Program Term Generation Through Enumeration of Indexed datatypes (Thesis Proposal)

CAS VAN DER REST

ACM Reference Format:
Cas van der Rest. 2019. Program Term Generation Through Enumeration of Indexed datatypes (Thesis Proposal). 1, 1 (February 2019), 21 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Author's address: Cas van der Rest.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2019 Association for Computing Machinery.
XXXX-XXXX/2019/2-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

A common way of asserting a program's correctness is by defining properties that should universally hold, and asserting these properties over a range of random inputs. This technique is commonly referred to as *property based testing*, and generally consists of a two-step process. Defining properties that universially hold on all inputs, and defining *generators* that sample random values from the space of possible inputs. *QuickCheck* [5] is likely the most well known tool for performing property based tests on haskell programs.

Although coming up with a set of properties that propertly captures a program's behavious might initially seem to be the most involved part of the process, defining suitable generators for complex input data is actually quite difficult as well. Questions such as how to handle datatypes that are inhabited by an infinite numer of values arise, or how to deal with constrained input data. The answers to these questions are reasonably well understood for *Algebraic datatypes (ADT's)*, but no general solution exists when more complex input data is required. In particular, little is known about enumerating and generating inhabitants of *Indexed datatypes*.

The latter may be of interest when considering property based testing in the context of languages with a more elaborate type system than Haskell's, such as *Agda* or *Idris*. Since the techniques used in existing tools such as QuickCheck and SmallCheck for the most part only apply to regular datatypes, meaning that there is no canonical way of generating inhabitants for a large class of datatypes in these languages.

Besides the obvious applications to property based testing in the context of dependently typed languages, a broader understanding of how we can generate inhabitants of indexed datatypes may prove useful in other areas as well. Since we can often capture a programming language's semantics as an indexed datatype, efficient generation of inhabitants of such a datatype may prove useful for testing compiler infrastructure.

1.1 Problem Statement

What is the problem? Illustrate with an example. [2, 17]

1.2 Research Questions and Contributions

The general aim of this thesis is to work towards an answer to the following question:

How can we generically enumerate and/or sample values of indexed datatypes?

Obviously, this question is not easily answered and sparks quite a lot of new questions, of which many are deserving of our attention in their own right. Some examples of interesting further questions include:

- We know that enumeration and sampling is possible for regular datatypes. QuickCheck [5] and SmallCheck [17] do this to generically derive test data generators. However, the question remains for which universes of indexed datatypes we can do the same.
- For more complex datatypes (such as ASTs or lambda terms), the number of values grows *extremely* fast with their size: there are only a few lambda terms (up to α-equivalence) with depth 1 or 2, but for depth 50 there are a little under 10^66 [10] distinguished terms. How can we efficiently sample or enumerate larger values of such datatypes? Can we apply techniques such memoization to extend our reach?
- How can insights gained into the enumeration and sampling of indexed datatypes aid in efficient generation of program terms?
- What guarantees about enumeration or sampling can we give? Can we exhaustively enumerate all datatypes, or are there some classes for which this is not possible (if not, why)?

Intented research contributions.

1.3 Methodology

We use the programming language/proof assistant Agda [14] as our vehicle of choice, with the intention to eventually backport to Haskell in order to be able to investigate the practical applications of our insights in the context of program term generation.

2 BACKGROUND

What is the existing technology and literature that I'll be studying/using in my research [8, 12, 14, 20]

2.1 Prerequisites

The reader is assumed to be familiar (to some extent) with functional programming in general, and Agda and Haskell in particular.

2.2 Dependent Types

Dependent type theory extends a type theory with the possiblity of defining types that depend on values. In addition to familiar constructs, such as the unit type ($\top$) and the empty type $\bot$, one can use so-called Π-types and Σ-types. Π-types capture the idea of dependent function types, that is, *functions* whose output type may depent on the values of its input. Given some type A and a family P of types indexed by values of type A (i.e. P has type $A \to Type$), Π-types have the following definition:

$$\Pi_{(x:A)} P(x) \equiv (x : A) \to P(x)$$

In a similar spirit, Σ-types are ordered *pairs* of which the type of the second value may depend on te first value of the pair.

$$\Sigma_{(x:A)} P(x) \equiv (x : A) P(x)$$

The Curry-Howard equivalence extends to Π- and Σ-types as well: they can be used to model universal and existential quantification [18].

2.3 Agda

Agda is a programming language that implements dependent types [14]. Its syntax is broadly similar to Haskell's, though Agda's type system is vastly more expressive due to the possibility for types to depend on term level values. Agda has a dual purpose as proof assistent based on the Curry-Howard equivalence.

2.3.1 Codata and Sized Types. All definitions in Agda are required to be *total*, meaning that they should be defined and terminate in finite time on all possible inputs. The Halting problem states that it is impossible to define a general procedure that decides whether the latter condition. To ensure that only terminating definitions are accepted, Agda's termination checker uses a sound approximation. A logical consequence is that there are Agda programs that terminate, but are rejected by the termination checker. This means that we cannot work with infinite data in the same way as in the same way as in Haskell, which does not care about termination. This means that co-recursive definitions are often problematic. For example, the following definition is perfectly fine in Haskell:

```
nats :: [Int]
nats = 0 : map (+1) nats
```

meanwhile, an equivalent definition in Agda gets rejected by the Termination checker:

```
nats : List ℕ
nats = 0 :: map suc nats
```

This is no surprise, as the termination checker will reject any recursive calls where there is not at least one argument that is strictly smaller. However, in both Agda and Haskell, an expression such as take 10 nats evaluates to $[0, 1, \ldots, 9]$ in finite time.

Codata. To allow these kind of manipulations on infinite structures, the Agda Standard Library makes the lazy semantics that allow these operations explicit. In the case of lists, this means that we explicitly specify that the recursive argument to the :: constructor is a *Thunk*, which should only be evaluated when needed:

```
data Colist {a} (A : Set a) (i : Size) : Set a where
  [] : Colist A i
  _::_ : A → Thunk (Colist A) i → Colist A i
```

We can now define nats in Agda by wrapping the recursive call in a thunk:

```
nats : ∀ {i : Size} → Colist ℕ i
nats = 0 :: λ where .force → map suc nats'
```

Since colists are possible infinite structures, there are some functions we can define on lists, but not on colists. An example of this is a function calculating the length of a colist:

```
length : ∀ {a : Set} → Colist a ∞ → ℕ
length [] = 0
length (x :: xs) = suc (length' (xs .force))
```

Sized Types. Sized types extend the space of function definitions that are recognized by the termination checker as terminating by tracking information about the size of values in types [1]. Consider the folowing example of a function that increments every element in a list of naturals with its position:

```
incpos : List ℕ → List ℕ
incpos [] = []
incpos (x :: xs) = x :: incpos (map suc xs)
```

The recursive call to incpos gets flagged by the termination checker; we know that map does not alter the length of a list, but the termination checker cannot see this. For all it knows map equals const [1], which would make incpos non-terminating. The size-preserving property of map is not reflected in its type.

