Generic Enumerators

Cas van der Rest
c.r.vanderrest@students.uu.nl
Universiteit Utrecht

Wouter Swierstra
w.s.swierstra@uu.nl
Universiteit Utrecht

Manuel Chakravarty
manuel.chakravarty@iohk.io
...

Introduction

Since the introduction of QuickCheck [3], *property based testing* has proven to be effective for the discovery of bugs. However, defining the properties to test is only part of the story: it is equally important to *generate* suitable test data. In particular, requiring random test data to satisfy arbitrary preconditions can lead to skewed distributions: for example, naively generating random sorted lists will rarely yield long lists. As a result, developers need to design custom generators carefully—but these generators can become arbitrarily complex. When testing a compiler, for example, it can be quite challenging to define a good generator that is guaranteed to produce well-formed programs. [2, 6]

In this brief abstract we propose to address this problem using the observation that well-formed inputs can often be described by (indexed) inductive datatypes. By defining a *generic* procedure for *enumerating* indexed datatypes, we can obtain a way of safely generating precise test data.

Defining generators We will sketch how to define a generic enumerator for a collection of datatypes in several steps:

- We begin by defining some universe of types U together with its semantics of the form $[\![\_]\!] : U \to \mathsf{Set}$;
- Next, we define a datatype generic function

  $$\mathsf{enumerate} : (u : U) \to (n : \mathbb{N}) \to \mathsf{List}\ [\![\ u\ ]\!]$$

  This function produces a list of elements, bounded by some size parameter n;
- Finally, we formulate the key *completeness* property that we expect of our enumerators:

  $$\forall\ u \to (x : [\![\ u\ ]\!]) \to \Sigma[\ n \in \mathbb{N}\ ]\ (x \in \mathsf{enumerate}\ u\ n)$$

  This property states that for each possible x, there is some size n such than x occurs in our enumeration.

We will now sketch three increasingly complex universes, together with their corresponding generic enumerations.

Enumeration of regular types

One of the simplest universes that describes a wide class of algebraic datatypes is the *universe of regular types*. This universe contains the unit type, empty type, constant types, and is closed under products and coproducts.

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

The associated semantics, $[\![\_]\!] : \mathsf{Reg} \to \mathsf{Set} \to \mathsf{Set}$, maps values of type Reg to their corresponding pattern functor. By taking the fixpoint of such a pattern functor, we have a uniform representation of a wide class of (recursive) algebraic datatypes:

```
data Fix (c : Reg) : Set where
  In : ⟦ c ⟧ (Fix c) → Fix c
```

Examples of regular types and their respective codes include natural numbers (U ⊕ I) or lists (U ⊕ (K a ⊗ I)).

It is reasonably straightforward to define a generic enumeration function:

$$\mathsf{enumerate} : (c : \mathsf{Reg}) \to (n : \mathbb{N}) \to \mathsf{List}\ [\![\ \mathsf{Fix}\ c\ ]\!]$$

For example, the enumeration of a coproduct of two codes is a fair merge of the two recursive calls, and for products we enumerate all possible combinations of values.

Enumeration of Indexed Containers

What happens when we consider *indexed* datatypes? Initially, we will consider *indexed containers* [1, 5], a subset of all possible indexed types that are defined by induction over the index type.

Following the presentation in [5], we define indexed containers as a triple of *operations*, *arities* and *typing*:

```
Op : i → Reg
Ar : ∀ {x} → Fix (Op x) → Reg
Ty : ∀ {x} {op : Fix (Op x)} → Fix (Ar op) → i
```

The set Op i describes the set of available operations at index i; Ar op the arities of each constructor; finally, Ty ar gives the index corresponding to the recursive subtree at arity ar. Signatures are interpreted as a function from index to dependent pair, with the first element of the pair denoting a choice of constructor, and the second element being a function that maps each recursive subtree to a value of the type that results from applying the recursive argument with the index given by the typing discipline for that arity.

```
⟦ Op ◃ Ar | Ty ⟧ x = λ i →
  Σ[ op ∈ Fix (Op i) ] (ar : Fix (Ar op)) → x (Ty ar)
```

Interpretations of signatures live in I → Set, hence we also need adapt our fixpoint, Fix, accordingly.

