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

Cas van der Rest

January 31, 2019

Contents

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* [3] 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

1.2 Research Questions and Contributions

What is the problem? Illustrate with an example. [1, 12]

What is/are your research questions/contributions? [3]

2 Background

What is the existing technology and literature that I'll be studying/using in my research [6, 10, 11, 14]

2.1 Dependently Typed Programming & Agda

2.1.1 Dependent Type Theory

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:

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

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.

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

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

2.1.2 Codata and Sized Types

Agda requires all functions to be total, where total means that they should be defined on any possible input, and give a result in a finite amount of time. The latter means that Agda is equipped with a termination checker that tries to prove that functions terminate. It is implied by undecidability of the halting problem that such a checker cannot be both sound and complete. Agda's termination checker is sound, meaning that there are functions that terminate which get rejected. This means that we cannot represent infinite structures in the same way as in haskell. For example, we might use the following definition in Haskell: `nats = 0 : map (+1) nats`. A similar definition in Agda will get rejected by the termination checker.

2.2 Property Based Testing

2.2.1 Existing Libraries

2.2.2 Generating Test Data

2.3 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.

2.3.1 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 → Set where
  U     : Reg ⊥
  K     : (a : Set) → Reg a
  _⊕_   : ∀ {a : Set} → Reg a → Reg a → Reg a
  _⊗_   : ∀ {a : Set} → Reg a → Reg a → Reg a
  I     : 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.

2.3.2 Ornaments

Ornaments [5] 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, given by $\Sigma_{Vec}(\mathbb{N})$. Recall the definition of the Vec datatype in listing 1. It has the following relation between index and operations:

```
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: $\Sigma_{Vec} \triangleq \texttt{Op-vec} \triangleleft^{\texttt{Ty-vec}} \texttt{Ar-vec}$.

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

Listing 1: Definition of Vec

We can define signatures for non-indexed datatypes as well by choosing a trivial index, e.g. $I = \top$. This gives $\Sigma_{\mathbb{N}} \triangleq \texttt{Op-nat} \triangleleft^{\texttt{Ty-nat}} \texttt{Ar-nat}$ with the definitions given in listing 2:

2.3.3 Functorial Species

2.3.4 Indexed Functors

2.4 Blockchain Semantics

2.4.1 BitML

2.4.2 UTXO & Extended UTXO

- Libraries for property based testing (QuickCheck, (Lazy) SmallCheck, QuickChick, QuickSpec)
- Type universes (ADT's, Ornaments) [5, 8]
- Generic programming techniques. (pattern functors, indexed functors, functorial species)
- Techniques to generate complex or constrained data (Generating constrained random data with uniform distribution, Generators for inductive relations)
- Techniques to speed up generation of data (Memoization, FEAT)
- Formal specification of blockchain (bitml, (extended) UTxO ledger) [15, 16]
- Representing potentially infinite data in Agda (Colists, coinduction, sized types)

Below is a bit of Agda code:

```
Op-nat : ⊤ → Set
Op-nat tt = ⊤ ⊎ ⊤

Ar-nat : Op-nat tt → Set
Ar-nat (inj₁ x) = ⊥
Ar-nat (inj₂ y) = ⊤

Ty-nat : (op : Op-nat tt) → Ar-nat op → ⊤
Ty-nat (inj₁ x) ()
Ty-nat (inj₂ y) tt = tt
```

Listing 2: Signature definition for $\mathbb{N}$

```
Γ-match : (τ : Ty) → ⟨⟨ ωᵢ (λ Γ → Σ[ α ∈ Id ] Γ [ α ↦ τ ]) ⟩⟩
Γ-match τ μ ∅ = uninhabited
Γ-match τ μ (α ↦ σ :: Γ) with τ ≟ σ
Γ-match τ μ (α ↦ τ :: Γ) | yes refl = ⦇ (α , TOP)           ⦈
                                    ∥ ⦇ (Σ-map POP) (μ Γ) ⦈
Γ-match τ μ (α ↦ σ :: Γ) | no ¬p    = ⦇ (Σ-map POP) (μ Γ) ⦈
```

Listing 3: Definition of Γ-match

3 Preliminary results

3.1 Enumeration of Agda Types

We look at how to enumerate various datatypes in Agda, starting with simple examples such as $\mathbb{N}$ 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 $\mathbb{N}$ 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* [] and *LeanCheck* [], enumerating values up to a certain depth or size.

We admit the same approach as the SmallCheck library, defining an enumerator/generator

```
data Env : Set where
  ∅ : Env
  _↦_::_ : Id → Ty → Env → Env

data _[_↦_] : Env → Id → Ty → Set where

  TOP : ∀ {Γ α τ}
        → (α ↦ τ :: Γ) [ α ↦ τ ]

