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

CAS VAN DER REST

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 universally hold on all inputs, and defining *generators* that sample random values from the space of possible inputs. *QuickCheck* [**?**] is likely the most well known tool for performing property based tests on Haskell programs.

Although coming up with a set of properties that properly captures a program's behavior 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 number 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*.

1.1 Problem Statement

Let us consider an example property in the context of QuickCheck:

```
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. Note that any inhabitant of the type [a] can be used test data for reverse_preserves_length. However, a problem occurs when we want to test a conditional property:

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

If we invoke QuickCheck on (`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 testcases 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.

Data invariants, such as sortedness, can often be represented as an indexed datatype:

```
data Sorted {ℓ} : List ℕ → Set ℓ where
  nil    :  Sorted []
  single : ∀ {n : ℕ} → Sorted (n :: [])
```

Author's address: Cas van der Rest.

```
step :   ∀ {n m : ℕ} {xs : List ℕ} → n ⩽ m
         → Sorted {ℓ} (m :: xs) → Sorted {ℓ} (n :: m :: xs)
```

This means we can generate test data for properties with a precondition by generating values of a suitable indexed datatype. Or in this case, enumerating all indices for which the datatype is inhabited. A good understanding of how to generate inhabitants of indexed datatypes might aid in the generation of many types of complex test data, such as well-typed program terms.

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 is quite a broad question, and as such answering it in its entirety is not realistic. Some subproblems worth considering are:

- We know that enumeration and sampling is possible for regular datatypes. QuickCheck [?] and SmallCheck [?] 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} [?] distinct 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)?

Intended research contributions. automatic derivation of generators for at least a subset of indexed datatypes and an implementation in Haskell showing how such derivations can be applied to practical problems.

1.3 Methodology

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

2 BACKGROUND

In this section we discuss relevant existing work and background information about Agda, property based testing, test data generation and datatype generic programming.

2.1 Dependent Types

Dependent type theory allows the definition of 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 depend 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.

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

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

2.2 Agda

Agda is a programming language based on Martin Löf type theory [?]. Its syntax is broadly similar to Haskell's, though Agda's type system is more elaborate in the sense that types may depend on term level values. Agda is also a proof assistant, using the Curry-Howard equivalence to express propositions as types.

2.2.1 Codata and Sized Types. All definitions in Agda are required to be *total*, meaning that they must be defined on all possible inputs, and give a result in finite time. The Halting problem states that it is impossible to define a general procedure that decides the termination condition for all functions, so 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. 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.

We can prevent the termination checker from flagging these kind of operations by making the lazy semantics explicit, using *codata* and sized types. Codata is a general term for possible inifinite data, often described by a co-recursive definition. 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 [?]. 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))
```

In this case length is not accepted by the termination checker because the input colist is indexed with size ∞, meaning that there is no finite upper bound on its size. Hence, there is no guarantee that our function terminates when inductively defined on the input colist.

There are some cases in which we can convince the termination checker that our definition is terminating by using sized types. 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.

2.3 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. Libraries for property based testing often include some kind of mechanism to automatically generate collections of test values. Existing tools take different approaches towards generation of test data: *QuickCheck* [?] randomly generates values within the test domain, while *SmallCheck* [?] and *LeanCheck* [?] exhaustively enumerate all values in the test domain up to a certain point.

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

QuickCheck. Published in 2000 by Claessen & Hughes [?], 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 counterexample. 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 *Shrinking* 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 by modifying the seed of the random generator (which is used to generate the function's output) based on it's input.

(Lazy) SmallCheck. Contrary to QuickCheck, SmallCheck [?] 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 [?].

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:`$\perp$. It is not necessary to evaluate $\perp$ to determine that the input list is not ordered.

Partial values represent an entire class of values. That is, `1:0:`$\perp$ can be viewed as a representation of the set of lists that have prefix `[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* [?], 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`.

Hegdgehog. Hedgehog [?] is a framework similar to QuickCheck, that aims to be a more modern alternative. It includes support for monadic effects in generators and concurrent checking of properties.

