Feat: Functional Enumeration of Algebraic Types

Jonas Duregård   Patrik Jansson   Meng Wang

Chalmers University of Technology and
University of Gothenburg
{jonas.duregard,patrikj,wmeng}@chalmers.se

Abstract

In mathematics, an enumeration of a set *S* is a bijective function from (an initial segment of) the natural numbers to *S*. We define "functional enumerations" as efficiently computable such bijections. This paper describes a theory of functional enumeration and provides an algebra of enumerations closed under sums, products, guarded recursion and bijections. We partition each enumerated set into numbered, finite subsets.

We provide a generic enumeration such that the number of each part corresponds to the size of its values (measured in the number of constructors). We implement our ideas in a Haskell library called `testing-feat`, and make the source code freely available. Feat provides efficient "random access" to enumerated values. The primary application is property-based testing, where it is used to define both random sampling (for example QuickCheck generators) and exhaustive enumeration (in the style of SmallCheck). We claim that functional enumeration is the best option for automatically generating test cases from large groups of mutually recursive syntax tree types. As a case study we use Feat to test the pretty-printer of the Template Haskell library (uncovering several bugs).

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.2.5 [*Testing and Debugging*]: Testing tools

Keywords Enumeration, Property-based testing, Memoisation

1. Introduction

Enumeration is used to mean many different things in different contexts. Looking only at the *Enum* class of Haskell we can see two distinct views: the list view and the function view. In the list view *succ* and *pred* let us move forward or backward in a list of the form $[start \,..\, end]$. In the function view we have bijective function $toEnum :: Int \rightarrow a$ that allows direct access to any value of the enumeration. The *Enum* class is intended for enumeration types (types whose constructors have no fields), and some of the methods (*fromEnum* in particular) of the class make it difficult to implement efficient instances for more complex types.

The list view can be generalised to arbitrary types. Two examples of such generalisations for Haskell are SmallCheck [Runciman et al. 2008] and the less well-known *enumerable* package. SmallCheck implements a kind of $enumToSize :: \mathbb{N} \rightarrow [a]$ function that provides a finite list of all values bounded by a size limit. *Enumerable* instead provides only a lazy $[a]$ of all values.

Our proposal, implemented in a library called Feat, is based on the function view. We focus on an efficiently computable bijective function $index_a :: \mathbb{N} \rightarrow a$, much like *toEnum* in the *Enum* class. This enables a wider set of operations to explore the enumerated set. For instance we can efficiently implement $enumFrom :: \mathbb{N} \rightarrow [a]$ that jumps directly to a given starting point in the enumeration and proceeds to enumerate all values from that point. Seeing it in the light of property-based testing, this flexibility allows us to generate test cases that are beyond the reach of the other tools.

As an example usage, imagine we are enumerating the values of an abstract syntax tree for Haskell (this example is from the Template Haskell library). Both Feat *and* SmallCheck can easily calculate the value at position 10^5 of their respective enumerations:

```
*Main> index (10^5) :: Exp
AppE (LitE (StringL "")) (CondE (ListE [])
(ListE []) (LitE (IntegerL 1)))
```

But in Feat we can also do this:

```
*Main> index (10^100) :: Exp
ArithSeqE (FromR (AppE (AppE (ArithSeqE
(FromR (ListE []))) ... -- and 20 more lines!
```

Computing this value takes less than a second on a desktop computer. The complexity of indexing is (worst case) quadratic in the size of the selected value. Clearly any simple list-based enumeration would never reach this far into the enumeration.

On the other hand QuickCheck [Claessen and Hughes 2000], in theory, has no problem with generating large values. However, it is well known that reasonable QuickCheck generators are really difficult to write for mutually recursive datatypes (such as syntax trees) – sometimes the generator grows as complex as the code to be tested! SmallCheck generators are easier to write, but fail to falsify some properties that Feat can.

We argue that functional enumeration is the only available option for automatically generating useful test cases from large groups of mutually recursive syntax tree types. Since compilers are a very common application of Haskell, Feat fills an important gap left by existing tools.

For enumerating the set of values of type *a* we partition *a* into numbered, finite subsets (which we call *parts*). The number associated with each part is the size of the values it contains (measured in the number of constructors). We can define a function for computing the cardinality for each part i.e. $card_a :: Part \rightarrow \mathbb{N}$. We can also define $select_a :: Part \rightarrow \mathbb{N} \rightarrow a$ that maps a part number *p* and an index *i* within that part to a value of type *a* and size *p*. Using these functions we define the bijection that characterises our enumerations: $index_a :: \mathbb{N} \rightarrow a$.

We describe (in §2) a simple theory of functional enumeration and provide an algebra of enumerations closed under sums, products, guarded recursion and bijections. We present an efficient Haskell implementation (in §3). These operations make defining enumerations for Haskell data types (even mutually recursive ones) completely mechanical.

The efficiency of Feat relies on memoising (of meta information, not values) and consequently on *sharing*, which is illustrated in detail in §3 and §4.

We discuss (in §5) the generation of data types with invariants, and show (in §6) how to define random sampling (QuickCheck [Claessen and Hughes 2000] generators) and exhaustive enumeration in the style of SmallCheck and combinations of these. In §7 we show results from a case study using Feat to test the pretty-printer of the Template Haskell library and some associated tools.

2. Functional enumeration

For the type E of functional enumerations, the goal of Feat is an efficient indexing function $\mathit{index} :: E\ a \to \mathbb{N} \to a$. For the purpose of property-based testing it is useful with a generalisation of *index* that selects values by giving size and index. Inspired by this fact, we represent the enumeration of a (typically infinite) set S as a *partition* of S, where each part is a numbered finite subset of S representing values of a certain size. Our theory of functional enumerations is a simple algebra of such partitions.

DEFINITION 1 (Functional Enumeration). *A functional enumeration of the set S is a partition of S that is*

- Bijective*, each value in S is in exactly one part (this is implied by the mathematical definition of a partition).*
- Part-Finite*, every part is finite and ordered.*
- Countable*, the set of parts is* countable*.*

□

The countability requirement means that each part has a number. This number is (slightly simplified) the size of the values in the part. In this section we show that this algebra is closed under disjoint union, Cartesian product, bijective function application and guarded recursion. In table 1 there is a comprehensive overview of these operations expressed as a set of combinators, and some important properties that the operations guarantee (albeit not a complete specification).

To specify the operations we make a tiny proof of concept implementation that does not consider efficiency. In §3 and §4 we show an efficient implementation that adheres to this specification.

Representing parts The parts of the partition are finite ordered sets. We first specify a data type *Finite a* that represents such sets and a minimal set of operations that we require. The data type is isomorphic to finite lists, with the additional requirement of unique elements. It has two consumer functions: computing the cardinality of the set and indexing to retrieve a value.

$$\begin{aligned}&\mathit{card}_F :: \mathit{Finite}\ a \to \mathbb{N}\\&(!!_F)\ :: \mathit{Finite}\ a \to \mathbb{N} \to a\end{aligned}$$

As can be expected, $f\ !!_F\ i$ is defined only for $i < \mathit{card}_F\ f$. We can convert the finite set into a list:

$$\begin{aligned}&\mathit{values}_F :: \mathit{Finite}\ a \to [a]\\&\mathit{values}_F\ f = \mathit{map}\ (f\,!!_F)\ [0\,..\,\mathit{card}_F\ f - 1]\end{aligned}$$

The translation satisfies these properties:

$$\begin{aligned}&\mathit{card}_F\ f \equiv \mathit{length}\ (\mathit{values}_F\ f)\\&f\ !!_F\ i \quad \equiv (\mathit{values}_F\ f)\ !!\ i\end{aligned}$$

For constructing *Finite* sets, we have disjoint union, product and bijective function application. The complete interface for building sets is as follows:

Enumeration combinators:

$$\begin{aligned}&\mathit{empty} &&:: E\ a\\&\mathit{singleton} &&:: a \to E\ a\\&(\oplus) &&:: E\ a \to E\ b \to E\ (\mathit{Either}\ a\ b)\\&(\otimes) &&:: E\ a \to E\ b \to E\ (a,b)\\&\mathit{biMap} &&:: (a \to b) \to E\ a \to E\ b\\&\mathit{pay} &&:: E\ a \to E\ a\end{aligned}$$

