Generating Constrained Random Data with Uniform Distribution

Generating Constrained Random Data with Uniform Distribution

Koen Claessen, Jonas Duregård, and Michał H. Pałka

Chalmers University of Technology
{koen,jonas.duregard,michal.palka}@chalmers.se

Abstract. We present a technique for automatically deriving test data generators from a predicate expressed as a Boolean function. The distribution of these generators is uniform over values of a given size. To make the generation efficient we rely on laziness of the predicate, allowing us to prune the space of values quickly. In contrast, implementing test data generators by hand is labour intensive and error prone. Moreover, handwritten generators often have an unpredictable distribution of values, risking that some values are arbitrarily underrepresented. We also present a variation of the technique where the distribution is skewed in a limited and predictable way, potentially increasing the performance. Experimental evaluation of the techniques shows that the uniform derived generators are much easier to define than hand-written ones, and their performance, while lower, is adequate for some realistic applications.

1 Introduction

Random property-based testing has proven to be an effective method for finding bugs in programs [1, 4]. Two ingredients are required for property-based testing: a *test data generator* and a *property* (sometimes called oracle). For each test, the test data generator generates input to the program under test, and the property checks whether or not the observed behaviour is acceptable. This paper focuses on the test data generators.

The popular random testing tool QuickCheck [4] provides a library for defining random generators for data types. Typically, a generator is a recursive function that at every recursion level chooses a random constructor of the relevant data type. Relative frequencies for the constructors can be specified by the programmer to control the distribution. An extra resource argument that shrinks at each recursive call is used to control the size of the generated test data and ensures termination.

The above method for test generation works well for generating structured, well-typed data. But it becomes much harder when our objective is to generate well-typed data *that satisfies an extra condition*. A motivating example is the random generation of programs as test data for testing compilers. In order to successfully test different phases of a compiler, programs not only need to be grammatically correct, they may also need to satisfy other properties such as all variables are bound, all expressions are well-typed, certain combinations of constructs do not occur in the programs, or a combination of such properties.

In previous work by some of the authors, it was shown to be possible but very tedious to manually construct a generator that (a) could generate random well-typed programs

```
data Expr                    check :: [Type] → Expr → Type → Bool
  = Ap Expr Expr Type        check env (Vr i)      t          = env !! i == t
  | Vr Int                   check env (Ap f x tx) t          =
  | Lm Expr                    check env f (tx :→ t) && check env x tx
data Type = A | B | C        check env (Lm e)      (ta :→ tb) = check (ta : env) e tb
          | Type :→ Type     check env _           _          = False
```

Fig. 1: Data type and type checker for simply-typed lambda calculus. The *Type* in the *Ap* nodes represents the type of the argument term.

in the polymorphic lambda-calculus, and at the same time (b) maintain a reasonable distribution such that no programs were arbitrarily excluded from generation.

The problem is that generators mix concerns that we would like to separate: (1) what is the structure of the test data, (2) which properties should it obey, and (3) what distribution do we want.

In this paper, we investigate solutions to the following problem: Given a definition of the structure of test data (a data type definition), and given one or more predicates (functions computing a boolean), can we automatically generate test data that satisfies all the predicates and at the same time has a predictable, good distribution?

To be more concrete, let us take a look at Fig. 1. Here, a data type for typed lambda expressions is defined, together with a function that given an environment, an expression, and a type, checks whether or not the expression has the stated type in the environment. From this input alone, we would like to be able to generate random well-typed expressions with a good distribution.

What does a 'good' distribution mean? First, we need to have a way to restrict the size of the generated test data. In any application, we are only ever going to generate a finite number of values, so we need a decision on what test data sizes to use. An easy and common way to control test data size is to control the *depth* of a term. This is for example done in SmallCheck [10]. The problem with using depth is that the cardinality of terms of a certain depth grows extremely fast as the depth increases. Moreover, good distributions for, to give an example, the set of trees of depth d are hard to find, because there are many more almost full trees of depth d than there are sparse trees of depth d, which may lead to an overrepresentation of almost full trees in randomly generated values.

Another possibility is to work with the set of values of a given *size* n, where size is understood as the number of data constructors in the term. Previous work by one of the authors on FEAT [5] has shown that it is possible to efficiently index in, and compute cardinalities of, sets of terms of a given size n. This is the choice we make in this paper.

The simplest useful and predictable distribution that does not arbitrarily exclude values from a set is the *uniform distribution*, which is why we chose to focus on uniform distributions in this paper. We acknowledge the need for other distributions than uniform in certain applications. However, we think that a uniform distribution is at least a useful building block in the process of crafting test data generators. We anticipate methods for

controlling the distribution of our generators in multiple ways, but that remains future work.

Our first main contribution in this paper is an algorithm that, given a data type definition, a predicate, and a test data size, generates random values satisfying the predicate, with a perfectly uniform distribution. It works by first computing the cardinality of the set of all values of the given size, and then randomly picking indices in this set, computing the values that correspond to those indices, until we find a value for which the predicate is true. The key feature of the algorithm is that every time a value x is found for which the predicate is false, it is removed from the set of values, together with all other values that would have lead to the predicate returning false using the same execution path as x.