We can define an alternative version of the List datatype indexed with Size, which tracks the depth of a value in its type.

```
data List (a : Set) : Size → Set where
  [] : ∀ {i} → List' a i
  _::_ : ∀ {i} → a → List' a i → List' a (↑ i)
```

here ↑ i means that the depth of a value constructed using the :: constructor is one deeper than its recursive argument. Incidently, the recursive depth of a list is equal to its size (or length), but this is not necessarily the case. By indexing values of List with their size, we can define a version of map which reflects in its type that the size of the input argument is preserved:

```
map : ∀ {i} {a b : Set} → (a → b) → List a i → List b i
```

using this definition of map, the definition of incpos is no longer rejected by the termination checker.

2.4 Property Based Testing

Property Based Testing aims to assert properties that universally hold for our programs by parameterizing tests over values and checking them against a collection of test values. An example of a property (in Haskell) would be:

```
reverse_preserves_length :: [a] → Bool
reverse_preserves_length xs = length xs ≡ length (reverse xs)
```

We can *check* this property by taking a collection of lists, and asserting that reverse_preserves_length is true on all test inputs. Libraries for property based testing often include some kind of mechanism to automatically generate collections of test values. Existing tools take different approaches towards generatino of test data: *QuickCheck* [5] randomly generates values within the test domain, while *SmallCheck* [17] and *LeanCheck* [13] exhaustively enumerate all values in the test domain up to a certain point.

2.4.1 Existing Libraries. Many libraries exist for property based testing. This section briefly discusses some of them.

QuickCheck. Published in 2000 by Claessen & Hughes [5], QuickCheck implements property based testing for Haskell. As mentioned before, test values are generated by sampling randomly from the domain of test values. QuickCheck supplies the typeclass `Arbitrary`, whose instances are those types for which random values can be generated. A property of type a → Bool can be tested if a is an instance of `Arbitrary`. Instances for most common Haskell types are supplied by the library.

If a property fails on a testcase, QuickCheck supplies a counterexapmle. Consider the following faulty definition of reverse:

```
reverse :: Eq a ⇒ [a] → [a]
reverse []     = []
reverse (x:xs) = nub ((reverse xs) ++ [x, x])
```

If we now test our function by calling quickCheck reverse_preserves_length, we get the following output:

```
Test.QuickCheck> quickCheck reverse_preserves_length
*** Failed! Falsifiable (after 8 tests and 2 shrinks):
[7,7]
```

We see that a counterexample was found after 8 tests *and 2 shrinks*. Due to the random nature of the tested values, the counterexamples that falsify a property are almost never minimal counterexamples. QuickCheck takes a counterexample and applies some function that produces a collection of values that are smaller than the original counterexample, and attempts to falsify the property using one of the smaller values. By repeatedly *Schrinking* a counterexample, QuickCheck is able to find much smaller counterexamples, which are in general of much more use to the programmer.

Perhaps somewhat surprising is that QuickCheck is also able randomly generate values for function types. The general idea here is that for a function of type a → b, a case expression is generated that switches over the possible constructors for a, and returns a random value of type b for every branch.

(Lazy) SmallCheck. Contrary to QuickCheck, SmallCheck [17] takes an *enumerative* approach to the generation of test data. While the approach to formulation and testing of properties is largely similar to QuickCheck's, test values are not generated at random, but rather exhaustively enumerated up to a certain *depth*. Zero-arity constructors have depth 0, while the depth of any positive arity constructor is one greather than the maximum depth of its arguments. The motivation for this is the *small scope hypothesis*: if a program is incorrect, it will almost allways fail on some small input [3].

In addition to SmallCheck, there is also *Lazy* SmallCheck. In many cases, the value of a property is determined only by part of the input. Additionally, Haskell's lazy semantics allow for functions to be defined on partial inputs. The prime example of this is a property `sorted :: Ord a => [a] -> Bool` that returns `false` when presented with `1:0:`$\bot$. It is not necessary to evaluate $\bot$ to determine that the input list is not ordered.

Partial values represent an entire class of values. That is, `1:0:`$\bot$ can be viewed as a representation of the set of lists that start with `[1, 0]`. By checking properties on partial values, it is possible to falsify a property for an entire class of values in one go, in some cases greatly reducing the amount of testcases needed.

LeanCheck. Where SmallCheck uses a value's *depth* to bound the number of test values, LeanCheck uses a value's *size* [13], where size is defined as the number of construction applications of positive arity.

Both SmallCheck and LeanCheck contain functionality to enumerate functions similar to QuickCheck's `Coarbitrary`.

Feat. A downside to both SmallCheck and LeanCheck is that they do not provide an efficient way to generate or sample large test values. QuickCheck has no problem with either, but QuickCheck generators are often more tedious to write compared to their SmallCheck counterpart. Feat [9] aims to fill this gap by providing a way to efficiently enumerate algebraic types, employing memoization techniques to efficiently find the n^{th} element of an enumeration.

QuickChick. QuickChick is a QuickCheck clone for the proof assistent Coq [8]. The fact that Coq is a proof assistent enables the user to reason about the testing framework itself [16]. This allows one, for example, to prove that generators adhere to some distribution.

2.4.2 Generating Constrained Test Data. Defining a suitable generation of test data for property based testing is notoriously difficult in many cases, independent of whether we choose to sample from or enumerate the space of test values. Writing generators for mutually recursive datatypes with a suitable distribution is especially challanging. Another frequently occuring problem is that of how to test conditional properties with a sparse precondition. The canonical example of this is that of sorted lists. Suppose we have the following `insert` function (in Haskell):

```
insert :: Ord a ⇒ a → [a] → [a]
insert v [] = [v]
insert v (x:xs) | v ⩽ x     = v:x:xs
                | otherwise = x:insert v xs
```

We would like to ensure that sortedness of lists is preserved by `insert`. However, if we define a property to test this:

```
insert_preserves_sorted :: Int → [Int] → Property
insert_preserves_sorted x xs = (sorted xs) ==> sorted (insert' x xs)
```

and invoke QuickCheck in the usual manner (`quickCheck insert_preserves_sorted`), we get the following output:

```
Test.QuickCheck> quickCheck prop_insertPreservesSorted
*** Gave up! Passed only 70 tests; 1000 discarded tests.
```

In essence, two things go wrong here. The obvious problem is that QuickCheck is unable to generate a sufficient amount of relevant test cases due to the sparseness of the precondition. The second and perhaps more subtle problem is that the generated test data for which the precondition holds almost exclusively consists of small values (that is, lists of 0, 1 or 2 elements). These problems make testing both inefficient in terms of computational power required, as well as ineffective. Obviously, things will only get worse once we require more complex test data.

The solution to this problem is to define a custom generator that only generates sorted lists, and remove the precondition from the property. For sorted (integer) lists, defining such a generator is somewhat straightforward

```
gen_sorted :: Gen [Int]
gen_sorted = arbitrary >>= return ∘ diff
  where diff :: [Int] → [Int]
        diff []     = []
        diff (x:xs) = x:map (+x) (diff xs)
```

However, for more complex preconditions defining suitable generators is all but trivial.

2.4.3 Automatic Generation of Specifications. A surprising application of property based testing is the automatic generation of program specifications, proposed by Claessen et al. [6] with the tool *QuickSpec*. QuickSpec automatically generates a set of candidate formal specifications given a list of pure functions, specifically in the form of algebraic equations. Random property based testing is then used to falsify specifications. In the end, the user is presented with a set of equations for which no counterexample was found.

