JFP **25**, e8, 31 pages, 2015.
doi:10.1017/S0956796815000143

Generating constrained random data with uniform distribution

KOEN CLAESSEN, JONAS DUREGÅRD and MICHAŁ H. PAŁKA

Department of Computer Science and Engineering, Chalmers University of Technology, Gothenburg, Sweden

(*e-mail:* koen@chalmers.se, jonas.duregard@chalmers.se, michal.palka@chalmers.se)

Abstract

We present a technique for automatically deriving test data generators from a given executable predicate representing the set of values we are interested in generating. 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 that has better performance, but where the distribution is skewed in a limited, albeit predictable way. Experimental evaluation of the techniques shows that the automatically derived generators are much easier to define than handwritten 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 (Claessen & Hughes 2000; Arts *et al.* 2006). Two ingredients are required for property-based testing: a *test data generator* and a *property* (sometimes called a test 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 (Claessen & Hughes 2000) 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 ensure termination.

The above method for test generation works well for generating structured data. But it becomes much harder when the data must *satisfy 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

```
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.

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 in the polymorphic lambda calculus, and at the same time (b) maintain a reasonable distribution such that no programs were arbitrarily excluded from generation (Pałka *et al.* 2011; Pałka 2012).

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 executable predicates (functions computing a boolean value) on the data type, can we automatically generate test data that satisfy all the predicates and at the same time has a predictable, useful distribution?

To be more concrete, let us take a look at Figure 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 automatically 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 (Runciman *et al.* 2008). The problem with using depth is that the number of terms grows extremely fast as the depth increases (doubly exponential even for simple binary trees). Moreover, useful distributions for sets of trees of depth d are hard to find, because there are many more complete trees of depth d than there are sparse trees. This 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 Functional Enumeration of Algebraic Types (FEAT) (Duregård *et al.* 2012) 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 led to the predicate returning false with the same execution path as x. We also outline a proof that this sampling procedure is uniform.

Unfortunately, perfect uniformity turns out to be too inefficient 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.

This paper extends and improves a paper presented at FLOPS 2014 (Claessen *et al.* 2014). The technical content is essentially unchanged, but we made several presentation and restructuring modifications. In this version, we expand the description of the algorithm (Section 3), provide a detailed example of its operation (Section 3.2), demonstrate that the distribution of the generated values is uniform (Section 4), and discuss an alternative algorithm better suited for non-deterministic predicates (Section 5.4).

2 Generating values of algebraic data types

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 (Duregård *et al.* 2012), 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.

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 = Zr | Sc Nat
data ListNat = Nil | 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* (*Sc* (*Sc Zr*)) (*Cons Zr Nil*), *Cons* (*Sc Zr*) (*Cons* (*Sc Zr*) *Nil*) and *Cons Zr* (*Cons Zr* (*Cons Zr Nil*)) 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 (a procedure also known as *unranking* (Knuth 2006)).

$$\text{index}_{S,k} : \{0 \ldots |S_k| - 1\} \to S_k$$

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:

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

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

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

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 *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 the (disjoint) union 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.

It is possible to construct spaces with duplicate elements from these operations, although it is rarely useful in practice (typically the operands of :+: are disjoint and functions used in :$: are injective, particularly constructor functions). This means that in general there is a multiset of values of any size and whenever we speak of uniform sampling procedures it is understood to be uniform over the set of occurrences of values, not over the set of values themselves. Simply put: repeated values are overrepresented exactly as one might expect from a uniform sampler of a multiset.

A convenient operator on spaces is the lifted application operator that takes a space of functions and a space of parameters and produces the space of all results from applying the functions to the parameters:

```
(<*>) :: Space (a → b) → Space a → Space b
s1 <*> s2 = (λ(f, a) → f a) :$: (s1 :*: s2)
```

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

```
spNat :: Space Nat
spNat = Pay (Pure Zr :+: (Sc :$: spNat))
spListNat :: Space ListNat
spListNat = Pay (Pure Nil :+: (Cons :$: spNat <*> spListNat))
```

Unit constructors are represented with *Pure*, whereas compound constructors are mapped on the spaces for the types they contain. In this example, *Pay* is applied each time we introduce a constructor, which makes the size of values equal to the number of constructors they contain. This is a common pattern, but the user may choose to assign costs differently, which would change the sizes of individual values and consequently the distribution of size-limited generators. The only rule when assigning costs is that all recursion is guarded by at least one *Pay* operation, otherwise the sets of values of a given size may be infinite, causing non-terminating cardinality computations.

2.3 Uniform sampling by size

Uniform sampling on spaces can be reduced to two subproblems: Extracting the finite set of values of a particular size, and uniform sampling from such sets.

Assume we have a data type *Set a* for finite multisets, constructed by combining the empty set ($\{\}$), singleton sets ($\{a\}$), (disjoint) union ($\uplus$) and Cartesian product ($\times$). We can also apply a function f to all members set s with *fmap f s*, such that *fmap f a* = *fmap f* (*sized a k*). From the definition of a finite set, its cardinality can be defined as follows:

$$
\begin{aligned}
|\{\}| &= 0 \\
|\{a\}| &= 1 \\
|a \times b| &= |a| * |b| \\
|a \uplus b| &= |a| + |b| \\
|fmap\ f\ a| &= |a|
\end{aligned}
$$

Guided by this definition, we can define an indexing function on the type that maps integers in the range $(0, |a| - 1)$ to (occurrences of) values in the multiset:

$$
\begin{aligned}
&indexSet\ \{a\}\ \ 0\ \ = a \\
&indexSet\ (a \uplus b)\ \ i \mid i < |a| = indexSet\ a\ i \\
&indexSet\ (a \uplus b)\ \ i \mid i \geqslant |a| = indexSet\ b\ (i - |a|) \\
&indexSet\ (a \times b)\ \ i\ \ = (indexSet\ a\ (i \div |b|), indexSet\ b\ (i \bmod |b|)) \\
&indexSet\ (fmap\ f\ a)\ i\ \ = f\ (indexSet\ a\ i)
\end{aligned}
$$

Since *indexSet* is bijective from a finite integer range into the values of the multiset, the only remaining component for uniform sampling from sets is a function for uniform sampling from ranges. For this purpose, suppose we have a monad *Random a* with the only side-effect of generating random values, and the following function:

$$uniformRange :: (Integer, Integer) \rightarrow Random\ Integer$$

Computing *uniformRange* (lo, hi) returns a uniformly random integer in the inclusive interval (lo, hi). On top of this, we can build the following procedure for uniform sampling from finite sets:

