Programming with Enumerable Sets of Structures

Ivan Kuraj
CSAIL, MIT, USA
ivan.kuraj@csail.mit.edu

Viktor Kuncak
EPFL, Switzerland
viktor.kuncak@epfl.ch

Daniel Jackson
CSAIL, MIT, USA
dnj@mit.edu

Abstract

We present an efficient, modular, and feature-rich framework for automated generation and validation of complex structures, suitable for tasks that explore a large space of structured values. Our framework is capable of exhaustive, incremental, parallel, and memoized enumeration from not only finite but also infinite domains, while providing fine-grained control over the process. Furthermore, the framework efficiently supports the inverse of enumeration (checking whether a structure can be generated and fast-forwarding to this structure to continue the enumeration) and lazy enumeration (achieving exhaustive testing without generating all structures). The foundation of efficient enumeration lies in both direct access to encoded structures, achieved with well-known and new pairing functions, and dependent enumeration, which embeds constraints into the enumeration to avoid backtracking. Our framework defines an algebra of enumerators, with combinators for their composition that preserve exhaustiveness and efficiency. We have implemented our framework as a domain-specific language in Scala. Our experiments demonstrate better performance and shorter specifications by up to a few orders of magnitude compared to existing approaches.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging — testing tools; D.3.3 [*Programming Languages*]: Language Constructs and Features — frameworks

General Terms Algorithms; Languages; Verification

Keywords Dependent enumeration, data generation, invariant, pairing function, algebra, exhaustive testing, random testing, lazy evaluation, program inversion, DSL, SciFe

1. Introduction

Automated data structure generation techniques are useful in many contexts: software testing [10, 11, 18, 45], synthesis [30, 32], bounded and unbounded verification [3, 15, 25, 28], theorem proving [6, 12], and others [4, 9, 22, 24, 31]. The structures in question are not just the common algorithmic data structures that are the backbone of common libraries, but also domain-specific structures, such as abstract models of programs in compilers, image formats, or DOM trees in browsers. A complex data structure is often characterized not only by an inherent structure (for example, being a tree) that can be defined with typing information, but also by specific constraints or invariants (for example, that the tree is ordered or balanced). One of the main challenges in automated structure generation is meeting these constraints. In practice, many tools use *generate-and-test* approaches, which generate all possible values, testing each one in turn to filter out those that fail to satisfy the invariant. This quickly becomes unfeasible when the vast majority of generated instances fail to pass the test.

State of the art structure generation techniques. Over a decade of research in automated testing has brought a variety of approaches that differ in the trade-off they make between expressiveness, efficiency, flexibility and ease of use [4, 11, 13, 14, 18, 28, 42–44]. Our approach focuses primarily on providing what has been called *bounded-exhaustive testing* (BET), which amounts to generating all structures within a given bound [45] (in contrast to BET tools, however, it also supports enumeration from unbounded, infinite domains). In the context of software testing (the main application of SciFe), this approach has the advantage of high test coverage. It covers all "corner cases" within the given bound that might otherwise be difficult to construct [24]. A similar approach is taken in model-checking tools such as Alloy [25], where the claim that high proportion of bugs can be found by exhaustive consideration of all cases up to some size is referred to as the *small scope hypothesis*.

BET approaches face a trade-off between the expressiveness of specifications and the performance of the generation process. *Declarative* BET techniques (or filtering as defined in [18]) allow writing specifications as declarative predicates; the tool then searches for valid structures that satisfy them [4, 28, 43]. *Constructive* (or generating) approaches require more prescriptive specifications which directly construct valid structures [10, 11, 42], and thus do not require explicit search. Later work introduced hybrid techniques, in which both types of specifications are used in conjunction, aiming to hit a sweetspot in the tradeoff space [18, 40].

The design space. Different generation techniques provide different sets of features. While some techniques rely on a costly generate-and-test approach [22, 32, 42], efficient techniques usually avoid explicit enumeration of all possible structures [13, 28, 43]. Most of the prior work employs some form of backtracking: with a constraint solver [28, 40, 43], using custom search [4, 40], or non-deterministic execution [18]. Often, the search problem is offloaded (after some translation) to a search engine, which means that the generator specifications highly dependent on the intricate details that affect the search. Although declarative approaches allow expressing constraints in a natural way, the performance of the generation process becomes highly sensitive to to the details of how the specification is written, sometimes unpredictably so [4, 30, 43]—seemingly equivalent specifications may differ drastically in performance [13], § 8.1. Consequently, in order to improve the impractical performance of generation, specifications become too complex and verbose, and may need to be tweaked, making them more complex and verbose [13, 40]). The key advantages of declarative specification—namely concise expression of the desired properties and the ability to compose specifications in a modular fashion—may thus be lost.

