Universiteit Utrecht

Faculty of Science

Dept. of Information and Computing Sciences

Thesis title

Author

C.R. van der Rest

Supervisor

Dr. W.S. Swierstra

Dr. M.M.T. Chakravarty

Dr. A. Serrano Mena

April 17, 2019

Contents

Declaration

Thanks to family, supervisor, friends and hops!

Abstract

Abstract

1
Introduction

2
Background

3
Literature Review

4

A Combinator Library for Generators

4.1 The Type of Generators

We have not yet specified what it is exactly that we mean when we talk about *generators*. In the context of property based testing, it makes sense to think of generators as entities that produce values of a certain type; the machinery that is responsible for supplying suitable test values. As we saw in section **??**, this can mean different things depending on the library that you are using. *SmallCheck* and *LeanCheck* generators are functions that take a size parameter as input and produce an exhaustive list of all values that are smaller than the generator's input, while *QuickCheck* generators randomly sample values of the desired type. Though various libraries use different terminology to refer to the mechanisms used to produce test values, we will use *generator* as an umbrella term to refer to the test data producing parts of existing libraries.

4.1.1 Examples in Existing Libraries

When comparing generator definitions across libraries, we see that their definition is often more determined by the structure of the datatype they ought to produce values of than the type of the generator itself. Let us consider the *Nat* datatype (definition **??**). In QuickCheck, we could define a generator for the *Nat* datatype as follows:

```
genNat :: Gen Nat
genNat = oneof [pure Zero, Suc <$> genNat]
```

QuickCheck includes many combinators to finetune the distribution of values of the generated type, which are omitted in this case since they do not structurally alter the generator. Compare the above generator to its SmallCheck equivalent:

```
instance Serial m Nat where
  series = cons0 Zero \/ Cons1 Suc
```

Both generator definitions have a strikingly similar structure, marking a choice between the two available constructors (*Zero* and *Suc*) and employing a appropriate combinators to produce values for said constructors. Despite this structural similarity, the underlying types of the respective generators are wildly different, with *genNat* being an *IO* operation that samples

random values and the *Serial* instance being a function taking a depth and producing all values up to that depth.

4.1.2 Separating Structure and Interpretation

The previous example suggests that there is a case to be made for separating a generators structure from the format in which test values are presented. Additionally, by having a single datatype representing a generator's structure, we shift the burden of proving termination from a generator's definition to its interpretation, which in Agda is a considerable advantage. In practice this means that we define some datatype $Gen\ a$ that marks the structure of a generator, and a function $interpret{:}Gen\ a \to T\ a$ that maps an input structure to some $T\ a$, where T which actually produces test values. In our case, we will almost exclusively consider an interpretation of generators to functions of type $\mathbb{N} \to List\ a$, but we could have chosen T to by any other type of collection of values of type a. An implication of this separation is that, given suitable interpretation functions, a user only has to define a single generator in order to be able to employ different strategies for generating test values, potentially allowing for both random and enumerative testing to be combined into a single framework.

This approach means that generator combinators are not functions that operate on a a generator's result, such as merging two streams of values, but rather a constructor of some abstract generator type; *Gen* in our case. This datatype represents generators in a tree-like structure, not unlike the more familiar abstract syntax trees used to represent parsed programs.

4.1.3 The *Gen* Datatype

We define the datatype of generators, $Gen\ a\ t$, to be a family of types indexed by two types. One signifying the type of values that are produced by the generator, and one specifying the type of values produced by recursive positions.

Listing 4.1: Definition of the *Gen* datatype

```
data Gen : (a : Set) → (t : Set) → Set where
  Or   : ∀ {a t   : Set} → Gen a t → Gen a t → Gen a t
  Ap   : ∀ {a t b : Set} → Gen (b → a) t → Gen b t → Gen a t
  Pure : ∀ {a t   : Set} → a → Gen a t
  None : ∀ {a t   : Set} → Gen a t
  μ    : ∀ {a     : Set} → Gen a a
```

Closed generators are then generators produce that produce the values of the same type as their recursive positions:

```
𝔾 : Set → Set
𝔾 a = Gen a a
```

The *Pure* and *Ap* constructors make *Gen* an instance of *Applicative*, meaning that we can (given a fancy operator for denoting choice) denote generators in way that is very similar to their definition:

```
nat : G ℕ
nat = ⦇ zero  ⦈
    ‖ ⦇ suc μ ⦈
```

This serves to emphasize that the structure of generators can, in the case of simpler datatypes, be mechanically derived from the structure of a datatype. We will see how this can be done in chapter ??.