Unfortunately, even with this optimisation, uniformity turns out to be a very costly property in many practical cases. We have also developed a backtracking-based generator that is more efficient, but has no guarantees on the distribution. Our second main contribution is a hybrid generator that combines the uniform algorithm and the backtracking algorithm, and is 'almost uniform' in a precise and predictable way.

2 Generating Values of Algebraic Datatypes

In this section we explain how to generate random values of an algebraic data type (ADT) uniformly. Our approach is based on a representation of sets of values that allows efficient *indexing*, inspired by FEAT [5], which is used to map random indices to random values. In the next section we modify this procedure to efficiently search for values that satisfy a predicate.

Algebraic Data Types (ADTs) are constructed using units (atomic values), disjoint unions of data types, products of data types, and may refer to their own definitions recursively. For instance, consider these definitions of Haskell data types for natural numbers and lists of natural numbers:

data *Nat* = *Z* | *Suc Nat*
data *ListNat* = *Nill* | *Cons Nat ListNat*

In general, ADTs may contain an infinite number of values, which is the case for both data types above. Our approach for generating random values of an ADT uniformly is to generate values of a specific *size*, understood as the number of constructors used in a value. For example, all of *Cons* (*Suc* (*Suc Z*)) (*Cons Z Nill*), *Cons* (*Suc Z*) (*Cons* (*Suc Z*) *Nill*) and *Cons Z* (*Cons Z* (*Cons Z Nill*)) are values of size 7. As there is only a finite number of values of each size, we can create a sampling procedure that generates a uniformly random value of *ListNat* of a given size.

2.1 Indexing

Our method for generating random values of an ADT is based on an *indexing* function, which maps integers to corresponding data type values of a given size.

$$\text{index}_{S,k} : \{i \in \mathbb{N} \mid i < |S_k|\} \to S_k$$

Here, S is the data type, and S_k is the set of k-sized values of S. The intuitive idea behind efficient indexing is to quickly calculate *cardinalities* of subsets of the indexed set. For example, when $S = T \oplus U$ is a sum type, then indexing is performed as follows:

$$\text{index}_{T\oplus U,k}(i) = \begin{cases} \text{index}_{T,k}(i) & \text{if } i < |T_k| \\ \text{index}_{U,k}(i - |T_k|) & \text{otherwise} \end{cases}$$

When $S = T \otimes U$ is a product type, we need to consider all ways size k can be divided between the components of the product. The cardinality of the product can be computed as follows:

$$|(T \otimes U)_k| = \sum_{k_1+k_2=k} |T_{k_1}||U_{k_2}|$$

When indexing $(T \otimes U)_k$ using index i, we first select the division of size $k_1 + k_2 = k$, such that:

$$0 \le i' < |T_{k_1}||U_{k_2}| \quad \text{where} \quad i' = i - \sum_{\substack{l_1<k_1 \\ l_1+l_2=k}} |T_{l_1}||U_{l_2}|$$

Then, elements of T_{k_1} and U_{k_2} are selected using the remaining part of the index i'.

$$\text{index}_{T\otimes U,k}(i) = (\text{index}_{T,k}(i' \text{ div } |U_{k_2}|), \text{index}_{U,k}(i' \bmod |U_{k_2}|))$$

In the rest of this section, we outline how to implement indexing in Haskell.

2.2 Representation of Spaces

We define a Haskell Generalized Algebraic Data Type (GADT) *Space* to represent ADTs, and allow efficient cardinality computations and indexing.

```
data Space a where
  Empty :: Space a
  Pure  :: a          -> Space a
  (:+:) :: Space a -> Space a -> Space a
  (:*:) :: Space a -> Space b -> Space (a,b)
  Pay   :: Space a -> Space a
  (:$:) :: (a -> b) -> Space a -> Space b
```

Spaces can be built using four basic operations: *Empty* for empty space, *Pure* for unit space, $(:+:)$ for a sum of two spaces and $(:*:)$ for a product. Spaces also have an operator *Pay* which represents a unit cost imposed by using a constructor. The last operation $(:\$:)$, applies a function to all values in the space. We assume that spaces are constructed in such a way that all their elements are unique. If this is not the case, a 'uniform' sampling procedure would return repeated elements more often than unique ones.

A very convenient operator on spaces is the lifted application operator, that takes a space of functions and a space of parameters and produces a space of all applications of the functions to the parameters:

$$(<*>) :: Space\ (a \to b) \to Space\ a \to Space\ b$$
$$s_1 <*> s_2 = (\lambda(f,a) \to f\ a)\ :\$:\ (s_1\ :*:\ s_2)$$

With the operators defined above, the definition of spaces mirror the definitions of data types. For example, spaces for the *Nat* and *ListNat* data types can be defined as follows:

$$spaceNat :: Space\ Nat$$
$$spaceNat = Pay\ (Pure\ Z\ :+:\ (Suc\ :\$:\ spaceNat))$$
$$spaceListNat :: Space\ ListNat$$
$$spaceListNat = Pay\ (Pure\ Nill\ :+:\ (Cons\ :\$:\ spaceNat <*> spaceListNat))$$

Unit constructors are represented with *Pure*, whereas compound constructors are mapped on the subspaces of the values they contain. In this example, *Pay* is applied each time we introduce a constructor, which makes the size of values equal to number of constructors they contain, and is the usual practice. However, the user may choose to use another way of assigning costs, which would change the sizes of individual values and, as a result, the distribution of the generated values. The only rule that must be followed when assigning costs is that all recursion is guarded by at least one *Pay* operation, otherwise the sets of values of a given size might be infinite, which would lead to non-terminating cardinality computations.