Moreover, separating search parameters from the generator specification makes it harder to bound the search and provide incremental generation [4, 32, 35, 42]. This separation may significantly hamper the ability to control and reason about the generation process, which is necessary for performance [13]. Some tools eagerly explore a complete search space within the given bounds, even for enumeration of a few candidate instances. Such coarse bounds are undesirable since tools become very inefficient for larger bounds (and unusable when the bound is effectively infinity) [22, 32, 42], § 8.1.

In turn, many of the constructive approaches avoid searching unnecessary search space thus leading to efficient generation [11, 14, 42, 44]. These approaches take advantage of properties that are naturally easier to express with constructive definitions for certain domains. (For instance, it can be easier to construct a sorted list from a set of elements, or a sequence of random numbers sampled from a generator with an adequate distribution, than to express constraints that describe such results). However, constructive approaches tend to suffer from lack of expressiveness and resort to a form of generate-and-test in the presence of complex structural invariants [11, 14, 42]. For generating data structures with invariants that cannot be captured within the expressive power of their constructive definitions, performance may asymptotically reach that of generate-and-test, even with optimizations (such as lazy evaluation and memoization) [13].

Motivation. Although achievable in different ways, including straightforward generate-and-test approaches, the main challenge in BET comes from the efficiency of generation and the complexity of generator specifications. The premise of this work is that there are *sweetspots in the tradeoff space* yet to be explored. In particular, it seems that many of the drawbacks of search-based approaches arise precisely from one of their appealing aspects: namely a clean separation of the expression of the constraint from the generation process. While this separation of concerns does allow use of off-the-shelf solvers, a major opportunity is lost by decoupling the structure of the specification from the structure of the generation. This work aims to explore this opportunity with the opposite strategy, by using an expressive algebra of enumerators to make generation compositional over the structure of the constraint to achieve both expressiveness for a wide range of structures and efficiency in the generation process. It fills the gap between the specification and the search process, while providing *direct control* of the search process itself, without offloading to external procedures.

Our new framework: dependent enumeration We present a new framework, implemented within our tool SciFe[1], for enumeration of complex data structures, with better performance than previous tools and additional features besides exhaustive generation, such as the ability to recognize structures, incremental and lazy enumeration, and non-standard enumeration orders (for example efficient access to a structure in the middle of the enumeration series).

The framework is based on enumerator objects and enumerator combinators. The enumerators are defined as computable bijective functions between subsets of the natural numbers and values of the respective encoded sets of structures. This definition allows efficient querying of an enumerator for elements at particular positions in the enumeration sequence. Enumerator combinators provide means of both composing and refining enumerators. Additionally, when enriched with dependent enumerators that are mappings between values and other enumerators, our framework achieves expressiveness and flexibility for constructing enumerators that efficiently enumerate complex data structures.

A key aspect of SciFe is that its expressive framework makes it convenient to write and reason about the generation process by taking complex invariants into account. Whilst developers write code that defines structures to be enumerated and ultimately guides the generation process, SciFe allows them to focus on the important aspects of constructing a data structure at hand rather than on the mechanical production of its instances. Instead of manually constructing a data structure, the developer defines an enumerator that encodes a whole class of structures that can be queried to enumerate many valid instances that satisfy specified structural properties and invariants.

Enumerators in SciFe are specified in a *constructive*, albeit functional style, with a domain-specific language (DSL) that allows *concise* specifications that can express *efficient* generation of a wide range of complex structures. SciFe guarantees *exhaustive* enumeration of encoded values along with multiple useful features: *modularity*, which allows treating enumerators as self-contained building blocks, and thus allows debugging, reusing, composing, refining and optimizing them; *memoization*, which allows storing and sharing computed intermediate results, making enumeration more efficient; *random access*, which allows random testing, in addition to exhaustive enumeration,

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

by choosing data structures to enumerate at random index in the series; *fine-grained control* of enumeration, which allows incremental enumeration, and specialized enumeration orderings; *expressiveness* for complex structures with dependent enumeration, where constraints are associated with enumerators to avoid unnecessary search; *flexibility* of combinators, which allows arbitrary generation and transformation of enumerated instances; enumeration from *infinite domains*, for sequences of structures that are naturally unbounded; *performance guarantees* of arbitrarily complex enumerators, by relying only on well-defined composition operators with predictable behavior.

Exploring new portions of the design space for data structure generation opened up opportunities for novel features:

- *invertible enumeration* effectively runs enumeration backwards and computes an index of a given structure if it can be enumerated by an enumerator;
- *enumerator as an invariant*, as a form of inverse, allows enumerators to be used both for efficient enumeration and for structure recognition, thus suitable both for preconditions and postconditions of method contracts;
- *fast-forwarding* starts enumerating from the point of a given structure, in enumerators of potentially large domains;
- *parallelization* builds upon efficient direct access and allows distributing work and sharing intermediate results across parallel worker threads.
- *lazy enumeration* allows coupling of an enumerator and code under test to avoid exhaustively enumerating test cases that are guaranteed to pass the test.