2.5 Techniques for Generating Test Data

As discussed in section 2.4.2, proper generation of test data is a hard problem, and involves a lot of details and subtleties. This section discusses some related work that attempts to tackle this problem.

2.5.1 Lambda Terms. A problem often considered in literature is the generation of (well-typed) lambda terms [4, 10, 15]. Good generation of arbitrary program terms is especially interesting in the context of testing compiler infrastructure, and lambda terms provide a natural first step towards that goal.

Claessen et al. [4] adapt the techniques described in [9] to allow efficient generation of constrained data. They use a variation on rejection sampling, where the space of values is gradually refined by rejecting classes of values through partial evaluation (similar to SmallCheck [17]) until a value satisfying the imposed constrained is found.

An alternative approach centered around the semantics of the simply typed lambda calculus is described in [15]. Contrary to [4], where typechecking is viewed as a black box, they utilize definition of the typing rules to devise an algorithm for generation of random lambda terms. The basic approach is to take some input type, and randomly select an inference rule from the set of rules that could have been applied to arrive at the goal type. Obviously, such a procedure does not guarantee termination, as repeated application of the function application rule will lead to an arbitrarily large goal type. As such, the algorithm requires a maximum search depth and backtracking in order to guarantee that a suitable term will eventually be generated.

2.5.2 Inductive Relations in Coq. An approach to generation of constrained test datat for Coq's QuickChick was proposed by Lampropoulos et al. [11] in their 2017 paper *Generating Good Generators for Inductive Relations*. They observe a common pattern where the required test data is of a simple type, but constrained by some precondition. The precondition is then given by some inductive dependent relation indexed by said simple type. The Sorted datatype is a prime example of this.

They derive such generators by abstracting over dependent inductive relations indexed by simple types. For every constructor, the resulting type uses a set of expressions as indices, that may depend on the constructor's arguments and universally quantified variables. These expressions induce a set of unification constraints that apply when using that particular constructor. These unification constraints are then used when constructing generators to ensure that only values for which the dependent inductive relation is inhabited are generated.

2.6 Generic Programming & Type Universes

If we desire to abstract over the structure of datatypes, we need a suitable type universe to do so. Many such universes have been developed and studied; this section discusses a few of them.

2.6.1 Regular Datatypes. The term *regular datatypes* is often used to refer to the class of datatypes that can be assembled using any combination of products, coproducts, unary constructors, constants (a position that is inhabited by a value of another type) and recursive positions.

Any value that lives in universe induced by these combinators describes a regular datatype, and is generally referred to as a *pattern functor*. We can define a datatype in agda that captures these values:

```
data Reg : Set where
  U : Reg
  _⊕_ : Reg → Reg → Reg
  _⊗_ : Reg → Reg → Reg
  I : Reg
  K : Set → Reg
```

Pattern functors can be interpreted as types in such a way that inhabitants of the interpreted type correspond to inhabitants of the type that is represented by the functor.

```
⟦_⟧ : Reg → Set → Set
⟦ U          ⟧ r = ⊤
⟦ K a        ⟧ r = a
⟦ reg₁ ⊕ reg₂ ⟧ r = ⟦ reg₁ ⟧ r ⊎ ⟦ reg₂ ⟧ r
⟦ reg₁ ⊗ reg₂ ⟧ r = ⟦ reg₁ ⟧ r × ⟦ reg₂ ⟧ r
⟦ I          ⟧ r = r
```

Notice that recursive positions are left explicit. This means that we require an appropriate fixed-point combinator:

```
data μ (f : Reg) : Set where
  ‘μ : ⟦ f ⟧ (μ f) → μ f
```

Example. Consider the pattern functor corresponding to the definition of *List*:

```
List' : Set → Set
List' a = μ (U ⊕ (K a ⊗ I))
```

Notice that this pattern functor denotes a choice between a unary constructor ([]), and a constructor that takes a constant of type *a* and a recursive positions as arguments (::). We can define conversion functions between the standard *List* type, and the interpretation of our pattern functor:

```
fromList : ∀ {a : Set} → List a → List' a
fromList [] = 'μ (inj₁ tt)
fromList (x :: xs) = 'μ (inj₂ (x , fromList xs))

toList : ∀ {a : Set} → List' a → List a
toList ('μ (inj₁ tt)) = []
toList ('μ (inj₂ (fst , snd))) = fst :: toList snd
```

Using such isomorphisms, we can automatically derive functionality for datatypes that can be captured using pattern functors. We will see an example of this in section 3.1.4, where we will derive enumeration of inhabitants for arbitrary pattern functors.

2.6.2 Ornaments. *Ornaments* [7] provide a type universe in which we can describe the structure of indexed datatypes in a very index-centric way. Indexed datatypes are described by *Signatures*, consisting of three elements:

- A function $Op : I \to Set$, that relates indices to operations/constructors
- A function $Ar : Op\ i \to Set$, that describes the arity (with respect to recursive positions) for an operation
- A typing discipline $Ty : Ar\ op \to I$, that describes indices for recursive positions.

When combined into a single structure, we say that Σ_D gives the signature of some indexed datatype $D : I \to Set$:

$$\Sigma_D(I) = \begin{cases} Op : I \to Set \\ Ar : Op\ i \to Set \\ Ty : Ar\ op \to I \end{cases}$$

Example. Let us consider the signature for the Vec type, denoted by $\Sigma_{Vec}(\mathbb{N})$. Recall the definition of the Vec datatype:

```
data Vec {a} (A : Set a) : ℕ → Set a where
  [] : Vec A zero
  _::_ : ∀ {n} (x : A) (xs : Vec A n) → Vec A (suc n)
```

It has the following relation between indices and operations (available constructors):

```
Op-vec : ∀ {a : Set} → ℕ → Set
Op-vec zero = ⊤
Op-vec {a} (suc n) = a
```

If the index is *zero*, we have only the unary constructor [] at our disposal, hence `Op-vec zero = top`. If the index is *suc n*, the number of possible constructions for Vec corresponds to the set of inhabitants of its element type, hence we say that `Op-vec (suc n) = a`.

The [] constructor has no recursive argument, so its arity is $\bot$. Similarly, *cons a* takes one recursive argument, so its arity is $\top$:

```
Ar-vec : ∀ {a : Set} → (n : ℕ) → Op-vec {a} n → Set
Ar-vec zero tt = ⊥
Ar-vec (suc n) op = ⊤
```

The definition of :: dictates that if the index is equal to *suc n*, the index of the recursive argument needs to be *n*. We interpret this as follows: if a vector has length (suc n), its tail has length n. This induces the following typing discipline for Vec:

```
Ty-vec : ∀ {a : Set} → (n : ℕ) → (op : Op-vec {a} n) → Ar-vec n op → ℕ
Ty-vec zero a ()
Ty-vec (suc n) a tt = n
```

This defines the signature for Vec: $\Sigma_{Vec} \triangleq$ `Op-vec` $\triangleleft^{\texttt{Ty-vec}}$ `Ar-vec`.

