SciFe: Scala Framework for Efficient Enumeration of Data Structures with Invariants

Ivan Kuraj Viktor Kuncak

ABSTRACT

We introduce SciFe, a tool for automated generation of complex structures, suitable for tasks such as automated testing and synthesis. SciFe is capable of exhaustive, memoized enumeration of values from finite or infinite domains. SciFe is based on the concept of an enumerator, defined as an efficiently computable bijection between natural numbers and values from a given set. SciFe introduces higher-order enumerators which define enumerators that depend on additional parameters. SciFe also includes combinators that can construct more complex enumerators from existing ones while preserving exhaustiveness and efficiency. SciFe is a Scala library that implements a domain-specific language. This tool demo presents an overview of SciFe as well as its use to generate complex structures such as search trees and models of class hierarchies. Our experiments demonstrate better performance and shorter specifications when compared to existing approaches.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

Keywords

Enumeration, Test Generation, Property-based Testing, Exhaustive Search, Embedded Language, Memoisation, Metaprogramming, Scala

1. INTRODUCTION

One of the challenges in automated testing is automating the process of test input generation. Test inputs may require instances of complex data structures such as red-black trees or models of class hierarchies. Such data structures have complex structure (defined with typing information) and are additionally constrained with data structure invariants. A procedure for automated and efficient data structure generation can have a great value not just in software testing [6, 3], but also in software verification and synthesis [9, 11], and software engineering in general [7, 2, 10, 4, 1]. Achieving both the expressiveness for complex structures and efficiency with such an automated procedure is a challenging task. Recent advances in automated testing brought a variety of approaches that differ in the trade-off between efficiency, flexibility and ease of use [6, 4, 5, 12, 1, 13]. SciFe aims at providing convenient and flexible ways for expressing specifications for automated, efficient and exhaustive enumeration of complex data structures.

SciFe[1] provides a framework for enumeration for complex data structures which allows operations such as efficient direct access to an element given by an index. The framework is based on enumerator objects and enumerator combinators. Enumerators are defined as computable bijective functions between natural numbers and values of the given (encoded) set of values. Such definition allows efficient operations such as querying an enumerator for elements at particular positions. Enumerator combinators provide means of composing and refining enumerators. With these core concepts and an extension of higher-order enumerators, our framework achieves expressiveness and flexibility for constructing enumerators that efficiently enumerate complex data structures.

SciFe allows developers to write enumerators and query them to enumerate the specified data structures. Using SciFe, a developer can focus on the important aspects of a data structure at hand rather than on the mechanical production of its instances. Instead of manually constructing instances, the developer writes an enumerator that can be queried to enumerate many valid instances that satisfy specified structural properties and invariants.

SciFe focuses on bounded-exhaustive testing (BET) approach of generating data, which produces all structures within the given bound [14]. This approach has the advantage of high test coverage — it covers all "corner cases" within the given bound (that could be otherwise difficult to find and write manually) — and follows the principle akin to the *small scope hypothesis* behind model-checking tools such as Alloy [8]. BET approaches face a trade-off between expressiveness of specifications and the performance of their generation. *Declarative* BET techniques (or filtering as mentioned in [6]) allow writing specifications as declarative predicates; the tool searches for valid structures that satisfy them. *Constructive* (or generating [6]) approaches re-

Scala '14, Uppsala, Sweden

[1] SciFe is an anagram using initial letters from "**Sc**ala **F**ramework for efficient **E**numeration of data structures with **I**nvariants"

quire more instructive specifications on how are valid structures constructed. In cases of generating complex data structures with invariants, the former often may allow more natural specifications but incur performance overheads, while the later tend to be insufficiently expressive. SciFe enumerators are specified in a *constructive* style, with concise specifications that are expressive enough for many complex data structures with invariants.

Enumerators specified with SciFe provide several attractive properties. *Modularity* allows treating enumerators as self-contained building blocks. *Memoization* allows storing and sharing computed intermediate results. *Random access* allows random testing, in addition to exhaustive enumeration, and *fine-grain* enumeration, with precise choice of which instances to enumerate. Note that simple generate-and-filter random testing approaches stop being efficient in the presence of complex invariants, because the vast majority of generated instance candidates fails to satisfy the invariant. A key aspect of SciFe is that it makes it convenient to write generators that take into account the invariants during the generation process itself, which makes generation more efficient.