2.3 Indexing on Spaces

Indexing on spaces can be reduced to two subproblems: Extracting the finite set of values of a particular set, and indexing into such finite sets. Assume we have some data type for finite sets constructed by combining the empty set ($\{\}$), singleton sets ($\{a\}$), disjoint union ($\uplus$) and Cartesian product ($\times$). From the definition of such a finite set, its cardinality can be computed as follows:

$$|\{\}| = 0 \qquad |a \times b| = |a| * |b|$$
$$|\{a\}| = 1 \qquad |a \uplus b| = |a| + |b|$$

Using this function it is possible to define an indexing function on the type:

$$indexFin\ \{a\}\ \ 0 = a$$
$$indexFin\ (a \uplus b)\ i \mid i < |a| = indexFin\ a\ i$$
$$indexFin\ (a \uplus b)\ i \mid i \geqslant |a| = indexFin\ b\ (i - |a|)$$
$$indexFin\ (a \times b)\ i = (indexFin\ a\ (i \div |b|), indexFin\ b\ (i \mod |b|))$$

With these definitions at hand, all we have to do to index in spaces is to define a function *sized* which extracts the finite set of values of a given size k from a space.

$$sized\ Empty\ \ k = \{\} \qquad sized\ (Pay\ a)\ \ k = sized\ a\ (k-1)$$
$$sized\ (Pure\ a)\ 0 = \{a\} \qquad sized\ (a\ :+:\ b)\ k = sized\ a\ k \uplus sized\ b\ k$$
$$sized\ (Pure\ a)\ k = \{\} \qquad sized\ (f\ :\$:\ a)\ k = \{f\ x : x \in sized\ a\ k\}$$
$$sized\ (Pay\ a)\ \ 0 = \{\}$$

We define *sized Pure* to be empty for all sizes except 0, since we want values of an exact size. For *Pay* we get the values of size $k-1$ in the underlying space. Union and function application translate directly to union and application on sets. Selecting k-sized values of a product space requires dividing the size between its components. Thus, we can consider the set as a disjoint union of the $k+1$ different ways of dividing size between the components:

$$sized\ (a :*: b)\ k = \biguplus_{k_1+k_2=k} sized\ a\ k_1 \times sized\ b\ k_2$$

Knowing how to index in finite sets, we can implement an indexing function on spaces by composing the *sized* function with the *indexFin* function.

```
indexSized :: Space a → Int → Integer → a
indexSized s k i = indexFin (sized s k) i
```

Computing cardinalities and indexing requires arbitrarily large integers, which are provided by Haskell's *Integer* type. Calculating cardinalities can be computationally intensive, and to be practical requires memoising cardinalities of recursive data types, which is implemented using another constructor of the *Space a* data type not shown here.

3 Predicate-Guided Indexing

Having solved the problem of generating members of algebraic data types, we extend the problem with a predicate that all generated values must satisfy.

A first approach for uniform generation is to choose a size, generate values of that size, test them against the predicate and keep the ones for which the predicate is *True*. This works well for cases where the proportion of values that satisfy the predicate is large enough, for example larger than 1%, but is far too inefficient in many practical situations.

In order to speed up random generation of values satisfying a given predicate, we use the lazy behaviour of the predicate to know its result on sets of values, rather than individual values, similarly to [10]. For instance, consider a predicate that tests if a list is sorted by checking the inequality of each pair of consecutive elements in turn starting from the front. Applying the predicate to $1:2:1:3:5:[\,]$ will yield *False* after the pair $(2,1)$ is encountered, before the predicate looks at the later elements, which means that it will return *False* for all lists starting with $1,2,1$. Once we have computed a set of values for which the predicate is going to return false, we remove all of these values from our original set.

To detect this we can exploit Haskell's call-by-need semantics by applying the predicate to a partially-defined value. In this case, observing that our predicate returns *False* when applied to a partially-defined list $1:2:1:\bot$, can lead us to conclude that $\bot$ can be replaced with any value without affecting the result. Thus, we could remove all lists that start with $1,2,1$ from the space. For many realistic predicates this removes a large number of values with each failed generation attempt, improving the chances of finding a value satisfying the predicate next time.

We implement this by using the function *valid*, that determines whether a given predicate needs to investigate its argument or not in order to produce its result. The function *valid* returns *Nothing* if the predicate needed its argument, and *Just b* if the predicate returns *b* regardless of its argument.

$$valid :: (a \to Bool) \to Maybe\ Bool$$

For example $valid\ (\lambda a \to True) == Just\ True$, $valid\ (\lambda a \to False) == Just\ False$, $valid\ (\lambda x \to x+1>x) == Nothing$. Implementing *valid* involves applying the predicate to $\bot$ and catching the resulting exception if there is one. Catching the exception is an impure operation in Haskell, so the function *valid* is also impure (specifically, it breaks monotonicity).

The function *valid* is used to implement the indexing function, which takes the predicate, the space, the size and a random index.

$$index :: (a \to Bool) \to Space\ a \to Int \to Integer \to Space\ a$$

It returns a space of values containing at least the value at the given index, and any number of values for which the predicate yields the same result. When the returned space contains values for which the predicate is false, the top level search procedure (not shown here) removes all these values from the original enumeration and retries with a new index in the now smaller enumeration.

