Fast and Sound Random Generation for Automated Testing and Benchmarking in Objective Caml

Benjamin Canou Alexis Darrasse *

Équipe APR - Département CALSCI
Laboratoire d'Informatique de Paris 6 (CNRS UMR 7606)
Université Pierre et Marie Curie (UPMC - Paris 6)
4 place Jussieu, 75005 Paris, France
{benjamin.canou,alexis.darrasse}@lip6.fr

Abstract

Numerous software testing methods involve random generation of data structures. However, random sampling methods currently in use by testing frameworks are not satisfactory: often manually written by the programmer or at best extracted in an ad-hoc way relying on no theoretical background. On the other end, random sampling methods with good theoretical properties exist but have a too high cost to be used in testing, in particular when large inputs are needed.

In this paper we describe how we applied the recently developed Boltzmann model of random generation to algebraic data types. We obtain a fully automatic way to derive random generators from Objective Caml type definitions. These generators have linear complexity and, the generation method being uniform, can also be used as a sound sampling back-end for benchmarking tools.

As a result, we provide testing and benchmarking frameworks with a sound and fast generation basis. We also provide a testing and benchmarking library, available for download (1), showing the viability of this experiment.

Categories and Subject Descriptors D.3.4 [*Programming languages*]: Processors; G.2.1 [*Discrete Mathematics*]: Combinatorics; D.2.5 [*Software Engineering*]: Testing and Debugging—Testing tools

General Terms Languages, Algorithms, Verification

Keywords Random generation, specification-based testing, Algebraic Data Types, Boltzmann model

1. Introduction

Testing has always been a major part of software development but also one of the most tedious. In particular, testing algorithms requires the programmer to write a lot of input data to cover a significantly wide amount of cases. The same input generation problem appears in benchmarking: when one wants to compute a practical average efficiency. Moreover, in this case, one also wants the generated input to uniformly cover the set of all possible inputs of a given size in order to soundly obtain statistical properties.

In our context of statically typed languages with algebraic data-types, we have an exact and complete formal definition of the shape of the values (types such as integers, strings, etc. will be treated as leaves throughout this paper). It is then straightforward that values of such a type can be automatically generated by extracting random generators from type definitions. Moreover, functional languages appear to be a very good environment for fully automatic specification-based testing techniques, as shown by QuickCheck [Claessen and Hughes 2000] (2)[1].

However even if there have been numerous experiments on fully automatic testing, the random generation methods currently in use are not satisfactory enough. Input data is generated

- by handcrafted generators,
- by automatically extracted generators with no theoretical background to ensure probabilistic properties of the generation, or
- by a constraint-based method which is better on a theoretical point of view but too expensive and restricts it to small sizes.

For all of these methods, it is usually extremely difficult to calculate the probability for a given value to appear and thus to estimate the bias of the generation.

Creating random generators with good properties is one of the topics in the field of combinatorics. The methods it proposed until recently were however not applicable in this context. They were either specific to some structure and thus not automatically applicable to large enough classes of objects or in the case of the recursive method [Flajolet et al. 1994] too costly (with a simple implementation) or too complicated.

In this paper, we propose a solution to solve this problem by using the recently developed Boltzmann model of random generation from the combinatorics research field. The Boltzmann model provides uniform random sampling, meaning that each object in

* work partially supported by ANR contract GAMMA, n°BLAN07-2_195422

ML'09, August 30, 2009, Edinburgh, Scotland, UK.

[1] We use the notation (n) to reference external links; see section Links at the end of the document for full URLs.

the selected class has the exact same probability to be produced. Moreover, the generators it produces are very simple and have a linear-time complexity. We designed a translation from Objective Caml [Leroy et al. 2008] (3) algebraic data-types to combinatorics specifications, and use it to automatically extract generators from type definitions.

Practically, we provide testing framework designers with a random sampling core for generating (possibly very large) objects that respect a given algebraic data-type. The probability for an object to appear is known and the cost of the generation is linear to the size of the object. We integrated this generation into Objective Caml via a syntax extension and tested its viability on a testing and benchmarking library prototype.

This work can be directly adapted to any functional language. It can also be used to generate tree-like structures in any programming language.

In section 2, we give an overview of software testing methods, explain how our work can interact with them and present some related work in typed functional languages. Section 3 first gives the theoretical background and section 4 explains the application to random sampling of algebraic data-types in Objective Caml. Then, section 5 demonstrates the practical applications of our random generation core and our associated testing and benchmarking library prototype in the form of a tutorial. We then give some performance results on the generation speed in section 6. Finally, we present our future work on this project as well as other ongoing experiments involving the Boltzmann model and programming languages.

2. Context

In this section, we describe the many sorts of source code level program testing frameworks. We then explain how these framework can integrate our work, and why they should.

Program testing The most widespread software testing technique in use in software development industry is called *unit testing*. In a regular industrial development environment, the work is organized around a well defined methodology called a *development model*. The most venerable is the *V-Model*, which splits the software development work in layers of abstraction, going from global conception of the system to detailed source code level development. To each conception task, this model associates a verification one. Unit testing consists in finding tests cases and procedures for each unit of code (methods, functions, procedure, etc. depending on the programming language). It is therefore the counterpart of detailed software conception in this model.

Independently from, and for best results in addition to, unit testing, there exists another method usually called *specification-based testing*. This approach tests a function by applying a validation predicate over pairs of values from its domain and the associated results. Whereas the goal of unit testing was to check that the function performs correctly on hand-written typical or pathological input values, specification-based testing tries to find bugs by checking the results of the function over a significant amount of inputs. The problem, which we are not addressing, is then how to define which inputs are *significant* and how to obtain them.

If the domain of the function to test is finite and small enough, the programmer can write every case and then obtain an exhaustive test. Since we focus on a functional language with well defined algebraic data-types, the procedure to obtain all the values of a given finite type can be automatically derived from its definition. By defining a size operator over a non-finite type, exhaustive testing can also be used to test the function for inputs up to a given size. This approach has already been experimented in [Runciman et al. 2008] for the Haskell language.

