Generic Generation of Indexed Datatypes

C.R. van der Rest

1 Introduction

Since the introduction of QuickCheck [1], *Property Based Testing* has proven to be effective for both the discovery of bugs in programs (BRON), as well as providing guidance to programmers in the process of formal verification (BRON). However, crafting a set of suitable properties that covers the desired domain is only part of the story; obtaining enough suitable test data can be just as challenging (BRON). Especially in the case of sparse preconditions (such as sortedness for lists), simple rejection sampling will result only a few trivial test cases. A special generator is required in order to find enough suitable test cases. However, defining these generators becomes difficult quickly once we require more complex test data.

We tackle this problem using the observation that constrained data can often be modelled as a family of datatypes, indexed by the type of values the desired precondition ranges over. By defining a generic procedure for the generation of such indexed datatypes, we obtain a way of generating constrained test data. We show that it is possible to derive generators for many indexed datatypes by deriving generators from codes representing these types.

1.1 Defining generators

We represent generators as a deep embedding of the combinators exposed by the Monad and Alternative typeclasses, with additional constructors for recursive positions, calls to other generators and empty generators.

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

This results in a tree-like structure that can consequently be mapped to any desired interpretation.

2 Generation for 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.

2.1 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)
```

We ignore isomorphisms for a moment, but intuitively it seems reasonable to assume that completeness is preserved when applying a bijection to the values produced by a generator. By explicitly supplying a top-level code to deriveGen, we can construct the desired proof by induction over c.

3 Generation for Indexed Types

We extend the approach used for regular types to indexed types by applying the same techniques to two other type universes. First we consider *indexed containers* (BRON), that allow us to describe a small subset of indexed types. Specifically, those types of which the indices of recursive subtrees of a value are uniquely determined by the index of the value

itself. Subsequently we look at the universe of *indexed descriptions* (BRON), which exteds the set of types we can describe by including a first-order combinator for sigma types.

3.1 Indexed Containers

Indexed containers (BRON) describe datatypes as a Signature, which is 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.

3.1.1 Representable datatypes

Many familiar indexed datatypes can be described using the universe of indexed containers. Examples include (Fin), (Vec), and well scoped lambda terms. However, indexed containers fall short once we try to describe datatypes of which the indices of recursive subtrees are not uniquely determined by the index of a value itself. An example of such a type is 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))
```

Both n and m depend on each other in this case, so we cannot describe them using a mapping from arity to index.

3.1.2 Deriving generators for 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.

3.2 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}
```

3.2.1 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.

4 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

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