2.6.3 Functorial Species. [20]

2.6.4 Indexed Functors. The most notable downside to the encoding described in section 2.6.1 is the lack of ability to encode mutually recursive datatypes. This makes geneneric operations on regular types of limited use in the context of program term generation, as abstract syntax trees often make heavy use of mutual recursion.

Löh and Magalhães [12] describe a universe that allows for these kind of mutual recursive structures to be encoded. Codes are indexed with an input and output type (both in Set), and are interpreted as a function between indexed functors. That is, a code of type I ▶ O gets interpreted as a function of type (I → Set) → O → Set. Compared to 2.6.1, a number of combinators are added to the universe, such as a construct for dependent pairs or isomorphisms.

3 PRELIMINARY RESULTS

This section discusses the progress made in the Agda development accompanying this proposal. The main contribution of this development is a set of proven complete combinators that can be used to assemble generators for Agda types, as well as a proven complete derivation mechanism that automatically constructs generators for all Agda types for which an isomorphism exists to some pattern functor.

These isomorphisms are included for a number of common types, together with proofs asserting equivalence between manually defined and derived generators for these types.

3.1 Enumerating Regular Types in Agda

We look at how to enumerate various datatypes in Agda, starting with simple examples such as ℕ or *Bool*, and progressively working towards more complex data. The first question we encounter is what the result of an enumeration should be. The ovious answer is that *enumeratea* should return something of type *Lista*, containing all possible values of type *a*. This is however not possible, as *List* in Agda can only represent a finite list, and many datatypes, such as ℕ have an infinite number of inhabitants. To solve this, we may either use the *Codata* functionality from the standard library, or index our result with some kind of metric that limits the number of solutions to a finite set. The latter approach is what is used by both *SmallCheck*[17] and *LeanCheck*[13], enumerating values up to a certain depth or size.

We admit the same approach as the SmallCheck library, defining an enumerator/generator to be a function of type ℕ → *List a*, where input argument signifies the maximum depth. By working with *List*, ensuring termination becomes a lot easier, since it is by definition a finite structure. Furthermore, proving properties about generators becomes more straightforward, as we can simply prove the desired properties about the *List* type, and lift the result to our generator type.

3.1.1 Basic Combinators. We can define a few basic combinators to allow composition of generators.

Constants. Generators can yield a constant value, e.g. *true* for the *Bool* type. Unary constructors have a recursive depth of zero, so we simply return a singleton list:

```
𝔾-pure : ∀ {a : Set} {n : ℕ} → a → 𝔾 a n
𝔾-pure x _ = [ x ]
```

Application. Many datatypes are constructed by applying a constructor to a value of another datatype. An example is the *just* constructor that takes a value of type *a* and yields a value of type *Maybea*. We can achieve this by lifting the familiar *map* function for lists to the generator type:

```
𝔾-map : ∀ {a b : Set} {n : ℕ} → (a → b) → 𝔾 a n → 𝔾 b n
𝔾-map f x n = map f (x n)
```

Product. When a constructor takes two or more values (e.g. _, _), enumerating all values that can be constructed using that constructor comes down to enumerating all possible combinations of its input values, and applying the constructor. Again, we can do this by defining the canonical cartesian product on lists, and lifing it to the generator type:

```
list-ap : ∀ {ℓ} {a b : Set ℓ} → List (a → b) → List a → List b
list-ap fs xs = concatMap (λ f → map f xs) fst

𝔾-ap : ∀ {a b : Set} → 𝔾 (a → b) → 𝔾 a → 𝔾 b
𝔾-ap f x n = list-ap (f n) (x n)
```

Note that in addition to $\mathbb{G} - ap$, one also needs $\mathbb{G} - map$ to construct values using constructors with arity greater than one. Assuming f generates values of type a, and g generates values of type b, we can generate values of type $a \times b$ using the following snippet:

```
pair : ∀ {a b : Set} → 𝔾 a → 𝔾 b → 𝔾 (a × b)
pair f g = 𝔾-ap (𝔾-map _,_ f) g
```

Notice that $\mathbb{G} - map$, $\mathbb{G} - pure$ and $\mathbb{G} - ap$ make $\mathbb{G}$ an instance of both *Functor* and *Applicative*, allowing us to use Agda's *idiom brackets* to define generators. This allows us to write

```
pair : ∀ {a b : Set} {n : ℕ} → 𝔾 a n → 𝔾 b n → 𝔾 (a × b) n
pair f g = (| f , g |)
```

instead.

Choice. Choice between generators can be defined by first defining a *merge* function on lists

```
merge : ∀ {ℓ} {a : Set ℓ} → List a → List a → List a
merge []       ys = ys
merge (x :: xs) ys = x :: merge ys xs
```

and lifting it to the generator type:

```
_||_ : ∀ {a : Set} {n : ℕ} → 𝔾 a n → 𝔾 a n → 𝔾 a n
x || y = λ n → merge (x n) (y n)
```

Allowing for choice between constructors to be denoted in a very natural way:

```
bool : 𝔾 Bool
bool = (| true |)
     || (| false |)
```

Recursion. Simply using implicit recursion is the most natural way for defining generators for recursive datatypes. However, the following definition that generates inhabitants of $\mathbb{N}$ gets rejected by the termination checker:

```
nats : 𝔾 ℕ
nats = ⟪ zero ⟫
     ∥ ⟪ suc nats ⟫
```

Though the above code does terminate, the termination checker cannot see this. Since the input depth is threaded through the applicative combinators, it is not immediately clear that the depth parameter decreases with the recursive call. We solve this by making recursive positions explicit:

```
nat : 𝔾 ℕ → 𝔾 ℕ
nat μ = ⟪ zero ⟫
      ∥ ⟪ suc μ ⟫
```

and defining an appropriate fixed-point combinator:

```
fix : ∀ {a : Set} → (𝔾 a → 𝔾 a) → 𝔾 a
fix f 0       = []
fix f (suc n) = f (fix f) n
```

This definition of fix gets rejected by the termination checker as well. We will see later how we can fix this. However, it should be apparent that it is terminating under the assumption that f is well-behaved, i.e. it applies the n supplied by fix to its recursive positions.

3.1.2 Indexed Types. Indexed types can be generated as well. Indexed generators can simply be defined as a Π-type, where the generated type depends on some input index:

```
𝔾ᵢ : ∀ {i : Set} → (i → Set) → Set
𝔾ᵢ {i = i} a = (x : i) → 𝔾 (a x)
```

The previously defined combinators can then be easily lifted to work with indexed types:

```
_∥ᵢ_ : ∀ {i : Set} {a : i → Set} → 𝔾ᵢ a → 𝔾ᵢ a → 𝔾ᵢ a
(f ∥ᵢ g) i = f i ∥ g i
```

Throughout the code, a subscript i is used to indicate that we deal with indexed types.