SciFe also allows *generation of graph-like structures*, whereas many predecessor tools have been limited to trees.

SciFe is implemented as an embedded DSL in Scala that expresses the enumerator algebra in a generic and modular way. SciFe expresses concise generators that outperform prior tools by *orders of magnitude*, as shown by experimental comparison with a wide range of tools for exhaustive generation, on both existing and new benchmarks. Furthermore, by offering unique additional operations that go beyond exhaustive generation, SciFe demonstrates applicability to a variety of new scenarios.

2. Motivating Examples

Consider the problem of writing an operation for key-ordered (search) trees. Such a tree is given by a recursive type definition, as well as a set of invariants, including that tree nodes are ordered (that is, an in-order traversal of the tree would give a sorted list). In Scala, we can define a simple tree that maps integer keys using a set of case classes such as the following[2]:

```
trait Tree[T]; case object Leaf extends Tree[T]
case class Node[T](l:Tree, k:Int, v:T, r:Tree) extends Tree[T]
```

We might want to test the preservation of the invariant on some operations. Consider a method for inserting key x into the given Tree instance (for brevity, we omit the function body and the field v because our concern is with the invariant):

```
def +(x: Int): Tree = { require(isBST); ... } ensuring {
  (res: Tree) ⇒ res.isBST && res.keys == keys ++ Set(x) &&
    res.size == size + 1 || res == this }
```

The method performs a functional update, returning a new tree extended with a given integer key. Whereas type declarations ensure that the result is a tree, they provide no guarantee that the tree is ordered. To capture the preservation of such an invariant, we may use standard Scala preconditions (**require**) and postconditions (**ensuring**) to state that + takes and returns a tree that satisfies the predicate isBST [37]. To test this method, we can use runtime checking of contracts on each method invocation (e.g. as a part of a unit test), but still need to come up with test inputs, namely valid Tree instances, to exercise the operation. Using SciFe, we can instead provide an enumerator of trees up to a given size, and use it in both the precondition and postcondition: in the former to generate well formed trees, and in the latter (as an alternative to an executable predicate) to check that the result is well formed.

Exhaustive Testing of Search Trees using SciFe. Binary search trees can be constructed in an intuitive way by reusing the inductive property of the structure: we can construct a tree of size s by combining a value for the root with two smaller trees with sizes s_l and s_r, such that $s_l + s_r = s$. We may formulate such strategy for construction as a recursive function that guarantees construction of trees of a particular size and range of values. In a functional programming language, we can leverage a monadic definition of such a construction. Here is a suitable enumerator, defined using SciFe:

```
import enum._                                    // import SciFe's DSL
val bst_fc = rec[(Int, Range), Tree]({ case (self, (s, r)) ⇒
  if (s == 0) Leaf                               // enumerate only Leaf
  else for (m ← r;                               // choose root value
    rl = r.start until m; rr = m+1 to r.end;     // define ranges
    sl ← 0 until s; sr = s−sl;                   // choose subtree sizes
    tl ← self(sl, rl); tr ← self(sr, rr);        // enumerate subtrees
  ) yield Node(tl, m, tr) })
```

The enumerator exploits the inductive structure of trees by combining smaller subtrees, while respecting the orderedness property. The function rec is part of SciFe's DSL (see § 7.1) and is similar to the Y-combinator: its argument self is matched to the enumerator's recursive definition. The rest of the definition is standard Scala code that is transformed into an enumerator (with implicit conversions; see § 7.1). The argument (s, r) represents the given size and range of constructed trees. The enumerator starts by enumerating all values from the range for the root element m∈r. Due to orderedness, the chosen m in turn restricts the possible ranges for the left rl and right rr subtrees. Similarly, it enumerates all possible sizes for the left subtree sl from the range [0..s-1] and from that computes the right tree size sr. Using the computed size and range, self(sl, rl) recursively constructs left subtrees, which are then enumerated in tl (and similarly for the right subtrees tr). The last line of the for-comprehension completes the inductive construction by constructing the final trees.

[2] **trait** and **case object** mark abstract and leaf nodes in type hierarchies [37]

Although Scala's type system invokes implicit conversion that transforms the given for-comprehension code into an enumerator, SciFe uses a modular combinator framework to construct enumerators, control memoization of shared intermediate results, and optimize enumeration. To enable efficient enumeration and all features of SciFe, the developers write the following definition:

```
val bst = rec[(Int, Range), Tree]({ case ((s, r), self) ⇒ {
  if (size == 0) Leaf
  else {                    // corresponds to previously given for
  ((0 until size) ⊗ range) ⊘
    (self ↓ { case (ls, m) ⇒ (ls, r.start to (m−1)) } ⊗
    self ↓ { case (ls, m) ⇒ (s − ls − 1, (m+1) to r.end) }) ↑
    { case ((_, root), (lt, rt)) ⇒ Node(lt, root, rt) }
}}})
```

This enumerator definition follows the same idea as the for-comprehension definition given above, but illustrates the more general DSL provided by SciFe for defining enumerators. (This enumerator definition is explained in more details, in § 3.1.)