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 [?] 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 assistant Coq [?]. The fact that Coq is a proof assistant enables the user to reason about the testing framework itself [?]. This allows one, for example, to prove that generators adhere to some distribution.

2.3.2 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 challenging.

We run into prolems when we desire to generate test data for properties with a precondition. If a property's precondition is satisfied by few input values, it becomes unpractical to test such a property by simply generating random input data. Few testcases will be relevant (meaning they satisfy the precondition), and the testcases that do are often trivial cases. The usual solution to this problem is to define a custom test data generator that only produces data that satisfies the precondition. In some cases, such as the insert_preserves_sorted from section ??, a suitable generator is not too hard to define:

```
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.

2.3.3 Automatic Generation of Specifications. A surprising application of property based testing is the automatic generation of program specifications, proposed by Claessen et al. [?] 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.

2.4 Techniques for Generating Test Data

This section discusses some existing work regarding the generation of test data satisfying invariants, such as well-formed λ-terms.

2.4.1 Lambda Terms. A problem often considered in literature is the generation of (well-typed) lambda terms [? ? ?]. 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 and Duregaard [?] adapt the techniques described by Duregaard [?] 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 [?]) until a value satisfying the imposed constrained is found.

An alternative approach centered around the semantics of the simply typed lambda calculus is described by Pałka et al. [?]. Contrary to the work done by Claessen and Duregaard [?], 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, though it is not guaranteed that such a term exists if a bound on term size is enforced [?].

Wang [?] considers the problem of generating closed untyped lambda terms.

2.4.2 Inductive Relations in Coq. An approach to generation of constrained test data for Coq's QuickChick was proposed by Lampropoulos et al. [?] 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 shown in section **??** is a good example of this

They derive generators for such datatypes 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.

2.5 Generic Programming & Type Universes

Datatype generic programming concerns techniques that allow for the definition of functions by inducting on the *structure* of a datatype. Many approaches towards this goal have been developed, some more expressive than others. This section discusses a few of them.

2.5.1 SOP (Sum of Products). On of the more simple representations is the so called *Sum of Products* view [?], where datatypes are respresented as a choice between an arbitrary amount of constructors, each of which can have any arity. This view corresponds to how datatypes are defined in Haskell. As we will see (for example in section **??**), other universes too employ sum and product combinators to describe the structure of datatypes, though they do not necessarily enforce the representation to be in disjunctive normal form.

Sum of Products, in its simplest form, cannot represent mutually recursive families of datatypes. An extension that allows this has been developed in [?].

2.5.2 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 (a *code*) represents a regular datatype. 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
```

Codes can be interpreted as types in such a way that inhabitants of the interpretation correspond to inhabitants of the type that is represented by the code.

```
⟦_⟧ : 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
```

The interpretation of a code is a function with type Set → Set, called a *pattern functor*, whose input argument is the type of recursive positions. By using an appropriate fixed-point combinator, we can fix a pattern functor to obtain a type that is isomorphic with the type that is represented by the code.

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

Example. Consider (the fixed point of) a 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 derive functionality for regular datatypes. We will see an example of this in section **??**, where we will derive enumeration of inhabitants for all regular datatypes.

Similar to the pure Sum of Products representation, extensions to this universe have been developed that allow for the encoding of mutually recursive structures [**?**].

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

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

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

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`.

2.5.4 Combinatorial Species. *Combinatorial species* [?] were originally developed as a mathematical framework, but can also be used as an alternative way of looking at datatypes. A *species* can, in terms of functional programming, be thought of as a type constructor with one polymorphic argument. Haskell's ADTs (or regular types in general) can be described by defining familiar combinators for species, such as sum and product.

2.5.5 Indexed Functors. The most notable downside to the encoding described in section **??** is the lack of ability to encode mutually recursive datatypes. This makes generic 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 [?] 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 **??**, 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 regular types, as well as a proven complete derivation mechanism that automatically constructs generators for all 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 obvious answer is that enumerate a should return something of type List a, 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 (see **??**), 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 given depth or size.