3.1.3 Guarantueeing Termination. We can prove termination for our fixed-point combinator if we somehow enforce that its input function is well behaved. Consider the following example of a generator that does not terminate under our fixed-point combinator:

```
bad : 𝔾 ℕ → 𝔾 ℕ
bad μ _ = map suc (μ 1)
```

Clearly, the base case of fix is never reached. We can solve this by indexing generators with a natural number, and requiring generators to be called with their index, yielding the following alternative definition for $\mathbb{G}$:

```
𝔾 : Set → ℕ → Set
𝔾 a m = (p : Σ[ n ∈ ℕ ] n ≡ m) → List a
```

We then use the following type for recursive generators:

```
⟨⟨_⟩⟩ : (ℕ → Set) → Set
⟨⟨ a ⟩⟩ = ∀ {n : ℕ} → a n → a n
```

Meaning that the resulting generator can only apply *its own input number* to recursive positions. If we now decrease the index explicitly in the fixed-point combinator, the termination checker is able to see that fix allways terminates.

```
fix : ∀ {a : Set} → (n : ℕ) → ⟨⟨ 𝔾 a ⟩⟩ → 𝔾 a n
fix zero    f (.0 , refl)       = []
fix (suc n) f (.suc n , refl) = f {n} (fix n f) (n , refl)
```

Let us reconsider the previous counterexapmle:

```
bad : ⟨⟨ 𝔾 ℕ ⟩⟩
bad μ n = map suc (μ (1 , {!!}))
```

It is impossible to complete this definition when applying any other value than n to the recursive position.

3.1.4 Deriving Enumeration for Regular Types. One may have noticed that the way in which generators are defined is structurally *very* similar to how one would define the corresponding datatypes in Haskell. This similarity is intentional, and serves to illustrate that the definition of many generators is completely mechanical with respect to the structure of the underlying datatype.

If we consider the universe of regular datatypes described in section 2.6.1, we see that there is a clear correspondence between our generator combinators, and the constructors of the *Reg* datatype. We can utilize this correspondence to automatically derive generators for datatypes, given an isomorphism with the fixed-point of some pattern functor.

Generating pattern functors. Recall that by fixing the interpretation of some value f of type *Reg*, we get a type whose inhabitants correspond to the inhabitants of the type that is represented by f. If we thus construct a generator that produces all inhabitants of this type, we have a generator that is isomorphic to a complete generator for the type represented by f. Doing this generically amounts to constructing a function of the following type:

```
deriveGen : (f : Reg) → ⟨⟨ 𝔾 (μ f) ⟩⟩
deriveGen = {!!}
```

Intuitively, this definition is easily completed by pattern matching on f, and returning the appropriate combinator. However, due to the intertwined usage of two fixed-point combinators to deal with recursion, there are quite a few subtleties that need to be taken into account.

We simplify the definition slightly by expanding the generator type: μ has one constructor, with one argument, so we replace $\mu\ f$ by its constructor's argument: $[\![f]\!]\ (\mu\ f)$.

Let us now consider the branch of *deriveGen* that deals with coproducts. We would like to simply write the following:

```
deriveGen (f₁ ⊕ f₂) μ = ⦇ inj₁ (deriveGen f₁ μ) ⦈ ∥ ⦇ inj₂ (deriveGen f₂ μ) ⦈
```

This definition is incorrect, however. The recursive call $deriveGen\ f_1$ yields a generator of type $\langle\langle\ \mathbb{G}\ ([\![\ f_1\ ]\!]\ (\mu\ f_1))\ \rangle\rangle$, meaning that two things go wrong: The recursive argument μ we apply to the recursive call has the wrong type, and recursive positions in f_1 refer to values of type $\mu\ f_1$ instead of $\mu\ (f_1 \oplus f_2)$. A similar problem occurs when attempting to define a suitable definition for products.

We solve this issue by *remembering* the top-level pattern functor for which we are deriving a generator when entering recursive calls to *deriveGen*. This can be done by having the recursive argument be a generator for the interpretation of this top-level pattern functor:

```
deriveGen : ∀ {n : ℕ} → (f g : Reg) → 𝔾 ([[ g ]] (μ g)) n → 𝔾 ([[ f ]] (μ g)) n
```

By using the type signature defined above instead, the previously shown defintion for the coproduct branch is accepted.

In most cases, the initial call to *deriveGen* will have the same value for f and g. Observe that $\forall f \in Reg\ .\ deriveGen\ f\ f : \mathbb{G}\ (\llbracket\ f\ \rrbracket\ (\mu\ f))\ n \to \mathbb{G}\ (\llbracket\ f\ \rrbracket\ (\mu\ f))\ n$, thus we can use fix to obtain a genrator that generates values of type $\llbracket\ f\ \rrbracket\ (\mu\ f)$).

Deriving for the K-combinator. Since we can refer to arbitrary values of Set using the K-combinator, there is no general procedure to construct generators of type $\mathbb{G}\ (\llbracket$ K a $\rrbracket\ (\mu$ g)) for any a and g. At first glance, there are two ways to resolve this issue:

(1) Restrict the set of types to which we can refer using K to those types for which we can automatically derive a generator (i.e. the regular types).
(2) Somehow require the programmer to supply generators for all occurenses of K in the pattern functor, and use those generators

The first approach has as a downside that it limits the expressiveness of derived generators, and excludes references to irregular types, hence we choose to require the user to supply a suitable set of generators that can be used whenever we encounter a value constructed using K.

Since it is likely that we will need to record other information about K constructors beyond generators at some point, we use a separate metadata structure that records whatever auxiliary information necessary. This metadata structure is indexed by some value of the Reg datatype. Values of this type have the exact same structure as their index, with the relevant data stored at the K leaves:

```
data RegInfo (P : Set → Set) : Reg → Set where
  U~    : RegInfo P U
  _⊕~_  : ∀ {f₁ f₂ : Reg}
          → RegInfo P f₁ → RegInfo P f₂
          → RegInfo P (f₁ ⊕ f₂)
  _⊗~_  : ∀ {f₁ f₂ : Reg}
          → RegInfo P f₁ → RegInfo P f₂
          → RegInfo P (f₁ ⊗ f₂)
  I~    : RegInfo P I
  K~    : ∀ {a : Set} → P a → RegInfo P (K a)
```

This means that deriveGen gets an additional parameter of type RegInfo (λ a → ⟨⟨ $\mathbb{G}$ a ⟩⟩) f, where f is the pattern functor we are *currently* deriving a generator for (so not the top level pattern functor):

```
deriveGen : ∀ {f g : Reg} {n : ℕ} → RegInfo (λ a → ⟨⟨ 𝔾 a ⟩⟩) f
          → 𝔾 (⟦ g ⟧ (μ g)) n → 𝔾 (⟦ f ⟧ (μ g)) n
```

In the K branch of deriveGen, we can then simply return the generator that is recorded in the metadata structure:

```
deriveGen {K a} {g} {n} (K~ x) rec = ⟨ x ⟩
```

Deriving generators from isomorphism. We use the following record to witness an isomorphism betwen type a and b:

```
record _≅_ (a b : Set) : Set where
  field
    from : a → b
```

```
to    : b → a
iso₁  : ∀ {x : a} → to (from x) ≡ x
iso₂  : ∀ {y : b} → from (to y) ≡ y
```

The functions *from* and *to* allow for conversion between a and b, while iso_1 and iso_2 assert that these conversion functions do indeed form a bijection between values of type a and type b. Given an isomorphism $a \cong b$, a generator $\mathbb{G}\ a\ n$ can easily be converted to a generator $\mathbb{G}\ b\ n$ by using $(\!|\ \_\cong\_.to\ gen\ |\!)$.