To exhaustively generate test inputs, one might write:

```
val e = bst_fc(15, 1 to 15)     // enumerate all trees of size 15
for (tree ← e) test(tree)                    // feed into test
```

This constructs trees of size 15 and range [1..15] and feeds them to some function test. Enumerators also support efficient indexing of structures, which is suitable for *randomized testing*:

```
for(i ← 1 to 10) test(e(rnd.nextInt(e.size))) // 10 random trees
```

Rather than enumerating sequentially, this code randomly generates 10 integers within the bounds of the enumerator (given by e.size) and uses them to index into the enumeration sequence defined by e to obtain random trees.

The enumerator, due to the inductive construction, computes all subtrees tl and tr only once and reuses them through memoization when constructing bigger trees. For trees of size 15 and a range of values [1..15], the enumerator enumerates all binary search trees (of which there are more than $9.6 \cdot 10^6$), in just a few seconds. This is *orders of magnitude faster* than state-of-the-art tools such as Korat, UDITA, HyTek, and constraint logic programming; see § 8.1.

Enumerators instead of contracts. SciFe alleviates the need to write an additional predicate to check the invariant after execution, by employing the enumerator as a *structure recognizer*. In addition to generating a data structure encoded at a particular index, the enumerator can play the role of an oracle—given a data structure, it answers with `true` or `false`, whether the data structure can be generated with the enumerator (and thus also whether it is correct)[3]. We can easily turn our enumerator into an oracle to test insertion without any additional contracts:

```
for (x ← 1 to 15; tree ← bst(14, 1 to 15)) {
  val newT = tree + x            // insertion without contracts
  assert(newT == tree || bst(15, 1 to 15).member(newT)) }
```

The member operation effectively runs the enumerator definition bst backwards; it starts from the end, decomposing the tree and trying to recognize components using the enumerator's inner structure; see § 6.2. In addition to saving developers' effort, enumerators jointly optimize and reuse memoization for enumeration and membership. As a result, the fast structure lookup provided here outperforms naively executing an **ensuring** predicate by 4.55x on average for inserting values $x \in (1..15)$ into trees of size 14; see § 8.3.

Lazy enumeration. The developers can also request lazy enumeration in our framework, which can provide savings especially in cases where a large portion of generated structures does not need to examined by the code at hand. This feature follows the idea of lazy evaluation, a technique which delays computation until its results are needed. In the case of lazy enumeration, the enumerator definition and the testing method remain the same; the developer simply includes a different set of implicit definitions when specifying the enumerator:

```
import enum.lzy._          // import "lazy" enum combinators
val bst = rec(...)                 // define enumerator as before
```

The result of this import is that the enumerator is embedded into the field computation of the structure and is triggered only when needed. The default enumeration is strict as laziness may result in unpredictable computations and overheads; this is because SciFe uses costly Scala lazy values [37]; see § 7. Lazy enumeration can be performed with a for-comprehension:

```
for (v ← 1 to 15; e = bst(14, 1 to 15); t ← e) {   // skips trees
  e.reset; test(t + v) }                // insert v and track accesses
```

The enumerator still accepts queries for enumeration at a given index, but the for-loop now achieves exhaustive enumeration without actually generating all trees. The key insight behind this feature is that if an operation (for a given input) does not use a part of a data structure, it will surely not use it if the rest of the structure remains unchanged. The developers just need to "signal" tracking for a new input with reset. The enumerator internally tracks accesses to the structure and uses the information to skip enumeration of whole sets of structure parts; see § 6.1. In the running example, lazy enumeration avoids enumeration of 23.2M, about 62%, of the total number of possible key-tree pairs and still guarantess exhaustive coverage; see § 8.3.