The function *index* is implemented by recursion on its *Space a* argument, and composing the predicate with traversed constructor functions, until its result is independent of which value from the current space is chosen. In particular, *index* on a function application $(:\$:)$ returns the current space if the predicate p' returns the same result regardless of its argument, which is determined by calling $valid\ p'$. Otherwise, it calls *index* recursively on the subspace, composing the predicate with the applied function.

$$\begin{array}{l} index\ p\ (f\ :\$:\ a)\ k\ i = \textbf{case}\ valid\ p'\ \textbf{of} \\ \quad Just\ \_\ \ \to f\ :\$:\ a \\ \quad Nothing \to f\ :\$:\ index\ p'\ a\ k\ i \\ \quad \textbf{where}\ p' = p \circ f \end{array}$$

3.1 Predicate-Guided Refinement Order

When implementing *index* for products, it is no longer possible to choose a division of size between the components, as was the case for indexing in Section 2. Determining the size of components early causes problems when generalising to sets of partial values, as the same partial value may represent values where size is divided in different ways.

We solve this problem using the algebraic nature of our spaces to eliminate products altogether. Disregarding the order of values when indexing, spaces form an algebraic semi-ring, which means that we can use the following algebraic laws to eliminate products.

$$\begin{array}{lll} a \otimes (b \oplus c) \equiv (a \otimes b) \oplus (a \otimes c) & & \textit{[distributivity]} \\ a \otimes (b \otimes c) \equiv (a \otimes b) \otimes c & & \textit{[associativity]} \\ a \otimes 1 & \equiv a & \textit{[identity]} \\ a \otimes 0 & \equiv 0 & \textit{[annihilation]} \end{array}$$

Expressing these rules on our Haskell data type is more complicated, because we need to preserve the types of the result, i.e. we only have associativity of products if we provide a function that transforms the left associative pair back to a right associative one, etc. The four rules defined on the *Space* data type expressed as a transformation operator ($***$) are as follows:

$$
\begin{array}{lll}
a *** (b :+: c) = (a :*: b) :+: (a :*: c) & & \textit{[distributivity]} \\
a *** (b :*: c) = (\lambda((x,y),z) \to (x,(y,z))) :\$: ((a :*: b) :*: c) & & \textit{[associativity]} \\
a *** (\mathit{Pure}\ x) = (\lambda y \to (y,x)) :\$: a & & \textit{[identity]} \\
a *** \mathit{Empty} = \mathit{Empty} & & \textit{[annihilation]}
\end{array}
$$

In addition to this, we need two laws for eliminating *Pay* and function application.

$$
\begin{array}{lll}
a *** (\mathit{Pay}\ b) = \mathit{Pay}\ (a :*: b) & & \textit{[lift-pay]} \\
a *** (f :\$: b) = (\lambda(x,y) \to (x, f\ y)) :\$: (a :*: b) & & \textit{[lift-fmap]}
\end{array}
$$

The first law states that paying for the component of a pair is the same as paying for the pair, the second that applying a function f to one component of a pair is the same as applying a modified (lifted) function on the pair. If recursion is always guarded by a *Pay*, we know that the transformation will terminate after a bounded number of steps.

Using these laws we could define *index* on products by applying the transformation, so $\mathit{index}\ p\ (a :*: b) = \mathit{index}\ p\ (a *** b)$. This is problematic, because $***$ is a right-first traversal, which means that for our generators the left component of a pair is never generated before the right one is fully defined. This is detrimental to generation, since the predicate may not require the right operand to be defined. To guide the refinement order by the evaluation order of the predicate, we need to 'ask' the predicate which component should be defined first. We define a function similar to *valid* that takes a predicate on pairs:

$$\mathit{inspectsRight} :: ((a,b) \to \mathit{Bool}) \to \mathit{Bool}$$

The expression *inspectsRight p* is *True* iff p evaluates the right component of the pair before the left. Just like *valid*, *inspectsRight* exposes some information of the Haskell runtime, which can not be observed directly.

To define indexing on products we combine *inspectsRight* with another algebraic law: commutativity of products. If the predicate 'pulls' at the left component, the operands of the product are swapped before applying the transformation for the recursive call.

$$
\begin{array}{l}
\mathit{index}\ p\ (a :*: b)\ k\ i = \textbf{if}\ \mathit{inspectsRight}\ p \\
\quad \textbf{then}\ \mathit{index}\ p\ (a *** b)\ k\ i \\
\quad \textbf{else}\ \ \mathit{index}\ p\ (\mathit{swap} :\$: (b *** a))\ k\ i \\
\quad \textbf{where}\ \mathit{swap}\ (a,b) = (b,a)
\end{array}
$$

The end result is an indexing algorithm that gradually refines the value it indexes to, by expanding only the part that the predicate needs in order to progress. With every refinement, the space is narrowed down until the predicate is guaranteed to be true or false for all values in the space. In the end the algorithm removes the indexed subspace from the search space, so no specialisations of the tested value are ever generated.

Note that the generation algorithm is still uniform because we only remove values for which the predicate is false from the original set of values. The uniformity is only concerned with the set of values for which the predicate is true.