We can say that some type a is `Regular` if there exists some value f of type *Reg* such that a is isomorphic to $\mu\ f$. We capture this notion using the following record:

```
record Regular (a : Set) : Set where
  field
    W : Σ[ f ∈ Reg ] (a ≅ μ f)
```

Given a value of type *Regular a*, we can now derive a generator for a by deriving a generator for f, and traveling through the isomorphism by applying the aforementioned conversion:

```
isoGen : ∀ {n : ℕ} → (a : Set) → ⦃ p : Regular a ⦄
           → RegInfo (λ a → ⟨⟨ 𝔾 a ⟩⟩) (getPf p) → 𝔾 a n
isoGen a ⦃ record {W = f , iso} ⦄ reginfo =
  (|_≅_.to iso ∘ `μ) ⟨ deriveGen {f = f} {g = f} reginfo ⟩ |)
```

3.2 Proving Generator Correctness

Since generators are essentially an embellishment of the *List* monad, we can reasonably expect them to behave according to our expectations. However, it would be better to prove that generators behave as intended. Before we can start reasoning about generators, we need to formulate our properties of interest:

Productivity. We say that a generator g *produces* some value x if there exists some $n \in \mathbb{N}$ such that x is an element of gn. We denote this by $g \rightsquigarrow x$. Below is the Agda formulation for this property:

```
_⇝_ : ∀ {a : Set} → (∀ {n : ℕ} → 𝔾 a n) → a → Set
f ⇝ x = ∃[ n ] (x ∈ f (n , refl))
```

Completeness. A generator $g : \mathbb{G}\ a\ n$ is complete when for all $x : a$, $g \rightsquigarrow x$. Informally, this means that a complete generator will eventually produce any inhabitant of the type it generates, provided it is given a large enough depth bound. We can formulate this in Adga as follows:

```
Complete : ∀ {a : Set} → (∀ {n : ℕ} → 𝔾 a n) → Set
Complete {a} f = ∀ {x : a} → f ⇝ x
```

Equivalence. Informally, two generators of type $\mathbb{G}\ a\ n$ can be considered equivalent if they produce the same elements. We formulate this as a bi-implication between productivity proofs, i.e. for all $x : a$, $g_1 \rightsquigarrow x$ if and only if $g_2 \rightsquigarrow x$. In Agda:

```
_~_ : ∀ {a} (g₁ g₂ : ∀ {n} → 𝔾 a n) → Set
g₁ ~ g₂ = (∀ {x} → g₁ ⇝ x → g₂ ⇝ x) × (∀ {x} → g₂ ⇝ x → g₁ ⇝ x)
```

Notice that equivalence follows trivially from completeness, i.e. if two generators produce the same type, and they are both complete, then they are equivalent:

```
Complete→eq : ∀ {a} {g₁ g₂ : ∀ {n} → 𝔾 a n}
            → Complete g₁ → Complete g₂ → g₁ ~ g₂
Complete→eq p₁ p₂ = (λ _ → p₂) , (λ _ → p₁)
```

3.2.1 Combinator Correctness. A natural starting point is to prove that properties are preserved by combinators. This section is by no means intended to exhaustively enumerate all possible combinations of combinators and properties and prove them correct, but rather serves to illustrate the general structure which can be used to construct such proofs.

We take productivity of choice as an example, hence our goal is to show that if, for some generator $g_1 : \mathbb{G}\ a\ n$ and $x : a$, $g_1 \rightsquigarrow x$, then for all generators g_2 we have that $(g_1 \parallel g_2) \rightsquigarrow x$. Since the $\parallel$-combinator is defined in terms of *merge*, we first prove a similar property over the *merge* function.

```
merge-complete-left : ∀ {ℓ} {a : Set ℓ} {xs ys : List a} {x : a}
                    → x ∈ xs → x ∈ merge xs ys
merge-complete-left (here)           = here
merge-complete-left {xs = _ :: xs} (there p) =
  merge-cong {xs = xs} (merge-complete-left p)
```

merge-cong is a lemma stating that if $y \in merge\ xs\ ys$, then $y \in merge\ (x :: xs)\ ys$; its definition is omitted for conciseness. Armed with the above lemma that asserts left-completeness of the *merge* function, we can set out to prove left-completeness for the $\parallel$-combinator. The key insight here is that the depth bound at which x occurs does not change, thus we can sipmly reuse it, and lift the above lemma to the generator type:

```
∥-complete-left : ∀ {a : Set} {x : a} {f g : ∀ {n : ℕ} → 𝔾 a n}
                → f ⇝ x → (f ∥ g) ⇝ x
∥-complete-left (n , p) = n , merge-complete-left p
```

We can construct a similar proof for products by first proving similar properties about lists, and lifting them to the generator type. Proofs about the productivity of combinators can, in a similar fashion, be lifted to reason about completeness. This allows us to show that if the two operands of a choice are both complete, then the resulting generator is complete as well:

```
∥-Complete : ∀ {a b : Set} {f : ∀ {n : ℕ} → 𝔾 a n} {g : ∀ {n : ℕ} → 𝔾 b n}
           → Complete f → Complete g
           → Complete (⟪ inj₁ f ⟫ ∥ ⟪ inj₂ g ⟫)
```

The definition of $\parallel$-Complete is not particular interesting, as it essentially boils down to invoking previously defined lemmas, with some extra work to deal with the unification of produced values as coproducts.

Depth monotonicity. Contrary to coproducts, the depth bound at which values occur in the production of a generator is not preserved by products. If a value x occurs at depth n, it is by no means guaranteed that (x , y) occurs at depth n for any value y. This poses the following problem: suppose f ⇝ x and g ⇝ y, what depth do we chose when we aim to show that ⟪ f , g ⟫ ⇝ (x , y)?

We might say that the lowest depth that at which the product generator produces the pair (x , y) is equal to max (depth (f ⇝ x)) (depth (g ⇝ y)). However, this includes the implicit assumption that if a generator produces a value at depth n, it will also produce this value at depth m for any $m \geqslant n$. This property follows automatically from the intended meaning of the term *depth bound*, but is in no way enforced in Agda. This means that we cannot complete the proof for product generators without adding the following postulate:

```
postulate depth-monotone :
  ∀ {a : Set} {x : a} {n m : ℕ} {g₁ : ∀ {n : ℕ} → 𝔾 a n}
  → n ⩽ m → x ∈ g₁ (n , refl) → x ∈ g₁ (m , refl)
```

Of course, adding such a postulate is dangerous, since it establishes depth monotonicity for *any* inhabitant of the generator type, while the generator type itself in no way enforces that its inhabitants are actually depth monotone. A better solution would be to make the completeness proof for product generators depend on the depth monotonicity of its operands, shifting the reponsibility to the programmer defining the generator. Additionally, we could write monotonicity proofs for our combinators, hopefully allowing monotonicity proofs to be constructed automatically for derived generators.

