A completely unique account of enumeration

ANONYMOUS AUTHOR(S)

How can we enumerate the inhabitants of an algebraic datatype? This paper explores a *datatype generic* solution that works for all *regular types* and *indexed families*. The enumerators presented here are provably both *complete* and *unique*—they will eventually produce every value exactly once—and *fair*—they avoid bias when composing enumerators. Finally, these enumerators memoise previously enumerated values whenever possible, thereby avoiding repeatedly recomputing recursive results.

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 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. Rather than generate *random* input values as QuickCheck does, this paper explores how to *enumerate* the values inhabiting a given datatype. While this is not a new problem, or even an entirely new idea—this paper makes several novel contributions:

- We give a datatype generic account of the enumeration of both *regular* datatypes (Section 3) and *indexed* families (Section 4). Most existing approaches to enumerating datatypes rely on some form of user-defined size bound and the datatype generic case is rarely considered. We show how to make the recursive structure of enumerators explicit, thereby cleanly separating the *definition* of enumerators from their *execution*. This allows for different interpretations of enumerators, including a coinductive stream of values, or indeed, the finite set of elements up to some bound. Furthermore, while there is large body of work on enumerating regular types [Braquehais 2017; Duregård et al. 2012; Runciman et al. 2008], the case for indexed families has remained unexplored until now.
- We identify and establish properties of these enumerators and provide a formalised proof that these properties hold for all of our generic definitions. While some of these properties are mentioned or established informally in existing work, their mechanization has lagged behind. We establish several different properties of all of our enumerators: *completeness* and *uniqueness* for both regular datatypes and indexed families (Sections 3.3 and 4.3), and the *fairness* of the enumerators we used in our generic definitions (Section 2).
- Finally, we show how naive enumeration has an efficiency problem: the repeated recomputation of recursive results. We can address this by memoising previous computations readily enough for regular datatypes (Section 3), extending these results to indexed families is less obvious. If we restrict ourselves to regular indices, however, we can show how to construct a generic trie to memoise the enumeration of indexed families (Section 5). Finally, we illustrate how these generic definitions can be used to enumerate well typed expressions (Section 6).

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

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 often omit universally quantified implicit arguments, making liberal use of Agda's **variable** construct. 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
```

Yet many recursive datatypes, 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 considering the enumeration of recursive datatypes (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 → ℕ
| 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 an 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, capturing the intuition that the ++ operation should respect the ordering of elements in its argument lists:

```
(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. As the ++ operation does not respect the order of elements of its argument lists, we consider it to be *unfair*.

Fair choice. So what is a fair notion of choice operator? Unsurprisingly, the solution is to draw elements, alternating between the two argument 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 does respect the order of elements in the resulting list. 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 ys xs p) ≺ (interleave∈-left  ys xs q)
```

We can additionally prove the other two possible combinations—left-left and right-right—hold as expected. Note that the append operator also satisfies both these properties.

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

```
_⟨|⟩_ : (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 A fair pair

Besides the choice operator, ⟨|⟩, we would like to compute the cartesian product of two enumerators:

```
pair : Enumerator A B → Enumerator A C → Enumerator A (B × C)
```

The usual definition of the cartesian product of two lists relies on the concatMap function:

```
cp : List A → List B → List (A × B)
cp xs ys = concatMap (λ x → map (λ y → x , y) ys) xs
```

Yet concatenation-based cartesian products are not a suitable choice if we care about the fairness of our enumerators. Just as the append operator is biased towards the first list, this definition of the cartesian product using concatenation will favour pairs whose first component appears earlier on in the first argument list xs. Instead, we introduce the following custom prod function, inspired by the product of power series [McIlroy 1999]:

```
prod : List A → List B → List (A × B)
prod []       ys       = []
prod (x ∷ xs) []       = []
prod (x ∷ xs) (y ∷ ys) = (x , y) ∷ interleave (map (λ y → (x , y)) ys) (prod xs (y ∷ ys))
```

The prod function also computes the cartesian product of its two argument lists. The interesting case, when both lists are non-empty, uses the interleave function to alternate between the elements with x as their first component and the elements whose first component is drawn from the tail xs. By interleaving these two intermediate lists, we can show that the prod function is fair.

To make this more precise, we need to start by showing that the prod function will produce all possible elements of the cartesian product of its two inputs:

```
prod∈ : (xs : List A) → (ys : List B) → x ∈ xs → y ∈ ys → (x , y) ∈ prod xs ys
```

We can now formulate and prove the desired fairness property using the above lemma:

```
prodFair : (p₁ : x₁ ∈ xs) (p₂ : y₁ ∈ ys) (q₁ : x₂ ∈ xs) (q₂ : y₂ ∈ ys) →
  p₁ ≺ q₁ → p₂ ≺ q₂ → prod∈ xs ys p₁ p₂ ≺ prod∈ xs ys q₁ q₂
```

The proof itself follows by induction over the positions passed as arguments. Although it requires several auxiliary lemmas about the fairness of map and interleave, the proof itself is not particularly complicated. The only non-trivial insight required is that prod∈ is a monotonically increasing

function: if an element x occurs in the i-th position in one of the input lists, pairs with the value x as their first component will not occur before position i in the list of pairs produced. Using this prod function, we can define the fair pairing operation on enumerators as follows:

```
pair : Enumerator A B → Enumerator A C → Enumerator A (B × C)
pair e₁ e₂ = λ cs → prod (e₁ cs) (e₂ cs)
```

Finally, we can use the prod function to also define the familiar applicative combinator:

```
_⊛_ : Enumerator C (A → B) → Enumerator C A → Enumerator C B
e₁ ⊛ e₂ = λ cs → map apply (prod (e₁ cs) (e₂ cs))
  where
  apply : (A → B) × A → B
  apply (f , x) = f x
```

Although we will not use this combinator in our generic constructions, it can be useful for some of the example enumerators we will define by hand in the remainder of this section.