The problem is then how to obtain *meaningful* sample inputs of large size. There are several methods for generating random values of a given type. In most test frameworks, these values are created by hand-crafted generators. QuickCheck, for example, defines several random generators for basic types and combinators to build from them generators for more complex types. With such a manual approach, it is very hard not to bias the form of generated values, and thus to unknowingly concentrate the domain of tested values to an arbitrary subset of values. A solution to control the shape of generated values is to use constraints-based random generation. However, it is not a viable solution in terms of generation time for large values.

Our proposition The solution we propose is to use uniform random generation. In other words, a generator of values of a given size gives each value of this size the exact same probability to appear. With such an approach, we know that we are concentrating on the subset of the most representative values. To achieve this property, generators are not hand-written but automatically derived from a type definition as we explain in the following section. Moreover, the derived generators are capable of producing values of a given size in linear complexity. Furthermore, we provide a mechanism of adding a measurable bias on the distribution, in order to target different value subsets.

Applications As a direct result, our work can provide any kind of testing framework based on random generation with a sound and fast generation basis, in particular for inputs of large size.

Moreover, as we have just said, we use uniform (unbiased) random generation. From a statistical point of view, this means that if we run a test over a great number of generated values, we can obtain statistical properties over the values of given type and size.

In particular, we can obtain a good evaluation of average performance of programs. This can be used to obtain the practical performance of an algorithm, since obtaining a theoretical average complexity is often tedious, and many algorithms with bad worst-case complexity can be very efficient in practice because of their average behaviour. Moreover, it can reveal errors like complexity miscalculations due to the use of an unsuited underlying primitive data structure. Our work can thus be used use as well by mean-time complexity checking or benchmarking tools.

Related works In this paper, we argue that the most widesreapd testing frameworks lack a sound generation basis. We propose the use of uniform generation and present some advantages of this method. For instance, uniformity of generators prevent them from systematically missing subsets of input values because of a bias. Other research works share the same goal to add a theoretical background to test input generation. We can cite [Sen et al. 2005], focusing on the notion of coverage thus generating inputs to exercise the maximum number of control paths. In a nearer field, we can also mention [Fischer and Kuchen 2008] in which the authors define a notion of data-flow coverage of declarative programs and produce generators accordingly.

On the benchmarking side, we use uniformity to give practical information on the performance of the program, as well as a sound statistical measure of average complexity. Other works focus on producing inputs to show worst case complexity [Burnim et al. 2009] and could be used to obtain complementary complexity information.

Related OCaml projects Research experiments on automatic code generation from Objective Caml type definitions have already been leaded in the past, including random generation (without theoretical background). We can cite for example OCaml Templates [Maurel 2004] or multi-stage [Taha 2003] programming with

MetaOcaml (4). Nowadays, efforts in the community are done to work with types. We use type-conv (5) for type-definition preprocessing code and dyn (6) for run-time type information. Other research solutions exists, one can cite for example [Henry et al. 2007] which modifies the compiler to add first-class run-time type representation to the language.

On the side of testing frameworks, a few tools exist and are maintained for the Objective Caml language, namely oUnit and mlquickcheck which have recently been unified by the Kaputt (7) project. Other functional languages have more widely used tools like Haskell with QuickCheck [Claessen and Hughes 2000].

3. Underlying theory: the Boltzmann model

Our approach is based on the random sampling of combinatorial structures, within the frame of the Boltzmann model, as introduced by [Duchon et al. 2004]. The main feature of this model is uniform generation with linear complexity, thus allowing for generation of much larger objects than was possible before.

Given a finite class of objects $\mathcal{C}$, the generation is uniform, meaning that any object of $\mathcal{C}$ is produced with equal probability $1/||\mathcal{C}||$, where $||\mathcal{C}||$ is the number of objects in $\mathcal{C}$. In the Boltzmann model, each object γ, with size $|\gamma|$, is generated with probability proportional to $x^{|\gamma|}$, where x is a control parameter, thus the generation is uniform for a sub-class of objects of the same size.

Though uniform for a given size, the size distribution of Boltzmann sampling is spread over the whole class $\mathcal{C}$ (this is different from most random generators, that, given a size n, output a random object of size exactly n). However the expected size of generated objects can be tuned by the choice of parameter x, thus giving approximate size sampling as detailed in section 3.3. In the case of testing, this feature is not a restriction: when generating a large set of very large objects, say with one million elements, the exact size of the objects is not relevant up to a few percent. And the important result is that relaxation of exact size from a small percentage is rewarded by a linear time complexity of generation.

The Boltzmann method is generic and can be applied to classes described by specifications based on a rich set of constructors, such as disjoint union, Cartesian product, sequences, sets, cycles, ... (as illustrated in figure 1 by some tree class specifications). The method relies on transforming a system of specifications into a system of functional equations involving generating functions, and working on these functions with analytical techniques (this is the domain of analytical combinatorics, described in [Flajolet and Sedgewick 2009]). By these means, sampling can be automatically compiled from specifications.

The generation algorithms are very simple: for instance generating a couple of objects (a, b) in class $\mathcal{A} \times \mathcal{B}$, simply reduces in independently generating $a \in \mathcal{A}$ and $b \in \mathcal{B}$, and the probability is correct; for disjoint union, generating an object in $\mathcal{A} \cup \mathcal{B}$ is obtained by throwing a biased coin and derive either a generation in $\mathcal{A}$ or a generation in $\mathcal{B}$. In this case, the bias of the coin is computed by evaluating generating functions at parameter x.

Boltzmann samplers are particularly efficient if we accept some variability in the size of the generated structures: fixing a target size n and a margin of error ε, generating random structures untill we get one of size belonging to $[(1-\varepsilon)n, (1+\varepsilon)n]$ can be completed in mean time $O(n)$ (whereas exact size average complexity can be up to quadratic). This *relaxed size* generator has a measurable bias towards smaller sized trees which should not be a problem for practical applications.