SciFe is a Scala library that implements a domain specific language (DSL) for specifying generic, reusable, and composable enumerators. SciFe fits and interoperates well with standard Scala constructs. We expect SciFe to be a useful and practical tool for development and testing of Scala programs. Furthermore, it gives an example of a high-performance tool written in Scala: our current experiments show favorable results when compared to existing test generation approaches.

2. OVERVIEW OF SciFe

In this section we describe the main concepts and techniques behind enumerators and show examples of their usage.

2.1 Enumerators

SciFe enumerators provide ways for efficient *encoding* and *enumeration* of sequences of elements from a particular domain. In our framework, an enumerator of elements of some type T is an instance of trait Enum[T]:

```
trait Enum[T] extends Function[Int, T] {
  def apply(ind: Int): T
  def hasDefiniteSize: Boolean
  def size: Int }
```

Note that the method hasDefiniteSize should be used in conjunction with size, since computing the size of an infinite enumerator may lead to a diverging computation[2]. Method apply defines an indexing function — for a given enumerator e and index i, e(i) returns the i-th element of the sequence encoded by e. Method size returns the length of the encoded sequence — every value from the range (0..e.size) is a valid input to the indexing function. Enum[T] extends the Scala function type, thus every enumerator can be viewed as a (bijective) function from natural numbers to values of type T. Additionally, enumerators can also be viewed as and constructed from Scala collections, and SciFe enables implicit conversions for the two concepts interchangeably.

[2]This adheres to the convention of handling infinite collections in the Scala standard library

name	syntax	semantics/type
merge	$x \oplus y$	disjoint union of x and y (x:Enum[T], y:Enum[T])⇒Enum[T]
product	$x \otimes y$	Cartesian product of x and y (x:Enum[T], y:Enum[V])⇒Enum[(T, V)]
map	$x \uparrow f$	mapping elements of x with f (x:Enum[T], f:T⇒V)⇒Enum[V]
filter	$x \succ p$	filtering elements of x with p (x:Enum[T], y:T⇒Boolean)⇒Enum[T]

Table 1: Enumerator combinators.

```
val dates = (1 to 31) ⊗ (1 to 12) ⊗ Stream.from(2014) ≻
  { isValid(_) } ↑ { case ((d, m), y) ⇒ new Date(y, m, d) }
```

Figure 1: Enumerator that enumerates all valid dates, starting from January 1, 2014, as instances of Java Date class

To go beyond simple enumerators, SciFe adds expressiveness by providing a set of *combinators* that allow composition and refinement of existing enumerators. The set of combinators is summarized in Table 1. These combinators have intuitive semantics and are analogous to common operations defined on collections in functional programming languages.

2.1.1 Enumeration of valid dates

The following example demonstrates enumeration of valid Java Date objects, constructed from multiple enumerators by composing with combinators. The example is given in Figure 1. We enumerate all possible combinations of values for date, month, and year by using the product combinator ⊗ — it effectively produces Cartesian product of all three sets of encoded values (note that since Stream.from(2014) is infinite, there is an infinite number of tuples of this product). Then, we filter these tuples with ≻ and function isValid which determines whether the enumerated combinations of values for date, month, and year represent a valid date, taking into account, for example, whether the year is leap. All valid tuples are then mapped with a function that constructs date objects.

The code that constructs the enumerator for valid date objects is specified with an expression that resembles mathematical notation. Our DSL enables such constructions by implicit conversions from Scala collections and rules for applying combinators. Enumerators can be constructed from any standard collection. The DSL aims at enabling natural and convenient ways for defining enumerators as well as invoking their operations that fit well into Scala programs. Having the enumerator dates defined in Figure 1, there are multiple ways for enumerating its elements:

```
dates.hasDefiniteSize // returns false
for (d ← dates) yield print(d) // prints all dates (does not stop)

dates(209) // gives 209th element (start date of Scala 2014)
for (i ← 0 until 2) yield dates(209+i) // workshop days
```

Since SciFe treats enumerators as standard Scala collections, enumeration can be performed by any other standard means for traversing collections, such as for-comprehensions. Note

name	syntax	description/type
merge	$x \oplus y$	point-wise disjoint union of x and y (x:Depend[D,T], y:Depend[D,T])⇒Depend[D,T]
product	$x \otimes y$	point-wise Cartesian product of x and y (x:Depend[D,T], y:Depend[D,V])⇒Depend[D,(T, V)]
inmap	$x \downarrow f$	mapping the dependent argument of x with f (x:Depend[D,T], f:V⇒D)⇒Depend[V,T]
chain	$z \oslash x$	chaining of z with x (z:Enum[D], x:Depend[D,T])⇒Enum[T]

Table 2: Higher-order combinators.

that the developer can use standard Scala constructs in the enumerator definition and SciFe will take care of constructing (or converting to) appropriate and optimal enumerators according to the given specification (e.g. enumerators for finite and infinite collections differ in their implementation).