3.2 Relaxed Uniformity Constraint

When our uniform generator finds a space for which the predicate is false, the algorithm chooses a new index and retries, which is required for uniformity. We have implemented two alternative algorithms.

The first one is to backtrack and try the alternative in the most recent choice. Such generators are no longer uniform, but potentially more efficient. Even though the algorithm start searching at a uniformly chosen index, since an arbitrary number of backtracking steps is allowed the distribution of generated values may be arbitrarily skewed. In particular, values satisfying the predicate that are 'surrounded' by many values for which it does not hold may be much more likely to be generated than other values.

The second algorithm also performs backtracking, but imposes a bound b for how many values the backtracking search is allowed to skip over. When the bound b is reached, a new random index is generated and the search is restarted. The result is an algorithm which has an 'almost uniform' distribution in a precise way: the probabilities of generating any two values differ at most by a factor $b+1$. So, if we pick $b = 1000$, generating the most likely value is at most 1001 times more likely than the least likely value.

The bounded backtracking search strategy generalises both the uniform search (when the bound b is 0) and the unlimited backtracking search (when the bound b is infinite).

We expected the backtracking strategy to be more efficient in terms of time and space usage than the uniform search, and the bounded backtracking strategy to be somewhere in between, with higher bounds leading to results closer to unlimited backtracking. Our intention for developing these alternative algorithms was that trading the uniformity of the distribution for higher performance may lead to a higher rate of finding bugs. Section 4 contains experimental verification of these hypotheses.

3.3 Parallel Conjunction

It is possible to improve the generation performance by introducing the parallel conjunction operator [10], which makes pruning the search space more efficient. Suppose we have a predicate $p\,x = q\,x \;\&\&\; r\,x$. Given that && is left-biased, if $\mathit{valid}\,r == \mathit{Just\ False}$ and $\mathit{valid}\,q == \mathit{Nothing}$ then the result of $\mathit{valid}\,p$ will be *Nothing*, even though we expect that refining q will make the conjunction return *False* regardless of what $q\,x$ returns.

We can define a new operator &&& for parallel conjunction with different behaviour when the first operand is undefined: $\bot$ &&& $\mathit{False} == \mathit{False}$. This may make the indexing algorithm terminate earlier when the second operand of a conjunction is false, without needing to perform refinements needed by the first operand at all. Similarly, we can define parallel disjunction that is *True* when either operand is *True*.

4 Experimental Evaluation

We evaluated our approach in four benchmarks. Three of them involved measuring the time and memory needed to generate 2000 random values of a given size satisfying a predicate. The fourth benchmark compared a derived simply-typed lambda term generator against a hand-written one in triggering strictness bugs in the GHC compiler. Some benchmarks were also run with a naïve generator that generates random values from a space, as in Section 2, and filters out those that do not satisfy a predicate.

4.1 Trees

Our first example is binary search trees (BSTs) with Peano-encoded natural numbers as their elements, defined as follows.

```
data Tree a = L
  | N a (Tree a) (Tree a)
isBST :: Ord a ⇒ Tree a → Bool
data Nat = Z | Suc Nat
```

```
instance Ord Nat where
  _     < Z     = False
  Z     < Suc _ = True
  Suc x < Suc y = x < y
```

The *isBST* predicate (omitted) decides if the tree is a BST, and uses a supplied lazy comparison function for type *Nat* for increased laziness.

We measured the time and space needed to generate 2000 trees for each size from a range of sizes, allowing at most 300 s of CPU time and 4 GiB of memory to be used. Derived generators based on three different search strategies (see Section 3.2) were used: One performing uniform sampling (*uniform*), one bounded backtracking allowed to skip at most 10k values (*backtracking 10k*), and one performing unbounded backtracking (*backtracking*). A naïve generate-and-filter generator was also used for comparison.

Both *backtracking 10k* and *backtracking* generators produce non-uniform distributions of values. The skew of the *backtracking 10k* generator is limited, as the least likely values are generated at most 10k times less likely than the most common ones, as mentioned in Section 3.2.

Fig. 2 shows the time and memory consumed the runs with resource limits marked by dotted lines in the plots. Run times for all derived generators rise sharply with the increased size of generated values and seem to approach exponential growth for larger sizes. The backtracking generator performs best of all, and has a slower exponential growth rate for large sizes than the other derived generators. The *backtracking 10k* generator achieved similar performance as the *uniform* one when generating values that are about 11 size units larger. The generate-and-filter generator was not able to complete any of the runs in time, and is omitted from the graphs.

4.2 Simply-typed Lambda Terms

Generating random simply-typed lambda terms was our motivating application. Simply-typed lambda terms can be turned into well-typed Haskell programs and used for testing compilers. Developing a hand-written recursive generator for them requires the use of backtracking, because of the inability of predicting whether a given local choice

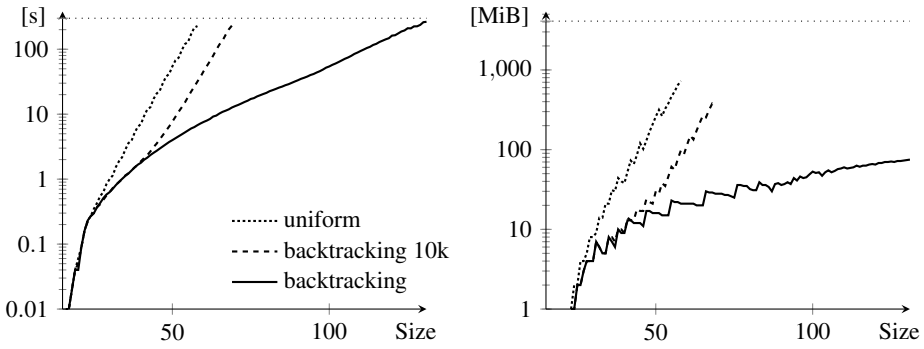

Fig. 2: Run times in (left) and memory consumption (right) of derived generators generating 2000 BSTs depending on the size of generated values.