```
uniformSet :: Set a → Random a
uniformSet s | |s| == 0 = error "empty set"
             | otherwise = do
                 i ← uniformRange (0, |s| − 1)
                 return (indexSet s i)
```

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

$$
\begin{aligned}
&sized :: Space\ a \rightarrow Int \rightarrow Set\ a \\
&sized\ Empty\ \ k = \{\} \\
&sized\ (Pure\ a)\ \ 0 = \{a\} \\
&sized\ (Pure\ a)\ \ k = \{\} \\
&sized\ (Pay\ a)\ \ 0 = \{\} \\
&sized\ (Pay\ a)\ \ k = sized\ a\ (k - 1)
\end{aligned}
$$

$$sized\ (a\ :+:\ b)\ k = sized\ a\ k \uplus sized\ b\ k$$
$$sized\ (f\ :\$:\ a)\ k = fmap\ f\ (sized\ a\ k)$$

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 the components of the resulting pair. 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 sample finite sets, we can implement a sampling procedure on spaces by composing the *sized* function with the *uniformSet* function.

$$uniformSized :: Space\ a \to Int \to Random\ a$$
$$uniformSized\ s\ k = uniformSet\ (sized\ s\ k)$$

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

3 Predicate-guided uniform sampling

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

A first approach for uniform generation is to use a simple form of *rejection sampling*. To generate a value that satisfies $p :: a \to Bool$ of a desired size, we simply generate values until we find one:

```
uniformFilter :: (a → Bool) → Space a → Int → Random a
uniformFilter p s k = do
  a ← uniformSized s k
  if p a then return a
         else uniformFilter p s k
```

The procedure returns values of a given size from the space that satisfy the predicate with a uniform distribution over the occurrences of these values in the space, as formally stated in Section 4.

This approach works well for cases where the proportion of values that satisfy the predicate is large enough, but it is far too inefficient in many practical situations (and if there are no values that satisfy the predicate it will never terminate).

In order to speed up random generation of values satisfying a given predicate, we propose another sampling procedure that uses the lazy behaviour of the predicate to know its result on sets of values, rather than individual values, similarly to Runciman

et al. (2008). For instance, consider a predicate *ordered* that tests if a list is sorted by comparing each pair of consecutive elements, starting from the front.

$$\begin{aligned}
&\mathit{ordered} :: \mathit{Ord}\ a \Rightarrow [a] \to \mathit{Bool}\\
&\mathit{ordered}\ [\,] \qquad\quad = \mathit{True}\\
&\mathit{ordered}\ [x] \qquad\quad = \mathit{True}\\
&\mathit{ordered}\ (x : y : xs) = x <= y\ \&\&\ \mathit{ordered}\ (y : xs)
\end{aligned}$$

Applying the predicate to $1 : 2 : 1 : 3 : 5 : [\,]$ will yield *False* after the pair $(2, 1)$ is encountered, before the predicate inspects the later elements, thanks to the short-circuiting && operator. This means that *ordered* is *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 *Space*.

To detect this we 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 $1 : 2 : 1 : \bot$, implies that the undefined part ($\bot$) 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 using the function *universal* that determines if a given predicate is universally true, universally false, or if it (potentially) depends on its argument. The function *universal* returns *Nothing* if the predicate need to inspects its argument to yield a result, and *Just True* if the predicate is universally true and *Just False* if is universally false.

$$\mathit{universal} :: (a \to \mathit{Bool}) \to \mathit{Maybe}\ \mathit{Bool}$$

For example, $\mathit{universal}\ (\lambda a \to \mathit{True}) = \mathit{Just}\ \mathit{True}$, $\mathit{universal}\ (\lambda a \to \mathit{False}) = \mathit{Just}\ \mathit{False}$, $\mathit{universal}\ (\lambda x \to x + 1 > x) = \mathit{Nothing}$. Implementing *universal* 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 *universal* is also impure (specifically, it breaks monotonicity).

The function *universal* is used to implement a new *sized* function: *sizedP*, which takes a predicate as a parameter along with a space and a size. Where *sized* resulted in just a set of values ($\mathit{Set}\ a$), sampling a value from the result of *sizedP* will either give a value that satisfies the predicate or an updated space that excludes a non-zero number of values that falsify the predicate.

$$\mathit{sizedP} :: (a \to \mathit{Bool}) \to \mathit{Space}\ a \to \mathit{Int} \to \mathit{Set}\ (\mathit{Either}\ a\ (\mathit{Space}\ a))$$

The intention of *sizedP* is best explained by implementing a sampling procedure that uses it. The sampling procedure picks a random value from the resulting set, if it is $\mathit{Left}\ x$, x satisfies the predicate and the procedure terminates, otherwise it recursively searches the new smaller space for a satisfying value:

```
sizedP :: (a → Bool) → Space a → Int → Set (Either a (Space a))
sizedP p (f :$: a) k = case universal p' of
  Just False → replicateSet |sized a k| (Right Empty)
  _          → fmap (apply f) (sizedP p' a k)
  where p' = p ∘ f
        apply f x = case x of
          Left x  → Left (f x)
          Right a → Right (f :$: a)
sizedP p (a :*: b) k = if inspectsFst p
    then sizedP p (swap :$: (b ⋆⋆⋆ a)) k
    else sizedP p (a ⋆⋆⋆ b) k
    where swap (a,b) = (b,a)
sizedP p (a :+: b) k = rebuild (:+: b) (sizedP p a k) ⊎
                       rebuild (a :+:) (sizedP p b k)
  where
    rebuild :: (Space a → Space a) → Set (Either a (Space a)) → Set (Either a (Space a))
    rebuild f s = fmap (fmap f) s
sizedP p (Pay a)  k
  | k > 0         = fmap (fmap Pay) $ sizedP p a (k − 1)
sizedP p (Pure a) 0
  | p a           = {Left a}
  | otherwise     = {Right Empty}
sizedP _ _        _ = {}
```

Fig. 2. Definition of a predicate-guided *sized*-function.