Properties:

$$\begin{aligned}&\mathit{index}\ (\mathit{pay}\ e)\ i &&\equiv \mathit{index}\ e\ i\\&(\mathit{index}\ e\ i_1 \equiv \mathit{index}\ e\ i_2) &&\equiv (i_1 \equiv i_2)\\&\mathit{pay}\ (e_1 \oplus e_2) &&\equiv \mathit{pay}\ e_1 \oplus \mathit{pay}\ e_2\\&\mathit{pay}\ (e_1 \otimes e_2) &&\equiv \mathit{pay}\ e_1 \otimes e_2\\& &&\equiv e_1 \otimes \mathit{pay}\ e_2\\&\mathit{fix}\ \mathit{pay} &&\equiv \mathit{empty}\\&\mathit{biMap}\ f\ (\mathit{biMap}\ g\ e) &&\equiv \mathit{biMap}\ (f \circ g)\ e\\&\mathit{singleton}\ a \otimes e &&\equiv \mathit{biMap}\ (a,)\ e\\&e \otimes \mathit{singleton}\ b &&\equiv \mathit{biMap}\ (,b)\ e\\&\mathit{empty} \oplus e &&\equiv \mathit{biMap}\ \mathit{Right}\ e\\&e \oplus \mathit{empty} &&\equiv \mathit{biMap}\ \mathit{Left}\ e\end{aligned}$$

Table 1. Operations on enumerations and selected properties

$$\begin{aligned}&\mathit{empty}_F &&:: \mathit{Finite}\ a\\&\mathit{singleton}_F &&:: a \to \mathit{Finite}\ a\\&(\oplus_F) &&:: \mathit{Finite}\ a \to \mathit{Finite}\ b \to \mathit{Finite}\ (\mathit{Either}\ a\ b)\\&(\otimes_F) &&:: \mathit{Finite}\ a \to \mathit{Finite}\ b \to \mathit{Finite}\ (a,b)\\&\mathit{biMap}_F &&:: (a \to b) \to \mathit{Finite}\ a \to \mathit{Finite}\ b\end{aligned}$$

The operations are specified by the following simple laws:

$$\begin{aligned}&\mathit{values}_F\ \mathit{empty}_F &&\equiv [\,]\\&\mathit{values}_F\ (\mathit{singleton}_F\ a) &&\equiv [a]\\&\mathit{values}_F\ (f_1 \oplus_F f_2) &&\equiv\\&\quad \mathit{map}\ \mathit{Left}\ (\mathit{values}_F\ f_1) +\!\!+\ \mathit{map}\ \mathit{Right}\ (\mathit{values}_F\ f_2)\\&\mathit{values}_F\ (f_1 \otimes_F f_2) &&\equiv\\&\quad [(x,y) \mid x \leftarrow \mathit{values}_F\ f_1, y \leftarrow \mathit{values}_F\ f_2]\\&\mathit{values}_F\ (\mathit{biMap}_F\ g\ f) &&\equiv \mathit{map}\ g\ (\mathit{values}_F\ f)\end{aligned}$$

To preserve the uniqueness of elements, the operand of biMap_F must be bijective. Arguably the function only needs to be injective, it does not need to be surjective in the type b. It is surjective into the resulting set of values however, which is the image of the function g on f.

A type of functional enumerations Given the countability requirement, it is natural to define the partition of a set of type a as a function from $\mathbb{N}$ to *Finite a*. For numbers that do not correspond to a part, the function returns the empty set (empty_F is technically not a part, a partition only has non-empty elements).

$$\begin{aligned}&\textbf{type}\ \mathit{Part} = \mathbb{N}\\&\textbf{type}\ E\ a = \mathit{Part} \to \mathit{Finite}\ a\\[4pt]&\mathit{empty} :: E\ a\\&\mathit{empty} = \mathit{const}\ \mathit{empty}_F\\[4pt]&\mathit{singleton} :: a \to E\ a\\&\mathit{singleton}\ a\ 0 = \mathit{singleton}_F\ a\\&\mathit{singleton}\ \_\ \_ = \mathit{empty}_F\end{aligned}$$

Indexing in an enumeration is a simple linear search:

```
index :: E a → ℕ → a
index e i = go 0 where
  go p = if i < card_F (e p)
           then e p !!_F i
           else index e (i − card_F (e p))
```

This representation of enumerations always satisfies countability, but care is needed to ensure bijectivity and part-finiteness when we define the operations in Table 1.

The major drawback of this approach is that we can not determine if an enumeration is finite, which means expressions such as *index empty* 0 fail to terminate. In our implementation (§3) we have a more sensible behaviour (an error message) when the index is out of bounds.

Bijective-function application We can map a bijective function over an enumeration.

```
biMap f e = biMap_F f ∘ e
```

Part-finiteness and bijectivity are preserved by *biMap* (as long as it is always used only with bijective functions). The inverse of *biMap f* is *biMap* f^{-1}.

Disjoint union Disjoint union of enumerations is the pointwise union of the parts.

```
e₁ ⊕ e₂ = λp → e₁ p ⊕_F e₂ p
```

It is again not hard to verify that bijectivity and part-finiteness are preserved. We can also define an "unsafe" version using *biMap* where the user must ensure that the enumerations are disjoint:

```
union :: E a → E a → E a
union e₁ e₂ = biMap (either id id) (e₁ ⊕ e₂)
```

Guarded recursion and costs Arbitrary recursion may create infinite parts. For example in the following enumeration of natural numbers:

```
data N = Z | S N deriving Show
natEnum :: E N
natEnum = union (singleton Z) (biMap S natEnum)
```

All natural numbers are placed in the same part, which breaks part-finiteness. To avoid this we place a *guard* on (at least) all recursive enumerations called *pay*, which pays a "cost" each time it is executed. The cost of a value in an enumeration is simply the part-number associated with the part in which it resides. Another way to put this is that *pay* increases the cost of all values in an enumeration:

```
pay e 0 = empty_F
pay e p = e (p − 1)
```

This definition gives *fix pay* ≡ *empty*. The cost of a value can be specified given that we know the enumeration from which it was selected.

```
cost :: E t → t → ℕ
cost (singleton _) _         ≡ 0
cost (a ⊕ b)       (Left x)  ≡ cost a x
cost (a ⊕ b)       (Right y) ≡ cost b y
cost (a ⊗ b)       (x,y)     ≡ cost a x + cost b y
cost (biMap f e)   x         ≡ cost e (f⁻¹ x)
cost (pay e)       x         ≡ 1 + cost e x
```

We modify *natEnum* by adding an application of *pay* around the entire body of the function:

```
natEnum = pay (union (singleton Z) (biMap S natEnum))
```

Now because we pay for each recursive call, each natural number is assigned to a separate part:

```
*Main> map values_F (map natEnum [0..3])
[[], [Z], [S Z], [S (S Z)]]
```

Cartesian product Product is slightly more complicated to define. The specification of *cost* allows a more formal definition of part:

DEFINITION 2 (Part). *Given an enumeration e, the part for cost p (denoted as P_e^p) is the finite set of values in e such that*

$$(v \in P_e^p) \Leftrightarrow (cost_e\ v \equiv p)$$

□

The specification of cost says that the cost of a product is the sum of the costs of the operands. Thus we can specify the set of values in each part of a product: $P_{a\otimes b}^p = \bigcup_{k=0}^p P_a^k \times P_b^{p-k}$. For our functional representation this gives the following definition:

```
e₁ ⊗ e₂ = pairs where
  pairs p = concat_F (conv (⊗_F) e₁ e₂ p)
concat_F :: [Finite a] → Finite a
concat_F = foldl union_F empty_F
conv :: (a → b → c) → (ℕ → a) → (ℕ → b) → ℕ → [c]
conv f fx fy p = [fx k ‘f‘ fy (p − k) | k ← [0..p]]
```

For each part we define *pairs p* as the set of pairs with a combined cost of *p*, which is the equivalent of $P_{e_1\otimes e_2}^p$. Because the sets of values "cheaper" than *p* in both e_1 and e_2 are finite, *pairs p* is finite for all *p*. For surjectivity: any pair of values (a,b) have costs $ca = cost_{e_1}\ a$ and $cb = cost_{e_2}\ b$. This gives $(a,b) \in (e_1\ ca \otimes_F e_2\ cb)$. This product is an element of $conv\ (\otimes_F)\ e_1\ e_2\ (ca+cb)$ and as such $(a,b) \in (e_1 \otimes e_2)\ (ca+cb)$. For injectivity, it's enough to prove that *pairs p1* is disjoint from *pairs p2* for $p1 \not\equiv p2$ and that (a,b) appears once in *pairs* $(ca+cb)$. Both these properties follow from the bijectivity of e_1 and e_2.