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 datatype 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 : ℕ → 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 Tree → List Tree. 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 by iterating its argument enumerator a fixed number of times.

```
enumerate : Enumerator A A → ℕ → List A
enumerate e n = iterate n e []
```

```
iterate : ∀ {A} → ℕ → (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. Phrasing our enumerators in this style ensures that we can efficiently enumerate recursive datatypes in a datatype generic fashion, reusing previous computations whenever possible. Note that there are other ways to apply our enumerator functions, such as producing an infinite stream of lists of increasingly 'large' elements:

```
stream : Enumerator A A → Stream (List A) ∞
stream e = Codata.Stream.iterate e []
```

For the purpose of this paper, however, we will only concern ourselves with the enumerate function that produces a finite approximation of the elements of a given datatype.

2.4 Enumerator completeness & uniqueness

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

```
boolsWrong₁ : Enumerator Bool Bool
boolsWrong₁ = ∅
```

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 : ℕ) → 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_1 is a complete enumerator; *a fortiori*, we can prove that boolsWrong_1 is not complete.

Besides completeness, we will define enumerators that list elements without duplicates. The following enumerator for booleans is complete:

```
boolsWrong₂ : Enumerator Bool Bool
boolsWrong₂ = λ bs → true ∷ true ∷ false ∷ []
```

Yet this enumerator contains two occurrences of true. To rule out enumerating the same element more than once, our enumerators will only ever produce lists that are *unique*:

```
Unique : List A → Set
Unique xs = (x : A) → (p₁ p₂ : x ∈ xs) → p₁ ≡ p₂
```

All our enumerators will preserve *uniqueness*: given a list of unique 'smaller' terms as its input, our enumerators will produce a new unique list of elements.

In what follows, we will rarely write completeness and uniqueness proofs by hand, but rather define *generic* enumerators that are built from the combinators presented here that guarantee these properties hold. It is important to stress that the uniqueness and completeness are extensional properties that we can verify within our proof assistant. Fairness, on the other hand, is an intensional property that we can prove of all the combinators we use—but cannot be observed from an enumerator's input-output behaviour alone.

3 GENERIC ENUMERATION OF REGULAR TYPES

In the previous section, we defined example enumerators for booleans and trees. In this section, we show to generalise these results and write a generic enumerator for a collection of simple algebraic datatypes; 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 algebraic datatypes.

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:

```
data Desc (P : Set → Set) : Set₁ 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
```

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. Note that the Desc type, as presented here, is *large* as the constant constructor k quantifies over all types. To avoid size problems, this construction can by stratified by only drawing constants drawn from some smaller universe U : Set.

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 datatypes such trees and lists. The Fix datatype ties the recursive knot:

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

This explicit description of datatypes enables 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 (const ⊤)
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 ℕ):

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

Here tt is the single constructor of the unit type ⊤; inj_1 and inj_2 are the two constructors for a disjoint sum of types.

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 enumerator, genumerator, with the following type:

```
genumerator : (D : Desc List) → Enumerator (Fix D) (Fix D)
```

Given any description D we will enumerate the recursive datatypes 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 genumerator 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) from the description that describes the type of recursive positions (D'):

```
enumerator : ∀ (D : Desc List) {D' : Desc List} → Enumerator (Fix D') (⟦ D ⟧ (Fix D'))
```

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 enumerator function is now immediate, using all the auxiliary functions defined in the previous section:

```
enumerator : ∀ (D : Desc List) {D' : Desc List} → Enumerator (Fix D') (⟦ D ⟧ (Fix D'))
enumerator zero        = ∅
enumerator one         = pure tt
enumerator (k A {as})  = const as
enumerator var         = rec
enumerator (D₁ ⊕ D₂)   = (inj₁ <$> enumerator D₁) <|> (inj₂ <$> enumerator D₂)
enumerator (D₁ ⊗ D₂)   = pair (enumerator D₁) (enumerator 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, the case for zero returns the empty enumerator, ∅. 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 pair function that computes this Cartesian product is defined in Section 2.

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

```
genumerator : (D : Desc List) → Enumerator (Fix D) (Fix D)
genumerator D = λ ts → map In (enumerator D ts)
```

This function simply calls the enumerator 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) → ℕ → List (Fix (ListD xs))
lists xs = enumerate (genumerator (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 datatype declaration.

In the enumerator for lists, we assumed that we had a list of elements enumerating the elements of the list. This is fine if we aim to draw elements from some (finite) fixed set, such as ASCII characters. Alternatively, however, we may want to pass a size-bounded finite list, $\mathbb{N} \to \mathsf{List}\ \mathsf{A}$, enabling the enumeration of lists with elements drawn from an infinite type. To do so, we would only need to change the type signature of the enumerator function, updating the implicit information stored for constant types. In the genumerator and enumerate functions, this size bound can then be passed on accordingly. For sake of simplicity, however, we have chosen the more simple approach—restricting ourselves to lists of constant types—as this keeps the execution of enumerators (using a size bound) and their definition clearly separated.