```
uniform :: (a → Bool) → Space a → Int → Random a
uniform p s k = do
  x ← uniformSet (sizedP p s k)
  case x of Left  a  → return a
            Right s' → uniform p s' k
```

A complete definition of *sizedP* can be found in Figure 2. The function is implemented by recursion on its *Space a* argument, in a similar way to *sized* from Section 2. One difference is that it also reconstructs the updated space for the values for which the predicate fails. As the recursion proceeds, it composes the predicate with constructor functions (from the :$: constructor). The cases for *Pay*, sum (:+:) and *Empty* follow the pattern set by the definition of *sized*. The case for *Pay* decreases the size parameter for the recursive call, and applies *Pay* to all residual spaces returned by it. The case for (:+:) returns a sum of sets returned by the recursive calls, and similarly applies (:+: *b*) or (*a* :+:) to the residual spaces contained in them. The *Empty* space is handled by the default case returning the empty set {}.

For function applications (:$:), *sizedP* constructs a new predicate p' by composing the input predicate with the applied function. If the new predicate is universally false (*universal* p' returns *Just False*), then *sizedP* returns an empty space. This reflects the fact that for a universally false predicate, there can be no values in the space that satisfy it. If the predicate is universally true or unknown, *sizedP* is called recursively on the underlying space with the updated predicate (this could be optimised to use

sized in the universally true case). The *apply* function applies the function f to either the returned space or the returned value.

A key point of *sizedP* is that the cardinality of the resulting set does not depend on the predicate. In fact, it is always the case that $|sizedP\ p\ s\ k| = |sized\ s\ k|$, in other words the result of *sizedP* contains one element for every value of size k regardless of how many of them that satisfy p. This allows efficient computations of cardinalities through memoisation. To accommodate this in the definition of *sizedP*, we extend our multiset type with a new construct $replicateSet :: Integer \to a \to Set\ a$ that adds a given number of occurrences of a value to the multiset (similar to the Haskell *replicate* function on lists). It is specified as follows:

$$\begin{aligned} &indexSet\ (replicateSet\ n\ a) = a \\ &|replicateSet\ n\ a| \qquad\quad = n \end{aligned}$$

The final case to discuss is the one for $(:*:)$, which is a little more involved, as we need to decide which component of the pair to refine. The following section describes how to make this choice based on the order of evaluation of the predicate.

3.1 Predicate-guided refinement order

When implementing *sizedP* for products, it is no longer possible to divide the size between the components, as was done in the implementation of *sized* (see Section 2). The reason is that it is not possible to split a predicate on pairs into two independent predicates for the first and second components.

We solve this problem using the algebraic nature of our spaces to eliminate products altogether. We can use the following algebraic laws to eliminate products:

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

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 equalities above can be used to define an operator $(***)$ on spaces that pushes top level products inwards without loss of information:

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

Additionally, we need two cases for lifting *Pay* and function application.

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

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 *sizedP* on products by applying the transformation, so $sizedP\ p\ (a\ :*:\ b) = sizedP\ p\ (a\ ***\ b)$. This is problematic, because $(***)$ imposes a right-first order of evaluation, which means that for our generators the first component of a pair is never generated before the right one is fully defined. This is detrimental to performance, since the predicate may not require the right operand to be defined at all. In the end, this would mean that when the predicate is falsified *sizedP* would not remove as many values from the space as it potentially could.

To change this, and 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 *universal* that takes a predicate on pairs:

$$inspectsFst :: ((a, b) \to Bool) \to Bool$$

The expression *inspectsFst p* is *True* iff *p* evaluates the first component of the pair before the second. Just like *universal*, *inspectsFst* exposes some information of the Haskell runtime, which cannot be observed directly.

Thus, to define the final, product $(:*:)$ case of *sizedP* in Figure 2 we combine *inspectsFst* with another algebraic law: commutativity of products. If the predicate 'pulls' at the first component, the operands of the product are swapped before applying the transformation for the recursive call.

The end result is an algorithm that gradually refines a value, by expanding only the part of the space that the predicate needs in order to progress. With every refinement, the space is narrowed down until the predicate is guaranteed to be false for all values in the space, or until a single value satisfying the predicate is found.

3.2 Example

To further illustrate how our algorithm operates, we provide an example of using it to find lambda terms of a given type and size. The example is similar to that in the introduction but slightly simplified for ease of presentation. For the purpose of this example, we define a simple data type representing lambda terms with De Bruin indices.

$$\textbf{data}\ Term = Ap\ (Term, Term) \mid Lam\ Term \mid Var\ Nat$$

We are not required to 'uncurry' the Ap constructor, but it helps the presentation of the example. The space of all lambda terms is defined as follows:

$$\begin{aligned}
&spTerm, spAp, spLam, spVar :: Space\ Term\\
&spTerm = Pay\ (spAp\ :+:\ spLam\ :+:\ spVar)\\
&spAp\ \ \ = Ap\ \ \ :\$:\ (spTerm\ :*:\ spTerm)\\
&spLam\ = Lam\ :\$:\ spTerm\\
&spVar\ \ = Var\ \ :\$:\ spNat
\end{aligned}$$

Furthermore, this example assumes we have a type checking function, which decides whether a lambda term is well scoped and of a given type.

$$\textbf{data}\ \mathit{Type} = \mathit{TInt} \mid \mathit{Type} \mapsto \mathit{Type}$$
$$\mathit{check} :: \mathit{Type} \to \mathit{Term} \to \mathit{Bool}$$

For instance, $\mathit{check}\ (\mathit{TInt} \mapsto \mathit{TInt})\ (\mathit{Lam}\ (\mathit{Var}\ Z))$ is *True* and $\mathit{check}\ \mathit{TInt}\ (\mathit{Lam}\ (\mathit{Var}\ Z))$ is *False*. The type checker is lazy so for instance $\mathit{check}\ \mathit{TInt}\ (\mathit{Lam}\ \bot)$ is *False*. We deliberately omit the implementation of the type checker, and treat it as a black box: We can only observe its behaviour by executing it, and using the functions *universal* and *inspectsFst*.

We will not directly step through the execution of the *uniform* function described earlier in this section. Instead, we show the algorithm as a series of transformations on spaces, which illustrate how indexing is performed.

The first step in every iteration of the algorithm is to transform the space we are currently working with into this normal form:

$$\mathit{Pay}^n\ (f\ \ :\$:\ (s_1\ \ :+:\ \ldots\ :+:\ s_k))$$

Here, Pay^n represents n applications of the *Pay* constructor. The general idea is that the partial value $f\ \bot$ is the result of all the choices we have made up to this point, and the sum is the next choice we would have to make in order to further define the value. To determine if further choices are necessary to decide predicate p, we use the *universal* function from Section 3. If $\mathit{universal}\ (p \circ f)$ is *Nothing*, the space must be further refined before the predicate yields a result. This is done by choosing one of the summands s_i of the sum, weighted by the cardinalities to guarantee uniformity. If the result of *universal* is $\mathit{Just}\ b$, we know that the result of p is b for all values in the space.