3. Implementation

The implementation in the previous section is thoroughly inefficient; the complexity is exponential in the cost of the input. The cause is the computation of the cardinalities of parts. These are recomputed on each indexing (even multiple times for each indexing). In Feat we tackle this issue with *memoisation*, ensuring that the cardinality of each part is computed at most once for any enumeration.

Finite sets First we implement the *Finite* type as specified in the previous section. *Finite* is implemented directly by its consumers: a cardinality and an indexing function.

```
type Index    = Integer
data Finite a = Finite {card_F :: Index
                       , (!!_F)  :: Index → a
                       }
```

Since there is no standard type for infinite precision *natural* numbers in Haskell, we use *Integer* for the indices. All combinators follow naturally from the correspondence to finite lists (specified in §2). Like lists, *Finite* is a monoid under append (i.e. union):

```
(⊕_F) :: Finite a → Finite a → Finite a
f₁ ⊕_F f₂ = Finite car ix where
  car = card_F f₁ + card_F f₂
  ix i = if i < card_F f₁
           then f₁ !!_F i
           else f₂ !!_F (i − card_F f₁)
```

```
emptyF = Finite 0 (λi → error "Empty")
instance Monoid (Finite a) where
  mempty = emptyF
  mappend = (⊕F)
```

It is also an applicative functor under product, again just like lists:

```
(⊗F) :: Finite a → Finite b → Finite (a,b)
(⊗F) f1 f2 = Finite car sel where
  car = cardF f1 ∗ cardF f2
  sel i = let (q,r) = (i ‘divMod‘ cardF f2)
          in (f1 !!F q,f2 !!F r)
singletonF :: a → Finite a
singletonF a = Finite 1 one where
  one 0 = a
  one _ = error "Index out of bounds"
instance Functor Finite where
  fmap f fin = fin { (!!F) = f ∘ (fin!!F) }
instance Applicative Finite where
  pure    = singletonF
  f ⟨∗⟩ a = fmap (uncurry ($)) (f ⊗F a)
```

For indexing we split the index $i < c_1 * c_2$ into two components by dividing either by c_1 or c_2. For an ordering which is consistent with lists (s.t. $values_F\ (f \langle * \rangle a) \equiv values_F\ f \langle * \rangle values_F\ a$) we divide by the cardinality of the second operand. Bijective map is already covered by the *Functor* instance, i.e. we require that the argument of *fmap* is a bijective function.

Enumerate As we hinted earlier, memoisation of cardinalities (i.e. of *Finite* values) is the key to efficient indexing. The remainder of this section is about this topic and implementing efficient versions of the operations specified in the previous section. A simple solution is to explicitly memoise the function from part numbers to part sets. Depending on where you apply such memoisation this gives different memory/speed tradeoffs (discussed later in this section).

In order to avoid having explicit memoisation we use a different approach: we replace the outer function with a list. This may seem like a regression to the list view of enumerations, but the complexity of indexing is not adversely affected since it already does a linear search on an initial segment of the set of parts. Also the interface in the previous section can be recovered by just applying (!!) to the list. We define a data type *Enumerate a* for enumerations containing values of type *a*.

```
data Enumerate a = Enumerate {parts :: [Finite a]}
```

In the previous section we simplified by supporting only infinite enumerations. Allowing finite enumerations is practically useful and gives an algorithmic speedups for many common applications. This gives the following simple definitions of empty and singleton enumerations:

```
empty :: Enumerate a
empty = Enumerate [ ]

singleton :: a → Enumerate a
singleton a = Enumerate [singletonF a]
```

Now we define an indexing function with bounds-checking:

```
index :: Enumerate a → Integer → a
index = index′ ∘ parts where
  index′ [ ]        i = error "index out of bounds"
  index′ (f : rest) i
     | i < cardF f  = f !!F i
     | otherwise    = index′ rest (i − cardF f)
```

This type is more useful for a propery-based testing driver (see §6) because it can detect with certainty if it has tested all values of the type.

Disjoint union Our enumeration type is a monoid under disjoint union. We use the infix operator (♢) = *mappend* (from the library Data.Monoid) for both the *Finite* and the *Enumerate* union.

```
instance Monoid (Enumerate a) where
  mempty  = empty
  mappend = union
union :: Enumerate a → Enumerate a → Enumerate a
union a b = Enumerate $ zipPlus (♢) (parts a) (parts b)
  where
    zipPlus :: (a → a → a) → [a] → [a] → [a]
    zipPlus f  (x : xs) (y : ys) = f x y : zipPlus f xs ys
    zipPlus _ xs        ys       = xs ++ ys
```

It is up to the user to ensure that the operands are really disjoint. If they are not then the resulting enumeration may contain repeated values. For example *pure True* ♢ *pure True* type checks and runs but it is probably not what the programmer intended. If we replace one of the *True*s with *False* we get a perfectly reasonable enumeration of *Bool*.

Cartesian product and bijective functions First we define a *Functor* instance for *Enumerate* in a straightforward fashion:

```
instance Functor Enumerate where
  fmap f e = Enumerate (fmap (fmap f) (parts e))
```

An important caveat is that the function mapped over the enumeration must be *bijective* in the same sense as for *biMap*, otherwise the resulting enumeration may contain duplicates.

Just as *Finite*, *Enumerate* is an applicative functor under product with singleton as the lifting operation.

```
instance Applicative Enumerate where
  pure    = singleton
  f ⟨∗⟩ a = fmap (uncurry ($)) (prod f a)
```

Similar to *fmap*, the first operand of ⟨∗⟩ must be an enumeration of bijective functions. Typically we get such an enumeration by lifting or partially applying a constructor function, e.g. if *e* has type *Enumerate a* then *f* = *pure* (,) ⟨∗⟩ *e* has type *Enumerate* (*b* → (*a*, *b*)) and *f* ⟨∗⟩ *e* has type *Enumerate* (*a*, *a*).

Two things complicate the computation of the product compared to its definition in §2. One is accounting for finite enumerations, the other is defining the convolution function on lists.

A first definition of *conv* (that computes the set of pairs of combined cost *p*) might look like this (with *mconcat* equivalent to *foldl* (⊕$_F$) *empty*$_F$):

```
badConv :: [Finite a] → [Finite b] → Int → Finite (a,b)
badConv xs ys p = mconcat
  (zipWith (⊗F) (take p xs) (reverse (take p ys)))
```

The problem with this implementation is memory. Specifically it needs to retain the result of all multiplications performed by (⊗$_F$) which yields quadratic memory use for each product in an enumeration.

Instead we want to perform the multiplications each time the indexing function is executed and just retain pointers to e_1 and e_2. The problem then is the reversal. With partitions as functions it is trivial to iterate an inital segment of the partition in reverse order, but with lists it is rather inefficient and we do not want to reverse a linearly sized list every time we index into a product. To avoid this we define a function that returns all reversals of a given list. We then define a product funtion that takes the parts of the first operand and all reversals of the parts of the second operand.

```
reversals :: [a] → [[a]]
reversals = go [] where
  go _   []     = []
  go rev (x:xs) = let rev′ = x:rev
                  in  rev′ : go rev′ xs
```

```
prod :: Enumerate a → Enumerate b → Enumerate (a,b)
prod e₁ e₂ = Enumerate $
  prod′ (parts e₁) (reversals (parts e₂))
prod′ :: [Finite a] → [[Finite b]] → [Finite (a,b)]
```

In any sensible Haskell implementation evaluating an inital segment of *reversals xs* uses linear memory in the length of the segment, and constructing the lists is done in linear time.

We define a version of *conv* where the second operand is already reversed, so it is simply a concatenation of a *zipWith*.

```
conv :: [Finite a] → [Finite b] → Finite (a,b)
conv xs ys = Finite
  (sum $ zipWith (∗) (map card_F xs) (map card_F ys))
  (λi → mconcat (zipWith (⊗_F) xs ys) !!_F i)
```

The worst case complexity of this function is the same as for the *conv* that reverses the list (linear in the list length). The best case complexity is constant however, since indexing into the result of *mconcat* is just a linear search. It might be tempting to move the *mconcat* out of the indexing function and use it directly to define the result of *conv*. This is semantically correct but the result of the multiplications are never garbage collected. Experiments show an increase in memory usage from a few megabytes to a few hundred megabytes in a realistic application.