2.2 Higher-order enumerators

In addition to having enumerators as bijective mappings between natural numbers and values, which we will refer to as *basic*, we extend the framework with *higher-order* enumerators. Higher-order enumerators represent bijective mappings between values and basic enumerators. They add the necessary expressiveness to allow defining enumerations of more complex data structures. In comparison to basic enumerators, rather than encoding a single set of values, a higher-order enumerator encodes multiple sets of values — a set of values per each value in the function's domain. Due to resemblance to dependent-type functions (a higher-order enumerator may be observed as a dependent type function where the codomain, i.e. the set of values to be enumerated, depends on the argument), we refer to values from the domain as *dependent*. Note that this definition does not restrict the domain of higher-order enumerators — it allows them to dependent on arbitrary parameters.

Higher-order enumerators in SciFe are declared as:

```
trait Depend[I, +O] extends Function[I, Enum[O]] {
  def apply(param: I): Enum[O] }
```

An instance of Depend[I,O], t_d, is a function I ⇒ Enum[O] which encodes an inner basic enumerator $t_d(p)$ for each value p of the dependent parameter. Effectively, t_d behaves as a function $D \Rightarrow N \Rightarrow S$, where D is the dependent domain, N is the range of the corresponding inner enumerator $t_d(p)$, and S is the set of elements encoded by $t_d(p)$.

Along with higher-order enumerators, SciFe extends the combinator set with additions shown in Table 2. First two combinators combine two higher-order enumerators by means of applying $\oplus$ and $\otimes$ pointwise: for each value of the dependent parameter, corresponding inner enumerators are composed. inmap applies f to the given value of the dependent parameter before querying x; it can be useful in cases where the dependent parameter needs to be adapted to the desired type. chain allows enumerating elements in the inner enumerators of y according to dependent values enumerated from z. This operation effectively encodes a flatMap operation defined for standard Scala collections.

Higher-order enumeration, with the extended set of combinators, provides the necessary expressiveness to construct efficient enumerators for complex data structures. Such enumerators allow efficient enumeration that goes beyond generate-and-test approach (which generates all possible values and then filters out those that are not correct); it allows capturing invariants in the generation itself and avoiding the exploration of unnecessary search space.

2.2.1 Enumeration of binary search trees

We demonstrate the expressiveness of SciFe, including higher-order enumerators, by enumerating ordered binary search trees. Figure 3 defines a higher-order enumerator bst that for a given pair of values (s, r) of type (Int, Range), which represent the tree size and range of values, enumerates all instances of Tree, as defined in Figure 2, that are valid binary search trees of size s and for which integer values of field v, for any node in the tree, belong to range r. Function rec, from the DSL, constructs a higher-order enumerator of type Depend[(Int, Range), Tree] from the given partial function. Function's argument is matched to the enumerator's recursive definition, self, and its dependent argument, a pair (size, range).

We generate trees by a case analysis: for dependent parameter where size s is equal to 0, inner enumerator contains only Leaf (since it is the only tree with size 0). If $s \geq 1$, due to the ordering property of BSTs, we may observe that value in the root node constrains values that may occur in the left and right subtree. Therefore, if the given range of all possible values for the whole tree is $r = (l..u)$, then fixing a value m at the root refines range for the left and right subtree, to $(l..m-1)$ and $(m+1..u)$, respectively. Similarly, by knowing the size of the whole tree and fixing the size of the left subtree l_s, we uniquely determine the size of the right subtree, which is calculated as $s - l_s - 1$. The enumerator bst is defined such that it, according to these constraints, enumerates all possible combinations of m, l_s, left and right subtrees, and combines them to form valid trees.

According to this reasoning, we transform dependent arguments and use them to recursively generate subtrees; for left subtrees, given range=$(l_s..m)$, we compute size and range for the left subtree as (ls, (range.start to (m−1))) and then use it to query the enumerator recursively using self. Enumeration of right subtrees is analogous and follows a similar reasoning. Having enumerators for subtrees, left and right, defined, we enumerate all possible pairs of left and right subtrees with left⊗right. Thus, in order to enumerate all valid trees: we enumerate values of size of left subtree (l_s) with (0 until size) and the root element (m) with range; pair them with ((0 until size) ⊗ range); and map all such pairs to produce all tuples that encode valid binary search trees of size s and range r. The resulting tuples are then mapped with a function that constructs instances of Node.