We use 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 compared to Colist, as we can simply prove the desired properties about the *List* type, and lift the result to our generator type.

This motivates the following definition for 𝔾 a, representing a generator that produces elements of type a:

```
𝔾 : Set → Set
𝔾 a = ℕ → List a
```

An example generator for the Bool type could be:

```
bool : 𝔾 Bool
bool _ = false :: true :: []
```

3.1.1 Basic Combinators. We can define a few basic combinators to allow composition of generators. The goal eventually is to choose our combinators such that generator definitions bear a close resemblance to a datatype's *structure*.

Constants. Generators can yield a constant value, e.g. true for the Bool type. 𝔾-pure lifts a constant value to the 𝔾 type. Unary constructors have a recursive depth of zero, so we simply return a singleton list, independent of the input depth bound.

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

Constructor 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 lifting 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 𝔾-ap, one also needs 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 × 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 𝔾-map, 𝔾-pure and 𝔾-ap make 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} → 𝔾 a → 𝔾 b → 𝔾 (a × b)
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)
```

We can now write a generator for the Bool type using only our combinators:

```
bool : 𝔾 Bool
bool = ⦇ true ⦈
     || ⦇ false ⦈
```

Recursion. Simply using implicit recursion seems to be the most obvious choice for defining recursive generators. 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 a 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. We will see later how we can resolve 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 Guaranteeing 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 which leads to nontermination when we take its fixpoint:

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

Since μ is allways called with 1, we never reach the branch fix f 0 = [] when applying bad to fix, leading to an infinite chain of calls to fix. We can resolve this by indexing generators with a natural number, and requiring that the input depth bound to a generator is allways the number it is indexed with. This yields the following alternative definition for 𝔾:

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

Notice that in addition to a natural number as input, a proof that the input bound is equal to the generator's index is required. We then use the following type synonym for recursive generators:

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

This definition requires that the recursive positions in a generator are allways called with the same depth bound as the generator itself. If we redefine bad using this new type for recursive generators, the type signature enforces that the input argument μ is called with depth bound n:

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

We cannot complete the above definition of bad since there exists no proof that n ≡ 1.

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)
```

3.1.4 Deriving Enumeration for Regular Types. Generator definitions are 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 **??**, 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 that there exists some isomorphism between said datatype and the fixed point of a 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 the fixed pattern functor, we can create a generator for the type represented by f. Hence we aim to construct the following function:

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

Intuitively, this definition is easily completed by pattern matching on f, and returning the appropriate combinator (recursing where necessary). 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 things 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 ⟨⟨ 𝔾 ($[\![\ f_1\ ]\!]\ (\mu\ f_1)$) ⟩⟩ rather than ⟨⟨ 𝔾 ($[\![\ \mathrm{f}_1\ ]\!]\ (\mu\ (\mathrm{f}_1 \oplus \mathrm{f}_2))$) ⟩⟩. This means 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 $\mu\ (f_1 \oplus f_2)$. This prevents us from unifying the results of recursive calls using the ∥ combinator. A similar problem occurs when attempting to define a suitable definition for products.

We solve this issue by *recording* 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, which means that we can use fix to obtain a generator that generates values of type $[\![\, f \,]\!]\ (\mu\ f))$.

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 𝔾 (⟦ K a ⟧ (μ 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 occurrences 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 non-regular 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 → ⟨⟨ 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 ⦇ _≅_.*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
```

3.2 Proving Generator Correctness

We would like to prove that our generators behave as intended. Most notably, we are interested in a proof of their *completeness*, i.e. a generator produces all elements of a certain type. We show that completeness is preserved by our combinators, and that the derived generators for regular types are complete.

3.2.1 Generator Properties. Before we set out to prove that our generators are complete, we need to establish what exactly completeness means in our context.

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} → Complete g₁ → Complete g₂ → g₁ ∼ g₂
```