Parallelizing enumeration. Thanks to the ability to access elements at a given index, it is straightforward to enumerate structures in parallel using SciFe. The following import statement constructs an enumerator that permits concurrent access:

```
import enum.par.lockFree._        // concurrency combinators
val bst = rec(...)                 // define enumerator as before
val enum = bst(15, 1 to 15)                    // trees of size 15
```

Here we import lock-free combinators for memoization, which perform recomputation rather than waiting. The developers can choose a favorite concurrency construct, such as Java concurrent executors, and enumerate with numThr threads:

```
def runner(e: Enum[Tree], b: Int) = new Runnable {
  def run = { for(i ← b until e.size by numThr) test(e(i)) } }
val ex = Executors.newFixedThreadPool(nThr)
for (t ← 0 until nThr) ex.submit(runner(enum,t))       // start
ex.shutdown                  // shutdown and wait for termination
```

[3] Moreover, enumerators can return indexes of structures in the enumeration series and fast-forward to them, § 6

Each worker enumerates a disjoint set of trees by enumerating a sequence that consists of every numThr-th element starting from an appropriate index (i.e. the enumeration step is numThr, instead of 1 as used previously). Having separate workers compute distinct instances of trees while still sharing intermediate results leads to great potential runtime improvements; see § 8.4. With 8 cores (and nThr==8), parallel enumeration yields a 4.1x speedup over execution with a single thread.

Benefits of our framework. The running example shows a new perspective on practical checking of code correctness. Efficient exhaustive enumeration and recognition in SciFe alleviate some limitations of other approaches applicable to such a scenario, such as random testing, where finding the appropriate input gets challenging due to generator's distribution [9, 24], and (unbounded) verification, which provides strong guarantees, but might be more restrictive in supported specifications.

The given enumerator demonstrates that encoding the invariant in the process of generation leads not just to more control and efficiency, but also more concise specifications. Some BET tools require writing finitization procedures, in addition to invariants, to control the search. Korat [4], e.g. requires more than 100 lines of specification (of imperative, Java) code, much more than the given enumerator; see § 8. SciFe's conciseness stems from leveraging Scala constructs: once the DSL is imported into the scope, the enumerator uses only combinators and rec (while the rest of the code is standard Scala, and the type inference and implicit conversions trigger the necessary construction, while **for** loops perform enumeration). Note that SciFe is implemented as an embedded DSL within the host language—there is no need for additional code compilation; see § 7.1.

The binary search tree example illustrates the key insight behind this work: by providing different modes of operation of the same combinators, the enumerator framework can enhance enumerator definitions with optimizations and new features. Note that while the given for-comprehension enumerator definition bst_fc demonstrates the idea and is a correct way of constructing enumerators, it is the direct application of the DSL that enables all features. The modularity of the given enumerator, where different enumerators are responsible for enumerating parts of the data structure, enabled further observations: generation according to a specification is just one of the two coupled tasks, alongside invariant checking, which allows the framework to reverse the behavior of combinators and transform enumerators into efficient checkers; and, enumerators can observe accesses to the parts of enumerated structures and use the feedback to control and optimize the exhaustive enumeration. Moreover, the running example shows positive impact of combining certain features, where the performance of both enumeration and recognition benefit from memoization. Note that these features do not require the developer to modify the specification; the enumerator is adapted to the data structure interface, while both the definition and code under test are treated as black box.

Support for a variety of data structures. The running example demonstrates the main idea behind expressive enumeration, which is also applicable to a variety of more complex data structures, such as Red-Black trees, B-trees and image formats; see § 8. In addition to the given enumerator, SciFe allows enumerating search trees as pairs of values that encode the tree structures and key ordering separately. This is similar to composition of orthogonal specifications in declarative approaches [43]; in addition, it brings additional performance gains, see § 8.1. Furthermore, constructive definitions in SciFe are geared towards, but not limited to, inductive and immutable data structures—SciFe supports enumerating arbitrary mutable data structures, possibly defined in external libraries, since it does not require their definitions, but only their constructors to be exposed.

Unlike related tools that focus only on tree-like structures [14, 42, 43], SciFe supports graph-like structures by leveraging specialized encoding of inductive algebraic representation of graphs (similar in spirit to the functional-style inductive graphs [16]). § 8 presents, among others, results for enumerating graph-like structures, while § 7.3 describes how graph structures are enumerated with SciFe.

3. Overview of SciFe

SciFe enumerators provide efficient *encoding* and *enumeration* of sets of elements from particular domains. We next illustrate the basic constructs of our enumerator DSL and their usage.

3.1 Basic enumerator concepts

In the implementation of SciFe, the basic enumerator type that enumerates a set elements of type T is defined as:

```
trait Enum[T] extends Function[Int, T] {
  def apply(ind: Int): T            // enumerate ind-th structure
  def hasDefiniteSize: Boolean      // whether it is finite
  def size: Int }
```

Enum[T] can be viewed as a function from a subset of natural numbers to values of type T (we restrict the function to be bijective; see § 4.1). The apply method defines the *indexing function* which returns encoded values—in Scala, this allows us to write e(i), which returns the i-th element of the set of elements encoded by e. The size of the encoded set of elements can be retrieved with e.size. Only values in the range [0..e.size-1] are valid inputs to the indexing function if hasDefiniteSize==true; otherwise, the enumerator is infinite and can be indexed with any positive integer.