For trees of size 15 and range (1..15) this code enumerated all, more than $9.6 \cdot 10^6$, trees in just a few seconds. The code outperform test generation tools such as Korat, which timeouts for sizes greater than 11 and is slower 3 orders of magnitude for size 10, and constraint-based generation which is slower up to 2 orders of magnitude. Additionally, specification in Korat takes more than 100 lines of code, while SciFe enumerator takes around 25. Seeing the capabilities of enumerators in the light of flexible data structure generation and bounded-exhaustive testing, SciFe offers a concise and modular way for specifying enumerators that are very efficient in generating complex data structures.

3. EXPERIMENTAL EVALUATION

```
trait Tree
case object Leaf extends Tree
case class Node(l: Tree, v: Int, r: Tree) extends Tree
```

Figure 2: Definition of the binary search tree data structure

```
val bst = rec[(Int, Range), Tree]({ case ((size, range), self) ⇒ {
  if (size == 0) Leaf
  else {
    val left: Depend[(Int, Int), Tree] = self ↓
      { case (ls, m) ⇒ (ls, range.start to (m−1)) }
    val right: Depend[(Int, Int), Tree] = self ↓
      { case (ls, m) ⇒ (size − ls − 1, (m+1) to range.end) }

    ((0 until size) ⊗ range) ⊘ (left ⊗ right) ↑ {
      case ((_, root), (lt, rt)) ⇒ Node(lt, root, rt) }
}}})
```

Figure 3: Enumeration of (ordered) binary search trees

We ran a set of experiments and evaluated the performance of generating several data structures. We compared SciFe to two existing approaches. All benchmarks and details about the experimental methodology are publicly available on the tool's website given in the Appendix.

Platform for Experiments. The experiments were ran on a processor with 3.4Ghz clock speed and 8MB of cache. The environment consisted of 64b Linux, Java (and the JVM) 1.7.0_51 and GNUProlog 1.3. SciFe had Scala 2.10.3 as its current version. Experiments were execute with imposed limits of 200s for the timeout and 24GB for the total working memory. Timings were collected using ScalaMeter and per-tool built-in timers, in case of SciFe and other tools, respectively.

Evaluation. Table 3 summarizes the comparison of measured performance. Data structures were generated with SciFe (with memoization enabled), the constraint logic programming (CLP) approach [13], and Korat [1]. We used specifications available from their official repositories whenever possible (for some data structures we wrote specifications from scratch). For each benchmark, structures of different sizes were enumerated. For ordered data structures, given ranges for the structure keys were determined by its size. DAG and (more complex variant) Class DAG encode directed acyclic graphs, which are not generated with CLP, since CLP is limited to simple tree-like structures[13]. Class DAGs represents generation of all valid models of Java class hierarchies with methods; each type can be either a class or an interface, while methods may be overloaded and sealed (the results are given when the number of methods is set to 2). The table tabulates the running time needed to generate all valid structures of a given size. Note that Korat and CLP generate structures of the given size, while SciFe enumerators enumerate all structures up to that size, when data structures have inductive definitions. The table omits numbers of generated data structures; all three approaches generate all valid structures according to the given bound (i.e. size).

In all experiments, SciFe outperformed both Korat and CLP, except for generation of red-black trees, where CLP used a highly-optimized logic specification[3]. We believe this is due to the aggressive optimizations that allow little to no backtracking (and thus smaller constant overhead) of the search with Prolog, not due to worse scalability of SciFe to larger red-black trees. Note that for some benchmarks, such as heap arrays of sizes above 10, our framework experienced garbage collection overheads.[4]

4. CONCLUSION AND FUTURE WORK

SciFe demonstrates that efficient automated generation of complex data structures can be achieved by means of enumerator objects and their composition. Higher-order enumerators add the expressiveness to enumerate complex data structures with invariants without the need for costly generate-and-filter approach. Although writing specifications by defining enumerators differs from the existing constructive and declarative approaches, such specifications tend to be more expressive and shorter when compared to some of the existing approaches. We believe SciFe can be useful and fit well in the toolbox of Scala developers.

APPENDIX

A. APPENDIX: PROJECT REPOSITORY

To find additional and updated information about SciFe, please visit `http://kaptoxic.github.io/SciFe`. The source code is available at `http://github.com/kaptoxic/SciFe`.

B. REFERENCES

[1] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on Java predicates. In *ISSTA*, pages 123–133, 2002.

