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

Cas van der Rest

January 24, 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 $A \rightarrow \mathit{Type}$), Π-types have the following definition:

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

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

Agda requires all functions to be total. This means that they are required to yield a result on all inputs in a *finite* amount of time. This means that we cannot work with infinite structures in the same way as in Haskell. For example, the following is perfectly fine in Haskell:

```
infinity :: Nat
infinity = Suc infinity

isSmaller :: Nat → Nat → Bool
isSmaller (Suc n) Zero    = False
isSmaller Zero    (Suc n) = True
isSmaller (Suc n) (Suc m) = isSmaller n m
```

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. Roughly, this class consists of ADT's in haskell, though mutual recursion is not accounted for.

Any value that lives in the 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, i.e. values of the *Reg* datatype, can be interpreted as types. Inhabitants of the interpretation of a pattern functor correspond to the inhabitants of the type that is represented by said pattern functor. We use the following interpretation function:

```
⟦_⟧ : 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 to find a pattern functor's representation in **Set**.

```
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
```

With these definitions, it is now trivial to show that there is indeed an isomorphism between the two:

```
isoList₁ : ∀ {a : Set} {xs : List a} → toList (fromList xs) ≡ xs
isoList₁ {xs = []} = refl
isoList₁ {xs = x :: xs} = cong (_::_ x) isoList₁


isoList₂ : ∀ {a : Set} {xs : List' a} → fromList (toList xs) ≡ xs
isoList₂ {xs = ‘μ (inj₁ x)} = refl
isoList₂ {xs = ‘μ (inj₂ (fst , snd))} = cong (‘μ ∘ inj₂ ∘ _,_ fst) isoList₂
```

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, 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:

$$\begin{aligned}
&\mathsf{Op\text{-}vec} : \forall \{\mathsf{a} : \mathsf{Set}\} \to \mathbb{N} \to \mathsf{Set}\\
&\mathsf{Op\text{-}vec\ zero} = \top\\
&\mathsf{Op\text{-}vec}\ \{\mathsf{a}\}\ (\mathsf{suc\ n}) = \mathsf{a}
\end{aligned}$$

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 $\top$:

$$\begin{aligned}
&\mathsf{Ar\text{-}vec} : \forall \{\mathsf{a} : \mathsf{Set}\} \to (\mathsf{n} : \mathbb{N}) \to \mathsf{Op\text{-}vec}\ \{\mathsf{a}\}\ \mathsf{n} \to \mathsf{Set}\\
&\mathsf{Ar\text{-}vec\ zero\ tt} = \bot\\
&\mathsf{Ar\text{-}vec}\ (\mathsf{suc\ n})\ \mathsf{op} = \top
\end{aligned}$$

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:

$$\begin{aligned}
&\mathsf{Ty\text{-}vec} : \forall \{\mathsf{a} : \mathsf{Set}\} \to (\mathsf{n} : \mathbb{N}) \to (\mathsf{op} : \mathsf{Op\text{-}vec}\ \{\mathsf{a}\}\ \mathsf{n}) \to \mathsf{Ar\text{-}vec\ n\ op} \to \mathbb{N}\\
&\mathsf{Ty\text{-}vec\ zero\ a}\ ()\\
&\mathsf{Ty\text{-}vec}\ (\mathsf{suc\ n})\ \mathsf{a\ tt} = \mathsf{n}
\end{aligned}$$

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:

```
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 $\mathbb{N}$

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:

```
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

```
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*

3 Preliminary results

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.

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

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

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

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

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.