3.3 Completeness & uniqueness

Completeness. We now briefly sketch the *completeness* proof, establishing that our generic enumerators will eventually produce every possible value. Proving our generic enumerators are complete amounts to showing that for all $\mathsf{x} : \mathsf{Fix\ D}$ there is a number $\mathsf{n} : \mathbb{N}$ such that x occurs in the list enumerate (genumerator 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 fix-point to produce x. We refer to this number as the depth of a given tree; it can be readily computed as follows:

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

  gdepth : (D : Desc P) → Fix D → ℕ
  gdepth D (In x) = succ (depth D x)
```

To prove that some $\mathsf{x} : \mathsf{Fix\ D}$ 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:

```
completeness-lemma : (D : Desc List) (x : ⟦ D ⟧ (Fix D')) (xs : List (Fix D')) →
  ((y : Fix D') → gdepth D' y ≤ depth D x → y ∈ xs) →
  x ∈ enumerator 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 completeness-lemma, while passing itself recursively to prove the completeness of any recursive calls:

```
completeness : (D : Desc List) → Complete (genumerator D)
```

We have chosen to ignore constant types in this proof sketch. To complete the proof, we have extended the completeness-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 completeness-lemma are fairly straightforward—once these definitions are fixed—spanning about twenty lines of proof and using a handful of auxiliary lemmas.

Uniqueness. Now that we have shown that our generic enumerators are complete, we will prove that the lists they produce are free of duplicates. The proof itself follows a similar pattern as we saw for completeness: a lemma by induction on D and a main result that relies upon this lemma. The key uniqueness-lemma required can be formulated as follows:

```
uniqueness-lemma : (D D' : Desc List) (xs : List (Fix D')) →
  Unique xs → Unique (enumerator D xs)
```

Proving this lemma is a bit harder than the completeness result we sketched previously. Intuitively, for completeness we only need to *construct* a proof that x ∈ enumerator D xs, whereas for uniqueness, we need to *destruct* such proofs. As a result, the proof requires a series of lemmas about the interleave and prod functions. There is a pleasant duality between these lemmas and those presented in Section 2: where the prod∈ and interleave∈ lemmas can be read as the familiar introduction rules of propositional logic, the lemmas required to prove uniqueness are their dual elimination rules:

```
interleave-elim : (xs ys : List A) → x ∈ interleave xs ys → (x ∈ xs) ⊎ (x ∈ ys)
prod-elim : ∀ (xs : List A) (ys : List B) → (x , y) ∈ prod xs ys → (x ∈ xs) × (y ∈ ys)
```

The proof of the uniqueness-lemma itself proceeds by induction on the description, using these lemmas and the assumption that the list xs is free of duplicates, to prove that the call to the enumerator preserves uniqueness, Unique (enumerator D xs). Once again, to complete the proof we require an additional assumption that all the enumerators for constant types contain unique elements. Using this uniqueness lemma, we can now prove our main result:

```
uniqueness : (D : Desc List) (n : ℕ) → Unique (enumerate (genumerator D) n)
```

This proof proceeds by straightforward induction on the natural number n. In the base case, the list is produced empty and hence uniqueness proof is trivial. For the inductive case, we apply our uniqueness-lemma, using our induction hypothesis and the fact that the `In` constructor is injective, to establish that the resulting list is free of duplicates.

4 GENERIC ENUMERATORS FOR INDEXED FAMILIES

While regular types are fairly straightforward to enumerate, the enumeration of *indexed* families 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 families.

4.1 Universe Definition

The universe of *indexed descriptions* describes a wide collection of indexed families. 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 for regular types directly as values 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 → Set₁
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) : Set₁ 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 datatype 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 pairs* (‘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.

Note that this universe is not minimal—we could always encode coproducts (⊕) using dependent pairs (‘Σ), but it can be useful to discriminate between the ‘choice of constructor’ and dependent types—the prior being much simpler to handle than the latter.

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 datatype 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) : ℕ → 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 (const ⊤) ℕ
VecF _ zero     = one
VecF A (succ n) = 'Σ A (const (var n))
```

We choose the indexed description VecF carefully such that Fix (VecF A) n is isomorphic to Vec A n. Rather than modeling 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 key challenge to enumerating indexed families will be the treatment of dependent pairs, 'Σ S {e} f. In this case, we are given an enumerator e that produces elements of type S, together with a *function* f that computes an indexed description for each value of S. Crucially, we cannot make a recursive call directly to these descriptions—but rather need a *monadic bind* to pass all possible inputs to f, before recursing. In what should be a familiar pattern, we do avoid using the usual bind of the underlying list monad implemented using concatMap. To avoid bias, we define a version that interleaves all its intermediate results fairly:

```
bind : List A → (A → List B) → List B
bind []       f = []
bind (x : xs) f = interleave (f x) (bind xs f)
_≫=_ : IEnumerator F A → (A → IEnumerator F B) → IEnumerator F B
(e ≫= f) = λ cs → bind (e cs) (λ x → f x cs)
```

To illustrate this combinator in action, we can write an enumerator for *dependent pairs* by hand as follows:

```
sigmas : IEnumerator F A → ((x : A) → IEnumerator F (B x)) → IEnumerator F (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.

We can now define the generic enumerator for indexed families. It 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, ienumerator, 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.

```
ienumerator : (desc : IDesc List I) → IEnumerator (Fix φ) (⟦ desc ⟧ (Fix φ))
ienumerator zero          = ∅
ienumerator one           = pure [ tt ]
ienumerator (D₁ ⊗ D₂)     = pair (ienumerator D₁) (ienumerator D₂)
ienumerator (D₁ ⊕ D₂)     = (inj₁ ⟨$⟩ enumerator D₁) ⟨|⟩ (inj₂ ⟨$⟩ enumerator D₂)
ienumerator (var i)       = λ rec → rec i
ienumerator (‘Σ S {e} f)  = sigmas e (ienumerator ∘ f)
```

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 ienumerator function using f s as the new description to enumerate.