[2] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. ARTOO : Adaptive Random Testing for Object-Oriented Software Categories and Subject Descriptors. *ICSE*, pages 71–80, 2008.

[3] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP*, pages 268–279, 2000.

[4] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov. Automated testing of refactoring engines. *Proc. 6th Jt. Meet. Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng. - ESEC-FSE '07*, page 185, 2007.

[5] J Duregå rd, P Jansson, and Meng Wang. Feat: functional enumeration of algebraic types. *Proc. 2012 Haskell Symp.*, pages 61–72, 2012.

[6] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov. Test generation through programming in UDITA. In *ICSE*, pages 225–234, 2010.

[7] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. In *PLDI*, pages 27–38, 2013.

[8] Daniel Jackson. Alloy: A lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, April 2002.

[3] the optimization is performed with the help of an external tool in multiple steps, which complicates the specification significantly[13]

[4] there are more than 10^8 heaps of size 10, storing just pointers to their roots can thus take more than 4GB of memory

Size	Binary search trees			Heap arrays			Red-black trees			Sorted lists			DAGs		Class DAGs	
	SciFe/e	CLP	Korat	SciFe	CLP	Korat	SciFe	CLP/no	Korat	SciFe	CLP	Korat	SciFe	Korat	SciFe	Korat
1	0/0	0	0.12	0	0	0.1	0	0/0	0.14	0	0	0.11	0	0.11	0	0.15
2	0/0	0	0.12	0	0	0.1	0	0/0	0.15	0	0	0.11	0	0.11	0	0.39
3	0/0	0	0.12	0	0	0.1	0	0/0	0.15	0	0	0.11	0	0.11	0.01	10.32
4	0/0	0	0.14	0	0	0.11	0	0/0	0.19	0	0	0.12	0	0.12	0.34	t/o
5	0/0	0	0.18	0	0	0.16	0	0/0	0.23	0	0	0.12	0	0.17	-	-
6	0/0	0	0.23	0	0	0.34	0	0/0	0.28	0	0	0.15	0.01	0.3	-	-
7	0/0	0	0.42	0.01	0.01	0.45	0.01	0/0	0.38	0	0	0.22	0.91	8.33	-	-
8	0/0	0.01	1.6	0.06	0.08	1.31	0.03	0/0.01	0.62	0	0	0.29	126.69	t/o	-	-
9	0/0	0.03	11.8	0.52	0.81	12.19	0.05	0/0.04	1.8	0.01	0	0.41	-	-	-	-
10	0.01/0	0.12	94.91	5.86	8.59	140.53	0.08	0/0.14	8	0.01	0.01	0.78	-	-	-	-
11	0.03/0.01	0.47	t/o	28.21	110.25	t/o	0.14	0.01/0.51	42.62	0.03	0.04	2.47	-	-	-	-
12	0.06/0.03	1.84	t/o	-	-	-	0.24	0.02/1.94	t/o	0.1	0.17	11.43	-	-	-	-
13	0.21/0.11	7.31	t/o	-	-	-	0.36	0.05/7.26	t/o	0.36	0.64	47.96	-	-	-	-
14	0.82/0.44	29.42	t/o	-	-	-	0.52	0.1/27.42	t/o	1.39	2.6	197.34	-	-	-	-
15	2.57/1.44	109.11	t/o	-	-	-	0.8	0.22/t/o	t/o	5.51*	9.78	t/o	-	-	-	-

Table 3: Performance of generating data structures. SciFe/e denotes generation only of encoding of structures, and CLP/no unoptimized red-black specification. Execution times are given in seconds (timeouts with t/o). Times with $*$ include garbage collection, while - denotes the number of structures being greater than $2^{31}-1$ (thus not enumerable by the considered tools).

[9] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. Synthesis modulo recursive functions. *SIGPLAN Not.*, 48(10):407–426, October 2013.

[10] Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. Constraints as control. In *POPL*, pages 151–164, 2012.

[11] Ivan Kuraj. Interactive code generation. Master's thesis, EPFL, February 2013.

[12] Colin Runciman, M Naylor, and F Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. *Acm sigplan Not.*, (September), 2008.

[13] Valerio Senni and Fabio Fioravanti. Generation of test data structures using Constraint Logic Programming. 2012, 2012.

[14] Kevin Sullivan, Jinlin Yang, and David Coppit. Software assurance by bounded exhaustive testing. *ACM SIGSOFT Softw. ...*, 31(4):328–339, 2004.