3.2.2 Correctness of Derived Generators. When assembling a completeness proof for derived generators, the question arises which metadata structure to use to deal with K-combinators; we need both a generator of the type referred to by the K leave, as well as a proof that it is correct. The natural choice for metadata is then a dependent pair with a generator and a completeness proof. This gives rise to the following formulation of the completeness theorem for derived generators:

```
deriveGen-Complete :
  ∀ {f : Reg} → (md : RegInfo (λ a → Σ[ gen ∈ ⟨⟨ 𝔾 a ⟩⟩ ] Complete ⟨ gen ⟩) f)
              → Complete ⟨ deriveGen {f = f} {g = f} {!!} ⟩
```

Dealing with the K-combinator (again). The question remains what metadata structure to pass to deriveGen. Luckily, usinging an appropriate mapping function, we can transform the input metadata structure into a new structure that is suitable as input for deriveGen. Notice that map-reginfo differs from the regular map in that it requires its input function to be polymorphic in the index of the metadata type.

```
map-reginfo : ∀ {f : Reg} {P Q : Set → Set}
            → (∀ {a : Set} → P a → Q a) → RegInfo P f → RegInfo Q f
map-reginfo f U~          = U~
map-reginfo f (ri ⊕~ ri₁) = map-reginfo f ri ⊕~ map-reginfo f ri₁
map-reginfo f (ri ⊗~ ri₁) = map-reginfo f ri ⊗~ map-reginfo f ri₁
map-reginfo f I~          = I~
map-reginfo f (K~ x)      = K~ (f x)
```

Resulting the following result type:

```
Complete ⟨ deriveGen {f = f} {g = f} (map-reginfo proj₁ info) ⟩
```

Assembling the proof. When attempting assemble a completeness proof we encounter similar issues to when defining deriveGen. Especially in the case of products and coproducts, we would like to recurse on the left- and right subtree before combining the result into the desired proof. This is again problematic, since the proofs resulting from the recursive calls will have the wrong type. To solve this, we use an auxiliary lemma that establishes a productivity proof for deriveGen, where we keep track both of the top level pattern functor for which we are deriving the proof, as well as the top level metadata structure (which is needed for the I-combinator).

```
deriveGen-complete :
  ∀ {f g : Reg} {x : ⟦ f ⟧ (μ g)}
  → (info₁ : RegInfo (λ a → Σ[ gen ∈ ⟨⟨ 𝔾 a ⟩⟩ ] Complete ⟨ gen ⟩) f)
```

```
  → (info₂ : RegInfo (λ a → Σ[ gen ∈ ⟨⟨ 𝔾 a ⟩⟩ ] Complete ⟨ gen ⟩) g)
  → (deriveGen {f = f} {g = g} (map-reginfo proj₁ info₁)
     ⟨ deriveGen {f = g} {g = g} (map-reginfo proj₁ info₂) ⟩) ⇝ x
```

Notice that this type definition unifies the type of recursive calls by applying the fixed point of deriveGen applied to the top level pattern functor. If we choose f and g to be the same pattern functor, we can take the fixed point of deriveGen. Observe that, by definition of fix, gen ⟨ gen ⟩ (n , refl) ≡ ⟨ gen ⟩ (suc n , refl) for any gen : ∀ {n : ℕ} → 𝔾 a n. Hence we can finish the completeness theorem with the following definition:

```
deriveGen-Complete {f} info {x}
  with deriveGen-complete {f = f} {g = f} {x = x} info info
  ... | n , p = suc n , p
```

3.2.3 Equivalence with manually defined generators. With a completeness proof for derived generators at hand, we can prove that generators derived from pattern functors are equivalent their manually defined counterparts. Consider the following generator that generates values of the Maybe type:

```
maybe : ∀ {a : Set} → ⟨⟨ 𝔾 a ⟩⟩ → ⟨⟨ 𝔾 (Maybe a) ⟩⟩
maybe a _ = ⦇ nothing ⦈
          ∥ ⦇ just ⟨ a ⟩ ⦈
```

Given a dependent pair with a generator with type ⟨⟨ 𝔾 a ⟩⟩, and a proof that the fixed point of that generator is a complete generator for values of type a, we can construct a proof that maybe is a complete generator:

```
maybe-Complete : ∀ {a : Set} → (sig : Σ[ gen ∈ ⟨⟨ 𝔾 a ⟩⟩ ] Complete ⟨ gen ⟩)
                 → Complete ⟨ maybe (proj₁ sig) ⟩
maybe-Complete sig {just x} with (proj₂ sig) {x}
... | n , snd =
  suc n , merge-cong {xs = []}
    (++-elem-left (map-preserves-elem snd))
maybe-Complete sig {nothing} = 1 , here
```

The proof basically points out that nothing will allways be at the head of its production, and uses the input dependent pair to establish that all possible values using the just constructor are generated as well. ++-elem-left states that if x ∈ xs, then x ∈ (xs ++ ys) for all ys, and map-preserves-elem that if x ∈ xs, then f x ∈ map f xs.

Assuming an instance argument is in scope of type Regular (Maybe a), we can derive a generator for the Maybe type as well:

```
maybe' : ∀ {n : ℕ} → (a : Set) → ⟨⟨ 𝔾 a ⟩⟩ → 𝔾 (Maybe a) n
maybe' a gen = isoGen (Maybe a) (K~ gen ⊕~ U~)
```

In order to show the completeness of maybe', we need to establish completeness of the generator derived by isoGen. The proof itself is slightly technical so it is omitted here, but it comes down to the following: isoGen works by deriving a generator for pattern functor corresponding to a regular type, and traveling through some isomorphism. We know that generators produced by deriveGen are complete, thus we need to show that the completeness property is preserved when applying an isomorphism. The key insight here is that if g : 𝔾 a n is a complete generator for type a, and f : a → b is a bijection, then ⦇ f g ⦈ : 𝔾 b n is a complete generator for type b.

Given that isoGen-Complete establishes completeness for derived generator, equivalence between the manual and derived generator for the maybe type now trivially follows from their respective completeness:

```
maybe~maybe' : ∀ {a : Set} → (sig : Σ[ gen ∈ ⟨⟨ 𝔾 a ⟩⟩ ] Complete ⟨ gen ⟩)
  → ⟨ maybe (proj₁ sig) ⟩ ~ maybe' a (proj₁ sig)
maybe~maybe' {a} sig = Complete→eq (maybe-Complete sig)
                                   (isoGen-Complete ((K~ sig) ⊕~ U~))
```

3.3 Generalization to Indexed Datatypes

Although having a well understood and proven set of definitions for the enumeration of regular types is defenitely useful, we would like to achieve something similar for indexed datatypes. As described in section 3.1.2, our existing set of combinators can be easily adapted to work with indexed datatypes, meaning that generators for indexed types can be defined in a very natural way. For example, for the Fin datatype:

```
fin : ⟨⟨ 𝔾ᵢ Fin ⟩⟩
fin _ zero     = uninhabited
fin μ (suc n) = ⦇ zero ⦈
              ∥ ⦇ suc (μ n) ⦈
```

Here, uninhabited denotes that a type is uninhabited for a certain index, and is simply defined as const []. Note that uninhabited should be used with care, since it has the potential to be source of inefficiency!