Examples Many familiar indexed datatypes can be described using the universe of indexed containers, such as finite types (Fin), well-scoped lambda terms, or the type of vectors given below:

```
Σ-vec a =
  let op-vec = (λ {zero → U; (suc n) → K a})
      ar-vec = (λ {{zero} tt → Z; {suc n} x → U})
      ty-vec = (λ {{suc n} {a} (In tt) → n})
  in op-vec ◃ ar-vec | ty-vec
```

Each index is associated with a unique operation. We map suc n to a constant type in op-vec, since the :: constructor stores a value along its recursive subtree. The empty vector, [], has no recursive subtrees, hence it arity is the empty type. Any non-empty vector has one subtree, so we assign its arity to be the unit type. This single subtree has an index that is one less than the original index, as described by ty-vec.

Generic enumerators In the definition of indexed containers, we carefully restricted the type of operations and arities to the universe of regular types. As a result, we can reuse the enumeration of regular types to write a generic enumerator for indexed containers. The second component of a signature's interpretation is a function type, so we require an enumerator for function types. Inspired by SmallCheck [7] we can define such an enumerator:

```
cogenerate :
  (ℕ → List a) → (c : Reg) → ℕ → List (⟦ c ⟧ → a)
```

This allows us to define enumerators for both components of the dependent pair:

```
enumOp : ∀ i → ℕ → List ⟦ Op i ⟧
enumAr : ∀ i → (x : ⟦ Op i⟧)
         → ℕ → List ⟦ (y : Ar x) → r (Ty y) ⟧
```

We then sequence these operations using the monadic structure of lists:

```
λ n → enumOp n >>= (λ op → op , enumAr n op)
```

Intuitively, this defines the enumeration of a signature as the union of the enumerations of its constructors.

Indexed Descriptions

Not all indexed families may be readily described as indexed containers. Consider, for example, the type of binary trees indexed by their number of nodes:

```
data Tree (a : Set) : ℕ → Set where
  Leaf : Tree a 0
  Node : ∀ {n m} → Tree a n → a → Tree a m
         → Tree a (suc (n + m))
```

Without introducing further equalities, it is hard to capture the decomposition of the index suc (n + m) into two subtrees of size n and m.

The universe of indexed descriptions, as defined in [4], is capable of representing arbitrary indexed families. This makes two key modifications to the universe of regular types: firstly, recursive positions must store an additional field corresponding to their index. Secondly, a new combinator, Σ is added.

```
I   : (i : I) → IDesc I
'Σ : (S : Set) → (T : S → IDesc I) → IDesc I
```

Their interpretation is rather straightforward.

```
⟦ I i    ⟧ r = r i
⟦ 'Σ S T ⟧ r = Σ[ s ∈ S ] ⟦ T s ⟧ r
```

With the added 'Σ and 'var, we can now describe the Tree datatype:

```
tree : Set → ℕ → IDesc ℕ
tree a zero    = '1
tree a (suc n) = 'Σ (Σ (ℕ × ℕ) λ {(n' , m') → n' + m' ≡ n})
  λ {((n , m) , refl) → I n ⊗ K a ⊗ I m}
```

The dependency between the indices of the left- and right subtrees of nodes is captured by having their description depend on a pair of natural numbers together with a proof that they sum to the required index.

Enumerators for indexed descriptions Since the IDesc universe largely exposes the same combinators as the universe of regular types, we only really need to define enumerate for the 'Σ combinator. This is straightforward once we can enumerate its first component.

```
enumerate : (δ : IDesc I) → ℕ → List ⟦ δ ⟧
enumerate ('Σ s g) =
  λ n → gen n >>= (λ x → x , enumerate (g s) n)
```

However, since 'Σ may range over any type in Set, we have no generic procedure to obtain a suitable enumerator. This creates a separation between the parts of a datatype for which an enumerator can be assembled mechanically, and those parts for which this would be too difficult.

In the case of the Tree datatype, we see that those elements that make it hard to generically enumerate inhabitants of this datatype emerge quite naturally; we merely need to supply a generator inverting addition:

```
+⁻¹ : (n : ℕ) → ℕ
    → List (Σ (ℕ × ℕ) λ {(n' , m') → n' + m' ≡ n})
```

enumerate needs nothing beyond $+^{-1}$ in order to be able to enumerate inhabitants of Tree.

TODO: Dit is in zekere zin het meest interessante aan de hele abstract–leg dit beter uit: omdat je moet Sigmas toestaat over een arbitrary set (of is het niet beter om een expliciete constraint constructor toe te voegen?), kun je geen generieke generator geven. Dus verwacht je die van de gebruiker.

TODO: Noem heel kort quickchick als alternatief

References

[1] Altenkirch, T., Ghani, N., Hancock, P., McBride, C., and Morris, P. Indexed containers. *Journal of Functional Programming 25* (2015).

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

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

[4] Dagand, P.-E. *A Cosmology of Datatypes.* PhD thesis, Citeseer, 2013.

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

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

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