Any space can be transformed into this form provided that it contains at least one sum. We call this transformation *lifting choices*. It involves applying the distributivity, associativity, identity and annihilation laws of $:*:$ mentioned earlier in this section, along with the following laws:

$$f\ \ :\$:\ \mathit{Pay}\ s \quad \equiv \mathit{Pay}\ (f\ \ :\$:\ s)$$
$$f\ \ :\$:\ (g\ \ :\$:\ s) \equiv (f \circ g)\ \ :\$:\ s$$

To demonstrate the execution of the algorithm, we consider generating well-typed terms of type $\mathit{Int} \to \mathit{Int}$ of size 11, for which we will use the space *spTerm* and the predicate $p = \mathit{typeCheck}\ (\mathit{Int} \mapsto \mathit{Int})$. The number of lambda terms of size 11 can be computed from the definition of the space (see Section 2), and for our space it is $c = 465$.

To generate a random well-typed term, our *uniformSet* function chooses a candidate index between 0 and $c - 1$ uniformly at random – let us say that it chose 407. The first iteration of the algorithm starts with the space *spTerm*. Transforming it to normal form only requires applying the identity function inside of the *Pay* constructor.

$$\mathit{Pay}\ (\mathit{id}\ \ :\$:\ (\mathit{spAp}\ \ :+:\ \mathit{spLam}\ \ :+:\ \mathit{spVar}))$$

Next we use *universal* to check if the type checker p cares at all about which value we use. In this case, *universal* $(p \circ id)$ is *Nothing*, so we need to define at least some part of the term to know if the predicate holds or not.

The space needs to be refined by committing to one of the three summands in $spAp \ :+: \ spLam \ :+: \ spVar$. The refined spaces corresponding to the choices are as follows:

$$Pay \ (id \ :\$: \ spAp)$$
$$Pay \ (id \ :\$: \ spLam)$$
$$Pay \ (id \ :\$: \ spVar)$$

The choice is made based on the candidate index selected earlier, and the cardinalities of the refined spaces we may choose. For size 11, the cardinalities are 257, 207 and 1 respectively. Since $257 \leqslant 407 \leqslant 257 + 207$, we choose the second space, and obtain a residual index of $407 - 257 = 150$. Expanding the definition of *spLam*, lifting the choice it contains, and eliminating a composition with *id* we get:

$$Pay^2 \ (Lam \ :\$: \ (spAp \ :+: \ spLam \ :+: \ spVar))$$

Next, *universal* $(p \circ Lam)$ evaluates to *Nothing*, indicating that the space needs to be further refined. The refinement is selected by the same procedure as in the previous iteration, now using the residual index from the last choice (150) instead of a random index. Lambda is chosen again yielding $Pay^2 \ (Lam \ :\$: \ spLam)$ as our refined space, in normal form it is

$$Pay^3 \ (Lam \circ Lam \ :\$: \ (spAp \ :+: \ spLam \ :+: \ spVar))$$

At this point, we test *universal* $(p \circ Lam \circ Lam)$, and get *Just False*, which means that there are no lambda terms of type $Int \rightarrow Int$ with two head lambdas.

Now our only option is to discard the current space (the space of terms with two head lambdas), choose a new random index, and restart. To reconstruct the space of remaining terms, we reiterate through the choices we have made to get to this point, and build a space from the sum of all the options we did not choose (see the definition of the :+: case in Figure 2). In this case, we made two choices, and each had two alternative options. Thus the space we construct is this:

$$Pay \ (spAp \ :+: \ spVar) \ :+: \ Pay^2 \ (Lam \ :\$: \ (spAp \ :+: \ spVar))$$

Analysing the space we see that it excludes only the values that start with two lambdas, because we have four choices that represent starting with *Ap* or *Var* or starting with *Lam* followed by *Ap* or *Var*.

The cardinality of this new space is 371, which means that the next candidate index must be in range from 0 to 370. Let us say that this time we choose an index, which leads to *Pay spAp* being selected as the refined space. Expanding the definition of *spAp* leads to $Pay \ (Ap \ :\$: \ (spTerm \ :*: \ spTerm))$, which cannot be trivially transformed into a normal form. A choice needs to be lifted up from one of the operands of the product. This is done by applying either left or right distributivity depending on which component of the pair is requested by the predicate. To determine which component is required, we evaluate *inspectsFst* $(p \circ Ap)$. In our

example, *inspectsFst* returned *True*, so we know that the predicate decided to evaluate the parameter of the function application first. This means that we get our new space from the *** transformation defined earlier (which expands its right operand) and get the following sum:

$$
\begin{aligned}
&Pay\ (Ap\ \ :\$:\ (spTerm\ ***\ spTerm)) \equiv \\
&\quad Pay^2\ (Ap\ \ :\$:\ \ \ (\ \ (spTerm\ \ :*:\ spAp) \\
&\qquad\qquad\qquad\qquad :+:\ (spTerm\ \ :*:\ spLam) \\
&\qquad\qquad\qquad\qquad :+:\ (spTerm\ \ :*:\ spVar)))
\end{aligned}
$$

At this point, further iterations with our particular index will lead to a space with no further choices, whose normal form is a function application on a unit. Since there is only one value in the space in this special case we can simply construct it and type check it. One possible value is $Ap\ (Lam\ (Var\ Z))\ (Ap\ (Lam\ (Var\ Z))\ (Lam\ (Var\ Z)))$, or $(\lambda x \to x)\ ((\lambda x \to x)\ (\lambda x \to x))$ in a more readable syntax. As the predicate p answers *True* for it, it is a term of the type we were looking for.

4 Uniformity of the generators

It is easy to prove that *uniformSet s* is uniform over the occurrences in s, by proving that *indexSet s* is bijective and that *uniformRange* (lo, hi) is uniform over the inclusive interval (lo, hi). Many of the subsequent proofs in this section rely on the *Set* type being an accurate representation of multisets, so properties like commutativity and distributivity of union on *Set* follow from the corresponding theorems for multisets. These properties should be straightforward to prove by providing bijections between the indexes of *indexSet*.

From these preliminaries, we can prove that the rejection sampler *uniformFilter p s k* (described in Section 3) provides uniform sampling of values of size k from the space s, constrained by the predicate p.

Theorem 1
Consider space s, a non-negative size k, and a predicate p. Let the multiset $a =$ *sized s k* and $b = \{x \mid x \in a, p\ x\}$. If b is non-empty, then *uniformFilter p s k* is uniform over b.

Proof
Let $n = |a|$, $m = |b|$ and x be an occurrence in b. We will calculate the probability of $x =$ *uniformFilter p s k*. In every iteration of *uniformFilter*, a uniformly random value y is drawn from a. The probability that $y \notin b$ (causing the procedure to retry) is $(n - m) / n$, whereas the probability that $y = x$ is $1 / n$. Thus, the probability of x being drawn during the ith iteration is equal to $(1/n)((n-m)/n)^{i-1}$. Finally, the probability of x being drawn after any number of retries becomes the limit of the geometric series $(1/n)\sum_{i=0}^{\infty}((n-m)/n)^i = 1/m$. □

Proving uniformity of the predicate-guided *uniform* function is more difficult, as the mapping from indexes to values depends on the evaluation order of the predicate. Particularly, the space is transformed differently depending on which component

of a pair is inspected first. The two possible resulting spaces will contain the same values but in different internal ordering. However, if we assume that the evaluation order of the predicate is deterministic, this also determines a fixed order of elements in the space, which defines an injective mapping.

First, we need a lemma that the $(***)$ operator preserves equality of spaces if we disregard the order of elements returned by indexing of multisets.

Lemma 1
Let $a :: Space\ a$ and $b :: Space\ b$ be two spaces, and $k :: Int$ a non-negative integer. Then, the following equivalence holds between multisets:

$$sized\ (a\ :*:\ b)\ k = sized\ (a *** b)\ k$$

Proof sketch
