Peter Dybjer, Qiao Haiyan, and Makoto Takeyama. 2005. Random Generators for Dependent Types. In *Theoretical Aspects of Computing - ICTAC 2004*, Zhiming Liu and Keijiro Araki (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 341–355.

Burke Fetscher, Koen Claessen, Michal H. Palka, John Hughes, and Robert Bruce Findler. 2015. Making Random Judgments: Automatically Generating Well-Typed Terms from the Definition of a Type-System. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science)*, Jan Vitek (Ed.), Vol. 9032. Springer, 383–405. https://doi.org/10.1007/978-3-662-46669-8_16

Qiao Haiyan. 2007. Testing and Proving Distributed Algorithms in Constructive Type Theory. In *Tests and Proofs - 1st International Conference, TAP 2007, Zurich, Switzerland, February 12-13, 2007. Revised Papers (Lecture Notes in Computer Science)*, Yuri Gurevich and Bertrand Meyer (Eds.), Vol. 4454. Springer, 79–94. https://doi.org/10.1007/978-3-540-73770-4_5

Ralf Hinze, Andres Löh, and Bruno C. d. S. Oliveira. 2006. "Scrap Your Boilerplate" Reloaded. In *Functional and Logic Programming, 8th International Symposium, FLOPS 2006, Fuji-Susono, Japan, April 24-26, 2006, Proceedings (Lecture Notes in Computer Science)*, Masami Hagiya and Philip Wadler (Eds.), Vol. 3945. Springer, 13–29. https://doi.org/10.1007/11737414_3

R. John Muir Hughes. 1986. A novel representation of lists and its application to the function "reverse". *Information processing letters* 22, 3 (1986), 141–144. https://doi.org/{10.1016/0020-0190(86)90059-1}

Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. 2018. Generating good generators for inductive relations. *Proc. ACM Program. Lang.* 2, POPL (2018), 45:1–45:30. https://doi.org/10.1145/3158133

Max S. New, Burke Fetscher, Robert Bruce Findler, and Jay McCarthy. 2017. Fair enumeration combinators. *Journal of Functional Programming* 27 (2017), e19. https://doi.org/10.1017/S0956796817000107

Ulf Norell. 2009. Dependently typed programming in Agda. In *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*, Andrew Kennedy and Amal Ahmed (Eds.). ACM, 1–2. https://doi.org/10.1145/1481861.1481862

Michal H. Palka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test, AST 2011, Waikiki, Honolulu, HI, USA, May 23-24, 2011*, Antonia Bertolino, Howard Foster, and J. Jenny Li (Eds.). ACM, 91–97. https://doi.org/10.1145/1982595.1982615

Zoe Paraskevopoulou, Cătălin Hriţcu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C Pierce. 2015. Foundational property-based testing. In *International Conference on Interactive Theorem Proving*. Springer, 325–343.

Colin Runciman, Matthew Naylor, and Fredrik Lindblad. 2008. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell, Haskell 2008, Victoria, BC, Canada,*

25 September 2008, Andy Gill (Ed.). ACM, 37–48. https://doi.org/10.1145/1411286.1411292

Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. 2016. Dependent types and multi-monadic effects in F. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 256–270. https://doi.org/10.1145/2837614.2837655

Paul Tarau. 2015. On Type-directed Generation of Lambda Terms. In *Proceedings of the Technical Communications of the 31st International Conference on Logic Programming (ICLP 2015), Cork, Ireland, August 31 - September 4, 2015 (CEUR Workshop Proceedings)*, Marina De Vos, Thomas Eiter, Yuliya Lierler, and Francesca Toni (Eds.), Vol. 1433. CEUR-WS.org. http://ceur-ws.org/Vol-1433/tc_12.pdf

Paul Van Der Walt and Wouter Swierstra. 2012. Engineering proof by reflection in Agda. In *Symposium on Implementation and Application of Functional Languages*. Springer, 157–173.

Alexey Rodriguez Yakushev and Johan Jeuring. 2009. Enumerating Well-Typed Terms Generically. In *Approaches and Applications of Inductive Programming, Third International Workshop, AAIP 2009, Edinburgh, UK, September 4, 2009. Revised Papers (Lecture Notes in Computer Science)*, Ute Schmid, Emanuel Kitzelmann, and Rinus Plasmeijer (Eds.), Vol. 5812. Springer, 93–116. https://doi.org/10.1007/978-3-642-11931-6_5