The top-level generic igenumerator invokes ienumerator, instantiating the indexed description with φ i:

```
igenumerator : ∀ φ (i : I) → IEnumerator (Fix φ) (Fix φ i)
igenumerator φ i = In ⟨$⟩ ienumerator (φ i)
```

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) → ℕ → List (A i)
ienumerate f i zero     = []
ienumerate f i (succ n) = f i (λ i → ienumerate f i n)
```

4.3 Completeness & uniqueness

Completeness. One pleasant property of our development is that many definitions and proofs on the universe of regular types can be 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:

```
idepth  : (D : IDesc P I) → ⟦ D ⟧ (Fix φ) → ℕ
gidepth : Fix φ i → ℕ
```

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

```
icomplete : (D : IDesc List I) (x : ⟦ D ⟧ (Fix φ)) (xsi : (i : I) → List (Fix φ i)) →
  (∀ i → (y : Fix φ i) → gidepth y < idepth D x → y ∈ xsi i) →
  x ∈ ienumerator D xsi
```

The general structure of this proof is the same as we saw for the regular universe. 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 value of type Fix φ i will appear in the list associated with the index i. Each case 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 pair 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 pairs—we can mimic the completeness proof for pair, though we have to prove new auxiliary lemmas that establish that the monadic bind operation is well behaved. To prove completeness, we do require completeness of the enumerator for the set S that is used by the dependent pair—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:

```
igcomplete : ∀ (φ : I → IDesc List I) (i : I) (x : Fix φ i) (n : ℕ) →
  idepth x ≤ n → x ∈ ienumerate (igenumerator φ) n i
```

Its proof is analogous: pattern matching on the In constructor and calling the icomplete lemma sketched above. Once again, we have 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 (implicit) argument that assumes that the lists stored for these values are also complete.

Uniqueness. Besides completeness, we would also like to prove that the generic enumerators of indexed families produce *unique* values. Given that the apparent overlap between the two universes of types we have considered, this may seem straightforward. Indeed, we can generalise the key uniqueness-lemma for regular types to its counterpart for indexed families:

```
uniqueness-lemma : (D : IDesc List I) → (φ : I → IDesc List I) → (xs : (i : I) → List (Fix φ i))
  → (ih : ∀ i → Unique (xs i)) → Unique (ienumerator D xs)
```

Essentially, this lemma states that if we have a unique list of elements for each index, extending this list using the ienumerator preserves uniqueness. The proof proceeds by induction on the description D; most of the cases follow using the same lemmas about pairing and interleaving as we saw for regular types. The dependent pairs, however, prove more of a challenge.

Where most cases follow the proof for regular types, the dependent pairs prove more of a challenge. The auxiliary function, sigmas, uses the bind for lists. To reason about this, we need to prove the following lemma:

```
bind-elim : y ∈ bind xs f → ∃[ x ] ((x ∈ xs) ↑↑ × ↑↑ (y ∈ f x))
```

This states that whenever y occurs in the list bind xs f, there is some x, such that we can find a pair of proofs x ∈ xs and y ∈ f xs. We can use this to establish a similar 'elimination principle' for dependent pairs:

```
sigmas-elim: (xs : IEnumerator F A) (f : (x : A) → IEnumerator F (B x))
  → (x , y) ∈ sigmas xs f rec → x ∈ xs rec × y ∈ f x rec
```

Finally, we need to show both these functions are injective. This is the key component of each uniqueness proof. Given two elements of our enumeration, we can recursively deconstruct them

into their constituent parts; using our induction hypothesis, together with the injectivity of our constructors, we can then establish uniqueness.

The full proof requires an additional assumption, similar to the one we mentioned for regular types, stating that all the constant types (that occur in the ‘Σ constructor) are enumerated uniquely. Interestingly, however, we do *not* require any injectivity of the *functions* associated with these constructors—these functions are only used to express the type dependency; to prove uniqueness, we simply have to prove that pair of values occurs no more than once.

5 MEMOISATION

There is one important difference between the enumerators for regular datatypes and indexed families: where the enumerators for regular types are passed a *list* of previously constructed values, those for indexed families are passed a *function* of type (i : I) → List (A i), returning the list of previously constructed values at each index. In Section 2.3, we illustrated the importance of not recomputing previously enumerated values over and over again: doing so quickly leads to an exponential slowdown in enumeration. Now consider the following example of an indexed family:

```
data Tree : ℕ → Set where
  Leaf : Tree 0
  Node : Tree n → Tree n → Tree (succ n)
```

When enumerating such trees, we run into the same problem as we encountered in the naive enumeration of simply typed binary trees: in the case for nodes, we will make two calls to the indexed enumerator passed. We have lost the sharing of previous results by shifting to the enumeration of indexed families.

Fortunately, there is a well known representation of *functions* as *data structures* [Hinze 2000]: if we restrict ourselves to an index set that is regular, we can *memoise* the enumeration of indexed families, thereby avoiding superfluous recomputation. This section sets out to achieve just that.

5.1 Generic tries

The key idea behind memoisation is to define an alternative—yet equivalent—representation of functions. We will sketch the interface of the required functions first, before presenting their datatype generic implementation. This section leans heavily on previous work [Hinze 2000; Hinze et al. 2004], but extends these results to a total and dependently typed setting.

First and foremost, we need a type representing generic lookup structures or *tries*:

```
_↦_ : Set → Set → Set
```

As we are interested in memoising a *dependent* function of the form (i : I) → List (A i), we will need to generalise this slightly:

```
_↦_ : (A : Set) → (B : A → Set) → Set
```

In addition to the type of generic tries, we will define a pair of functions to create and consult tries:

```
trie   : ((x : A) → B x) → (A ↦ B)
untrie : (A ↦ B) → ((x : A) → B x)
```

As its name suggests, the trie function constructs a trie; the untrie function looks up the value associated with a key of type A in a given trie. Finally, we require that untrie is the left inverse of trie, that is untrie ∘ trie ≡ id.

We will now implement this interface generically for any function whose domain consists of a regular type. That is, we consider the special case of memoising functions of the form:

```
(x : Fix D) → B x
```

This restriction enables us to define a memoisation structure, Memo D, by induction on the description D:

```
Memo : (D : Desc P) → (R : (Fix D' → Set) → Set) → (B : ⟦ D ⟧ (Fix D') → Set) → Set
Memo zero       R B = ⊤
Memo one        R B = B tt
Memo var        R B = R B
Memo (k S)      R B = (s : S) → B s
Memo (D₁ ⊗ D₂) R B = Memo D₁ R (λ x → Memo D₂ R (λ y → B (x , y)))
Memo (D₁ ⊕ D₂) R B = Memo D₁ R (λ x → B (inj₁ x)) × Memo D₂ R (λ y → B (inj₂ y))
```

The type of Memo may seem daunting at first blush. The Memo type takes three arguments: the description D, the type of recursive tries R, and the codomain of the function being memoised B. When reading this definition, you may want to keep in mind that the type variable for recursive tries, R, will later be instantiated with the _↦_ type—corresponding to a (corecursive) Memo structure. Most clauses of this definition follow from the familiar laws of exponentiation:

$$C^{A+B} = C^A \times C^B \qquad C^{A\times B} = C^{A^B} \qquad C^0 = 1 \qquad C^1 = C$$

The only remaining cases are those for recursive parameters, constants k, and recursive parameters one. For the sake of simplicity, we do not memoise functions over constant types for the moment—although we could use the additional P parameter in descriptions to store more efficient memoisation structures, such as Patricia trees for fixed width integers for instance. For recursive types, var, we use the the additional type parameter R. We tie the recursive knot by defining the coinductive _↦_ type as follows:

```
record _↦_ (D : Desc P) (B : Fix D → Set) : Set where
  coinductive
  constructor mkMemo
  field
    getMemo : Memo D (λ X → D ↦ X) (λ x → B (In x))
```

Each such trie D ↦ B is given by a memo structure—determined by D—where the recursive components themselves are new memo structures. At this point, it may be instructive to consider an example. We can define the description of natural numbers natD as follows:

```
natD = one ⊕ var
```

Every natural number is either zero (given by inj_1 tt) or a successor (given by inj_2 n for some natural number n). Given these definitions, we can consider the type of tries memoising computations of the form (x : Fix natD) → B x, that is: what is the type natD ↦ B? Unfolding the Memo and _↦_ definitions above, this gives rise to the coinductive datatype arising as the greatest fix-point of the following equation:

```
MemoNat B = B zero × MemoNat (B ∘ succ)
```

The MemoNat structure corresponds to a stream of values of type B 0, B 1, B 2, and so forth. In this way, the type natD ↦ B describes the tabulation of a (dependent) function over natural numbers.

To create such tabulations, we define a pair of functions, gtrie and trie. Given a function (x : Fix D) → B x, the gtrie function corecursively constructs the corresponding generic trie, introducing the mkMemo constructor and calling the trie function:

```
gtrie : (D : Desc P) → ((x : Fix D) → B x) → D ↦ B
gtrie D f = mkMemo (trie D (λ x → f (In x)))
```

The trie function proceeds by induction on its argument description, passing increasingly complex arguments to the function argument f in each recursive call.

```
trie : (D : Desc P) → ((x : ⟦ D ⟧ (Fix D')) → B x) → Memo D (λ X → D' ↦ X) B
trie zero       f = tt
trie one        f = f tt
trie var        f = gtrie D' f
trie (k S)      f = f
trie (D₁ ⊗ D₂) f = trie D₁ (λ x → trie D₂ (λ y → f (x , y)))
trie (D₁ ⊕ D₂) f = (trie D₁ (λ x → f (inj₁ x)) , trie D₂ (λ x → f (inj₂ x)))
```

How can we use such generic tries? Given any argument of type Fix D, we can traverse the trie structure to find the corresponding result. The pair of mutually recursive functions guntrie and untrie do precisely that:

```
guntrie : (D : Desc P) → (D ↦ B) → (x : Fix D) → B x
guntrie D m (In t) = untrie D (getMemo m) t

untrie : (D : Desc P) → Memo D (λ X → D' ↦ X) B → (x : ⟦ D ⟧ (Fix D')) → B x
untrie one       m tt          = m
untrie var       m x           = guntrie D' m x
untrie (k S)     m x           = m x
untrie (D₁ ⊗ D₂) m (x , y)     = untrie D₂ (untrie D₁ m x) y
untrie (D₁ ⊕ D₂) (m₁ , m₂) (inj₁ x) = untrie D₁ m₁ x
untrie (D₁ ⊕ D₂) (m₁ , m₂) (inj₂ y) = untrie D₂ m₂ y
```

Once again, the heavy lifting is done by the untrie function that proceeds by induction on D. In the case for the unit type one, we return the memoised value m. In the case for constants, we apply the function m to the argument x. The cases for products and coproducts pattern match on the argument x and recurse accordingly. Finally, the case for recursive subtrees var calls the guntrie function to peel off the In constructor and recurse.

We can now compose gtrie and guntrie to create a trie and lookup the argument x in it:

```
gmemo : (D : Desc P) (B : Fix D → Set) → ((x : Fix D) → B x) → (x : Fix D) → B x
gmemo D B f x = guntrie D (gtrie D f) x
```

Of course, this gmemo should simply be (a more expensive version of) the identity function. Indeed, we can prove the following lemma by immediate induction on the description D:

```
correct : ∀ (D : Desc P) (f : (x : Fix D) → B x) (x : Fix D) → gmemo D B f x ≡ f x
```

Before using these generic tries to memoise the indexed enumerators we saw previously, however, it is worth pointing out that these definitions are not immediately accepted by Agda. The codatatype for tries, _↦_, is not identified to be strictly positive, even though we can readily check by hand that each of the right-hand sides of the Memo definition is. Furthermore, the trie function is incorrectly marked as non-terminating. Whereas this definition is obviously guarded—the only corecursive call in the branch for recursive types, var, is immediately guarded by the MkMemo constructor—the definition is rejected by the guardedness checker and Agda's 'musical notation' for coinductive definitions. In practice, however, as long as we can prove that the gmemo function behaves as the identity, however, there need not be a problem with disabling the termination and positivity checkers for these definitions.

5.2 Memoising enumerators

With these generic tries defined, we can now use them to avoid recompution during enumeration. The key insight is that, when the index set is itself regular, we can replace the argument *function* with a *generic trie* storing the previously computed results. To achieve this, we define the following type for memoising indexed enumerators, or *memorators* for short:

```
Memorator : (D : Desc P) → (Fix D → Set) → Set → Set
Memorator D A B = (D ↦ (List ∘ A)) → List B
```

It is worth comparing this type with that of indexed enumerators defined on page 13. The gtrie and guntrie functions defined previously can be used to convert between these two representations, when the index set I is of the form Fix D for some description D.

We can now define a memoising version of our indexed enumerators. To do so, we follow the familiar iterative pattern we have seen twice already:

```
gmemorate : ((i : Fix D) → Memorator D B (B i)) → (i : Fix D) → ℕ → List (B i)
gmemorate f i zero     = []
gmemorate f i (succ n) = f i (gtrie D (λ i → gmemorate f i n))
```

This definition closely mimics the enumerate and ienumerate functions defined previously. The key difference, however, is that it maintains a trie storing the previously produced values at every index. Similarly, we need to adapt the indexed enumerators from the previous section slightly to produce memorators, rather than the indexed enumerators we saw previously. The required changes are mostly superficial modifications in the type signatures, hence we refrain from presenting the code here.

Performance. As Agda uses a call-by-need evaluation strategy, the memoised computations are shared during enumeration. Although the current implementation of the compiler[1] and abstract machine[2] have some limitations when memoising coinductive computations, the presentation here avoids these. To illustrate the performance gain memoisation offers, consider the following indexed description of the perfect trees we saw previously:

```
treeD : Fix natD → IDesc List (Fix natD)
treeD (In (inj₁ tt)) = one
treeD (In (inj₂ n))  = var n ⊗ var n
```

We can now define a small benchmarking suite, calling the memoising and non-memoising enumerator for such perfect trees:

```
memo-tree     = λ size → gmemorate (gmemorator treeD)   (fromNat size) (succ size)
non-memo-tree = λ size → ienumerate (igenumerator treeD) (fromNat size) (succ size)
```

Given an argument size, these functions call one of the enumerate functions we have seen. The index we are interested in is given by fromNat size, converting a given natural number to its corresponding generic representation in Fix natD. Finally, we bound the computation by succ size—providing just enough room to unfold the recursive structure of our enumerators.

Using the agda-bench[3] benchmarking tool we can enumerate such trees with and without memoisation, measuring the time necessary to enumerate perfect trees of various sizes. The results are shown in the Table 1. While the absolute numbers are not so important, these figures clearly highlight the performance gain that memoisation offers: enumerating the perfect tree of depth 16 using

[1] https://github.com/agda/agda/issues/2918
[2] https://github.com/agda/agda/issues/5722
[3] https://github.com/UlfNorell/agda-bench/

Tree size	Without memoisation	With memoisation
1	82.28 µs	104.3 µs
2	154.6 µs	181.2 µs
4	616.7 µs	386.8 µs
8	12.27 ms	1.523 ms
16	3.522 s	111.2 ms

Table 1. Profiling the enumeration of perfect trees

memoisation is more than 30 times faster than the naive enumerators presented in the previous section. While these results are encouraging, it is worth noting that there is only ever *one* perfectly balanced binary tree of a given depth. As a result, memoisation requires a modest amount of additional memory; as every recursive call is re-used, we can see substantial performance gains. Whether these numbers hold up in a more realistic example, however, will be explored in the next section.

6 CASE STUDY: ENUMERATING WELL-TYPED EXPRESSIONS

To illustrate how to use the datatype generic enumerators for indexed families, this section explores a small case study. In particular, we consider the types (t) and expression language (e) given by the following BNF equations:

$$t ::= \mathsf{nat} \mid \mathsf{bool} \qquad\qquad e ::= \mathbb{N} \mid \mathbb{B} \mid e + e \mid e \wedge e \mid e \leqslant e \mid x \mid \mathsf{let}\ e = x\ \mathsf{in}\ e$$

The types of our expression language correspond to natural numbers or booleans. The language itself is closed under literals, addition, conjunction, comparison, variables and let bindings. Before giving the indexed description corresponding to this expression language, it can be useful to give a direct datatype declaration for the types and expressions involved. Defining a datatype for the types of our language is entirely trivial:

```
data Type : Set where
  nat bool : Type
```

To model well typed (and well scoped) terms needs a bit more work. By indexing a datatype by both its type and *context*, we can ensure that we cannot construct ill formed expressions:

```
data Expr : List Type → Type → Set where
  nlit  : ℕ → Expr Γ nat
  blit  : Bool → Expr Γ bool
  add   : (x y : Expr Γ nat) → Expr Γ nat
  conj  : (x y : Expr Γ bool) → Expr Γ bool
  leq   : (x y : Expr Γ nat) → Expr Γ bool
  var   : t ∈ Γ → Expr Γ t
  letin : Expr Γ s → Expr (s : Γ) t → Expr Γ t
  wk    : Expr Γ t → Expr (s : Γ) t
```

Here we have modeled the context as a list of types, describing the types of all the variables in scope. A variable, given by the var constructor, simply refers to a particular element of this context. The remaining constructors are mostly unsurprising, with the exception of the wk constructor. By adding this explicit weakening operation to our language, we hope to facilitate memoisation—enabling us to re-use previously computed expressions after going under a let-binder.