A straightforward proof by structural induction on b is possible. Each of the cases can be proven using the corresponding laws for multisets and *Pay*. □

The next lemma contains most of the complexity of the proof. It shows that values of the form $Left\ x$ in the multiset returned by $sizedP\ p\ s\ k$ contains exactly the subset of values returned by $sized\ s\ k$ that satisfy p.

Lemma 2
Let $s :: Space\ a$ be a space, $p :: a \to Bool$ a boolean predicate, and $k :: Int$ a non-negative integer. Then, the following equivalence holds between multisets:

$$\{x \mid Left\ x \in sizedP\ p\ s\ k\} = \{x \mid x \in sized\ s\ k, p\ x\}$$

Proof sketch
The proof is carried out by induction and case analysis of *sizedP*. The most interesting part of the proof are cases for $a\ :*:\ b$ and $f\ :\$:\ a$. We first show the sketch for $a\ :*:\ b$, which requires proving the following equation.

$$\{x \mid Left\ x \in sizedP\ p\ (a\ :*:\ b)\ k\} = \{x \mid x \in sized\ (a\ :*:\ b)\ k, p\ x\}$$

Out of the two subcases, we will only show the more complex case when $inspectsFst\ p$, using equational reasoning. Note that we only consider the recursive case of the second *sizedP* invocation here. It can be shown that the other case, when $universal\ p' = Just\ False$, results in both sides of the equation being equal to $\{\}$.

$\{x \mid Left\ x \in sizedP\ p\ (a\ :*:\ b)\ k\}$
 {-Definition of sizedP -}
$\{x \mid Left\ x \in sizedP\ p\ (swap\ :\$:\ (b *** a))\ k\}$
 {-Definition of sizedP (recursive case) -}
$\{x \mid Left\ x \in \{apply\ swap\ y \mid y \in sizedP\ (p \circ swap)\ (b *** a)\ k\}\}$
 {-Simplification -}
$\{swap\ x \mid Left\ x \in sizedP\ (p \circ swap)\ (b *** a)\ k\}$
 {-Induction hypothesis -}
$\{swap\ x \mid x \in sized\ (b *** a)\ k, (p \circ swap)\ x\}$
 {-From Lemma 1 -}

$\{swap\ x \mid x \in sized\ (b\ :*:\ a)\ k, (p \circ swap)\ x\}$
{-Commutativity of space products -}
$\{x \mid x \in sized\ (a\ :*:\ b)\ k, p\ x\}$

Notably, to be well founded this inductive step requires an external convergence measure where $b *** a$ is smaller than $a\ :*:\ b$, which is a little intricate to construct. Without going into extreme detail, we present an outline of such a measure. The measure consists of three components (k, r, q), ordered lexicographically from left to right. The component k is the size parameter of *sizedP*. The measure r keeps track of how many steps the algorithm might need to take before the next *Pay* constructor is consumed, and corresponds, more less, to the longest path without a *Pay* constructor. The measure q keeps track of the number of times that $***$ needs to be applied to remove the $:*:$ from the top level of the space. The measure q depends on the predicate p, and for a space that is a tree of nested products, q is the position (in a breadth first order) of the leaf that is requested by the evaluation of the predicate, as indicated by repeated application of *inspectsFst*.

With this definition, it can be proven that for a given invocation of *sizedP* with measure $m = (k, r, q)$, the measure for a recursive call performed by it is strictly smaller than m. For the $:*:$ case above, the measure k for the recursive call is the same as for the original call, but either r or q is always decreased.

One complication in the proof is that fact that $a *** b$ might add an extra $:\$:$ constructor to the result space, which is immediately removed by the next invocation of *sizedP*. The convergence measure needs to account for this fact.

Furthermore, the convergence measure requires the evaluation of the predicate to be deterministic, as otherwise the repeated application of the $***$ operator might lead to an infinite loop. Specifically, *inspectsFst* is required to interact in a standard way with functions like *swap*, for example satisfying the law $inspectsFst\ p \Rightarrow not\ (inspectsFst\ (p \circ swap))$.

The next case of *sizedP* that we will consider is the $f\ :\$:\ a$ case, which requires proving the following equation:

$$\{x \mid Left\ x \in sizedP\ p\ (f\ :\$:\ a)\ k\} = \{x \mid x \in sized\ (f\ :\$:\ a)\ k, p\ x\}$$

The subcase when $universal\ (p \circ f) = Just\ False$ yields both sides of the equation to be equal to $\{\}$. The recursive subcase can be proven using equational reasoning.

$\{x \mid Left\ x \in sizedP\ p\ (f\ :\$:\ a)\ k\}$
{-Definition of sizedP -}
$\{x \mid Left\ x \in \{apply\ f\ y \mid y \in sizedP\ (p \circ f)\ a\ k\}\}$
{-Simplification -}
$\{f\ x \mid Left\ x \in sizedP\ (p \circ f)\ a\ k\}$
{-Induction hypothesis -}
$\{f\ x \mid x \in sized\ a\ k, (p \circ f)\ x\}$
{-Simplification -}
$\{x \mid x \in \{f\ y \mid y \in sized\ a\ k\}, p\ x\}$
{-Definition of sized -}
$\{x \mid x \in sized\ (f\ :\$:\ a)\ k, p\ x\}$

The recursive call in this case has the same measure k as the original call, but measure r is reduced by one.

The remaining cases can be proved in a similar way. □

If indexing in the result of *sizedP* hits an element that does not satisfy the predicate, the result is *Right s*, where s is the residual space, which is used by *uniform* to continue sampling. We show that the spaces returned by *sizedP* retain all elements from the original space that satisfy the predicate, and are strictly smaller than the original space.

Lemma 3
Let s be a space, p a boolean predicate, k a non-negative integer, and $Right\ s' \in sizedP\ p\ s\ k$. Then the following equation between multisets holds.

$$\{x \mid x \in sized\ s\ k, p\ x\} = \{x \mid x \in sized\ s'\ k, p\ x\}$$

Furthermore, $sized\ s'\ k$ is a proper subset of $sized\ s\ k$.

The proof of the above lemma is by induction and case analysis on *sizedP*, similarly to the proof of Lemma 2.

The final theorem shows the uniformity of the distribution of elements returned by *uniform*.

Theorem 2
Consider space s, a non-negative size k, and a predicate p. Let the multiset $a = sized\ s\ k$ and $b = \{x \mid x \in a, p\ x\}$. If b is non-empty, then $uniform\ p\ s\ k$ is uniform over b.

Proof
Let $n = |a|$, $m = |b|$ and x be an occurrence in b. We will show that the probability of $x = uniform\ p\ s\ k$ is equal $1/m$, by performing induction on n. When $uniform\ p\ s\ k$ executes, first $sizedP\ p\ s\ k$ is used to construct a multiset of indexing results, then one element of this set is selected uniformly at random using *uniformSet*. For the base case of the induction, we take that $n = m$. Then, by Lemma 2, all occurrences from the multiset are of the form $Left\ y$, and exactly 1 of them is the occurrence of $Left\ x$. Thus, the probability of generating x is $1/m$.

For the induction step, assume that $n > m$. From Lemma 2, $Left\ x$ is an occurrence in $sizedP\ p\ s\ k$ corresponding to x. Moreover, $n - m$ elements are of the form $Right\ s'$, since $|sizedP\ p\ s\ k| = |sized\ s\ k|$. Thus, the probability of generating x directly is $1/n$, whereas the probability of retrying is $(n - m)/n$. The *uniform* procedure will retry with a modified space s'. From Lemma 3, $sized\ s'\ k$ contains the same elements that satisfy p as $sized\ s\ k$, but fewer elements that do not satisfy p. We can now invoke the induction hypothesis, which gives that the probability of generating x by $uniform\ p\ s'\ k$ is $1/m$. Thus, the overall probability of generating x is $1/n + ((n - m)/n)(1/m) = 1/m$ □

4.1 Non-deterministic predicates

The proof above assumes that the evaluation order of the predicate is deterministic. There are two reasons for this. Firstly, there is a risk of non-termination when the

order of evaluation of the predicate is not consistent between different invocations of *inspectsFst*. This problem can be addressed by introducing a generalised *inspectsFst* that returns the index in a nesting of pairs that a predicate inspects, and a generalised associativity transformation.

Even so, there is a more subtle problem with non-deterministic evaluation order – it may cause biased distribution of the generated data. For example, consider a space containing the four total values of type $(Bool, Bool)$, and a predicate with non-deterministic evaluation order defined on this type.