SciFe provides the means for transforming standard programming language elements, such as collections and functions, into enumerators; see § 7. Enumerator objects together with *combinators* form an enumerator algebra; see § 4. Using enumerator combinators developers can compose and refine existing enumerators to build more complex ones. Tables 1 and 2 summarize combinators in SciFe (the combinators have both unicode and ASCII variants); we describe them in more detail in § 4.1.

Enumerator of dates. To demonstrate basic usage of the DSL, we construct an enumerator that exhaustively generates valid, standard Java, Date objects. This simple example shows how developers can use combinators from Table 1 to build interesting enumerators by composing simple ones.

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	$x \uparrow f$	map elements of x with f (x:Enum[T], f:T⇒V)⇒Enum[V]
filter	$x \succ p$	filter elements of x with p (x:Enum[T], y:T⇒Boolean)⇒Enum[T]

Table 1: Basic enumerator combinators.

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	$x \downarrow f$	map dependent argument of x with f (x:Depend[D,T], f:V⇒D)⇒Depend[V,T]
bind	$z \oslash x$	enumerate elements of x according to z (z:Enum[D], x:Depend[D,T])⇒Enum[(D, T)]

Table 2: Dependent combinators.

```
val dates = (1 to 31) ⊗ (1 to 12) ⊗ Stream.from(2015) ≻
  { isValid(_) } ↑ { case ((d, m), y) ⇒ new Date(y, m, d) }
```

The enumerator encodes all possible dates starting from the year of 2015. We compose enumerators of date, month, and year ranges, to enumerate their Cartesian product using the ⊗ combinator. Note that there is an infinite number of constructed tuples because the enumerator for years is infinite (which is constructed from infinite Scala streams [37]). The ≻ combinator filters the tuples using the isValid function defined elsewhere (which determines whether the given values for date, month, and year represent a valid date, taking into account, for example, leap years). Valid tuples are then transformed with the ↑ combinator and the anonymous function that constructs Date objects.

The enumerator can be used in different ways:

```
dates.hasDefiniteSize                          // returns false
for (i ← 3 to 365 by 7) print(dates(i))        // Sundays in 2015
dates(302)          // enumerate start date of OOPSLA 2015
for (d ← dates) yield print(d)       // prints all dates (diverges)
```

In addition to enumerating by invoking the indexing function directly, SciFe allows treating enumerators as standard collections (through the use of implicit conversions § 7.1), thus we can perform enumeration using, for example, for-comprehensions, as shown in the examples so far.

3.2 Dependent enumerators

In addition to having enumerators as mappings between natural numbers and values, which we will refer to as *first-order*, we extend the framework with *dependent* enumerators. Dependent enumerators add the necessary expressiveness to the algebra to allow efficient enumeration of complex structures, such as binary search trees (as shown in § 2). In comparison to first-order enumerators, rather than encoding a single set of values, a dependent enumerator encodes multiple sets of values—one per each value in the function's domain.

Dependent enumerators implement the Depend[I,O] trait:

```
trait Depend[I, O] extends Function[I, Enum[O]] {
  def apply(p: I): Enum[O] }          // returns an enumerator
```

Effectively, a dependent enumerator represents a function I ⇒ Enum[O] from arbitrary values to first-order enumerators, thus encoding a first-order enumerator for each value p of the dependent parameter[4].

To allow composing and refining dependent enumerators, the DSL supports the dependent combinators given in Table 2. As the example in § 2 illustrates, dependent enumeration adds the necessary expressiveness for defining efficient enumeration of data structures with complex invariants. By having enumerators depend on parameter values, they allow expressing constraint propagation during the generation process, instead of enumerating a larger space of structures and filtering undesired elements. This effectively allows capturing complex invariants by making generation compositional over the structure of the desired constraint.

Dependent enumerator of binary search trees. The enumerator definition bst, given in § 2, uses combinators and enables efficient indexing by using optimizations and memoization. It defines a dependent enumerator that depends on a pair of values (s, r), of type (Int, Range), which represent the tree size and range of values—for a given pair of values (s, r), bst enumerates all instances of Tree 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. We write definitions as (partial) functions; the rec construct then creates a dependent enumerator of type Depend[(Int, Range), Tree] from them; see § 7.1. The construction of trees follows the same case analysis as described for the enumerator bst_fc, defined with a for-comprehension: for size 0, we enumerate only the Leaf instance, while for $s \geq 1$, we choose size for the left subtree and a root value and compute pairs of values for size and range to enumerate both subtrees recursively. The ↓ combinator transforms enumerators so that they depend on pairs (ls, m) and can be used to enumerate subtrees with the recursive call self. The combinator ↑ maps the resulting tuples and constructs final trees.