```
ExprD : (Fix (k (List Type) ⊗ k Type)) ⟶ IDesc List ((Fix (k (List Type) ⊗ k Type)))
ExprD (In (Γ , t@nat))
  = (`Σ _ {[ 0 ]} λ _ ⟶ one)                          -- nlit
  ⊕ (var (In (Γ , nat)) ⊗ var (In (Γ , nat)))         -- add
  ⊕ (`Σ (t ∈ Γ) {vars t Γ} λ _ ⟶ one)                 -- var
  ⊕ ((var (In (Γ , nat)) ⊗ var (In (nat : Γ , t)))    -- let nat
  ⊕ (var (In (Γ , bool)) ⊗ var (In (bool : Γ , t))))  -- let bool
  ⊕ (case Γ of λ where [] ⟶ zero
                       (s : Γ) ⟶ var (In (Γ , t)))    -- wk
ExprD (In (Γ , t@bool))
  = `Σ Bool {true : false : []} (λ b ⟶ one)           -- blit
  ⊕ (var (In (Γ , bool)) ⊗ var (In (Γ , bool)))       -- conj
  ⊕ (var (In (Γ , nat)) ⊗ var (In (Γ , nat)))         -- leq
  ⊕ (`Σ (t ∈ Γ) {vars t Γ} λ _ ⟶ one)                 -- var
  ⊕ ((var (In (Γ , nat)) ⊗ var (In (nat : Γ , t)))    -- let nat
  ⊕ (var (In (Γ , bool)) ⊗ var (In (bool : Γ , t))))  -- let bool
  ⊕ (case Γ of λ where [] ⟶ zero
                       (s : Γ) ⟶ var (In (Γ , t)))
```

Fig. 1. A description of well-typed expressions

Figure 1 gives an indexed description corresponding to the Expr datatype. While the definition of ExprD seems rather complicated, it can be mechanically reconstructed from the definition of Expr. Doing so is quite straightforward: we match on the description's index, a tuple of a context and a type, and return a coproduct of descriptions for each constructor that can be used to construct an expression with that index. We have simplified this definition in one or two places, in an attempt to keep the number of expressions manageable. In the case for letin, we "inline" all possible combinations to avoid having to choose a type variable with the `Σ combinator. We also treat natural number and Boolean literals as constant types, drawn from a small set of possible choices. Although it is possible to derive such indexed descriptions for a large class of indexed families automatically using Agda's metaprogramming facilities, this is beyond the scope of this paper.

6.1 Using the Generic Enumerator

With the indexed description ExprD at hand, we can immediately obtain an enumerator for well-typed terms, simply by invoking the generic enumerator:

```
expressions : (Γ : List Type) ⟶ (t : Type) ⟶ ℕ ⟶ List (Fix ExprD (In (Γ , t)))
expressions Γ t n = ienumerate (igenumerator ExprD) (In (Γ , t)) n
```

It is worth pointing out that, we know that the resulting enumerator is complete, unique and constructed from fair combinators. Furthermore, we can construct a memoising enumerator equally easily.