```
spaceBool :: Space Bool
spaceBool = Pay (Pure False :+: Pure True)
spacePairB :: Space (Bool, Bool)
spacePairB = Pay ((,) :$: spaceBool :*: spaceBool)
oracle :: IO Bool
nonDetB :: (Bool, Bool) → Bool
nonDetB (a, b) = unsafePerformIO $ do
  x ← oracle
  if x then a `pseq` b `pseq` True
       else b `pseq` a `pseq` True
```

The predicate has access to non-deterministic *oracle* returning a boolean value. If the oracle returns *True*, the predicate evaluates the first component of the pair using *pseq*, then the second one before returning *True*. Otherwise, it evaluates the pair's components in the opposite order.

Depending on the value returned by *oracle*, indexing values of size 3 in *spacePairB* with the predicate *nonDetB* yields values in one of two orders.

oracle result	Index:	0	1	2	3
True		$(False, False)$	$(False, True)$	$(True, False)$	$(True, True)$
False		$(False, False)$	$(True, False)$	$(False, True)$	$(True, True)$

Now suppose that *oracle* contains a race condition, which is triggered by events in the part of the program that selects the randomly chosen index, and results in the following: if the index is 1 then *oracle* returns *False*, otherwise it returns *True*. Then, indexing using *nonDetB* will yield $(True, False)$ for indices 1 and 2, whereas $(False, True)$ will never be returned, leading to a biased distribution.

Although it is hard to implement *oracle* so it exhibits this behaviour making this particular example largely hypothetical, it highlights the risks of allowing evaluation order to influence semantics. In Section 5.4, we discuss an alternative algorithm with deterministic indexing. For our main algorithm, this example demonstrates the need for assuming deterministic evaluation order, or possibly a weakened assumption that the evaluation order is independent from the choice of index.

5 Efficient implementation and alternative algorithms

The previous sections give a high-level description of our algorithm. Implementing it required making a number of engineering choices, some of which had a considerable effect on the performance of the generator.

This section describes some of these choices that we found most important for performance, and also experiments with alternative versions of the core algorithm aimed at improving performance.

5.1 Relaxed uniformity constraint

When our uniform generator encounters a subspace for which the predicate is false, the algorithm must retry with a new random index in the reconstructed set. The new index must be chosen independently from the old in order to achieve uniform distribution. We have implemented two alternative algorithms that violate this restriction, compromising uniformity, in favour of better performance.

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 6 contains experimental verification of these hypotheses.

5.2 Parallel conjunction

It is possible to improve the generation performance by introducing the parallel conjunction operator (Runciman *et al.* 2008), 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 *universal r* is *Just False* and *universal q* is *Nothing*, then the result of *universal 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: $\perp$ &&& *False* = *False*. This may make the *sizedP* function 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 define parallel disjunction that is *True* when either operand is *True*.

Note that the parallel conjunction and disjunction still have the same evaluation order as their normal counterparts, that is when both operands are undefined, the left one is evaluated first.

5.3 Sharing in the representation of spaces

The implementation of the algorithm described in Sections 2 and 3 starts with a compact representation of the whole search space, where recursive references to the space are shared. The representation is subsequently expanded, and subspaces are created from it as a result of refinement.

We found it important to ensure that as much sharing as possible is achieved between the representations of subspaces in order to save memory, and share the results of the cardinality computations. The measures used to increase sharing included folding subspaces that have no choices left in them into single units, and rebalancing the tree representation of spaces.

Increasing sharing turned memory-bound computations into CPU-bound ones, while improving run time performance at the same time. As a result, most benchmarks that we ran, including the ones presented in Section 6 were limited by the run time rather than the available memory.

5.4 Deterministic indexing

As demonstrated in Section 4, allowing evaluation order to influence indexing is potentially problematic if the evaluation order is non-deterministic, for instance in parallel computations.

To address this problem, we propose to make the mapping of indices to values in the space independent of the predicate, restricting non-determinism to affect only which falsifying values are removed in an iteration of the top-level algorithm. This requires significant modifications of the algorithm, which involve replacing *sizedP* with three distinct steps:

1. A deterministic indexing procedure produces a tree structure containing all indexing choices (left or right operands of a union) required to produce a random total value, without considering the predicate at all.
2. A non-deterministic procedure prunes this tree, keeping only the choices required to build a partial value for which the given predicate terminates.
3. A subtraction procedure removes all values resulting from the pruned tree from the space.

The tree structure that containing indexing choices can be defined by this data type:

data *Select* = *This* | *Fst Select* | *Snd Select* | *Pair Select Select*

The *Fst* and *Snd* constructors mark the selection of the first or the second component of a union respectively. The *Pair* constructor combines choices made in the components of a product space. With this data type, the three steps above would correspond to functions of these types:

sizedDet :: *Space a* → *Int* → *Set Select*
pruneChoices :: (*a* → *Bool*) → *Space a* → *Select* → *Select*
subtractChoices :: *Space a* → *Select* → *Space a*

In addition, a fourth function is needed that returns the value identified by the choices.

selectedValue :: *Space a* → *Select* → *a*

Here, we discuss implementation of these functions, which is not included in the paper. Function *sizedDet* is a simple adaptation of *sized* (Section 2.3). Function *pruneChoices* uses *inspectsFst*, *universal* and *** similarly to how they are used in *sizedP* (Figure 2) in order to prune the tree of choices. Function *subtractChoices* performs algebraic transformations to remove the subspace specified by the choice tree from the original space. Function *selectedValue* is implemented using structural recursion on both arguments.

Preliminary experimentation with this approach showed that implementing *subtractChoices* efficiently is key for efficiency. Several implementations of it were tried but none were as fast as the original algorithm. Further experimentation with this approach remains a topic of future work.

5.5 In-place refinement

The algorithm described in Section 3 is implemented by applying the predicate to partial values and by throwing and catching exceptions, determine which part of the value needs to be further defined (this is the inner workings of *inspectsFst* and *universal*). This process might be computationally expensive as it requires repeated evaluation of the predicate.

As an alternative mechanism for observing the evaluation order of predicates, we experimented with a variant of the algorithm that uses Haskell's lazy evaluation with fully defined values. In this algorithm, the indexing function directly builds a random fully defined value, and attaches a Haskell IO-action to each subcomponent of it. When the predicate is applied to the value, the IO-actions will fire only for the parts that needs to be inspected to determine the outcome. Whenever the indexing function is required to make a choice, the corresponding IO-action records the option it took. After the predicate has terminated, the pruned choice tree can be constructed from the recorded trace.

This approach has the advantage of deterministic indexing, and reduces the maximal number of times the predicate is executed for each iteration of the algorithm from n to 1 (where n is the number of choices made, usually proportional to the

size of the value). In particular, the exact value returned by the indexing function will not depend on the evaluation order of the predicate.

The algorithm based on in-place refinement can be summarised as follows:

1. Sample the space for a lazily defined value uniformly at random. Inspecting any constructor of the sampled value makes a record in the trace.
2. Execute the predicate on the value.
3. If the result of the predicate is *True*, return the generated value, otherwise continue.
4. Determine which parts of the value were inspected by examining the trace. This information determines which choices had to be made by the indexing function, and which are redundant.
5. *Subtract* the space of all values from the previous point from the current space, and restart the algorithm.

Despite the clear advantage of not having to re-evaluate the predicate on many partial values for each falsifying total value, the generator based on this technique turned out to be slower than our original implementation for the predicates and spaces we used. On the other hand, this generator used less memory in most cases compared the original one.

In addition to the performance problems, 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 these reasons, our implementations of in-place refinement remains a separate branch of development and a topic of future work.

6 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 handwritten one in triggering strictness bugs in the GHC compiler. Some benchmarks were also run with a naive generator that generates random values from a space, as in Section 2, and filters out those that do not satisfy a predicate.

6.1 Trees

Our first example is binary search trees 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
```