For specifying *prod′* we can revert to dealing with only infinite enumerations i.e. assume *prod′* is only applied to "padded" lists:

```
parts = let rep = repeat empty_F in Enumerate $
  prod′ (parts e₁ ++ rep) (reversals (parts e₂ ++ rep))
```

Then we define *prod′* as:

```
prod′ xs rys = map (conv xs) rys
```

Analysing the behaviour of *prod* we notice that if e_2 is finite then we eventually start applying *conv xs* on the reversal of *parts* e_2 with a increasing chunk of $empty_F$ prepended. Analysing *conv* reveals that each such $empty_F$ corresponds to just dropping an element from the first operand (*xs*), since the head of the list is multiplied with $empty_F$. This suggest a strategy of computing *prod′* in two stages, the second used only if e_2 is finite:

```
prod′ xs@(_:xs′) (ys:yss) = goY ys yss where
  goY ry rys = conv xs ry : case rys of
    []          → goX ry  xs′
    (ry′:rys′)  → goY ry′ rys′
  goX ry = map (flip conv ry) ∘ tails
prod′ _ _                  = []
```

If any of the enumerations are empty the result is empty, otherwise we map over the reversals (in *goY*) with the twist that if the list is depleted we pass the final element (the reversal of all parts of e_2) to a new map (*goX*) that applies *conv* to this reversal and every suffix of *xs*. With a bit of analysis it is clear that this is semantically equivalent to the padded version (except that it produces a finite list if both operands are finite), but it is much more efficient if one or both the operands are finite. For instance the complexity of computing the cardinality at part *p* of a product is typically linear in *p*, but if one of the operands is finite it is *max p l* where *l* is the length of the part list of the finite operand (which is typically very small). The same complexity argument holds for indexing.

Assigning costs So far we are not assigning any costs to our enumerations, and we need the guarded recursion operator to complete the implementation:

```
pay :: Enumerate a → Enumerate a
pay e = Enumerate (empty_F : parts e)
```

To verify its correctness, consider that *parts* (*pay e*) !! 0 ≡ $empty_F$ and *parts* (*pay e*) !! $(p+1)$ ≡ *parts e* !! *p*. In other words, applying the list indexing function on the list of parts recovers the definition of *pay* in the previous section (except in the case of finite enumerations where padding is needed).

Examples Having defined all the building blocks we can start defining enumerations:

```
boolE :: Enumerate Bool
boolE = pay $ pure False ◊ pure True
blistE :: Enumerate [Bool]
blistE = pay $ pure []
             ◊ ((:) ⟨$⟩ boolE ⟨∗⟩ blistE)
```

A simple example shows what we have at this stage:

```
*Main> take 16 (map card_F $ parts blistE)
[0,1,0,2,0,4,0,8,0,16,0,32,0,64,0,128]
*Main> values_F (parts blistE !! 5)
[[False,False],[False,True],[True,False],[True,True]]
```

We can also very efficiently access values at extremely large indices:

```
*Main> length $ index blistE (10^1000)
3321
*Main> foldl1 xor $ index blistE (10^1000)
True
*Main> foldl1 xor $ index blistE (10^1001)
False
```

Computational complexity Analysing the complexity of indexing, we see that *union* adds a constant factor to the indexing function of each part, and it also adds one to the generic size of all values (since it can be considered an application of *Left* or *Right*). For product we choose between *p* different branches where *p* is the cost of the indexed value, and increase the generic size by one. This gives a pessimistic worst case complexity of $p * s$ where *s* is the generic size. If we do not apply *pay* directly to the result of another pay, then $p \leqslant s$ which gives s^2. This could be improved to *s log p* by using a binary search in the product case, but this also increases the memory consumption (see below).

The memory usage is (as always in a lazy language) difficult to measure exactly. Roughly speaking it is the product of the number of distinguished enumerations and the highest part to which these enumerations are evaluated. This number is equal to the sum of all constructor arities of the enumerated (monomorphic) types. For regular ADTs this is a constant, for non-regular ones it is bounded by a constant multiplied with the highest evaluated part.

Sharing As mentioned, Feat relies on memoisation and subsequently sharing for efficient indexing. To demonstrate this, we move to a more realistic implementation of the list enumerator which is parameterised over the underlying enumeration.

```
listE :: Enumerate a → Enumerate [a]
listE aS = pay $ pure []
               ◊ ((:) ⟨$⟩ aS ⟨∗⟩ listE aS)
blistE2 :: Enumerate [Bool]
blistE2 = listE boolE
```

This simple change causes the performance of *blistE2* to drop severely compared to *blistE*. The reason is that every evaluation of *listE2 aS* creates a separate enumeration, even though the argument to the function has been used previously. In the original we had *blistE* in the tail instead, which is a top level declaration. Any clever Haskell compiler evaluates such declarations at most once throughout the execution of a program (although it is technically not required by the Haskell language report). We can remedy the problem by manually sharing the result of the computation with a **let** binding (or equivalently by using a fix point combinator):

```
listE2 :: Enumerate a → Enumerate [a]
listE2 aS = let listE = pay $ pure [ ]
                        ◊ ((:) ⟨$⟩ aS ⟨∗⟩ listE)
            in listE
blistE3 :: Enumerate [Bool]
blistE3 = listE2 boolE
```

This is efficient again but it has one major problem, it requires the user to explicitly mark recursion. This is especially painful for mutually recursive data types since all members of a system of such types must be defined in the same scope:

```
data Tree a = Leaf a | Branch (Forest a)
newtype Forest a = Forest [Tree a]
treeE   = fst ∘ treesAndForests
forestE = snd ∘ treesAndForests
treesAndForests :: Enumerate a → (Enumerate (Tree a)
                                 ,Enumerate (Forest a))
treesAndForests eA =
  let eT = pay $ (Leaf ⟨$⟩ eA) ◊ (Branch ⟨$⟩ eF)
      eF = pay $ Forest ⟨$⟩ listE2 eT
  in (eT, eF)
```

Also there is still no sharing between different evaluations of *treeS* and *forestS* in other parts of the program. This forces everything into the same scope and crushes modularity. What we really want is a class of enumerable types with a single overloaded enumeration function.

```
class Enumerable a where
  enumerate :: Enumerate a
instance Enumerable Bool where
  enumerate = boolE
instance Enumerable a ⇒ Enumerable (Tree a) where
  enumerate = pay $ (Leaf   ⟨$⟩ enumerate)
                  ◊ (Branch ⟨$⟩ enumerate)
instance Enumerable a ⇒ Enumerable [a] where
  enumerate = listE2 enumerate
instance Enumerable a ⇒ Enumerable (Forest a) where
  enumerate = pay $ Forest ⟨$⟩ enumerate
```

This solution performs well and it's modular. The only potential problem is that there is no guarantee of *enumerate* being evaluated at most once for each monomorphic type. We write potential problem because it is difficult to determine if this is a problem in practice. It is possible to provoke GHC into reevaluating instance members, and even if GHC mostly does what we want other compilers might not. In the next section we discuss a solution that guarantees sharing of instance members.

4. Instance sharing

Our implementation relies on memoisation for efficient calculation of cardinalities. This in turn relies on sharing; specifically we want to share the instance methods of a type class. For instance we may have:

```
instance Enumerable a ⇒ Enumerable [a] where
  enumerate = pay $ pure [ ]
                  ◊ ((:) ⟨$⟩ enumerate ⟨∗⟩ enumerate)
```

The typical way of implementing Haskell type classes is using dictionaries, this essentially translates the instance above into a function similar to *enumerableList* :: *Enumerate a* → *Enumerate* [*a*]. Determining exactly when GHC or other compilers recompute the result of this function requires significant insight into the workings of the compiler and its runtime system. Suffice it to say that when re-evaluation does occur it has a significant negative impact on the performance of Feat. In this section we present a practical solution to this problem.

A monad for type-based sharing The general formulation of this problem is that we have a value $x :: C\ a \Rightarrow f\ a$, and for each monomorphic type T we want $x :: f\ T$ to be shared, i.e. to be evaluated at most once. The most direct solution to this problem seems to be a map from types to values i.e. *Bool* is mapped to $x :: f\ Bool$ and $()$ to $x :: f\ ()$. The map can then either be threaded through a computation using a state monad and updated as new types are discovered or updated with unsafe IO operations (with careful consideration of safety). We have chosen the former approach here.

The map must be dynamic, i.e. capable of storing values of different types (but we still want a type safe interface). We also need representations of Haskell types that can be used as keys. Both these features are provided by the *Typeable* class.