Tree type	A corresponding grammar
Ternary trees:	$\mathcal{T} = \mathcal{Z} + \mathcal{T} \times \mathcal{T} \times \mathcal{T}$
One-two trees:	$\mathcal{T} = \mathcal{Z} + \mathcal{U} + \mathcal{B}$, $\mathcal{U} = \mathcal{Z} \times \mathcal{T}$, $\mathcal{B} = \mathcal{Z} \times \mathcal{T} \times \mathcal{T}$
General trees:	$\mathcal{T} = \mathcal{Z} \times \mathcal{F}$, $\mathcal{F} = \mathrm{Seq}(\mathcal{T})$

Figure 1. Examples of tree specifications.

Tree type	Generating functions	ρ
Ternary trees:	$T(z) = z + T^3(z)$	$\rho = 2\sqrt{3}/9$
One-two trees:	$T(z) = z + U(z) + B(z)$ $U(z) = z \cdot T(z)$ $B(z) = z \cdot T^2(z)$	$\rho = 1/3$
General trees:	$T(z) = z \cdot F(z)$ $F(z) = \frac{1}{1-T(z)}$	$\rho = 1/4$

Figure 2. The generating functions and radius of convergence ρ of the example grammars.

This section is continued with a presentation of Boltzmann generation of trees. We first explain how to compute the parameters for the generator in order to obtain a linear complexity generation. Then we present the notion of *tree specification* which will be used for specifying the structure of the trees to be generated. Finally, we describe how to automatically derive a parametrized uniform tree generator that follows such a specification.

3.1 Generating functions

The Boltzmann method applies to the generation of structured objects, using the powerful tool of *generating functions*. Given a class $\mathcal{C}$, we consistently denote by $C(z)$ its generating function, which is the series $C(z) = \sum_{\gamma \in \mathcal{C}} z^{|\gamma|} = \sum_n c_n z^n$, where c_n is the number of objects of size n in $\mathcal{C}$.

The symbolic method [Flajolet and Sedgewick 2009] provides a dictionary for translating structural constructions into operators on generating functions: concerning tree constructions, the dictionary reduces to

$$\begin{aligned} \mathcal{C} = \mathcal{A} + \mathcal{B} &\rightarrow C(z) = A(z) + B(z), \\ \mathcal{C} = \mathcal{A} \times \mathcal{B} &\rightarrow C(z) = A(z) \cdot B(z), \\ \mathcal{C} = Seq(\mathcal{A}) &\rightarrow C(z) = \frac{z}{1 - A(z)}. \end{aligned}$$

Thus in a tree specification, each production rule (non-terminal elements) transforms into a corresponding *generating function* equation, and a grammar transforms into a polynomial system of equations.

To generate trees from these specifications, we need to evaluate these functions for a given value x of variable z by solving such systems of equations. The resolution is analytically coherent for $0 < x \leq \rho$, where ρ is a special value, called the *singularity* of the system.

Solving polynomial systems of equations is a very complex problem in general, but systems corresponding to specifications do have a structure, that can be exploited in the computations. In our implementation we use a combinatorial newton method that gives a very efficient solver [Pivoteau et al. 2008], that can also be used to calculate an approximation of the singularity ρ.

In figure 2, we show the generating functions for the previously introduced tree specifications and the value ρ for each of these systems.

In each case, using the values of these functions at $x = \rho$, the Boltzmann algorithms of section 3.2 derive a linear time generator with the property of *uniformity*: given a size n, two trees of that size have exactly the same probability of being generated. These generators however have the particularity that the generated trees are not all of size n, but have a random size, with a mean value depending on parameter x. We show in section 3.3 how to deal with this aspect, using ρ as the value for x.

3.2 Generation algorithms

In this paper, a *tree specification* will be a unambiguous context-free grammar with one terminal, $\mathcal{Z}$ and three operators: a unary operator to create a *Sequence* (denoted by Seq), where a sequence is made of and arbitrary number $k \geq 0$ of trees (an empty sequence is $\mathcal{Z}$); and two binary operators to compose trees: *Union* (denoted by $+$) and *Product* (denoted by $\times$).

The *size* of a tree T, denoted by $|T|$, will be the number of $\mathcal{Z}$ it contains. This means that the size of a tree is its number of nodes. Other choices for the size are of course possible, as long as there is always a finite number of trees of a given size.

A tree specification can be automatically transformed to a random generation algorithm. Let's see how to do it.

Each production rule of the specification describes a non-terminal. We deduce from it an algorithm to generate objects defined by this non-terminal and the corresponding generating function (we note $A(z)$ the generating function of $\mathcal{A}$). We detail here how to interpret the terminals and operators in the tree specification to obtain these.

$\mathcal{Z}$: the $\mathcal{Z}$ element in the tree specification corresponds to one size unit. Wherever there is a $\mathcal{Z}$ in the specification an object of size one is to be generated. The generating function of $\mathcal{Z}$ is z.

$\mathcal{B} \times \mathcal{C}$: first generate independently both an element $b \in \mathcal{B}$ and an element $c \in \mathcal{C}$. The result will be the couple (b, c). The corresponding function is $B(z) \cdot C(z)$.

$\mathcal{B} + \mathcal{C}$: with this construction, either an element in $\mathcal{B}$ or in $\mathcal{C}$ will be generated. The probability of generating in $\mathcal{B}$ is $B(z)/(B(z) + C(z))$, and the probability of generating in $\mathcal{C}$ is symmetric. A pseudo-random number is used to determine which element should be generated, with respect to the given probability. The generating function is $B(z) + C(z)$.

***Seq*$(\mathcal{B})$:** first the number k of components in the sequence is drawn, following a geometric law with parameter $B(z)$, and then k elements of type $\mathcal{B}$ are independently generated and returned as a sequence. The corresponding function is $\frac{z}{1-B(z)}$.

Figure 3 shows the generation algorithms for the specifications given in figure 1.