The question remains how to deal with constructors that refer to *other* types. For example, consider the type of lists (definition ??). We can define an appropriate generator following the structure of the datatype definition:

```
list : ∀ {a : Set} → G a → G (List a)
list a = ⦇ []          ⦈
       ‖ ⦇ { }? :: μ ⦈
```

It is however not immediately clear what value to supply to the remaining interaction point. If we inspect its goal type we see that we should supply a value of type *Gen a (List a)*: a generator producing values of type *a*, with recursive positions producing values of type *List a*. This makes little sense, as we would rather be able to invoke other *closed generators* from within a generator. To do so, we add another constructor to the *Gen* datatype, that signifies the invokation of a closed generator for another datatype:

```
Call : ∀ {a t : Set} → Gen a a → Gen a t
```

Using this definition of *Call*, we can complete the previous definition for *list*:

```
list : ∀ {a : Set} → G a → G (List a)
list a = ⦇ []           ⦈
       ‖ ⦇ (Call a) :: μ ⦈
```

4.1.4 Generator Interpretations

We can view a generator's interpretation as any function mapping generators to some type, where the output type is parameterized by the type of values produced by a generator:

```
Interpretation : (Set → Set) → Set
Interpretation T = ∀ {a t : Set} → G t → Gen a t → T a
```

From this definition of *Interpretation*, we can define concrete interpretations. For example, if we want to behave our generators similar to SmallCheck's *Series*, we might define the following concrete instantiation of the *Interpretation* type:

```
GenAsList : Set
GenAsList = Interpretation λ a → ℕ → List a
```

We can then define a generator's behiour by supplying a definition that inhabits the *GenAsList* type:

```
asList : GenAsList
asList gen = { }?
```

The goal type of the open interaction point is then $\mathbb{N} \to List\ a$. We will see in section 4.3 how we can flesh out this particular interpretation. We could however have chosen any other result type, depending on what suits our particular needs. An alternative would be to interpret generators as a *Colist*, omitting the depth bound altogether:

```
GenAsColist : Set
GenAsColist = ∀ {i : Size} → Interpretation λ a → Colist a i
```

4.2 Generalization to Indexed Datatypes

A first approximation towards a generalization of the *Gen* type to indexed types might be to simply lift the existing definition from *Set* to $I \to Set$.

```
Gᵢ : ∀ {I : Set} → (I → Set) → Set
Gᵢ {I} P = (i : I) → G (P i)
```

However, by doing so we implicitly impose the constraint that the recursive positions of a value have the same index as the recursive positions within it. Consider, for example, the *Fin* type (definition ??). If we attempt to define a generator using the lifted type, we run into a problem.

```
fin : Gᵢ Fin
fin zero    = None
fin (suc n) = ⟪ zero     ⟫
            ‖ ⟪ suc { }? ⟫
```

Any attempt to fill the open interaction point with the constructor fails, as it expects a value of $Gen\ (Fin\ n)\ (Fin\ suc\ n)$, but requires both its type parameters to be equal. We can circumvent this issue by using direct recursion.

```
fin : Gᵢ Fin
fin zero    = None
fin (suc n) = ⟪ zero ⟫
            ‖ ⟪ suc (Call (fin n)) ⟫
```

It is however clear that this approach becomes a problem once we attempt to define generators for datatypes with recursive positions which have indices that are not structurally smaller than the index they target. To overcome these limitations we resolve to a separate deep embedding of generators for indexed types.

And consequently the type of closed indexed generators.