3.2.2 Combinator Completeness. We show here how to prove completeness for the _||_ combinator, but proofs for other combinators follow a similar structure. 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 ||-combinator is defined in terms of *merge*, we first prove a similar property over the *merge* function by inducting over x ∈ xs.

```
merge-complete-left : ∀ {ℓ} {a : Set ℓ} {xs ys : List a} {x : a}
                      → x ∈ xs → x ∈ merge xs ys
```

With the above lemma that asserts left-completeness of the *merge* function, we can set out to prove left-completeness for the ||-combinator. The key insight here is that the depth bound at which x occurs does not change, thus we can simply 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
```

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 simply invokes ||-complete-left, though there is an intermediate step where we show that mapping inj₁ or inj₂ over a generator preserves its completeness.

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 assumes 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 responsibility to the programmer defining the generator. Additionally, we aim to provide proofs that our combinators preserve monotonicity.

3.2.3 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: Σ[gen ∈ ⟨⟨ 𝔾 a ⟩⟩] Complete ⟨ gen ⟩.

Proving completeness for the K-combinator. Consider the result type of our completeness proof:

```
Complete ⟨ deriveGen {f = f} {g = f} {!!} ⟩
```

The metadata structure that is required by deriveGen is different from the structure that the completeness proof gets as its input. This means that we require a mapping function that can be used to transform metadata structures:

```
map-reginfo : ∀ {f : Reg} {P Q : Set → Set}
                  → (∀ {a : Set} → P a → Q a) → RegInfo P f → RegInfo Q f
```

In this case, we simply need to extract the first element of the dependent pair, resulting in the following result type for our completeness proof:

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

where info refers to the metadata structure that was provided.

Assembling the proof. When attempting to assemble the final proof, we encounter much of the same problems as with the definition of 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). The general pattern is similar to that used in the definition of deriveGen.

3.2.4 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 to 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 proven complete generator for elements of type a, we can construct a proof that maybe is a complete generator. 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~)
```

We can prove the completeness of this derived generator using our completeness proof for deriveGen. The key observation is that completeness is preserved if we apply a bijective function to the outputs of a generator. Given that such a proof, equivalence between the manual and derived generator for the maybe type now trivially follows from their respective completeness.

3.3 Generation for Indexed Datatypes

Although having a well understood and proven set of definitions for the enumeration of regular types is useful, we would like to achieve something similar for indexed datatypes. As described in section **??**, 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 a source of inefficiency!

Although the current set of combinators can be used to manually assemble generators for some indexed datatypes, we still lack a generic procedure to derive generators for indexed datatypes as well as suitable combinators to capture more intricate dependencies between indices.

4 TIMETABLE AND PLANNING

In the remainder of this thesis, we aim to address the following topics:

(1) Generic derivation of generators for multisorted signatures.
(2) Enumeration for datatypes beyond multisorted signatures
(3) A port to Haskell of (part of) our work.

The remainder of this section contains a little bit of elaboration on each of these topics, identifying challenges and possible approaches, as well as (first) steps required. It is important to note that there is little use in starting on the latter two topics before we have finished generic generators for multisorted signatures.

4.1 Generic Generation for Multisorted Signatures

We intend to apply the approach used for generic derivation of generators for regular types to multisorted signatures. That is, automatically derive a generator from a signature that generates values of the fixed point of its interpretation and using a suitable isomorphism to obtain values of the desired type.

Given a family of types x : i → Set The interpretation function described by Dagand [?] interprets signatures as a function from index to a dependent pair (e.g. a Σ-type) consisting of a choice of constructor and a Π-type describing a mapping between arity and an element of x indexed with whatever is returned by the signature's typing discipline:

```
⟦_⟧ : ∀ {i : Set} → Sig i → (x : i → Set) → (i → Set)
⟦ Op ◃ Ar | Ty ⟧ x = λ i → Σ[ op ∈ ⟦ Op i ⟧ᵤ ] Π[ ⟦ Ar op ⟧ᵤ ] x ∘ Ty
```

This means that, if we desire to generically derive a generator for a signature's interpretation, we need a way to generate both Σ- and Π-types. A generic generator for Σ-types can be obtained by adapting a generator for pairs to deal with the dependencies between elements. Π-types, however, are a little more complicated, as we do not have any functionality for dealing with the generation of function types yet.