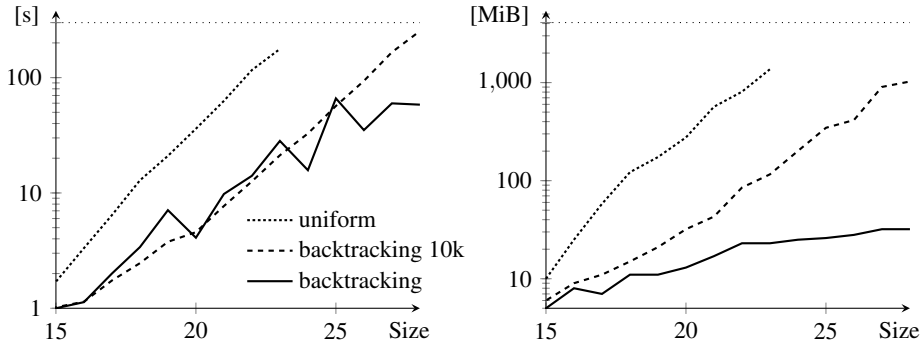

Fig. 3: Run times (left) and memory consumption (right) of derived generators generating 2000 simply-typed lambda terms depending on the size of generated terms.

can lead to a successful generation, and because typing constraints from two distant parts of a term can cause conflict. Achieving satisfactory distribution and performance requires careful tuning, and it is difficult to assess if any important values are severely underrepresented [9].

On the other hand, obtaining a generator that is based on our framework requires only the definitions from Fig. 1, and a relatively simple space definition, which we omit here. The code for the type checker is standard and uses a type stored in each application node (*tx* in *Ap f x tx*) to denote the type of the argument term for simplicity.

To evaluate the generators, we generated 2000 terms with a simple initial environment of 6 constants. The derived generator with three search strategies and one based on generate-and-filter were used. Fig. 3 shows the results. The uniform search strategy is capable of generating terms of size up to 23. For larger sizes, the generator exceeded the resource limits (300 s and 4 GiB, marked w/ dotted lines). The generator that used limited backtracking allowed generating terms up to size 28, using 9 times less CPU time and over 11 times less memory than the uniform one at size 23. Unlimited backtracking improved

Generator	Hand-written	Derived (size 30)
Terms per ctr ex. (k)	18.6	52.5
Gen. CPU time per ctr ex. (min)	1.7	14.0
Test CPU time per ctr ex. (min)	1.8	10.4
Tot. CPU time per ctr ex. (min)	3.5	24.4

Table 1: Performance of the reference hand-written term generator compared to a derived generator using backtracking with size 30. We compare the average number of terms that have to be generated before a counterexample (ctr ex.) is found, and how much CPU time the generation and testing consumes per found counterexample.

memory consumption dramatically, up to 30-fold, compared to limited backtracking. The run time is improved only slightly with unlimited backtracking. Finally, the generator based on generate-and-filter exceeded the run times for all sizes, and is not included in the plots.

4.3 Testing GHC

Discovering strictness bugs in the GHC optimising Haskell compiler was our prime reason for generating random simply-typed lambda terms. To evaluate our approach, we compared its bug finding power to a hand-written generator that had been developed before [9] using the same test property that had been used there.

Random simply-typed lambda terms were used for testing GHC by first generating type-correct Haskell modules containing the terms, and then using them as test data. In this case, we generated modules containing expressions of type $[Int] \to [Int]$ and compiled them with two different optimisation levels. Then, we tested their observable behaviour and compared them against each other, looking for discrepancies.

We implemented the generator using a similar data type as in Fig. 1 extended with polymorphic constants and type constructors. For efficiency reasons we avoided having types in term application constructors, and used a type checker based on type inference, which is more complex but still easily implementable. It allows generators to scale up to larger effective term sizes because not having types in the term representation increases the density of well-typed terms.

A generator based on this data type was capable of generating terms containing 30 term constructors, and was able to trigger GHC failures. Table 1 shows the results of testing GHC both with the hand-written simply-typed lambda term generator and our derived generator. The hand-written generator used for comparison generated terms of sizes from 0 to about 90, with most terms falling in the range of 20–50. It needed the least total CPU time to find a counterexample, and the lowest number of generated terms. The derived generator needs almost 7 times more CPU time per failure than the hand-written one.

The above results show that a generator derived from a predicate can be used to effectively find bugs in GHC. The derived generator is less effective than a hand-written one, but is significantly easier to develop. Developing an efficient type-checking predicate

Predicates	Backtracking	Backtracking c/o
1, 2, 3, 4, 5	13	15
1, 3, 4, 5	13	30
1, 3, 5	31	30

Table 2: Maximum practical sizes of values generated by derived program generators that use unlimited backtracking and backtracking with cut-off of 10k.

required for the derived generator took a few days, whereas the development and tuning of the hand-written generator took an order of months.