We define a data structure we call a dynamic map as an (abstract) data type providing type safe insertion and lookup. The type signatures of *dynInsert* and *dynLookup* are the significant part of the code, but the full implementation is provided for completeness.

```
import Data.Dynamic (Dynamic, fromDynamic, toDyn)
import Data.Typeable (Typeable, TypeRep, typeOf)
import Data.Map as M
newtype DynMap = DynMap (M.Map TypeRep Dynamic)
  deriving Show
dynEmpty :: DynMap
dynEmpty = DynMap M.empty
dynInsert :: Typeable a ⇒ a → DynMap → DynMap
dynInsert a (DynMap m) =
  DynMap (M.insert (typeOf a) (toDyn a) m)
```

To associate a value with a type we just map its type representation to the dynamic (type casted) value.

```
dynLookup :: Typeable a ⇒ DynMap → Maybe a
dynLookup (DynMap m) = hlp run ⊥ where
  hlp :: Typeable a ⇒
     (TypeRep → Maybe a) → a → Maybe a
  hlp f a = f (typeOf a)
  run tr = M.lookup tr m >>= fromDynamic
```

Lookup is also easily defined. The dynamic library provides a function *fromDynamic* :: *Dynamic* → *Maybe a*. In our case the *M.lookup* function has already matched the type representation against a type stored in the map, so *fromDynamic* is guaranteed to succeed (as long as values are only added using the *insert* function).

Using this map type we define a sharing monad with a function *share* that binds a value to its type.

```
type Sharing a = State DynMap a
runSharing :: Sharing a → a
runSharing m = evalState m dynEmpty
```

```
share :: Typeable a ⇒ Sharing a → Sharing a
share m = do
  mx ← gets dynLookup
  case mx of
    Just e    → return e
    Nothing → mfix $ λe → do
      modify (dynInsert e)
      m
```

Note that we require a monadic fixpoint combinator to ensure that recursive computations are shared. If it had not been used (i.e. if the *Nothing* case had been $m \gg= modify \circ dynInsert$) then any recursively defined *m* would eventually evaluate *share m* and enter the *Nothing* case. Using the fix point combinator ensures that a reference to the result of *m* is added to the map *before m* is computed. This makes any evaluations of *share m* inside *m* end up in the *Just* case which creates a cyclic reference in the value (exactly what we want for a recursive *m*). For example if we have $x = share\ (liftM\ pay\ x)$ the fixpoint combinator ensures that we get $runSharing\ x \equiv fix\ pay$ instead of $\bot$.

Self-optimising enumerations Now we have a monad for sharing and one way to proceed is to replace *Enumerate a* with *Sharing* (*Enumerate a*) and re-implement all the combinators for that type. We don't want to lose the simplicity of our current type though and it seems a very high price to pay for guaranteeing sharing which we are used to getting for free.

Our solution extends the enumeration type with a self-optimising routine, i.e. all enumerations have the same functionality as before but with the addition of an *optimiser* record field:

```
data Enumerate a = Enumerate
  {parts      :: [Finite a]
  , optimiser :: Sharing (Enumerate a)
  } deriving Typeable
```

The combinator for binding a type to an enumeration is called *eShare*.

```
eShare :: Typeable a ⇒ Enumerate a → Enumerate a
eShare e = e {optimiser = share (optimiser e)}
```

We can resolve the sharing using *optimise*.

```
optimise :: Enumerate a → Enumerate a
optimise e = let e′ = runSharing (optimiser e) in
  e′ {optimiser = return e′}
```

If *eShare* is used correctly, *optimise* is semantically equivalent to *id* but possibly with a higher degree of sharing. But using *eShare* directly is potentially harmful. It's possible to create "optimised" enumerations that differ semantically from the original. For instance $\lambda e \rightarrow eShare\ t\ e$ yields the same enumerator when applied to two different enumerators of the same type. As a general rule the enumeration passed to *eShare* should be a closed expression to avoid such problems. Luckily users of Feat never have to use *eShare*, instead we provide a safe interface that uses it internally.

An implication of the semantic changes that *eShare* may introduce is the possibility to replace the *Enumerable* instances for any type throughout another enumerator by simply inserting a value in the dynamic map before computing the optimised version. This could give unintuitive results if such enumerations are later combined with other enumerations. In our library we provide a simplified version of this feature where instances can be replaced but the resulting enumeration is optimised, which makes the replacement completely local and guarantees that *optimise* still preserves the semantics.

The next step is to implement sharing in all the combinators. This is simply a matter of lifting the operation to the optimised enumeration. Here are some examples where ... is the original definitions of *parts*.

```
fmap f e = e {...
  optimiser = fmap (fmap f) $ optimiser e}
f ⟨∗⟩ a = Enumerate {...
  optimiser = liftM2 (⟨∗⟩) (optimal f) (optimiser a)}
pure a = Enumerate {...
  optimiser = return (pure a)}
```

The only noticeable cost of using *eShare* is the reliance on *Typeable*. Since almost every instance should use *eShare* and consequently require type parameters to be *Typeable* and since *Typeable* can be derived by GHC, we chose to have it as a superclass and implement a default sharing mechanism with *eShare*.

```
class Typeable a ⇒ Enumerable a where
  enumerate :: Enumerate a
shared :: Enumerable a ⇒ Enumerate a
shared = eShare enumerate
optimal :: Enumerable a ⇒ Enumerate a
optimal = optimise shared
```

The idiom is that *enumerate* is used to define instances and *shared* is used to combine them. Finally *optimal* is used by libraries to access the contents of the enumeration (see §6).

Non-regular enumerations The sharing monad works very well for enumerations of regular types, where there is a closed system of shared enumerations. For non-regular enumerations (where the number of enumerations is unbounded) the monadic computation may fail to terminate. In these (rare) cases the programmer must ensure termination.

Free pairs and boilerplate instances There are several ways to increase the sharing further, thus reducing memory consumption. Particularly we want to share the cardinality computation of every sequenced application (⟨∗⟩). To do this we introduce the *FreePair* data type which is just like a pair except constructing one carries no cost i.e. the cost of the pair is equal to the total costs of its components.

```
data FreePair a b = FreePair a b
  deriving (Show, Typeable)
instance (Enumerable a, Enumerable b) ⇒
         Enumerable (FreePair a b) where
  enumerate = FreePair ⟨$⟩ shared ⟨∗⟩ shared
```

Since the size of *FreePair a b* is equal to the sum of the sizes of *a* and *b*, we know that for these functions:

```
f :: a → b → c
g :: FreePair a b → c
g (FreePair a b) = f a b
```

We have *f* ⟨$⟩ *shared* ⟨∗⟩ *shared* isomorphic to *g* ⟨$⟩ *shared* but in the latter case the product of the enumerations for *a* and *b* are always shared with other enumerations that require it (because *shared* :: *FreePair a b is* always shared. In other words *deep uncurrying* functions before applying them to *shared* often improve the performance of the resulting enumeration. For this purpose we define a function which is equivalent to *uncurry* from the Prelude but that operates on *FreePair*.

```
funcurry :: (a → b → c) → FreePair a b → c
funcurry f (FreePair a b) = f a b
```

Now in order to make an enumeration for a data constructor we need one more function:

```
unary :: Enumerable a ⇒ (a → b) → Enumerate b
unary f = f ⟨$⟩ shared
```

Together with *pure* for nullary constructors, *unary* and *funcurry* can be used to map any data constructor to an enumeration. For instance *pure* [ ] and *unary* (*funcurry* (:)) are enumerations for the constructors of [*a*]. In order to build a new instance we still need to combine the enumerations for all constructors and *pay* a suitable cost. Since *pay* is distributive over ◊, we can pay once for the whole type:

```
consts :: [Enumerate a] → Enumerate a
consts xs = pay $ foldl (◊) mempty xs
```

This gives the following instance for lists:

```
instance Enumerable a ⇒ Enumerable [a] where
  enumerate = consts [pure [ ], unary (funcurry (:))]
```

5. Invariants

Data type invariants are a major challenge in property-based testing. An invariant is just a property on a data type, one often wants to test that it holds for the result of a function. But we also want to test other properties only with input that is known to satisfy the invariant. In random testing this can sometimes be achieved by filtering: discarding the test cases that do not satisfy the invariant and generating new ones instead, but if the invariant is an arbitrary boolean predicate finding test data that satisfies the invariant can be as difficult as finding a bug. For systematic testing (with SmallCheck or Feat) this method is slightly more feasible since we do not repeat values which guarantees progress, but filtering is still a brute force solution.

In QuickCheck programmers can manually define custom test data generators that guarantee any invariant, but it may require a significant programmer effort and analysing the resulting generator to ensure correctness and statistical coverage can be difficult. Introducing this kind of complexity into testing code is hazardous since complex usually means error prone.

In Feat the room for customised generators is weaker (corresponding to the difference between monads and applicative functors). In theory it is possible to express any invariant by providing a bijection from a Haskell data type to the set of values that satisfy the invariant (since functional enumerations are closed under bijective function application). In practice the performance of the bijection needs to be considered because it directly affects the performance of indexing.