```
expressions-memo : (Γ : List Type) ⟶ (t : Type) ⟶ ℕ ⟶ List (Fix ExprD (In (Γ , t)))
expressions-memo Γ t n = gmemorate (gmemorator ExprD) (In (Γ , t)) n
```

It is worth comparing the performance of these two enumerators. Table 2 shows the execution times for enumerating all closed expressions of type nat up to various depths, together with the length of the lists involved. Once again, the absolute numbers are not particularly important. First

Depth	Without memoisation	With memoisation	Number of terms
1	96.79 µs	132.0 µs	1
2	699.0 µs	1.225 ms	5
3	11.57 ms	14.13 ms	143
4	—	—	208471

Table 2. Profiling the enumeration of well-typed expressions with type nat

of all, it is worth noting that the number of terms rapidly explodes—there are more than 200K expressions of depth four or less. While we can compute the total number of elements quickly enough, computing such long lists (and storing them in memory) is intractable. For property-based testing, however, one can still compute the first thousand elements at this depth in a handful of milliseconds.

Contrary to our previous benchmark, however, it appears that memoisation is *slower* than using the non-memoising enumerator. We ascribe this to the sheer length of the lists involved. The memoising enumerator will initially cache many relatively inexpensive computations. It is only as the cost of recomputation increases that memoisation will yield substantial performance gains. Unfortunately, as the depth parameter increases, the number of terms grows so quickly, that this becomes the dominating cost in enumeration.

Does this mean that the memoising enumerators presented in Section 5.2 are pointless? The good news is that, as far as we can tell, the overhead they introduce at small depths remains modest. On the other hand, we can imagine indexed families where the enumerations grow less rapidly, such as red-black trees or sorted lists, where there are fewer constructors and the type information is used to constrain or rule out certain ill-formed elements. In these situations, we expect memoising enumerations to outperform our naive enumerators eventually.

7 DISCUSSION

Related work

There is a large body of related work on property-based testing, datatype generic programming, and datatype 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]. We will roughly divide the related work into these two camps: random generators and enumeration.

Random generators. Since the initial work on QuickCheck, there have been numerous articles porting these ideas to proof assistants. Early work by Dybjer et al. [2005] and Haiyan [2007] was the first to explore 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 [Dénès et al. 2014] that ports QuickCheck to the Coq proof assistant.

Datatype generic generators for indexed families have been developed for QuickChick [Lampropoulos et al. 2018]. Like QuickCheck, the generic generators require an explicit size bound. QuickCheck identifies a completeness property, similar to the one in this paper, by mapping generators to the set of values with a non-zero probability and ensuring all possible inhabitants of a type have such non-zero probabilities of being enumerated.

Proving properties and writing generic instances of these random generators is less straightforward than the definitions given here [Paraskevopoulou et al. 2015] as "QuickChick uses an incomplete heuristic for trying out different sizes in an efficient way." The proofs about QuickChick's behaviour try to abstract over the size parameter where possible. Contrasting this to the approach presented here, where we make the recursive structure an enumerators explicit, we see that the only place where we use size bounds is in proving completeness of the top-level enumeration functions that 'tie the recursive knot' so to speak. The individual combinators, such as those for interleaving and cartesian products, do not mention sizes at all, nor do their proofs of fairness, uniqueness, and completeness.

Claessen et al. [2015] explores how to generate constrained random data that satisfies some predicate. The interface is similar to the combinators presented here, with constructs for handling products, coproducts, empty types, singleton values, and an applicative star operation. However, to handle recursive types requires an explicit Pay constructor to bound the size of the generated data and avoid divergence. Similarly, the proof of uniformity presented has not been formalised in a proof assistant.

Enumeration. To the best of our knowledge, the existing work on datatype enumeration focuses on regular datatypes, rather than the indexed families considered here. The approach we take here is similar in spirit to LeanCheck [Braquehais 2017], that strives to define enumerators using a minimal set of combinators. LeanCheck, however, structures its enumerators using *tiers*, much in the same way as the stream semantics we defined above. To ensure that the enumerators are productive, however, users may need to insert explicit delay when defining enumerators. Furthermore, LeanCheck does not attempt to prove completeness, fairness, or uniqueness of its enumerators, even in the case for regular datatypes.

The work on functional enumerations of algebraic types (FEAT) [Duregård et al. 2012] takes a similar approach to ours—instead of enumerating all possible values, we consider finite lists approximating the elements of an algebraic datatype. The work on FEAT, however, once again requires an explicit pay construct to handle recursive types. The ideas underlying the FEAT library is to sample random elements up to a given size efficiently by caching the size of sub-enumerations. These enumerations, however, are limited to regular datatypes.

Yakushev and Jeuring [2009] consider a similar problem in the context of Haskell, showing how to use the *spine views* [Hinze et al. 2006] on GADTs, extended with a form of existential quantification, to define enumerators—mostly with the aim of enumerating well-typed lambda terms. Their approach is, however, restricted to those invariants that can be expressed using a GADT, rather than the dependent types that can be expressed using the indexed families—including dependent pairs—covered in this paper.

New et al. [2017] give a more thorough treatment of fairness for enumerators, especially aimed at the fair enumeration of *infinite* lists. New et al. consider fairness of (infinite) enumerators to correspond to a non-starvation property—every sub-enumerator will eventually produce its values—the case for finite lists is considered degenerate, using list concatenation rather than the interleaving presented here. The notion of fairness presented in this paper relies on using dependent types extensively: 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. All in all, this work establishes a stronger notion of fairness that holds for finite enumerations.

The work by New et al. does, however, raise an important point that we have ignored in this paper: although binary products and binary sums suffice to model datatypes with any number of constructors, doing so may yield unbalanced enumerators. For example, representing a datatype

with three constructors using binary sums will necessarily skew the enumeration towards one of the three constructors. The solution is clear: generalising the binary products and sums to n-ary products and sums, much in the style of the 'true sums-of-products' approach to datatype generic programming [de Vries and Löh 2014]. Adapting our enumerators is reasonably straightforward, replacing interleaving with transposition and generalising the pair enumerator to compute n-ary cartesian products. We have refrained from doing so in this work, largely to keep the generic presentation as simple as possible.

Future work

Performance. When writing these enumerators, we have not focused on performance. There are plenty of other opportunities for optimisations, such as fusing the repeated map operations over intermediate lists, that we have not pursued in this paper. Furthermore, 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) datatypes studied here, where we may be able to show how another iteration of our generic enumerator extends the size of the (indexed) list of values generated in a predictable fashion. Using these ideas, we could then uniformly sample the inhabitants of an indexed family up to a given size.

Automation. As our case study shows, there is still quite some overhead involved in manually writing the descriptions corresponding to a user-defined datatype. Using Agda's reflection and metaprogramming facilities [Van Der Walt and Swierstra 2012], it should be possible to automate the derivation descriptions for datatypes, 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 datatypes.

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 programmatically configured to restrict the search depth, constructors used, or hypotheses available.

Conclusion

This paper shows how both regular datatypes and indexed families can be enumerated. We have sketched the mechanized proof of completeness and uniqueness for both these generic enumerators, guaranteeing that they eventually produce every possible inhabitant of every type exactly once; these enumerators use combinators that we have shown to be *fair*. Furthermore, we have shown how to avoid recomputation by sharing recursive calls and applying memoisation. The uniform presentation of these enumerators, the simplicity of our definitions, and the formal verification of their properties, provides a fairly complete account of datatype generic enumeration.

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

Edsko de Vries and Andres Löh. 2014. True sums of products. In *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming*. 83–94.

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.

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. 2000. Generalizing generalized tries. *Journal of Functional Programming* 10, 4 (2000), 327–351. https://doi.org/10.1017/S0956796800003713

Ralf Hinze, Johan Jeuring, and Andres Löh. 2004. Type-indexed data types. *Science of Computer Programming* 51, 1-2 (2004), 117–151.

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

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

M. Douglas McIlroy. 1999. Power series, power serious. *Journal of Functional Programming* 9, 3 (1999), 325–337. https://doi.org/10.1017/S0956796899003299

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

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 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