3.3 Parameter tuning and complexity

With the Boltzmann method, the size of generated trees is random, with a distribution that depends on the specification $\mathcal{C}$ and a mean value that goes from 0 to infinity when parameter x goes from 0 to ρ, and is equal to $xC'(x)/C(x)$. The probability for the result to be of size n is $c_n x^n \rho^{-n}$, which for most tree specifications and for large n is proportional to $n^{-\frac{3}{2}} x^n \rho^{-n}$. In all cases, the closest is x to the value of ρ, the biggest is the probability of generating large size trees.

The precise size distribution of the generator depends on the nature of the tree structure. We will identify three cases: finite classes, lists and trees[2], each case demanding a different strategy for the choice of parameter x in order to generate objects of size approximately n with a linear complexity (including the cost to generate trees outside the size target, that will be thrown away). The precise definition of the classes depends on the corresponding generating functions, allowing for an automatic classification of specifications. Finite classes have a value of $\rho = \infty$. The generating function of lists tends to infinity when z tends to ρ, while that of trees tends to a finite value.

[2] some structures may look like trees but are actually lists

Ternary trees:

```
TTree () =    if random 0 1 < x/T(x) then Leaf ()
              else Node (TTree ()) (TTree ()) (TTree ())
```

One-two trees:

```
OTTree () =   let r = random 0 1 in
              if r < x/T(x) then Leaf ()
              elsif r < (x + U(x))/T(x) then UTree ()
              else BTree ()
UTree () =    UNode (OTTree ())
BTree () =    BNode (OTTree ()) (OTTree ())
```

General trees:

```
GTree () =    GNode (Forest ())
Forest () =   let k = geom T(z) and res = ref [] in
              for i = 1 to k do res := GTree ()::!res done;
              !res
```

Figure 3. The generation algorithms for the example grammars.

For lists and finite classes, we have to choose x such that the mean size of the generated objects equals n. In that case, the theory [Duchon et al. 2004] guarantees that there will be a constant number of rejects before generating an object of size approximately n. Therefore the mean time complexity is linear.

In the case of trees, linear time complexity can be achieved by using either *pointing* or *singular sampling*. Pointing consists of differentiating the tree specification, akin to Huet's 'zipper' [Huet 1997]. The result of this transformation is a specification of the list class. For our implementation, we chose the second approach, consisting in taking ρ as the value of x. In the case of singular sampling, the mean size of the generated structures is infinite. We will never generate an infinite object, but there is a non-trivial probability of generating objects of sizes that we cannot handle. The solution to this problem is simple and consists in aborting the generating process as soon as we pass the upper bound of our target size.

The only remaining problem is to calculate the value of ρ (for trees) or of x given a target size n (for lists and finite classes). The details of this calculation out of scope for this paper; it uses the Newton algorithm of [Pivoteau et al. 2008] that evaluates the generating function of a given structure on a value x together with a dichotomy heuristic.

The value ρ is an algebraic real number for which we calculate a numeric approximation. The sizes up to which the generation is efficient depends on the precision of this approximation. Adding one digit allows to reach trees of one more order of magnitude. As an illustration, the probability plot given in figure 4 shows the probability of producing a tree of size n as a function of n, with different values of x. It is quasi-impossible to obtain a tree of size one hundred with a precision of $1/10$ for ρ, whereas it is likely to produce a tree of size ten million when ρ is approximated with a precision of $1/10^{10}$.

Our implementation uses double precision floating point numbers, which allows the calculation of ρ to a precision of at least

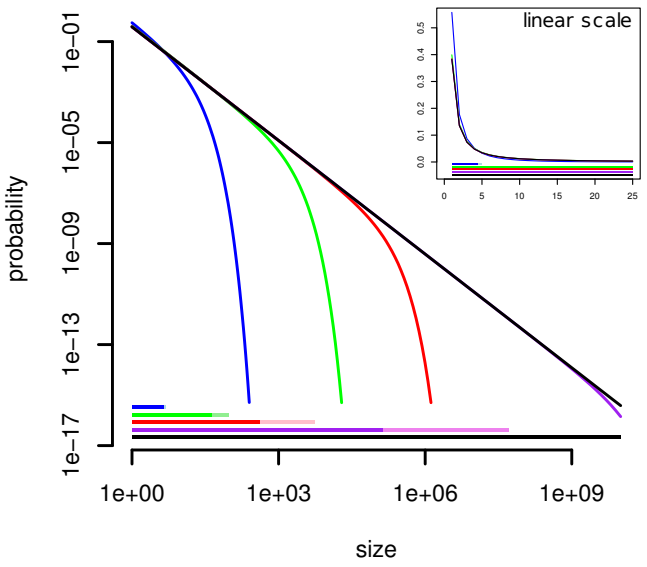
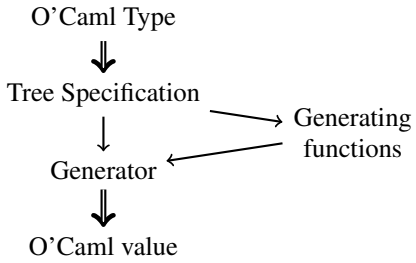


Figure 4. Probability distribution of sizes for trees generated with the Boltzmann method, with a parameter $x = 0.9\rho, 0.999\rho, 0.99999\rho, 0.9999999999\rho$, and ρ. The solid color bars show the range inside which the generators have a guaranteed linear complexity, which in practice extends to the whole colored range. In the main plot, both axes are in logarithmic scale, while the miniature plot is the same one in linear scale.

$1/10^{12}$. Thus, if tree size corresponds to bytes, the biggest trees we are expecting to generate will be of some gigabytes, the size of the main memory of a current desktop computer.

4. Application to O'Caml

The following schema summarizes the steps necessary to get from an algebraic data type to a random value respecting it.



The simple arrows represent transformations that were explained in the previous section, and this section deals with the double arrows. First we explain how to relate algebraic data types to tree specifications. Then, how to create a generator for O'Caml values rather than abstract trees.