As mentioned in the previous section, the difference between the two definitions, bst and bst_fc, is that for-comprehensions impose (a monadic) ordering of values through the computation, while direct combinator application allows: more control over the enumeration orderings (for reasons similar to the advantages

[4] If we view enumerators as a more precise description of types, then dependent enumerators correspond to dependent-types

of arrows over monads, see [23]), including enumeration from infinite domains (see § 5.2); modular composition with combinators; and optimizations (see § 7.2).

4. Enumerator Framework

This section formalizes the enumerator framework as an algebra expressible with a simple language and defines the semantics of enumeration. One appealing property of our enumerators is that they have very natural mathematical foundations. SciFe's DSL implements the core enumerator language and guarantees exhaustive enumeration, efficiency and composition.

4.1 Enumerator algebra

The enumerator algebra defines enumerators as mathematical objects, with combinators as its operations. One of the key insights behind SciFe is that basing a generator on a small, well-defined algebra—with only a small number of enumerator classes and combinators for their composition—is sufficient to express enumeration of complex structures. Additionally, all the features and strategies of enumeration are then easily defined by extending the algebra. All the presented enumerator features are defined as extensions of the algebra; see § 6.

First-order enumerators. The basic type of an enumerator is the *first-order* enumerator. In general, it can encode enumeration of both finite and countably infinite sets. Formally, it represents a bijection from a subset of natural numbers.

If p is a function (by which we mean a total function), then $dom(p)$ denotes its domain. Let $\mathbb{N}_0 = \{0,1,...\}$ denote natural numbers (non-negative integers), $A \xrightarrow{1:1} B$ denote a bijective function from A to B, and $\mathbb{N}_{0..|S|}$ denote $\{0,...,|S|-1\}$ if S is finite and $\mathbb{N}_0$ otherwise.

DEFINITION 4.1. *A first-order enumerator is a pair (S,p), where $p : \mathbb{N}_{0..|S|} \xrightarrow{1:1} S$ is a bijective function from a subset of natural numbers to a countable set of "encoded" elements S.*

Thus, we restrict the domain of the function to natural numbers smaller than the cardinality of the set of encoded elements ($dom(p) = \mathbb{N}_{0..|S|}$). As shown in § 3.1, in SciFe, `Enum[T]` designates a first-order enumerator $e = (S,p)$ where all elements of S are of type T, i.e. $\forall x \in S.x : T$.

The behavior of the indexing function, and thus the process of enumeration, is completely determined by the *mapping function* p of the respective enumerator object. In turn, mapping functions are constrained (but not uniquely determined) by the rules of constructing atomic first-order enumerators (see § 7.1) and their composition with combinators (as presented subsequently).

Dependent enumerators. Dependent enumerators are simply functions from some parameter set P to first-order enumerators. Therefore, applying a dependent enumerator to a parameter results in a first-order enumerator. Let $\pi_1(e) \equiv S$ simply project the set of encoded elements of $e = (S,p)$.

DEFINITION 4.2. *A dependent enumerator e_d is a tuple $(E,\ t)$ where E is a set of first-order enumerators (as defined in Definition 4.1) and t is a function $D \rightarrow E$ that maps values of D to enumerators from E, where $\forall i,i' \in D.i \neq i' \rightarrow \pi_1(t(i)) \cap \pi_1(t(i')) = \emptyset$.*

Effectively, dependent enumerators encode, for each value of the "dependent parameter" $d \in D$, a corresponding (inner) first-order enumerator. The condition $\pi_1(p(i)) \cap \pi_1(p(i')) = \emptyset$ restricts the dependent enumerator to encode inner enumerators that enumerate disjoint sets for different dependent values (to preserve bijectivity, as discussed subsequently). In SciFe, enumerator `Depend[I, O]` designates $e_d = (E,t)$, where t has the type `I`$\rightarrow$`Enum[O]` and encodes the inner enumerator $t(d)$ for a value `d: I`, such that $d \in D$. Note that we do not restrict the domain D; dependent enumerators can be queried with (i.e. to depend on) an arbitrary type of values, including tuples of values.

Combinators. Combinators represent operators of the enumerator algebra. They apply to enumerators and other values (such as functions). We define combinators as rules that govern the set of operands to which a combinator can apply and the result of such application.

DEFINITION 4.3. *A combinator @ defines a set $C_@$ of tuples (e,v,e_r) where e and e_r are enumerators, v is an enumerator or a value, and applying @ to e,v yields e_r, denoted $e_r = e@v$.*

We say that e_r is a *composite enumerator* composed with @. This definition serves us to define the semantics of combinator applications within our language. For example, $C_\oplus$ restricts the application of $e_r = e_1 \oplus e_2$ to tuples (e_1,e_2,e_r) where e_r encodes a disjoint union of elements from e_1 and e_2. We define the instantiated combinators in more details, in § 4.3.