3.3.1 Generation For Ornaments. Section 2.6.2 describes a universe for indexed datatypes called *ornaments*, which might be suitable for automatic derivation of generators for certain indexed datatypes. It can capture a large range of indexed datatypes, though there are some that cannot be described as a signature.

Generic Generators. The procedure for deriving generators for datatypes that can be described as an ornament would largely be the same as the approach we used for regular types: derive a generator that produces inhabitants of the fixed point of some signature, and travel through some isomorphism to obtain a generator for the intended type.

One of the challenges of automatically deriving generators for signature interpretations becomes clear when we recall the definition of the interpretation function defined in section 2.6.2: part of a signature is interpreted as a Π-type. This means that if we desire to derive generators for signatures, we need something similar to QuickCheck's CoArbitrary[5] or SmallCheck's CoSeries[17] to generate all inhabitants of the relevant function space.

Non-describable Datatypes. As mentioned above, not all indexed datatypes can be described as a signature. In particular, constructors are used with arity greater than 1 with dependencies between the indices of recursive calls are problematic. For example, consider the following datatype definition:

```
data Foo : ℕ → Set where
  bar : Foo zero
  baz : ∀ {n m : ℕ} → Foo n → Foo m → Foo (n + m)
```

When attempting to define a signature for this type, we cannot define a suitable typing discipline:

```
Ty-Foo : (n : ℕ) → (op : ⟦ Op-Foo n ⟧ᵤ) → ⟦ Ar-Foo n op ⟧ᵤ → ℕ
Ty-Foo (suc n) tt (inj₁ tt) = {!!}
Ty-Foo (suc n) tt (inj₂ tt) = {!!}
```

The definition of Foo requires that the sum of the last two branches is equal to suc n, but since they are independently determined, there is no way to enforce this requirement. In general this means that we cannot capture any datatype that has a constructor with recursive positions whose indices in some way depend on each other as a signature.

This limitation means, for example, that we cannot describe the simply typed lambda calculus as a signature, since similar dependencies occur when constructing typing judgements for function application.

Monadic Combinators. Perhaps surprisingly, we cannot even manually define a generator for Foo using our standard combinators. Consider the obvious definition:

```
foo : ⟨⟨ 𝔾ᵢ Foo ⟩⟩
foo μ zero     = ⦇ bar ⦈ ‖ ⦇ baz (μ 0) (μ 0) ⦈
foo μ (suc n) = ⦇ baz (μ {!!}) (μ {!!}) ⦈
```

It is not clear what index values to use for the recursive positions. More specifically, we need to know which index was used for the first recursive call in order to determine the index for the second recursive call. Applicative unfortunately is not expressive enough to carry around this kind of contextual information. We can define a Monad instance for 𝔾 to allow these kind of dependencies to exist between generated values:

```
𝔾-bind : ∀ {a b : Set} {n : ℕ} → 𝔾 a n → (a → 𝔾 b n) → 𝔾 b n
𝔾-bind f g = λ n → concatMap ((λ x → x n) ∘ g) (f n)
```

This allows us, for example, to define a generator for Σ-types:

```
Σ-gen : ∀ {a : Set} {P : a → Set} {n : ℕ}
        → ⟨⟨ 𝔾 a ⟩⟩ → ⟨⟨ 𝔾ᵢ P ⟩⟩ → 𝔾 (Σ[ x ∈ a ] P x) n
Σ-gen gₐ gₚ = do x ← ⟨ gₐ ⟩
                 y ← ⟨ gₚ ⟩ᵢ x
                 return (x , y)
```

How can I generalize these results? What problems have I identified or do I expect? [19]

4 TIMETABLE AND PLANNING

4.1 Roadmap

4.2 Timetable

What will I do with the remainder of my thesis? [4]

Give an approximate estimation/timetable for what you will do and when you will be done.

REFERENCES

[1] Abel, A. Miniagda: Integrating sized and dependent types. *arXiv preprint arXiv:1012.4896* (2010).

[2] Altenkirch, T., and McBride, C. Generic programming within dependently typed programming. In *Generic Programming*. Springer, 2003, pp. 1–20.

[3] Andoni, A., Daniliuc, D., Khurshid, S., and Marinov, D. Evaluating the "small scope hypothesis". In *In Popl* (2003), vol. 2, Citeseer.

[4] Claessen, K., Duregård, J., and Pałka, M. H. Generating constrained random data with uniform distribution. *Journal of functional programming 25* (2015).

[5] Claessen, K., and Hughes, J. Quickcheck: a lightweight tool for random testing of haskell programs. *Acm sigplan notices 46*, 4 (2011), 53–64.

[6] Claessen, K., Smallbone, N., and Hughes, J. Quickspec: Guessing formal specifications using testing. In *International Conference on Tests and Proofs* (2010), Springer, pp. 6–21.

[7] Dagand, P.-É. The essence of ornaments. *Journal of Functional Programming 27* (2017).

[8] Dénès, M., Hritcu, C., Lampropoulos, L., Paraskevopoulou, Z., and Pierce, B. C. Quickchick: Property-based testing for coq. In *The Coq Workshop* (2014).

[9] Duregård, J., Jansson, P., and Wang, M. Feat: functional enumeration of algebraic types. *ACM SIGPLAN Notices 47*, 12 (2013), 61–72.

[10] Grygiel, K., and Lescanne, P. Counting and generating lambda terms. *Journal of Functional Programming 23*, 5 (2013), 594–628.

[11] Lampropoulos, L., Paraskevopoulou, Z., and Pierce, B. C. Generating good generators for inductive relations. *Proceedings of the ACM on Programming Languages 2*, POPL (2017), 45.

[12] Löh, A., and Magalhaes, J. P. Generic programming with indexed functors. In *Proceedings of the seventh ACM SIGPLAN workshop on Generic programming* (2011), ACM, pp. 1–12.

[13] Matela Braquehais, R. *Tools for Discovery, Refinement and Generalization of Functional Properties by Enumerative Testing*. PhD thesis, University of York, 2017.

[14] Norell, U. Dependently typed programming in agda. In *International School on Advanced Functional Programming* (2008), Springer, pp. 230–266.

[15] Pałka, M. H., Claessen, K., Russo, A., and Hughes, J. Testing an optimising compiler by generating random lambda terms. In *Proceedings of the 6th International Workshop on Automation of Software Test* (2011), ACM, pp. 91–97.

[16] Paraskevopoulou, Z., Hriţcu, C., Dénès, M., Lampropoulos, L., and Pierce, B. C. Foundational property-based testing. In *International Conference on Interactive Theorem Proving* (2015), Springer, pp. 325–343.

[17] Runciman, C., Naylor, M., and Lindblad, F. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Acm sigplan notices* (2008), vol. 44, ACM, pp. 37–48.

[18] Wadler, P. Propositions as types. *Communications of the ACM 58*, 12 (2015), 75–84.

[19] Yakushev, A. R., Holdermans, S., Löh, A., and Jeuring, J. Generic programming with fixed points for mutually recursive datatypes. In *ACM Sigplan Notices* (2009), vol. 44, ACM, pp. 233–244.

[20] Yorgey, B. A. Species and functors and types, oh my! In *ACM Sigplan Notices* (2010), vol. 45, ACM, pp. 147–158.