4.1 O'Caml types to tree specifications

The size function has to be defined over O'Caml values. One may want to count the number of bytes in the memory representation of the value, or, in the contrary, only count "units" of information carried by the value. We made the choice of an intermediate path, knowing that it is very easy to change the implementation to adapt to a different one.

Base types (unit, int, float, bool, string) will have size one. A constructor (including the empty list, list constructor, tuple and record) add one to the size, so `[1]` and `(True,False)` are both of size 3.

We need a type to express the tree specifications, as presented in last section:

```
1 type grammar = (string * def) list
2 and  def     = Sum of def list
3              | Prod of def list
4              | Seq of def
5              | Ref of string
6              | Z
```

Listing 1. Type of tree specifications

where the `string`s correspond to type names.

A run-time representation of O'Caml types is needed. Many are available and we chose DynLib (6), which is itself based on Typeconv (5).

The translation itself is quite natural, as O'Caml lists correspond to sequences, constructions with a fixed number of arguments to products and variants to sums. Formally, for a type t in a set D of mutually recursive type definitions, we define the transformation as the recursive function $\mathcal{T}$ such as

$$\begin{aligned}
\mathcal{T}(\text{int}, \text{bool}, \ldots) &= \text{Z} \\
\mathcal{T}(t \in D) &= \text{Ref}(\text{``}t\text{''}) \\
\mathcal{T}(t \notin D) &= \text{Z} \\
\mathcal{T}(t \text{ list}) &= \text{Prod [Z; Seq (Prod [Z; } \mathcal{T}(t)\text{])]} \\
\mathcal{T}(t_1 * \ldots * t_n) &= \text{Prod [Z; } \mathcal{T}(t_1); \ldots; \mathcal{T}(t_n)\text{]} \\
\mathcal{T}(\{n_1 = t_1; \ldots; n_n = t_n\}) &= \text{Prod [Z; } \mathcal{T}(t_1); \ldots; \mathcal{T}(t_n)\text{]} \\
\mathcal{T}(\text{A1 of } t_1 \mid \ldots \mid \text{An of } t_n) &= \text{Sum [Prod [Z; } \mathcal{T}(t_1)\text{]; } \ldots; \text{ Prod [Z; } \mathcal{T}(t_n)\text{]]}
\end{aligned}$$

4.2 Tree sampler to O'Caml value sampler

Once we obtain a tree specification corresponding to the types, we can apply the methods of section 3 to obtain a tree sampler. It would then suffice to transform these trees back to O'Caml values and be done. It is however more efficient and as simple to create directly a generator for O'Caml values and use the tree specification only to calculate the generating function values, necessary for the generation.

We need to decide how to generate values for the leaves of our trees, which are either primitive types or types external to the generator. All testing frameworks include generators for primitive types and one generator will not fit all use cases, it is thus important to use a callback mechanism to allow for some flexibility. Our prototype provides some simple generators and allows for the user to name a generator for each type.

We will once again need the definition of the O'Caml type we are using, and this time we will translate it to a generator. Let's see in detail the algorithm to generate a value given its type:

- unit, int, float, bool and string: call the corresponding generator.
- $t \in D$: call the generator created by us.
- $t \notin D$: call the generator provided by the user.
- $(t_1 * \ldots * t_n)$: generate independently each element of the tuple.
- $\{n_1 = t_1; \ldots; n_n = t_n\}$: generate independently each element of the record.

- t list : first draw the list length using a geometric law with the parameter calculated using the generating function corresponding to t. Then generate independently each element of the list.
- A1 **of** t_1 | ... | An **of** t_n: choose randomly which constructor to use with the probabilities of drawing each one being calculated using the corresponding generating functions. Then generate the value of the constructor's argument.

4.3 Modifying the generation result

While we argue that it is important to know the exact probability distribution of the generator, we are aware that the only uniform distribution is not sufficient. For this reason, we propose the user a simple means of modifying the probability distribution, while still being able to easily calculate it. Each type and its variant constructor can be associated to a real number that will be its coefficient.

Adapting our framework to deal with coefficients is fairly straightforward. The generation algorithms are not modified, only the generating functions change. The new values affect the probabilities of choosing a constructor in a variant type or the length of a list. The default coefficient value of 1 changes nothing to the distribution, increasing it will make the corresponding type or constructor appear more often.

Tuning the coefficients is still a manual task though. Given a function on trees (e.g. depth, root degree, number of nodes at a given level), we would like the framework to calculate the coefficients that will make the generator produce trees with a given mean value for the function. The extension of the underlying theory to allow for this is work in progress.

5. Demonstration

In this section, we give the reader a practical overview of the different applications of our random generation core. In this regard, we developed a library handling specification-based testing and benchmarking based on it.

The implementation is available as a CamlP4[3] syntax extension along with a core library to be linked with the executable. To parse type definitions and obtain a run-time representation from them, we respectively rely on (5) and (6).

Downloading, building and launching a toplevel The tester must have at least version 3.11.0 of the Objective Caml compiler and tools. The aforementioned dependencies also have to be available. Our library (1) can then be installed with the usual `make install` command. The package provides a top-level pre-loaded with the library and the syntax extension; let's start by loading it with `genadttop`.