Listing 4.2: Definitiong of the Gen_i datatype

```
data Genᵢ {I : Set} : (I → Set) → (I → Set) → I → Set where
  Pureᵢ : ∀ {a t : I → Set} {i : I} → a i → Genᵢ a t i

  Apᵢ   : ∀ {a b t : I → Set} {i1 i2 : I}
          → Genᵢ (λ _ → b i2 → a i1) t i1 → Genᵢ b t i2 → Genᵢ a t i1

  Orᵢ   : ∀ {a t : I → Set} {i : I}
          → Genᵢ a t i → Genᵢ a t i → Genᵢ a t i

  μᵢ    : ∀ {a : I → Set} (i : I) → Genᵢ a a i

  Noneᵢ : ∀ {a t : I → Set} {i : I} → Genᵢ a t i

  Callᵢ : ∀ {t : I → Set} {i : I} {J : Set} {s : J → Set}
          → ((j : J) → Genᵢ s s j) → (j : J) → Genᵢ (λ _ → s j) t i
```

```
Gᵢ : ∀ {I : Set} → (I → Set) → Set
Gᵢ {I} P = (i : I) → Genᵢ P P i
```

Notice how the Ap_i constructor allows for its second argument to have a different index. The reason for this becomes clear when we

With the same combinators as used for the Gen type, we can now define a generator for the Fin type.

```
fin : Gᵢ Fin
fin zero    = empty
fin (suc n) = ⟪ zero    ⟫
            ‖ ⟪ suc (μᵢ n) ⟫
```

Now defining generators for datatypes with recursive positions whose indices are not structurally smaller than the index of the datatype itself can be done without complaints from the termination checker, such as well-scoped λ-terms (definition ??).

```
term : Gᵢ WS
term n = ⟪ var (Callᵢ {i = n} fin n) ⟫
       ‖ ⟪ abs (μᵢ (suc n))          ⟫
       ‖ ⟪ app (μᵢ n) (μᵢ n)         ⟫
```

It is important to note that it is not possible to call indexed generators from simple generators and vice versa with this setup. We can allow this by either parameterizing the $Call$ and $iCall$ constructors with the datatype they refer to, or by adding extra constructors to the Gen and Gen_i datatypes, making them mutually recursive.

4.3 Interpreting Generators as Enumerations

We will now consider an example interpretation of generators where we map values of the *Gen* or Gen_i datatypes to functions of type $\mathbb{N} \to List\ a$. The constructors of both datatypes mimic the combinators used Haskell's *Applicative* and *Alternative* typeclasses, so we can use the *List* instances of these typeclasses for guidance when defining an enumerative interpretation.

Listing 4.3: Interpretation of the *Gen* datatype as an enumeration

```
toList : Interpretation λ a → ℕ → List a
toList _ _           zero    = []
toList g (Or g1 g2)  (suc n) = merge (toList g g1 (suc n)) (toList g g2 (suc n))
toList g (Ap g1 g2)  (suc n) =
  concatMap (λ f → map f (toList g g2 (suc n))) (toList g g1 (suc n))
toList _ (Pure x)    (suc n) = x :: []
toList _ None        (suc n) = []
toList g μ           (suc n) = toList g g n
toList _ (Call g)    (suc n) = toList g g (suc n)
```

Similarly, we can define such an interpretation for the Gen_i datatype similar to listing 4.3 with the only difference being the appropriate indices getting passed to recursive calls. Notice how our generator's behaviour - most notably the intended semantics of the input depth bound - is entirely encoded within the definition of the interpretation. In this case by decrementing n anytime a recursive position is encountered.

4.4 Properties for Enumerations

4.5 Generating Function Types

4.6 Monadic Generators

There are some cases in which the applicative combinators are not expressive enough to capture the desired generator. For example, if we were to define a construction for generation of Σ types, we encounter some problems.

```
gen-Σ : ∀ {I : Set} {P : I → Set} → G I → ((i : I) → G (P i)) → G (Σ[ i ∈ I ] P i)
gen-Σ gi gp = ⦇ (λ x y → x , { }? ) (Call gi) (Call (gp { }? )) ⦈
```

We can extend the *Gen* datatype with a *Bind* operation that mimics the monadic bind operator (≫=) to allow for such dependencies to exist between generated values.

```
gen-Σ : ∀ {I : Set} {P : I → Set} → G I → ((i : I) → G (P i)) → G (Σ[ i ∈ I ] P i)
gen-Σ gi gp = (Call gi) >>= λ i → (Call (gp i)) >>= λ p → Pure (i , p)
```

5 Generic Generators for Regular types

5.1 Testing Literate Agda

5.1.1 Describing Well-Typed Λ terms

The following inductive relation can be used to describe all well-typed terms under a certain context, given a goal type:

Listing 5.1: Well typed lambda terms

```
data _⊢_ (Γ : Ctx) : Ty → Set where

  [Var] : ∀ {τ} → Γ ∋ τ
        → Γ ⊢ τ

  [Abs] : ∀ {α τ σ} → Γ , α : σ ⊢ τ
        → Γ ⊢ σ `→ τ

  [App] : ∀ {τ σ} → Γ ⊢ σ `→ τ → Γ ⊢ σ
        → Γ ⊢ τ
```

6

Deriving Generators for Indexed Containers

7

Deriving Generators for Indexed Descriptions

7.1 Universe Description

We utilize the generic description for indexed datatypes proposed by Dagand [Dag13] in his PhD thesis.

7.1.1 Definition

Indexed descriptions are not much unlike the codes used to describe regular types (that is, the *Reg* datatype), with the differences being:

1. A type parameter I:*Set*, describing the type of indices.
2. A generalized coproduct, $`\sigma$, that denotes choice between n constructors, in favor of the $\oplus$ combinator.
3. Recursive positions storing the index of recursive values
4. Addition of a combinator to encode Σ types which is a generalization of the K combinator.

This amounts to the definition of indexed descriptions described in listing 7.1.1.

Listing 7.1: The Universe of indexed descriptions

```
data IDesc (I : Set) : Set where
  `var : (i : I) → IDesc I
  `1   : IDesc I
  _`×_ : (A B : IDesc I) → IDesc I
  `σ   : (n : ℕ) → (T : Sl n → IDesc I) → IDesc I
  `Σ   : (S : Set) → (T : S → IDesc I) → IDesc I
```

The *Sl* datatype is used to select the right branch from the generic coproduct, and is isomorphic to the *Fin* datatype.

```
data Sl : ℕ → Set where
  · : ∀ {n} → Sl (suc n)
  _ : ∀ {n} → Sl n → Sl (suc n)
```

7.1.2 Examples

7.2 Generic Generators for Indexed Descriptions

8

Program Term Generation

9

Implementation in Haskell

10
Conclusion & Further Work

A

Datatype Definitions

A.1 Natural numbers

Listing A.1: Definition of natural numbers in Haskell and Agda

```
data Nat = Zero
         | Suc N
```

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```

A.2 Finite Sets

Listing A.2: Definition of finite sets in Agda

```
data Fin : ℕ → Set where
  zero : ∀ {n : ℕ} → Fin (suc n)
  suc  : ∀ {n : ℕ} → Fin n → Fin (suc n)
```

A.3 Vectors

Listing A.3: Definition of vectors (size-indexed listst) in Agda

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

A.4 Simple Types

Listing A.4: Definition of simple types in Haskell and Agda

```
data Type = T
          | Type :->: Type
```

```
data Ty : Set where
  `τ     : Ty
  _`→_ : Ty → Ty → Ty
```

A.5 Contexts

Listing A.5: Definition of contexts in Haskell and Agda

```
data Ctx = Empty
         | Cons Id Type Ctx
```

```
data Ctx : Set where
  ∅ : Ctx
  _,_:_ : Ctx → Id → Ty → Ctx
```

A.6 Raw λ-Terms

Listing A.6: Definition of raw λ-terms in Haskell and Agda

```
data RT = Var Id
        | Abs Id RT
        | App RT RT
```

```
data RT : Set where
  $_     : Id → RT
  Λ_⇒_ : Id → RT → RT
  _⊠_    : RT → RT → RT
```

A.7 Lists

Listing A.7: Definition lists and Agda

```
data List (a : Set) : Set where
  []   : List a
  _::_ : a → List a → List a
```

A.8 Well-scoped λ-terms

Listing A.8: Definition well-scoped λ-terms in Agda

```
data WS : ℕ → Set where
  var : ∀ {n : ℕ} → Fin n → WS n
  abs : ∀ {n : ℕ} → WS (suc n) → WS n
  app : ∀ {n : ℕ} → WS n → WS n → WS n
```

Code listings

Listing of tables

Bibliography

[Dag13] Pierre-Evariste Dagand. *A Cosmology of Datatypes*. PhD thesis, Citeseer, 2013.