4.4 Programs

The *Program* benchmark is meant to simulate testing of a simple compiler by generating random programs, represented by the following data type.

$$
\begin{aligned}
&\textbf{type } \mathit{Name} &&= \mathit{String}\\
&\textbf{data } \mathit{Program} &&= \mathit{New\ Name\ Program} \mid \mathit{Name} := \mathit{Expr} \mid \mathit{Skip}\\
&&&\mid \mathit{Program} :>> \mathit{Program} \mid \mathit{If\ Expr\ Program\ Program}\\
&&&\mid \mathit{While\ Expr\ Program}\\
&\textbf{data } \mathit{Expr} &&= \mathit{Var\ Name} \mid \mathit{Add\ Expr\ Expr}
\end{aligned}
$$

The programs contain some common imperative constructs and declarations of new variables using *New*, which creates a new scope.

A compiler may perform a number of compilation passes, which would typically transform the program into some kind of normal form that may be required by the following pass. Our goal is to generate test data that satisfy the precondition in order to test the code of each pass separately. We considered 5 predicates on the program data type that model simple conditions that may be required by some compilation phases: (1) *boundProgram* saying that the program is well-scoped, (2) *usedProgram* saying that all bound variables are used, (3) *noLocalDecls* requiring all variables to be bound on the top level, (4) *noSkips* forbidding the redundant use of $:>>$ and *Skip*, and (5) *noNestedIfs* forbidding nested *if* expressions.

Table 2 shows maximum value sizes that can be practically reached by the derived generators for the program data type with different combinations of predicates. All runs were generating 2000 random programs with resource limits (300 s and 4 GiB). When all predicates were used, the generators performed poorly being able to reach at most size 15. When the *usedProgram* predicate was omitted, the generator that uses limited backtracking improved considerably, whereas the one using unlimited backtracking remained at size 13. Removing the *noSkips* predicate turns the tables on the two generators improving the performance of the unlimited backtracking generator dramatically.

A generator based on generate-and-filter was also benchmarked, but did not terminate within the time limit for the sizes we tried.

4.5 Summary

All derived generators performed much better than ones based on generate-and-filter in three out of four benchmarks. In the fourth one, testing GHC, using a generator based on generate-and-filter was comparable to using our uniform or near-uniform derived generators, and slower than a derived generator using backtracking. In that benchmark the backtracking generator was the only that was able to find counterexamples, and yet it was less effective than a hand-written generator. However, as creating the derived generators was much quicker, we believe that they are still an attractive alternative to a hand-written generator.

The time and space overhead of the derived generators appeared to rise exponentially, or almost exponentially with the size of generated values in most cases we looked at, similarly to what can be seen in Figures 2 and 3.

In most cases the backtracking generator provided the best performance, which means that sometimes we may have to sacrifice our goal of having a predictable distribution. However, we found the backtracking generator to be very sensitive to the choice of the predicate. For example, some combinations of predicates in Section 4.4 destroyed its performance, while having less influence on the uniform and near-uniform generators. We hypothesise that this behaviour may be caused by regions of search space where the predicates evaluate values to a large extent before returning *False*. The backtracking search remain in such regions for a long time, in contrast to the other search that gives up and restarts after a number of values have been skipped.

Overall, the performance of the derived generators is practical for some applications, but reaching higher sizes of generated data might be needed for effective bug finding. In particular, being able to generate larger terms may improve the bug-finding performance when testing for GHC strictness bugs.

5 Related Work

Feat. Our representation of spaces and efficient indexing is based on FEAT (Functional Enumeration of Algebraic Types) [5]. The practicalities of computing cardinalities and the deterministic indexing functions are described there. The inability to deal with complex data type invariants is the major concern for FEAT, which is addressed by this paper.

Lazy SmallCheck and Korat. Lazy SmallCheck [10] uses laziness of predicates to get faster progress in an exhaustive depth-limited search. Our goal was to reach larger, potentially more useful values than Lazy SmallCheck by improving on it in two directions: using size instead of depth and allowing random search in sets that are too large to search exhaustively. Korat is a framework for testing Java programs [2]. It uses similar techniques to exhaustively generate size-bounded values that satisfy the precondition of a method, and then automatically check the result of the method for those values against a postcondition.

EasyCheck: Test Data For Free. EasyCheck is a library for generating random test data written in the Curry functional logic programming language [3]. Its generators define search spaces, which are enumerated using diagonalisation and randomising local choices. In this way values of larger sizes have a chance of appearing early in the enumeration, which is not the case when breadth-first search is used. The Curry language supports narrowing, which can be used by EasyCheck to generate values that satisfy a given predicate. The examples that are given in the paper suggest that, nonetheless, micro-management of the search space is needed to get a reasonable distribution. The authors point out that their enumeration technique has the problem of many very similar values being enumerated in the same run.

Metaheuristic Search. In the GödelTest [6] system, so-called metaheuristic search is used to find test cases that exhibit certain properties referred to as *bias objectives*. The objectives are expressed as fitness metrics for the search such as the mean height and width of trees, and requirements on several such metrics can be combined for a single search. It may be possible to write a GödelTest generator by hand for well typed lambda terms and then use bias objectives to tweak the distribution of values in a desired direction, which could then be compared to our work.