An algebraic data type We need a data type to work on, let's choose a simplified representation of text documents. It shows products (tuples), sum types and mutually recursive types.

```
type document = string * block list
and block =
  | Paragraph of text
  | Section of (string * block list)
and text =
  | Text of string * text
  | Attribs of attribs * text
  | Empty
and attribs =
  | Bold | Italic | Underlined
with sample
```

Listing 2. Our example data type

[3] The Objective Caml preprocessor included in the standard distribution

The ⌜**with** sample⌝[4] keyword is preprocessed by our CamlP4 syntax extension to extract its combinatorics grammar.

$$\mathcal{D} = \mathcal{Z} \times (\mathcal{Z} \times \mathrm{Seq}(\mathcal{B}))$$
$$\mathcal{B} = \mathcal{Z} \times \mathcal{T} + \mathcal{Z} \times (\mathcal{Z} \times (\mathcal{Z} \times (\mathcal{Z} \times \mathrm{Seq}(\mathcal{B}))))$$
$$\mathcal{T} = \mathcal{Z} \times (\mathcal{Z} \times \mathcal{T}) + \mathcal{Z} \times (\mathcal{A} \times \mathcal{T}) + \mathcal{Z}$$
$$\mathcal{A} = \mathcal{Z} + \mathcal{Z} + \mathcal{Z}$$

A note on size In this section, we shall talk about document size, let's make this notion clear. As explained in section 4, each constructor in a sum type, each product type as well as each base type has size one. For example, the following value has a size of 8.

```
("book", Paragraph (Text ("text", Empty)) :: [])
1  1          1         1       1        1     1  1
```

Listing 3. Size count example

The syntax extension produces the function to build a generator of documents of a given maximum size (the minimum size can be adjusted with another optional argument).

```
val sample_document
  : ?min:int max:int -> (unit -> document)
```

Listing 4. Generated sampler signature

We also provide a syntax to define specialized samplers in which the programmer can give coefficients to constructors or types, as well as specify the samplers for external types appearing in the definition.

```
let my_sample_document =
  <:sampler< document
             coeff 10 for Paragraph
             sample string with my_string_sampler >>
  : ?min:int max:int -> (unit -> document)
```

Listing 5. Tweaked sampler

In this example, we ask for values in which the ⌜Paragraph⌝ constructor appear much more often than in a uniform random generation by assigning a coefficient of 10 to it (by default, each constructor is assigned a coefficient of 1). We also require strings to be sampled by our own function rather than by the default string sampler.

Testing We shall now demonstrate how to use these samplers in specification-based testing. We provide several functions for this task. For example, the ⌜run_test⌝ one which runs an interactive loop, testing a function with a predicate and showing the problematic inputs/outputs on demand.

```
run_tests :
   int -> (unit -> 'a) ->
   ('a -> 'b) ->
   ('a -> 'b -> bool) ->
   ('a -> string) ->
   ('b -> string) ->
```

Listing 6. Interactive testing function signature

Here is an example testing the validity of our pretty printing function for this document format.

```
  run_tests
    10 (sample_document ~min:1 ~max:10_000)
    print_document
    print_endline
    to_string
    (fun d r -> d = of_string r)
```

Listing 7. Launching an interactive testing

[4] We use the notation ⌜code⌝ for code extracts in the text

This example reveals simple errors. For example, it seems that we did not make a difference between sections containing only the empty string and empty sections in our printer:

```
Test 5/100 failed.
Show input (size 11) (y/n) ? y
("x", [
  Paragraph (Text ("y",
             Attribs (Underlined,
                      Attribs (Bold, Empty))))
])
Show output (y/_) ? y
"x"  ( p "y" ( u ( b "" ) ) )
```

Listing 8. Interactive testing session

But our ability to easily generate large size values can also make other classes of errors show up, like the fact that one of our function is not as tail recursive as we thought:

```
Test 1/100 raised Stack overflow.
Show input (size 1000000) (y/n) ? n
```

Listing 9. Interactive testing session (2)

Also, checkers also have to be written correctly to handle checking of large size values:

```
Checker failed on test 13/100 with Stack overflow.
```

Listing 10. Interactive testing session (3)

Observing properties As explained in section 2, since we use uniform random generation, we can soundly obtain statistical properties about the values of a given size. For example, it is theoretically hard to obtain an average height for a non-trivial tree structure. Even if it quite meaningless in this example, let's see how to obtain the average number of sections in a random document of size ranging between 10,000 and 20,000.

```
1  range
2    sample_document      (* generator *)
3    10_000 20_000        (* size *)
4    100                  (* 100 samples *)
5    average              (* combinator *)
6    ["height", height]   (* property *)
```

Listing 11. Simple statistical property computing example

This example showed how to obtain a single value by applying the ⌜average⌝ function to all the results. We provide several functions to select the inputs (range, exact size, histogram, etc.). We also provide several others to combine the results (identity, average, min_max, etc.).

The next example show how we can obtain an histogram of all documents of size between 1000 and 100,000 with 10 subdivisions with the ⌜histogram⌝ selector and obtain the average of each subset with the ⌜average⌝ combinator. The samples can be used simultaneously on several properties, here on the height and the maximum length of paragraphs.

```
1  histogram
2    sample_document
3    1000 100_000      (* sizes from 1000 to 100,000 *)
4    10                (* ten subdivisions *)
5    10                (* 10 samples per subdivision *)
6    average
7    ["height", height ;
8     "paragraph␣length", parlen ]
```

Listing 12. Histogram with multiple properties

We provide an integrated display written with the standard ⌜Graphics⌝ module, in order to be able to easily see the results from the top-level in a single line. Figure 5 shows this display applied to the result of the previous example.

