Generic Enumeration of (Indexed) Datatypes

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

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

Introduction

Since the introduction of QuickCheck [3], *property based testing* has proven to be effective for the discovery of bugs in programs. 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 typically lead to shorter 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, 7]

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 ⟦_⟧ : U → Set;
- Next, we define a datatype generic function generate : (u : U) → (n : ℕ) → List ⟦ u ⟧, that produces a list of elements, bounded by some size parameter n;
- Finally, we formulate the key *completeness* property that we expect of our enumerators:

  ∀ u → (x : ⟦ u ⟧) → Σ[n ∈ ℕ] (x ∈ generate u n)

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

We will now sketch three increasingly complex universes, together with their associated enumeration.

Enumeration of regular types

Types in the universe of regular types are defined as elements of the following datatype. They are closed under both products and coproducts [6].

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

The associated semantics map values of Reg to a pattern functor whose fixed point is isomorphic to the type that a code represents. Examples of regular types and their respective codes include natural numbers (U ⊕ I) and Lists (U ⊕ (K a ⊗ I)). We fix pattern functors using the following fixed-point combinator:

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

We can procedurally assemble generator for any type that is isomorphic to the fixed point of some pattern functor in two steps. First, we derive a generator based on the code the pattern functor is derived from, producing values that inhabit its fixed-point. Next, we travel through the appropriate isomorphism to obtain values of the desired type. We assume a type Code : Set that represents codes, and a meta-theory ⟦_⟧ : Code → Set → Set, mapping codes to pattern functors. Enumerators for the various constructions of Reg arise quite naturally given their semantics. We refer the reader to section 3.2 of Ulf Norell and James Chapman's *Dependently Typed Programming in Agda* [6] for a more in depth explanation on how to define functionality over a set of types by induction on codes. We utilize a similar strategy to prove that enumerators derived from codes satisfy our completeness property

Enumeration of Indexed Containers

We extend the approach used for regular types to indexed types by applying the same techniques to two other universes capable of representing *dependent* types. First we consider *indexed containers* [1, 5]. Crucially, these are defined by induction over the index type.

Indexed containers consist of a a triple of *operations*, *arities* and *typing*, defined as the following record fields in Agda:

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

Op i describes the set of available operations/constructors at index i, Ar op the set of arities/recursive subtrees at operation op, and Ty ar the index of 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 describes, per arity, what the recursive subtrees of that operation look like.

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

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

Many familiar indexed datatypes can be described using the universe of indexed containers. Examples include the finite types (Fin), vectors (Vec), and the well-scoped lambda terms. We include an example description of Vec below.

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

Not all indexed families may be readily described in this fashion. 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))
```

The dependency between n and m cannot be expressed using a mapping from arity to index value, hence it is not possible to describe Tree as an indexed container.

Composing enumerators

We purposefully restrict the type of operations and arities to the universe of regular types, since this means that we can apply the techniques described in the previous section to derive enumerators for indexed containers. This requires an enumerator for function types. The authors of SmallCheck [8] provide a good insight into how we might do this for regular types.

Given these ingredients, we may define enumerators for both components of the dependent pair that make up a signature's interpretation.

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

We then sequence these operations using the monadic bind for lists (concatMap):

```
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 operations.

Indexed Descriptions

The universe of indexed descriptions [4] makes two key modifications to the universe of regular types. Recursive positions get an additional field storing their index, and constants may have descriptions depend on them.

```
‘var : (i : I) → IDesc I
‘Σ : (S : Set) (T : S → IDesc I) → IDesc I
```

Their interpretation is rather straightforward.

```
⟦ ‘var 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) → ‘var n ‘× ‘Σ a (λ _ → ‘1) ‘× ‘var 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.

Enumerating Indexed Descriptions

Since the IDesc universe largely exposes the same combinators as the universe of regular types, we only really need to define generate for the ‘Σ combinator.

```
generate : (δ : IDesc I) → ℕ → List ⟦ δ ⟧
generate (‘Σ s g)  =
  λ n → genS n ≫= (λ x → x , generate (g s) n)
```

Here, genS : ℕ → List S is an enumerator producing values of type S. Since S can by any value in Set, we have no generic formula for obtaining an appropriate enumerator. We choose to have the programmer supply this enumerator, leaving the choice between any of the generic approaches described here, or a custom enumerator to their judgement. In the case of the Tree datatype, we see that this results in a neat separation between the parts of an enumerator that can be derived mechanically, and te parts that require more thought.

Conclusion

Using the techniques described here, we may derive generators for a large class of indexed datatypes. Given the generic procedure to derive generators from indexed descriptions, we can create generators for any type that we can describe using these descriptions. This includes types that are more way more complex than the Tree example presented in this abstract, such as well-typed lambda terms, potentially making the work presented here useful in the domain of compiler testing.

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] Norell, U. Dependently typed programming in agda. In *International School on Advanced Functional Programming* (2008), Springer, pp. 230–266.

[7] 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.

[8] 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.