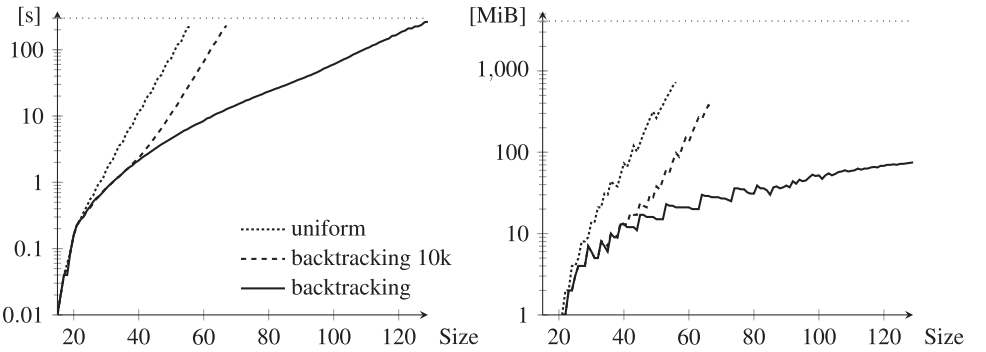

Fig. 3. Run times in (left) and memory consumption (right) of derived generators generating 2000 binary search trees depending on the size of generated values.

The *isBST* predicate decides if the tree is a binary search tree, and uses a supplied lazy comparison function for type *Nat* for increased laziness.

```
isBST :: Ord a ⇒ Tree a → Bool
isBST t  = aux Nothing Nothing t where
  Nothing ⊴ y  = True
  Just x  ⊴ y  = x ⩽ y
  x ≤ Nothing = True
  x ≤ Just y  = x ⩽ y
  aux _ _  L            = True
  aux lb ub (N x t1 t2) = lb ⊴ x  && x ≤ ub
    && aux lb (Just x) t1  && aux (Just x) ub t2
```

The predicate's auxiliary function accepts two optional bounds and a subtree and decides whether the subtree is a binary search tree with all elements within the bounds.

The benchmark involved measuring 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 5.1) were used: one performing uniform sampling (*uniform*), one bounded backtracking allowed to skip at most 10 k values (*backtracking 10 k*), and one performing unbounded backtracking (*backtracking*). A naive generate-and-filter generator was also used for comparison.

Both *backtracking 10 k* and *backtracking* generators produce non-uniform distributions of values. The skew of the *backtracking 10 k* generator is limited, as the least likely values are generated at most 10 k times less likely than the most common ones, as mentioned in Section 5.1.

Figure 3 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 10 k* 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.

6.2 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 handwritten recursive generator for them requires the use of backtracking, because of the inability of predicting whether a given local choice 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 (Pałka 2012).

On the other hand, obtaining a generator that is based on our framework requires only the definitions from Figure 1, and a space definition. Defining the space of closed expressions *spaceClosedExprs* also requires auxiliary definitions of spaces containing open expressions.

```
spaceExprs :: [Space Expr]
spaceExprs = go 0 where
  go k = let go_k_1 = go (k + 1)
             go_k   = Pay  $  (Ap  :$: go_k  :*: go_k  :*: spaceType)
                           :+: (Vr  :$: fin k, Lm  :$: head go_k_1)
         in go_k : go_k_1
spaceClosedExprs = head spaceExprs
```

To ensure sharing, we define the top-level list *spaceClosedExprs* of spaces of expressions with 0, 1, 2, and so on free variables. The definition of the *n*th space refers to the $n + 1$-th space, as the *Lm* constructor requires its body to be an expression with one more free variable.

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 six constants. The derived generator with three search strategies and one based on generate-and-filter were used. Figure 4 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 with 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 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.

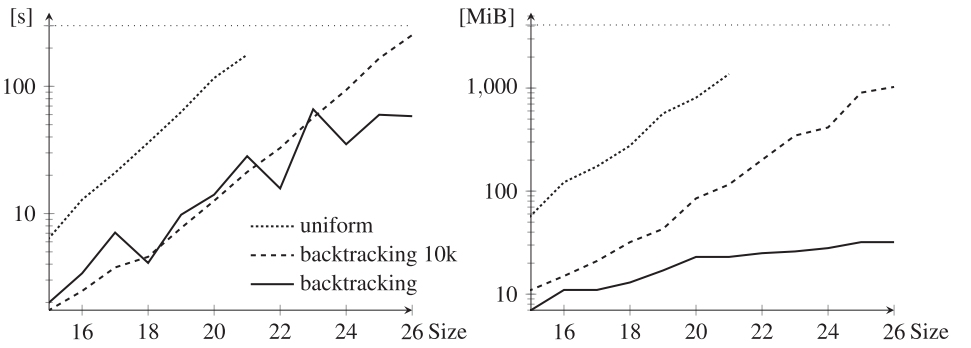

Fig. 4. Run times (left) and memory consumption (right) of derived generators generating 2000 simply typed lambda terms depending on the size of generated terms.

6.3 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 handwritten generator that had been developed before (Pałka 2012) 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 Figure 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 backtracking generator based on this data type was capable of generating terms containing 30 term constructors, and was able to trigger GHC failures. Other derived generators were not able to find counterexamples. Table 1 shows the results of testing GHC both with the handwritten simply typed lambda term generator and our derived generator. The handwritten 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 seven times more CPU time per failure than the handwritten 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 handwritten one, but is significantly easier to develop. Developing an efficient type-checking predicate required for the derived generator took a few days, whereas the development and tuning of the hand-written generator took an order of months.

Table 1. *Performance of the reference handwritten 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*

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

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

6.4 Programs

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

```
type Name    = String
data Program = New Name Program
             | Name := Expr
             | Skip
             | Program ≫ Program
             | If Expr Program Program
             | While Expr Program
data Expr    = Var Name
             | Add Expr Expr
```

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 five 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.

6.5 Summary

All derived generators performed much better than ones based on generate-and-filter in three out of four benchmarks. In the GHC benchmark, 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. The backtracking generator was the only automatic generator that found any counterexamples, although less efficiently than a handwritten generator. However, as creating the derived generators was much quicker, we consider them an appealing alternative to handwritten generators.

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 3 and 4.

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 6.4 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.