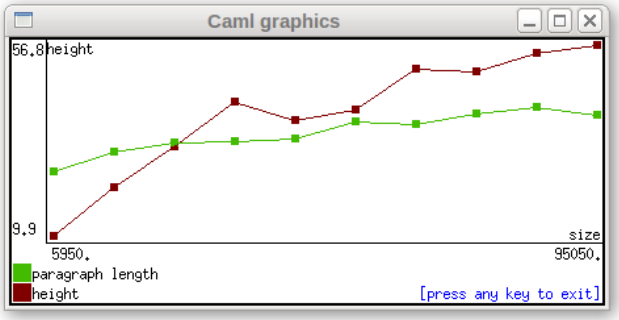


Figure 5. Integrated Graphics display

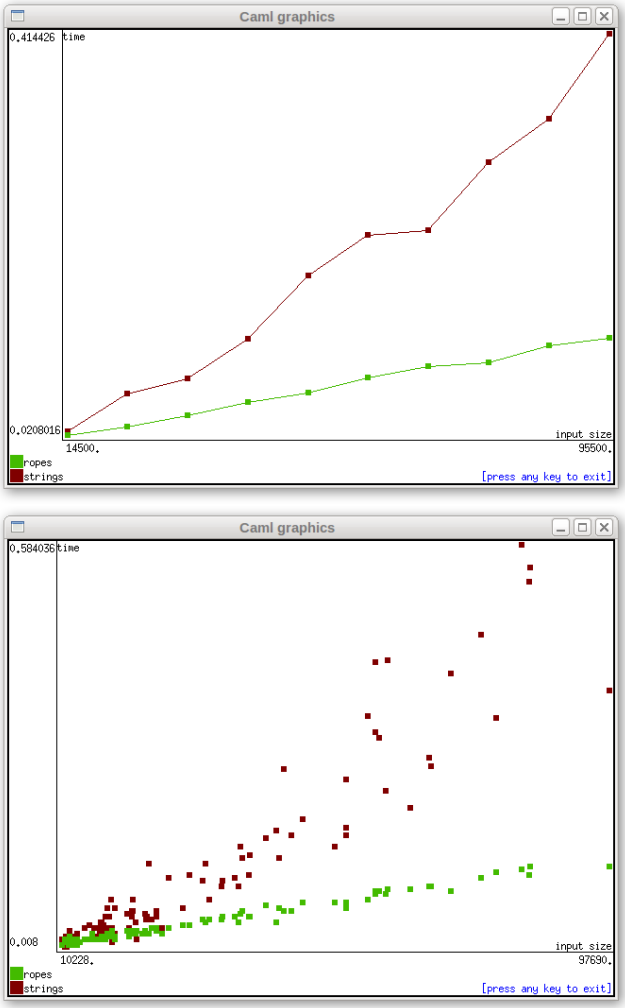


Figure 6. Display of benchmarking results for to_string implementations with strings and ropes. The top image has been obtained with the histogram function, the bottom one with range.

Benchmarking One big difficulty when writing complex programs is to figure out how the complexity of the implementation of some data structure can affect the global performance. As we can for any property, we can obtain statistical properties about complexity by running an algorithm over a number of samples.

We provide a benchmarking function taking a list of implementation and running them on the same samples. Figure 6 shows for example the performance of the ⌜to_string⌝ function with two different string implementations (standard strings and ropes).

6. Performance

In order to demonstrate the efficiency of the Boltzmann model, we measured the time[5] needed to generate values of the `document` type with our prototype implementation. The measures were made on a recent desktop computer (Intel Core 2 2.4GHz with 3GB of RAM). Since the running time of a generation depends on the number of rejections which is random, the numbers shown below are averages over 100 runs.

We compared these timings to the recursive generation implemented in the combstruct module of the comuter algebra system Maple version 10. This module has not been designed for generating very large objects, sizes over 2000 result in a stack overflow.

size	recursive	Boltzmann	
		exact	$\varepsilon = 0.1$
100	0.5s	0.07s	-
200	1.5s	0.25s	-
500	9.4s	1.6s	-
1000	42s	8.1s	0.037s
1500	1m46s	13.5s	0.062s
10 K	*fail*	-	0.4s
100 K	*fail*	-	4.2s
1 M	*fail*	-	50s

This experiment confirms that exact-size generation of trees using the Boltzmann method is quadratic, while approximate-size generation is linear. This includes the pre-processing time that is dependant on the specification size. O'Caml types will typically produce small grammars, but our prototype is capable of dealing with grammars containing several hundred rules in minutes.

The O'Caml code needed for this prototype is just a few hundred lines long. The generating function values are calculated by a seperate C library, shared with other projects, that is itself less than a thousand lines long.

7. Future works and other applications

Plans for this work The current implementation is still in prototype stage. We are willing to help its integration and adaptation by testing or benchmarking framework designers. If the community shows interest in this project and our library prototype, we shall be ready to develop and maintain it, but in this case we shall probably choose a better run-time type representation. Of course, this does not apply only to Objective Caml but also to other statically typed functional languages like SML or Haskell for which the adaptation of the model is straightforward.

One of the limitations is that we don't handle type parameters yet. The easy solution with the current tools would be to add an argument to the generator for each type parameter so the programmer can provide the sub-generator by hand. This means that type parameters are considered as leaves in the combinatorics specification. We would however like the combinatorics specification of each type parameter to be inlined into the main type so that the generation is uniform over the whole grammar. In practice, we want to obtain the instantiation of type parameters when building the generator. If we stay at the syntax extension level, this would require the programmer to manually explicit the instances along with some tricky type operations. A better solution could be to use the integrated run-time type representation, as proposed by [Henry et al. 2007] which automatically handles instantiation but requires a modified compiler.

Another limitation is that the generated values do not contain cycles or sharing. This is due to the fact that the theory only handles trees. However, generating graphs, or at least directed acyclic ones, could be very interesting for testing purposes. Work is done on the theoretical side, but until a satisfactory solution is found, we have practical ideas for ad-hoc methods. For example, a method already experimented in other applications is to tweak the generated values.

Work in progress about programming languages Any programming language with algebraic data types is amenable to the techniques presented in this paper. Haskell is particularly interesting, since the QuickCheck specification-based test library is widely adopted by this language's community. A small library extending QuickCheck in order to automatically generate random instances of algebraic data types is almost ready, and it's functioning principles are identical to those showed here.

Other tree-like data structures can also be generated with these methods, sometimes with a more involved translation process. XML documents respecting a given RelaxNG grammar is an example we have treated [Darrasse 2008]. Possible applications include stress-testing of web services and this can be easily adapted to related languages, like CDuce or OCamlDuce.

It might even be interesting to generate the tree skeleton of more complicated data structures, in cases where there is no efficient random generation yet. In a recent work [Mougenot et al. 2009], this method has been applied to generate randoms models that respect a given meta-model. The cross-references that appear in these structures are treated in an ad-hoc manner.

