Generic Enumeration of Datatypes

Cas van der Rest
Wouter Swierstra
c.r.vanderrest@students.uu.nl
w.s.swierstra@uu.nl
Universiteit Utrecht

Introduction

Since the introduction of QuickCheck [**?**], *property based testing* has proven to be effective for both the discovery of bugs in programs (BRON). 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.

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

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 aforementioned 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. We fix pattern functors using the following fixed-point combinator:

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

Unfortunately, we cannot define a function deriveGen : (c : Code) → Gen (Fix c) by induction on c, since Fix c implies that any recursive position in c is interpreted as a value of type Fix c. This becomes a problem when branching over (co)products, since the recursive position in both branches will refer to the fixed point of their respective part of the code, not the code as a whole. We can solve this by noting that ⟦ c ⟧ (Fix c) ≅ Fix c, and having deriveGen yield values of type ⟦ c ⟧ (Fix c') instead, where c is the code we are currently inducting over, and c' the *top-level* code (of which c is a subcode). This allows us to induct over codes without changing the type of recursive positions. Calling deriveGen with c ≡ c' allows us to leverage the appropriate isomorphisms to obtain a generator of the desired type.

Proving properties over generator interpretations

If we define a some generator interpretation interpret : Gen a → T a, we can use a similar approach to automatically construct a proof that a generic generator is well-behaved under that interpretation. The exact meaning of well-behaved obviously depends on what type T : Set → Set we interpret generators to.

For example, suppose we interpret generators as functions from ℕ to List a (not unlike SmallCheck's Serial type class (BRON)). We might be interested in proving completeness for all derived generators. That is, for all values x : a, there exists some $n \in \mathbb{N}$ such that x is an element of the list we get by applying n to the interpretation of deriveGen c, where c is the code representing a. This comes down to finding a value that inhabits the following type.

```
∀ {x c} → Σ[ n ∈ ℕ ] (x ∈ interpret (deriveGen c) n)
```

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* (BRON). Crucially, these are defined by induction over the index type.

Indexed containers consist of a a triple of *operations*, *arities* and *typing*:

$$Sig\ I \ = \begin{cases} Op \ : \ I \to Set \\ Ar \ : \ \forall \ \{i\} \ . \ Op\ i \to Set \\ Ty \ : \ \forall \ \{op\} \ . \ Ar\ op \to I \end{cases}$$

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 describes what the recursive subtrees of that operation look like. Interpretations of signatures live in I → Set, hence we need to lift Fix from (Set → Set) → Set to ((I → Set) → I → Set) → I → Set as well.

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.

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))
```

Generic enumeration of indexed containers

It is possible to derive generators for datatypes that are isomorphic to the fixed point of the interpretation of some signature once we can procedurally compose generators that produce values that inhabit said fixed point. Before we are able to achieve this goal, we need to take the following two steps:

1. Restrict the result of Op and Ar to regular types.
2. Derive co-generators for regular types, e.g. generators producting values of type R → b, where R is regular.

Once the above is in place, we have all the ingredients necessary at our disposal to derive generators from signatures. By restricting operations and arities to regular types, we can simple reuse the existing machinery for regular types.

Indexed Descriptions

The universe of indexed descriptions (BRON) 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
```

Again, we use a fixed point combinator that fixes values of kind (I → Set) → I → Set:

```
data Fix {I : Set} (φ : I → IDesc I) (i : I) : Set where
  In : ⟦ φ i ⟧ (Fix φ) → Fix φ i
```

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}
```

Generating Indexed Descriptions

We can take most of the implementation of deriveGen for regular types to derive generators for indexed descriptions, provided we lift it to the appropriate function space. This amounts to the following type signature for deriveGen:

```
deriveGen : ∀ {φ'} → (i : I) → (φ : I → IDesc I)
              → Gen (⟦ φ i ⟧ (Fix φ'))
```

The question remains how to handle the ‘Σ and ‘var combinators. In the case of ‘var, we simply modify Gen such that μ stores the index of recursive values. However, ‘Σ is a bit more tricky. We can easily map the second element from S → IDesc I to S → Gen. However, we have no generic procedure to derive generators for arbitrary values in Set. This means that we need to either restrict S to those types for which we can derive generators, or have the programmer supply an appropriate generator. We opt for the second solution, since it enables the programmer to guide the generation process according to their needs. In the case of the Tree datatype, this means that a programmer only needs to supply a generator that calculates all the inversions of + while deriveGen takes care of the rest.

Conclusion & future work

Using the techniques 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