We identify the following steps necessary in order to achieve generic generation for multisorted signatures:

(1) Defining suitable combinators to deal with the enumeration of function types, similar to SmallCheck's [?] *CoSerial* typeclass.
(2) Extending generators for pairs and function types to generators for Σ- and Π-types. The Applicative class is probably not expressive enough for this, since it does not allow for dependencies between the different parameters of a construction. For example, in the generator for pairs (⦇ x , y ⦈), x and y are generated independently, while in a Σ-type the type of y would depend on the generated value for x. The Monad class is expressive enough to capture such dependencies.
(3) Defining suitable isomorphisms between datatypes and the fixed point of some signature. For the most part this step is relatively trivial, though some challenge arises from the fact that the interpretation of signatures contains a function type.
(4) Formalizing completeness for function types. This step is necessary if we desire to eventually prove completeness for the generators derived from signatures. However, it may be challenging to do so, as it requires that we prove that any arbitrary function occurs in an enumeration.
(5) Automatically deriving generation for the inhabitants of a signature's fixed point. This should be rather straightforward given that we can generate Σ-types and Π-types.
(6) Formalizing completeness for said derivation, and extending it to generators derived from an isomorphism. This step completely depends on whether we succeed in formalizing completeness for function types.

4.2 Beyond Multisorted Signatures

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 for binary trees indexed with their size:

```
data Tree : ℕ → Set where
  leaf : Tree zero
  node : ∀ {n m : ℕ} → Tree n → Tree m → Tree (suc (n + m))
```

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

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

The definition of Tree requires that the sum of the last two branches is equal to 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.

Another problem is that _+_ is surjective on ℕ, so the indices of the recursive positions are not uniquely determined by the index of the result type. Multisorted signatures are only equipped to return a single index for their recursive positions.

These limitations mean, for example, that we cannot describe the simply typed lambda calculus as a signature, since similar dependencies occur when constructing a typing judgement for function application.

In case of the Tree datatype, we can define a generator if we have a way to invert _+_ (That is, for all n, find all pairs of numbers such that they sum to n) combined with an appropriate *Monad* instance for 𝔾.

Solving this problem generically will probably be too difficult, as it essentially equates to synthesizing proofs for arbitrary theorems (which is a hard problem [?]). However, the Tree example points us to a hybrid solution, where *most* of the generator can be mechanically defined based on the datatype definition, but input from the programmer is required to supply a suitable strategy to deal with the more difficult parts of generation (in this case, inversion of the _+_ function).

Hence a good approach may be to establish the common paterns that emerge when defining example generators for more complex datatypes (we have already done so for the simply typed lambda calculus and regular expressions) and trying to find out which part of the generator follow mechanically from the definition of the datatype, and which parts require more thought from the programmer.

4.3 Haskell Implementation

Implementing (part of) our work in Haskell gives us opportunity to work on a few more practical aspects:

(1) Developing a framework for generation and sampling of values of some Generic Algebraic Datatypes [?] based on our Agda development. Though generically enumerating all inhabitants of a GADT is probably a too difficult, we may be able to do so for at least the subset of GADTs that can be described as a multisorted signature. Additionally some work has been done on generic programming for datatypes beyond regular ADTs in Haskell [? ?], which might be useful for our purposes.

(2) Integration with our findings into existing testing facilities for Haskell, such as QuickCheck or SmallCheck, allowing for a wider range of test data to be automatically generated in these frameworks.

(3) Applying memoization techniques in order to achieve efficient sampling and/or generation of complex data. Memoization in the context of functional languages has been studied extensively [? ?] and has shown to be effective in the context of data generation [?], and it might prove effective for improving performance in our case.

Additionally, porting our work to Haskell allows for experimentation with the generation of realistic abstract syntax datatypes.