A simple and very common example of an invariant is the non-empty list. The function *uncurry* (:) is a bijection into non-empty lists of *a* from the type (*a*, [*a*]). The preferred way of dealing with these invariants in Feat is by defining a **newtype** for each restricted type, and a *smart constructor* which is the previously mentioned bijection and export it instead of the data constructor.

```
newtype NonEmpty a = MkNonEmpty {nonEmpty :: [a]}
  deriving Typeable
mkNonEmpty :: a → [a] → NonEmpty a
mkNonEmpty x xs = MkNonEmpty (x : xs)
instance Enumerable a ⇒ Enumerable (NonEmpty a) where
  enumerate = consts [unary (funcurry mkNonEmpty)]
```

To use this in an instance declaration, we only need the *nonEmpty* record function. In this example we look at the instance for the data type *Type* from the Template Haskell abstract syntax tree which describes the syntax of (extended) Haskell types. Consider the constructor for universal quantification:

```
ForallT :: [TyVarBndr] → Cxt → Type → Type
```

This constructor must not be applied to the empty list. We use *nonEmpty* to ensure this:

```
instance Enumerable Type where
  enumerate = consts [...
    ,funcurry $ funcurry $ ForallT ∘ nonEmpty]
```

Here *ForallT* ∘ *nonEmpty* has type:

```
NonEmpty TyVarBndr → Cxt → Type → Type
```

The only addition from the unrestricted enumeration is ∘ *nonEmpty*.

Enumerating Sets of natural numbers Another fairly common invariant is sorted lists of unique elements i.e. Sets. It is not obvious that sets can be built from our basic combinators. We can however define a bijection from lists of natural numbers to sets of natural numbers: *scanl* (((+) ∘ (+1)). For example the list [0,0,0] represents the set [0,1,2], the list [1,1,0] represents [1,3,4] and so on. We can define an enumerator for natural numbers using a bijection from *Integer*.

```
newtype Nat = Nat {nat :: Integer}
  deriving (Show, Typeable, Eq, Ord)
mkNat :: Integer → Nat
mkNat a = Nat $ abs $ a ∗ 2 − if a > 0 then 1 else 0
instance Enumerable Nat where
  enumerate = unary mkNat
```

Then we define sets of naturals:

```
newtype NatSet = MkNatSet {natSet :: [Integer]}
  deriving Typeable
mkNatSet :: [Nat] → NatSet
mkNatSet = MkNatSet ∘ scanl1 ((+) ∘ (+1)) ∘ map nat
```

Generalising to sets of arbitrary types Sets of naturals are useful but what we really want is a data type *Set a* = *MkSet* {*set* :: [*a*]} and a bijection to this type from something which we can already enumerate. Since we just defined an enumeration for sets of naturals, an efficient bijective mapping from natural numbers to *a* is all we need. Since this is the definition of a functional enumeration, we appear to be in luck.

```
mkSet :: Enumerate a → NatSet → Set a
mkSet e = MkSet ∘ map (index e) ∘ natSet
instance Enumerable a ⇒ Enumerable (Set a) where
  enumerate = unary (mkSet optimal)
```

This implementation works but it's slightly simplified, it doesn't use the cardinalities of *a* when determining the indices to use. This distorts the cost of our sets away from the actual size of the values.

6. Accessing enumerated values

This section discusses strategies for accessing the values of enumerations, especially for the purpose of property-based testing. The simplest function *values* is simply all values in the enumeration partitioned by size. We include the cardinalities as well, this is often useful e.g. to report to the user how many values are in a part before initiating testing on values. For this reason we give *values* type *Enumerate a* → [(*Integer*, [*a*])].

Given that Feat is intended to be used primarily with the type class *Enumerable* we have implemented the library functions to use class members, but provide non-class versions of the functions that have the suffix *With*:

```
type EnumL a = [(Integer, [a])]
values :: Enumerable a ⇒ [(Integer, [a])]
values = valuesWith optimal
valuesWith :: Enumerate a → [(Integer, [a])]
valuesWith = map (λf → (cardF f, valuesF f)) ∘ parts
```

Parallel enumeration A generalisation of *values* is possible since we can "skip" an arbitrary number of steps into the enumeration at any point. The function *striped* takes a starting index and a step size *n* and enumerates every n^{th} value after the initial index in the ordering. As a special case *values* = *striped* 0 0 1. One purpose of this function is to enumerate in parallel. If *n* processes execute *uncurry striped k n* where *k* is a process-unique id in the range $[0\,..\,n-1]$ then all values are eventually evaluated by some process and, even though the processes are not communicating, the work is evenly distributed in terms of number and size of test cases.

```
stripedWith :: Enumerate a → Index → Integer → EnumL a
stripedWith e o0 step = stripedWith′ (parts e) o0 where
  stripedWith′ (Finite crd ix : ps) o =
    (max 0 d, thisP) : stripedWith′ ps o′
    where
      o′      = if space ⩽ 0 then o − crd else step − m − 1
      thisP   = map ix (genericTake d $ iterate (+step) o)
      space   = crd − o
      (d, m)  = divMod space step
```

Bounded enumeration Another feature afforded by random-access indexing is the ability to systematically select manageable portions of gigantic parts. Specifically we can devise a function *bounded* :: *Integer* → *EnumL a* such that each list in *bounded n* contains at most *n* elements. If there are more than *n* elements in a part we systematically sample *n* values that are evenly spaced across the part.

```
samplePart :: Integer → Finite a → (Integer, [a])
samplePart m (Finite crd ix) =
   let step  = crd % m
   in if crd ⩽ m
      then (crd, map ix [0 .. crd − 1])
      else  (m,  map ix [ round (k ∗ step)
                        | k ← map toRational [0 .. m − 1]])
boundedWith :: Enumerate a → Integer → EnumL a
boundedWith e n = map (samplePart n) $ parts e
```

Random sampling A noticeable feature of Feat is that it provides random sampling with uniform distribution over a size-bounded subset of a type. This is not just nice for compatibility with *QuickCheck*, it is genuinely difficult to write a uniform generator even for simple recursive types with the tools provided by the QuickCheck library.

The function *uniform* :: *Enumerable a* ⇒ *Part* → *Gen a* generates values of the given size or smaller.

```
uniformWith :: Enumerate a → Int → Gen a
uniformWith = uni ∘ parts where
  uni :: [Finite a] → Int → Gen a
  uni [] _     = error "uniform: empty enumeration"
  uni ps maxp =  let (incl, rest) = splitAt maxp ps
                     fin          = mconcat incl
     in case cardF fin of
          0 → uni rest 1
          _ → do i ← choose (0, cardF fin − 1)
                 return (fin !!F i)
```

```
*Main> sample (sized uniform :: Gen [[Bool]])
[]
[[]]
[[], []]
[[True]]
[[False], [], []]
[[], [False, False, True]]
[[False, True, False, True, True]]
[[False], [], [], []]
[[True], [True], [], [False, True]]
[[False], [False, True, False, False, True]]
```

Table 2. Randomly chosen values from the enumeration of [*Bool*]

```
data Exp    = CaseE Exp [Match] | ...
data Match  = Match Pat Body [Dec]
data Body   = NormalB Exp | ...
data Dec    = FunD Name [Clause] | ...
data Clause = Clause [Pat] Body [Dec]
data Pat    = ViewP Exp Pat | ...
```

Table 3. Parts of the Template Haskell AST type. Note that all the types are mutually recursive.

Since we do not make any local random choices, performance is favourable compared to hand written generators. The typical usage is *sized uniform*, which generates values bounded by the QuickCheck size parameter. In Table 2 we present a typical output of applying the function *sample* from the QuickCheck library to the uniform generator for [[*Bool*]]. The function drafts values from the generator using increasing sizes from 0 to 20.

7. Case study: Enumerating the ASTs of Haskell

As a case study, we use the enumeration technique developed in this paper to generate values of Haskell ASTs, specifically the abstract syntax of Template Haskell, taken from the module Language.Haskell.TH.Syntax.

We use the generated ASTs to test the Template Haskell pretty-printer. The background is that in working with *BNFC-meta* [Duregård and Jansson 2011], which relies heavily on meta programming, we noticed that the TH pretty printer occasionally produced un-parseable output. BNFC-meta also relies on the more experimental package *haskell-src-meta* that forms a bridge between the *haskell-src-exts* parser and Template Haskell. We wanted to test this tool chain on a system-level.