Work in progress about combinatorics As we just said, we cannot handle sharing and cycles in the generated values. A rigorous treatment of structures with cross-references is being worked on, but the complexity rises with the number of shared objects (e.g. variables in λ-terms). Generating DAGs or graphs in general (other than specific classes that are in bijection with some kind of trees) is out of reach for the moment.

Boltzmann sampling theory is much more wide than the part that we present here. More constructions are available, especially to take symmetries into account. These could be easily integrated to this work, but these constructions are out of the range of expressiveness of ML's type system.

We believe however that there is room for more collaboration between the fields of combinatorial structures and functional languages. The theory of species [Bergeron et al. 1998], that treats combinatorial structures from a category theory perspective is central to such efforts, as can be seen for instance in the work on derivatives of [Abbott et al. 2003].

8. Conclusion

We applied the recent develoment of the Bolzmann random generation model to algebraic data types by defining a translation from type definitions to combinatorics specifications. We developed a syntax extension for Objective Caml generating combinatorics grammars from type definitions, and a random generation core generating values respecting such a grammar. In the end, we extract in a fully automated way random generators from Objective Caml type definitions.

[5] processor time in user mode

The theoretical results on these generators ensure soundness and efficiency properties that had never been reached simultaneously by existing generation techniques.

1. The generation is uniform: the generator for a given type and size gives the same probability to be produced to each possible value. In a testing context; this property ensures that no subclass will be missed because the generator is biased. Moreover, it also allows the method to be used as a sound input generator for benchmarking.
2. If we authorize a little approximation on the size, which is fine for testing purposes, the time and space complexity is linear. The method is thus able to produce very large objects.

Other generation techniques providing the first property have super-linear space and/or time complexity while techniques providing the second produce biased generators.

To demonstrate the method, we developed a small testing and benchmarking open source library for Objective Caml. We are ready and willing to maintain and extend it if the community is enthusiast. We are as well available to help testing and benchmarking frameworks integrate this method.

Links

(1) GenADT: our prototype implementation
http://www-apr.lip6.fr/~canou/genadt/
(2) Quickcheck, an automatic testing tool for Haskell
http://www.cs.chalmers.se/~rjmh/QuickCheck/
(3) Objective Caml
http://caml/inria.fr/
(4) MetaOcaml, multi-stage programming for Objective Caml
http://www.metaocaml.org/
(5) Type-conv: Support library for preprocessor type conversions
http://hg.ocaml.info/release/type-conv
(6) Dyn: Library for run-time type representation
http://www.pps.jussieu.fr/~till/dyn
(7) The Kaputt testing tool
http://kaputt.x9c.fr/

References

M. Abbott, T. Altenkirch, N. Ghani, and C. McBride. Derivatives of containers. *Lecture notes in computer science*, pages 16–30, 2003.

F. Bergeron, G. Labelle, and P. Leroux. *Combinatorial species and tree-like structures*. Cambridge University Press, 1998.

Jacob Burnim, Sudeep Juvekar, and Koushik Sen. Wise: Automated test generation for worst-case complexity. In *ICSE*, pages 463–473, 2009.

Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, New York, NY, USA, 2000. ACM. ISBN 1-58113-202-6. doi: 10.1145/351240.351266.

A. Darrasse. Random XML sampling the Boltzmann way. *ArXiv e-prints*, 2008. URL `http://arxiv.org/abs/0807.0992`.

Philippe Duchon, Philippe Flajolet, Guy Louchard, and Gilles Schaeffer. Boltzmann samplers for the random generation of combinatorial structures. *Combinatorics, Probability and Computing*, 13:577–625, 2004. doi: 10.1017/S0963548304006315.

Sebastian Fischer and Herbert Kuchen. Data-flow testing of declarative programs. *SIGPLAN Not.*, 43(9):201–212, 2008. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1411203.1411233.

P. Flajolet, P. Zimmerman, and B. Van Cutsem. A calculus for the random generation of labelled combinatorial structures. *Theoretical Computer Science*, 132(1-2):1–35, 1994.

Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics*. Cambridge University Press, 2009.

G. Henry, M. Mauny, and E. Chailloux. Typer la dé-sérialisation sans sérialiser les types. *ArXiv e-prints*, May 2007. URL `http://arxiv.org/abs/0705.1452`.

G. Huet. The zipper. *Journal of Functional Programming*, 7(05):549–554, 1997.

Xavier Leroy, Didier Rémy with Damien Doligez, Jacques Garrigue, and Jérôme Vouillon. The objective caml system release 3.11 documentation and user's manual. Technical report, Inria, november 2008.

François Maurel. Ocaml-templates, méta-programmation à partir des types. In *Actes des journées JFLA Journées francophones des langages applicatifs*, pages 21–36, Sainte-Marie-de-Ré, France, January 2004. INRIA. URL `http://hal.archives-ouvertes.fr/hal-00153820/en/`.

Alix Mougenot, Alexis Darrasse, Xavier Blanc, and Michèle Soria. Uniform random generation of huge metamodel instances. In *Fifth European Conference on Model-Driven Architecture Foundations and Applications*, 2009.

Carine Pivoteau, Bruno Salvy, and Michèle Soria. Boltzmann oracle for combinatorial systems. In *Fifth Colloquium on Mathematics and Computer Science Algorithms, Trees, Combinatorics and Probabilities*, DMTCS Proceedings, pages 475–488, 2008.

C. Runciman, M. Naylor, and F. Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Proceedings of the first ACM SIGPLAN symposium on Haskell*, pages 37–48. ACM New York, NY, USA, 2008.

Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 263–272, New York, NY, USA, 2005. ACM. ISBN 1-59593-014-0. doi: http://doi.acm.org/10.1145/1081706.1081750.

Walid Taha. A gentle introduction to multi-stage programming, 2003. Available from `http://www.cs.rice.edu/~taha/publications/journal/dspg04a.pdf`.