Lazy Nondeterminism. There is some recent work on embedding non-determinism in functional languages [7]. As a motivating example an *isSorted* predicate is used to derive a sorting function, a process which is quite similar to generating sorted lists from a predicate. The framework defined in [7] is very general and could potentially be used both for implementing SmallCheck style enumeration and for random generation.

Generating Lambda Terms. There are several other attempts at enumerating or generating well typed lambda terms. One such attempt uses generic programming to exhaustively enumerate lambda terms by size [11]. The description focuses mainly on the generic programming aspect, and the actual enumeration appears to be mainly proof of concept with very little discussion of the performance of the algorithm. There has been some work on counting lambda terms and generating them uniformly [8]. This includes generating well typed terms by a simple generate-and-filter approach.

6 Discussion

Performance of Limiting Backtracking. The performance of our generators depends on the strictness and evaluation order of the used predicate. The generator that performs unlimited backtracking was especially sensitive to the choice of predicate, as shown in Section 4.4. Similar effects have been observed in Korat [2], which also performs backtracking.

We found that for most predicates unbounded backtracking is the fastest. But unexpectedly, for some predicates imposing a bound on backtracking improves the run time of the generator. This also makes the distribution more predictable, at the cost of increased memory consumption. We found tweaking the degree of backtracking to be a

useful tool for improving the performance of the generators, and possibly trading it for distribution guarantees.

In-place Refinement. We experimented with a more efficient mechanism for observing the evaluation order of predicates, which avoids repeated evaluation of the predicate. For that we use an indexing function that attaches a Haskell IO-action to each subcomponent of the generated value. When the predicate is applied to the value, the IO-actions will fire only for the parts that the property needs to inspect to determine the outcome. Whenever the indexing function is required to make a choice, the corresponding IO-action records the option it did not take, so after the predicate has finished executing the refined search space can be reconstructed. Guiding the evaluation order is handled automatically by the Haskell run time system, which has call-by-need built into it.

In-place refinement is somewhat more complicated than the procedure described in Section 3. Also, defining parallel conjunction for this type of refinement is difficult, because inspecting the result of a predicate irreversibly makes the choices required to compute the result. For this reason our implementation of in-place refinement remains a separate branch of development and a topic of future work.

Conclusion. Our method aims at preserving the simplicity of generate-and-filter type generators, but supporting more realistic predicates that accept only a small fraction of all values. This approach works well provided the predicates are lazy enough.

Our approach reduces the risk of having incorrect generators, as coming up with a correct predicate is usually much easier than writing a correct dedicated generator. Creating a predicate which leads to an efficient derived generator, on the other hand, is more difficult.

Even though performance remains an issue when generating large test cases, experimental results show that our approach is a viable option for generating test data in many realistic cases.

Acknowledgements

This research has been supported by the Resource-Aware Functional Programming (RAW FP) grant awarded by the Swedish Foundation for Strategic Research.

References

[1] Thomas Arts et al. “Testing Telecoms Software with Quviq QuickCheck”. In: *Proc. 2006 Erlang Workshop*. ACM, 2006, pp. 2–10.

[2] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. “Korat: Automated Testing Based on Java Predicates”. In: *Proc. 2002 Inl. Symp. Software Testing and Analysis (ISSTA ’02)*. ACM, 2002, pp. 123–133.

[3] Jan Christiansen and Sebastian Fischer. “EasyCheck: test data for free”. In: *FLOPS’08*. Springer, 2008, pp. 322–336.

[4] Koen Claessen and John Hughes. “QuickCheck: a lightweight tool for random testing of Haskell programs”. In: *ICFP ’00*. ACM, 2000, pp. 268–279.

[5] Jonas Duregård, Patrik Jansson, and Meng Wang. “FEAT: functional enumeration of algebraic types”. In: *Proc. 2012 Symp. Haskell*. ACM, 2012, pp. 61–72.

[6] Robert Feldt and Simon Poulding. “Finding Test Data with Specific Properties via Metaheuristic Search”. In: *Proc. Intl. Symp. Software Reliability Engineering (ISSRE)*. IEEE, 2013.

[7] Sebastian Fischer, Oleg Kiselyov, and Chung chieh Shan. “Purely functional lazy nondeterministic programming”. In: *J. Funct. Program.* 21.4-5 (2011), pp. 413–465.

[8] Katarzyna Grygiel and Pierre Lescanne. “Counting and generating lambda terms”. In: *J. Funct. Program.* 23 (05 Sept. 2013), pp. 594–628.

[9] Michał H. Pałka. “Testing an Optimising Compiler by Generating Random Lambda Terms”. Licentiate Thesis. Chalmers University of Technology, Gothenburg, Sweden, 2012.

[10] Colin Runciman, Matthew Naylor, and Fredrik Lindblad. “Smallcheck and lazy smallcheck: automatic exhaustive testing for small values”. In: *Haskell ’08*. ACM, 2008, pp. 37–48.

[11] Alexey Rodriguez Yakushev and Johan Jeuring. “Enumerating Well-Typed Terms Generically”. In: *AAIP 2009*. Vol. 5812. LNCS. Springer, 2009, pp. 93–116.