The AST types We limited ourselves to testing expressions, following dependencies and adding a few **newtype** wrappers this yielded a system of almost 30 data types with 80+ constructors. A small part is shown in Table 3.

We excluded a few non-standard extensions (e.g. bang patterns) because the specification for these are not as clear (especially the interactions between different Haskell extensions).

Comparison to existing test frameworks We wanted to compare Feat to existing test frameworks. For a set of mutual-recursive datatypes of this size, it is very difficult to write a sensible QuickCheck generator. We therefore excluded QuickCheck from the case study.

On the other hand, generators for SmallCheck and Feat are largely boilerplate code. To avoid having the results skewed by trying to generate the large set of strings for names (and to avoid using GHC-internal names which are not printable), we fix the name space and regard any name as having size 1. But we do generate characters and strings as literals (and found bugs in these).

Test case distribution The result shows some interesting differences between Feat and SmallCheck on the distribution of the generated values. We count the number of values of each part (depth for SmallCheck and size for Feat) of each generator.

Size	1	2	3	4	5	6	20
SmallCheck	1	9	951	×	×	×	×
Feat	0	1	5	11	20	49	65072965

Table 4. The number of test cases below certain size

It is clear that for big datatypes such as ASTs, SmallCheck quickly hits a wall: the number of values below a fixed size grows aggressively, and we are not able to complete the enumeration of size 4 (given several hours of execution time). In the case of Feat, the growth in the number of values in each category is more controlled, due to its more refined definition of size. If we look more closely into the values generated by SmallCheck by sampling the first 10000 values of the series on depth 4. A count reveals that the maximum size in this sample is 35, with more than 50% of the values having a size more than 20. Thus, contrary to the goal of generating small values, SmallCheck is actually generating pretty large values from early on.

Testing the TH PrettyPrinter The generated AST values are used as test cases to find bugs in Template Haskell's prettyprinter (Language.Haskell.TH.Ppr). We start with a simple property: a pretty-printed expression should be syntactically valid Haskell. We use *haskell-src-exts* as a test oracle:

```
prop_parses e =
  case parse $ pprint (e :: Exp) :: ParseResult Exp of
    ParseOk _ → True
    ParseFailed _ s → False
```

After a quick run, Feat reports numerous bugs, some of which are no doubt false positives. A small example of a confirmed bug is expression $[Con..]$. The correct syntax has a space after the constructor name (i.e. $[Con\ ..]$). As we can see, this counter example is rather small (having size 6 and depth 4). However, after hours of testing SmallCheck is not able to find this bug even though many much larger (but not deeper) values are tested. Given a very large search space that is not exhaustible, SmallCheck tends to get stuck in a corner of the space and test large but similar values. The primary cause of SmallCheck's inability to deal with ASTs is that the definition of "small" as "shallowly nested" means that there are very many small values but many types can practically not be reached at all. For instance generating any *Exp* with a where-clause seems to require at least depth 8, which is far out of reach.

Comparatively, the behaviour of Feat is much better. It advances quickly to cover a wider range of small values, which maximises the chance of finding a bug. The guarantee "correct for all inputs with 15 constructors or less" is much stronger than "correct for all values of at most depth 3 and a few million of depth 4". When there is no bug reported, Feat reports a more meaningful portion of the search space that has been tested.

It is worth mentioning that SmallCheck has the facility of performing "depth-adjustment", that allows manual increment of the depth count of individual constructors to reduce the number of values in each category. For example, instead counting all constructors as 1, one may choose to count a binary constructor as having depth 2 to reflect the fact that it may create a larger value than a unary one (similar to our *pay* function). In our opinion, this adjustment is a step towards an imprecise approximation of size as used in our approach. Even if we put time into manually adjusting the depth it is unclear what kind of guarantee testing up to depth 8 implies, especially when the definition of depth has been altered away from generic depth.

Testing round trip properties We also tested an extension of this property that does not only test the syntactic correctness but also that the information in the AST is preserved when pretty printing. We tested this by making a round trip function that pretty prints the AST, parses it with *haskell-src-exts* and converts it back to Template Haskell AST with *haskell-src-meta*. This way we could test this tool chain on a system level finding bugs in *haskell-src-meta* as well as the pretty printer. The minimal example of a pretty printer error found was *StringL* `"\n"` which is pretty printed to `""`, discarding the newline character. This error was not found by SmallCheck partly because it is too deep (at least depth 4 depending on the character generator), and partly because the default character generator only tests alphabetical characters. Presumably an experienced SmallCheck tester would use a **newtype** to generate more sensible string literals.

8. Related Work

SmallCheck, Lazy SmallCheck and QuickCheck Our work is heavily influenced by the property based testing frameworks QuickCheck [Claessen and Hughes 2000] and SmallCheck [Runciman et al. 2008]. The similarity is greatest with SmallCheck and we improve upon it in two distinct ways:

- (Almost) Random access times to enumerated values. This presents a number of possibilities that are not present in SmallCheck, including random or systematic sampling of large values (too large to exhaustively enumerate) and overhead-free parallelism.
- A definition of size which is closer to the actual size. Especially for testing abstract syntax tree types and other "wide" types this seems to be a very important feature (see §7).

Since our library provides random generation as an alternative or complement to exhaustive enumeration it can be considered a "best of two worlds" link between SmallCheck and QuickCheck. We provide a QuickCheck compatible generator which should ease the reuse of existing properties.

SmallCheck systematically tests by enumerating all values bounded by depth of constructor nestings. In a sense this is also a partitioning by size. The major problem with SmallCheck is that the number of values in each partition grow too quickly often hitting a wall after a few levels of depth. For AST's this is doubly true (the growth is proportional to the number of constructors in the type, and it's unlikely you can ever test beyond depth 4 or so. This means that most constructors in an AST are never touched.

Lazy SmallCheck can cut the number of tests on each depth level by using the inherent laziness of Haskell. It can detect if a part of the tested value is evaluated by the property and if it is not it refrains from refining this value further. In some cases this can lead to an exponential decrease of the number of required test cases. In the case of testing a pretty printer (as we do in §7) Lazy SmallCheck would offer no advantage since the property fully evaluates its argument every time.

After the submission of this paper, a package named GenCheck is uploaded to Hackage [Uszkay and Carette 2012]. GenCheck is designed to generalise both QuickCheck and SmallCheck, which is similar to Feat in goal. This initial release has very limited documentation, which prevents a more comprehensive comparison at the moment.

EasyCheck In the functional logic programming language Curry [Hanus et al. 2006], one form of enumeration of values comes for free in the form of a search tree. As a result, testing tools such as EasyCheck [Christiansen and Fischer 2008] only need to focus on the traversal strategy for test case generation. It is argued in [Christiansen and Fischer 2008] that this separation of the enumeration scheme and the test case generation algorithm is particularly beneficial in supporting flexible testing strategies.

Feat's functional enumeration, with its ability to exhaustively enumerate finite values, and to randomly sample very large values, lays an excellent groundwork for supporting various test case generation algorithms. One can easily select test cases of different sizes with a desired distribution.

AGATA AGATA [Duregård 2009] is the previous work of Jonas Duregård. Although it is based entirely on random testing it is a predecessor of Feat in the sense that it attempts to solve the problem of testing syntactic properties of abstract syntax trees. It is our opinion that Feat subsumes AGATA in this and every other aspect.

Generating (Typed) Lambda Terms To test more aspects of a compiler other than the libraries that perform syntax manipulation, it is more desirable to generate terms that are type correct.

In [Yakushev and Jeuring 2009], well-typed terms are enumerated according to their costs—a concept similar to our notion of size. Similar to SmallCheck, the enumeration in [Yakushev and Jeuring 2009] adopts the list view, which prohibits the sampling of large values. On the other hand, the special-purpose QuickCheck generator designed in [Pałka et al. 2011], randomly generates well-typed terms. Unsurprisingly, it has no problem with constructing individual large terms, but falls short in systematicness.

It is shown [Wang 2005] that well-scoped (but not necessarily well-typed) lambda terms can be uniformly generated. The technique used in [Wang 2005] is very similar to ours, in the sense that the number of possible terms for each syntactic constructs are counted (with memoization) to guide the random generation for a uniform distribution. This work can be seen as a special case of Feat, and Feat can indeed be straightforwardly instrumented to generate well-scoped lambda terms.

Feat is at present not able to express complicated invariants such as type correctness of the enumerated terms. One potential solution is to adopt more advanced type systems as in [Yakushev and Jeuring 2009], so that the type of the enumeration captures more precisely its intended range.