  POP : ∀ {Γ α β τ σ} → Γ [ α ↦ τ ]
        → (β ↦ σ :: Γ) [ α ↦ τ ]
```

Listing 4: Envirionment definition and membership in *Agda*

$$TOP\ \frac{}{(a \mapsto t : \Gamma)[a \mapsto t]} \qquad POP\ \frac{\Gamma[a \mapsto t]}{(b \mapsto s : \Gamma)[a \mapsto t]}$$

$$VAR\ \frac{\Gamma[a \mapsto \tau]}{\Gamma \vdash a : \tau} \qquad ABS\ \frac{\Gamma, a \mapsto \sigma \vdash t : \tau}{\Gamma \vdash \lambda a \to t : \sigma \to \tau}$$

$$APP\ \frac{\Gamma \vdash f : \sigma \to \tau \quad \Gamma \vdash x : \sigma}{\Gamma \vdash fx : \tau} \qquad LET\ \frac{\Gamma \vdash e : \sigma \quad \Gamma, a \mapsto \sigma \vdash t : \tau}{\Gamma \vdash \texttt{ let } a := e \texttt{ in } t : \tau}$$

Listing 5: Semantics of the *Simply Typed Lambda Calculus*

to be a function of type $\mathbb{N} \to List\ a$, 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 $Maybe a$. 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) fs


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 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 2.3.1, 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 $\langle\langle\ \mathbb{G}\ (\llbracket\ f_1\ \rrbracket\ (\mu\ f_1))\ \rangle\rangle$, 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 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.

3.2 Proving Correctness of Generators

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, consequently be lifted to reason about completeness. This allows us to show that, for example, if the two operands of a choice are both complete, then the resulting generator is complete as well.

3.2.2 Correctness of Derived Generators

3.3 Generalization to Indexed Types

What examples can you handle already? [9]

What prototype have I built? [4, 7]

How can I generalize these results? What problems have I identified or do I expect? [13]

4 Timetable and planning

What will I do with the remainder of my thesis? [2]

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

References

[1] Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. In *Generic Programming*, pages 1–20. Springer, 2003.

[2] Koen Claessen, Jonas Duregård, and Michał H Pałka. Generating constrained random data with uniform distribution. *Journal of functional programming*, 25, 2015.

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

[4] Koen Claessen, Nicholas Smallbone, and John Hughes. Quickspec: Guessing formal specifications using testing. In *International Conference on Tests and Proofs*, pages 6–21. Springer, 2010.

[5] Pierre-Évariste Dagand. The essence of ornaments. *Journal of Functional Programming*, 27, 2017.

[6] Maxime Dénès, Catalin Hritcu, Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C Pierce. Quickchick: Property-based testing for coq. In *The Coq Workshop*, 2014.

[7] Jonas Duregård, Patrik Jansson, and Meng Wang. Feat: functional enumeration of algebraic types. *ACM SIGPLAN Notices*, 47(12):61–72, 2013.

[8] Hsiang-Shang Ko and Jeremy Gibbons. Programming with ornaments. *Journal of Functional Programming*, 27, 2016.

[9] Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C Pierce. Generating good generators for inductive relations. *Proceedings of the ACM on Programming Languages*, 2(POPL):45, 2017.

[10] Andres Löh and José Pedro Magalhaes. Generic programming with indexed functors. In *Proceedings of the seventh ACM SIGPLAN workshop on Generic programming*, pages 1–12. ACM, 2011.

[11] Ulf Norell. Dependently typed programming in agda. In *International School on Advanced Functional Programming*, pages 230–266. Springer, 2008.

[12] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Acm sigplan notices*, volume 44, pages 37–48. ACM, 2008.

[13] Alexey Rodriguez Yakushev, Stefan Holdermans, Andres Löh, and Johan Jeuring. Generic programming with fixed points for mutually recursive datatypes. In *ACM Sigplan Notices*, volume 44, pages 233–244. ACM, 2009.

[14] Brent A Yorgey. Species and functors and types, oh my! In *ACM Sigplan Notices*, volume 45, pages 147–158. ACM, 2010.

[15] Joachim Zahnentferner. An abstract model of utxo-based cryptocurrencies with scripts. *IACR Cryptology ePrint Archive*, 2018:469, 2018.

[16] Joachim Zahnentferner and Input Output HK. Chimeric ledgers: Translating and unifying utxo-based and account-based cryptocurrencies. Technical report, Cryptology ePrint Archive, Report 2018/262, 2018. https://epri nt. iacr. org . . . , 2018.