7 Related work

There is a substantial amount of research on generating combinatorial structures both in pure mathematics and in computer science. These structures sometimes include trees or even (recursive) ADTs. Although this work does not deal with data constrained by arbitrary predicates, some of it could potentially be adapted to it in a similar way as we do in this paper.

An efficient algorithm to index into a size-based enumeration of binary trees can be derived from a bijection to strings of nested parenthesis (Knuth 2006). Boltzmann

samplers can be used to generate objects from a wide range of combinatorial structures, with uniform distribution over values of an approximate or exact size (Flajolet *et al.* 1994, 2007). Yorgey has explored the relation between a class of combinatorial objects called *species* and ADTs (Yorgey 2010, 2014). This work can potentially be used for uniform random generation of algebraic types as well as some more complex structures involving symmetries.

7.1 FEAT

Our representation of spaces and efficient indexing is based on FEAT (Duregård *et al.* 2012). 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.

7.2 Lazy SmallCheck and Korat

Lazy SmallCheck (Runciman *et al.* 2008) 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 (Boyapati *et al.* 2002). 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.

SmallCheck has been applied to the problem of generating programs to test compilers (Reich *et al.* 2012). The work focuses on limiting the search space to include interesting programs without containing too many variants of similar programs. Notably some of these limitations, such as limiting function arity, arise from the use of depth bound as opposed to size bound.

7.3 Lazy instantiation

A framework for generating values satisfying a computable predicate has been proposed based on explicit term representation of computable predicates (Lindblad 2008). It uses logic variables to represent unrefined parts of the input data, and performs backtracking search with their successive refinement. The framework performs reductions of the predicate program explicitly, and shares its intermediate results for similar arguments, which may be beneficial for computationally expensive predicates. The framework can be adapted to perform random search, but only limited experiments have been performed on it.

7.4 EasyCheck: test data for free

EasyCheck is a library for generating random test data written in the Curry functional logic programming language (Christiansen & Fischer 2008). 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, micromanagement 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.

7.5 Metaheuristic search

In the GödelTest (Feldt & Poulding 2013) 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.

7.6 Lazy non-determinism

There is some recent work on embedding non-determinism in functional languages (Fischer *et al.* 2011). 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 is very general and could potentially be used both for implementing SmallCheck style enumeration and for random generation.

7.7 Generating lambda terms

There are several other attempts at enumerating or generating well-typed lambda terms. Generic programming has been used to exhaustively enumerate lambda terms by size (Yakushev & Jeuring 2009). 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 (Grygiel & Lescanne 2013). This includes generating well-typed terms by a simple generate-and-filter approach.

8 Conclusion

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 6.4. Similar effects have been observed in Korat (Boyapati *et al.* 2002), 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.

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, and often requires careful reasoning about its strictness and evaluation order.

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.

Acknowledgments

This research has been supported by the Resource-Aware Functional Programming grant awarded by the Swedish Foundation for Strategic Research. We would like to thank David Christiansen for his valuable feedback, and anonymous referees for their detailed and helpful reviews.

References

Arts, T., Hughes, J., Johansson, J. & Wiger, Ulf. (2006) Testing telecoms software with Quviq QuickCheck. In Proceedings of the Workshop on Erlang (Erlang 2006). New York, NY, USA: ACM, pp. 2–10.

Boyapati, C., Khurshid, S. & Marinov, D. (2002) Korat: Automated testing based on Java predicates. In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2002). New York, NY, USA: ACM, pp. 123–133.

Christiansen, J. & Fischer, S. (2008) EasyCheck: Test data for free. In Proceedings of the International Conference on Functional and Logic Programming (FLOPS 2008). LNCS, vol. 4989. Berlin, Heidelberg: Springer, pp. 22–336.

Claessen, K., Duregård, J. & Pałka, M H. (2014) Generating constrained random data with uniform distribution. In Proceedings of the International Conference on Functional and Logic Programming (FLOPS 2014), Codish, M. & Sumii, E. (eds), LNCS, vol. 8475, Springer International Publishing, pp. 18–34.

Claessen, K. & Hughes, J. (2000) QuickCheck: A lightweight tool for random testing of Haskell programs. In Proceedings of the International Conference on Functional Programming (ICFP 2000). New York, NY, USA: ACM, pp. 268–279.

Duregård, J., Jansson, P. & Wang, M. (2012) Feat: Functional enumeration of algebraic types. In Proceedings of the Haskell Symposium (Haskell 2012). ACM, pp. 61–72.

Feldt, R. & Poulding, S. (2013) Finding test data with specific properties via metaheuristic search. In Proceedings of the International Symposium on Software Reliability Engineering (ISSRE 2013). IEEE, pp. 350–359.

Fischer, S., Kiselyov, O. & Shan, C.-C. (2011) Purely functional lazy nondeterministic programming. *J. Funct. Program.* **21**(4–5), 413–465.

Flajolet, P., Éric F. & Pivoteau, C. (2007) Boltzmann sampling of unlabelled structures. In Proceedings of the Workshop on Analytic Algorithmic and Combinatorics. SIAM Press, pp. 201–211.

Flajolet, P., Zimmermann, P. & Cutsem, B. V. (1994) A calculus for the random generation of labelled combinatorial structures. *Theor. Comput. Sci.* **132**(1–2), 1–35.

Grygiel, K. & Lescanne, P. (2013) Counting and generating lambda terms. *J. Funct. Program.* **23**(9), 594–628.

Knuth, D. E. (2006) *The Art of Computer Programming, volume 4, fascicle 4: Generating All Trees–History of Combinatorial Generation (Art of Computer Programming)*. Addison-Wesley Professional.

Lindblad, F. (2008) Property directed generation of first-order test data. In Trends in Functional Programming (TFP 2007). Intellect, pp. 105–123.

Pałka, M. H. (2012) *Testing an Optimising Compiler by Generating Random Lambda Terms*. Licentiate thesis, Chalmers University of Technology, Gothenburg, Sweden.

Pałka, M. H., Claessen, K., Russo, A. & Hughes, J. (2011) Testing an optimising compiler by generating random lambda terms. Proceedings of the International Workshop on Automation of Software Test (AST 2011). ACM, pp. 91–97.

Reich, J. S, Naylor, M. & Runciman, C. (2012) Lazy generation of canonical test programs. In Implementation and Application of Functional Languages (IFL 2012). LNCS, vol. 7257, Springer, pp. 69–84.

Runciman, C., Naylor, M. & Lindblad, F. (2008) Smallcheck and Lazy Smallcheck: Automatic exhaustive testing for small values. In Proceedings of the Haskell Symposium (Haskell 2008). New York, NY, USA: ACM, pp. 37–48.

Yakushev, A. R. & Jeuring, J. (2009) Enumerating well-typed terms generically. In Proceedings of International Workshop of Approaches and Applications of Inductive Programming (AAIP 2009). LNCS, vol. 5812. Springer, pp. 93–116.

Yorgey, B. A. (2010) Species and functors and types, oh my! In Proceedings of the Haskell Symposium (Haskell 2010). New York, NY, USA: ACM, pp. 147–158.

Yorgey, B. A. (2014) *Combinatorial Species and Labelled Structures*. Ph.D. thesis, University of Pennsylvania.