Combinatorial species In mathematics a *combinatorial species* is an endo-functor on the category of finite sets and bijections. Each object A in this category can be described by its cardinality n and a finite enumeration of its elements: $f : \mathbb{N}_n \to A$. In other words, for each n there is a canoncial object (label set) $\mathbb{N}_n$. Each arrow $phi : A \to B$ in this category is between objects of the same cardinality n, and can be described by a permutation of the set $\mathbb{N}_n$. This means that the object action S_0 of an endofunctor S maps a pair (n,f) to a pair $S_0\ (n,f)$ whose first component is the cardinality of the resulting set (we call it *card n*). (The arrow action S_1 maps permutations on $\mathbb{N}_n$ to permutations on $\mathbb{N}_{card\ n}$.)

In the species library (decribed in [Yorgey 2010]) there is a method $enumerate : Enumerable\, f \Rightarrow [a] \to [f\ a]$ which takes a (list representation of) an object a to all f a-structures obtained by the S_0 map. The key to comparing this with our paper is to represent the objects as finite enumerations $\mathbb{N}_n \to a$ instead of as lists $[a]$. Then $enumerate' : Enumerable\, f \Rightarrow (\mathbb{N}_n \to a) \to (\mathbb{N}_{card\ n} \to f\ a)$. We can further let a be $\mathbb{N}_p$ and define $sel\ p = enumerate'\ id : \mathbb{N}_{card\ p} \to f\ \mathbb{N}_p$. The function *sel* is basically an inefficient version of the indexing function in the Feat library. The elements in the image of g for a particular n are (defined to be) those of weight n. The union of all those images form a set (a type). Thus a species is roughly a partition of a set into subsets of elements of the same size.

The theory of species goes further than what we present in this paper, and the species library implements quite a bit of that theory. We cannot (yet) handle non-regular species, but for the regular ones we can implement the enumeration efficiently.

Boltzmann samplers A combinatorial class is basically the same as what we call a "functional enumeration": a set C of combinatorial objects with a size function such that all the parts C_n of the induced partitioning are finite. A *Boltzmann model* is a probability distribution (parameterized over a small real number x) over such a class C, such that a uniform discrete probability distribution is used within each part C_n. A *Boltzmann sampler* is (in our terminology) a random generator of values in the class C following the Boltzmann model distribution. The datatype generic Bolztmann sampler defined in [Duchon et al. 2004] follows the same structure as our generic enumerator. We believe a closer study of that paper could help defining random generators for ASTs in a principled way from our enumerators.

Decomposable combinatorial structures. The research field of enumerative combinatorics has worked on what we call "functional enumeration" already in the early 1990:s and Flajolet and Salvy [1995] provide a short overview and a good entry point. They define a grammar for "decomposable" combinatorial structures including constructions for (disjoint) union, product, sequence, sets and cycles (atoms or symbols are the implicit base case). The theory (and implementation) is based on representing the counting sequences $\{C_i\}$ as generating functions as there is a close correspondance between the grammar constructs and algebraic operations on the generating functions. For decomposable structures they compute generating function *equations* and by embedding this in a computer algebra system (Maple) the equations can be symbolically manipulated and sometimes solved to obatin closed forms for the GFs. What they don't do is consider the pragmatic solution of just tabulating the counts instead (as we do). They also don't consider complex algebraic datatypes, just universal (untyped) representations of them. Complex ASTs can perhaps be expressed (or simulated) but rather awkwardly. They also don't seem to implement the index function into the enumeration (only random generation). Nevertheless, their development is impressive, both as a mathematical theory and as a computer library and we want to explore the connection further in future work.

9. Conclusions and Future work

Since there are now a few different approaches to property-based testing available for Haskell it would be useful with a library of properties to compare the efficiency of the libraries at finding bugs. The library could contain "tailored" properties that are constructed to exploit weaknesses or utilise strengths of known approaches, but it would be interesting to have naturally occurring bugs as well (preferably from production code). It could also be used to evaluate the paradigm of property-based testing as a whole.

Instance (dictionary) sharing Our solution to instance sharing is not perfect. It divides the interface into separate class functions for consuming and combining enumerations and it requires *Typeable*.

A solution based on stable names [Peyton Jones et al. 1999] would remove the *Typeable* constraint but it's not obvious that there is any stable name to hold on to (the stable point is actually the dictionary function, but that is off-limits to the programmer). Compiler support is always a possible solution (i.e. by a flag or a pragma), but should only be considered as a last resort.

Enumerating functions For completeness, Feat should support enumerating function values. We argue that in practice this is seldom useful for property-based testing because non trivial higher order functions often have some requirement on their function arguments, for instance the $*By$ functions in *Data.List* need functions that are total orderings, a parallel fold needs an associative function etc. This can not be checked as a precondition, the best bet is probably to supply a few manually written total orderings or possibly use a very clever QuickCheck generator.

Regardless of this, it stands to reason that *functional* enumerations should have support for functions. This is largely a question of finding a suitable definition of size for functions, or an efficient bijection from an algebraic type into the function type.

Invariants The primary reason why enumeration can not replace the less systematic approach of QuickCheck testing is invariants. QuickCheck can always be used to write a generator that satisfies an invariant, but often with no guarantees on the distribution or coverage of the generator.

The general understanding seems to be that it is not possible to use systematic testing and filtering to test functions that require e.g. type correct programs. Thus QuickCheck gives you something, while automatic enumeration gives you nothing. The reason is that the ratio type correct/syntactically correct programs is so small that finding valid non-trivial test cases is too time consuming.

It would be worthwhile to try and falsify or confirm the general understanding for instance by attempting to repeat the results of [Pałka et al. 2011] using systematic enumeration.

Invariants and costs We have seen any bijective function can be mapped on an enumeration, preserving the enumeration criterion. This also preserves the cost of values, in the sense that a value x in the enumeration $\mathit{fmap}\ f\ e$ costs as much as $f^{-1} x$.

This might not be the intention, particularly this means that a strong size guarantee (i.e. that the cost is equal to the number of constructors) is typically not preserved. As we show in §7 the definition of size can be essential in practice and the correlation between cost and the actual number of constructors in the value should be preserved as far as possible. There may be useful operations for manipulating costs of enumerations.

Conclusions We present an algebra of enumerations, an efficient implementation and show that it can handle large groups of mutually recursive datatypes. We see this as a step on the way to a unified theory of test data enumeration and generation. Feat is available as an open source package from the HackageDB repository:
`http://hackage.haskell.org/package/testing-feat`

Acknowledgements We thank Koen Claessen for helpful discussions on the implementation of Feat and Norman Ramsey for several helpful suggestions to improve presentation. We also thank the anonymous reviewers for several useful comments, especially suggestions for related work.

References

J. Christiansen and S. Fischer. Easycheck: test data for free. In *FLOPS'08*, pages 322–336. Springer, 2008.

K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP '00*, pages 268–279. ACM, 2000.

P. Duchon, P. Flajolet, G. Louchard, and G. Schaeffer. Boltzmann samplers for the random generation of combinatorial structures. *Combinatorics, Probability and Computing*, 13(4–5):577–625, 2004. doi: 10.1017/S0963548304006315.

J. Duregård. Agata: Random generation of test data. Master's thesis, Chalmers University of Technology, 2009.

J. Duregård and P. Jansson. Embedded parser generators. In *Haskell '11*, pages 107–117. ACM, 2011.

P. Flajolet and B. Salvy. Computer algebra libraries for combinatorial structures. *J. Symb. Comput.*, 20(5/6):653–671, 1995.

M. Hanus et al. *Curry: An Integrated Functional Logic Language*, version 0.8.2 edition, 2006. Available from `http://www.informatik.uni-kiel.de/~curry/report.html`.

M. H. Pałka, K. Claessen, A. Russo, and J. Hughes. Testing an optimising compiler by generating random lambda terms. In *AST '11*, pages 91–97. ACM, 2011.

S. Peyton Jones, S. Marlow, and C. Elliot. Stretching the storage manager: weak pointers and stable names in Haskell. In *IFL'99*, LNCS. Springer, 1999.

C. Runciman, M. Naylor, and F. Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Haskell '08*, pages 37–48. ACM, 2008.

G. J. Uszkay and J. Carette. Available from `http://hackage.haskell.org/package/gencheck`, 2012.

J. Wang. Generating random lambda calculus terms. Technical report, Boston University, 2005.

A. R. Yakushev and J. Jeuring. Enumerating well-typed terms generically. In *AAIP 2009*, volume 5812 of *LNCS*, pages 93–116. Springer, 2009.

B. A. Yorgey. Species and functors and types, oh my! In *Haskell '10*, pages 147–158. ACM, 2010.