Remarks on properties of enumerators. Let us examine some consequences of our definitions for the implementation of enumerators. As per Definition 4.1, for a first-order enumerator $e = (S,p)$, the bijectivity of p implies that:

- each enumerated element belongs to the encoded set, i.e. $\forall i \in \mathbb{N}_{0..|S|}.\ p(i) \in S$
- each encoded element gets (eventually) enumerated, i.e. $\forall s \in S.\exists i \in \mathbb{N}_{0..|S|}.\ p(i) = s$

To have a valid enumeration of a countable set S, the mapping function $p : \mathbb{N}_{0..|S|} \xrightarrow{1:1} S$ must define a deterministic ordering according to which e (eventually) enumerates all elements from S. In case of S being infinite, for each element $s \in S$, there must exist a finite sequence of encoded elements in which s is enumerated. Effectively, enumerators need to enumerate increasing subsets of S with increasing finite prefixes of the enumeration sequence, exhaustively without duplicates. Since composite enumerators combine encoded sets from multiple, potentially infinite, enumerators, this imposes intricate restrictions on the underlying mapping functions; see § 5.

To ensure sanity of the framework, the enumerator algebra must be closed over the instantiated combinators. The closure is guaranteed by the inference rules that restrict combinator applications according to the underlying enumerator types and specific instantiated combinators, as presented subsequently.

$$\frac{V(x)=e_x \quad e_x=(S,p) \quad r=|S|}{V,\texttt{let}\ x'=|x|\ \texttt{in}\ t \longrightarrow (V,y\mapsto r),\texttt{let}\ x'=y\ \texttt{in}\ t}\ \text{(R-SIZE)}$$

$$\frac{V(x)=e_x \quad e_x=(S,p) \quad V(y)=r \quad p(r)=r' \quad r\in dom(p)}{V,\texttt{let}\ x'=x[y]\ \texttt{in}\ t \longrightarrow (V,z\mapsto r'),\texttt{let}\ x'=z\ \texttt{in}\ t}\ \text{(R-IND)}$$

$$\frac{V(x)=e_x \quad e_x=(S_x,p_x) \quad V(y)=v \quad r=e_z \quad (e_x,v,e_z)\in C_@}{V,\texttt{let}\ x'=x@y\ \texttt{in}\ t \longrightarrow (V,z\mapsto r),\texttt{let}\ x'=z\ \texttt{in}\ t}\ \text{(R-COMB)}$$

Figure 1: Reduction rules for enumerators.

$P ::= l$	program
$l ::= \texttt{let}\ x=t\ \texttt{in}\ l \mid x$	let binding, variable
$t ::= l \mid x[y] \mid$	term, indexing function
$\quad\quad \lvert x\rvert \mid x@y$	size, combinator application

Figure 2: Syntax of the core language and the fragment, where x,y represent variables.

$V ::=$	$\emptyset \mid (V,y\mapsto r)$	environment $(y\notin dom(V))$
$v ::=$	$e \mid v$	values
$e ::=$	(S,p)	enumerator

Figure 3: Elements of the core language and the fragment.

4.2 Enumerator language

We formalize the enumerator language that expresses the enumerator algebra as a fragment within the host (core) language. The fragment is sufficiently expressive for defining and composing enumerators according to the algebra. The minimal restrictions imposed by the fragment make the enumerator framework amenable to implementation in a variety of host languages. Effectively, the semantics of the enumerator fragment ensures that an enumerator term is reducible to a value only if the construction and invoked operations follow the rules of the algebra.

To formalize the enumerator language fragment, we employ a standard approach based on the operational semantics for a core language [34]. (We only focus on the fragment, as our goal is not to fully formalize the core language.)

Core language syntax. Figure 2 shows the core language syntax. We simplify the core language and restrict our attention to let-bindings. Alongside the environment, which maps variables to values, the language includes elements that express the enumerator algebra. Figure 3 summarizes these elements: $x[i]$ indexes the enumerator x with i; $|x|$ returns the enumerator size; $x@y$, applies the given (binary) combinator $@$ to values in x and y, where $@$ belongs to the set of instantiated combinators in the enumerator framework, which define $C_@$.

Dynamic semantics. We use a small-step operational semantics to formalize the enumerator language fragment. Figure 1 shows reduction rules, written in the form $V,t \longrightarrow V',t'$: a rule defines that, in the environment V, term t is reduced to t' and the environment becomes V'.

According to the grammar shown in Figure 2, expressions are always reduced in the context of a let-binding. Rule R-SIZE shows that querying enumerator size must reduce the argument to an enumerator value before returning the cardinality of the encoded set of elements. R-INDEX shows reduction of the indexing operation: it restricts the argument values to belong to the domain of the enumerator's mapping, while the result is defined by the indexing function. Indexing with values outside the domain results in a stuck term (and the implementation should not allow it—SciFe throws an exception § 7). R-COMBINE shows that combinator application is reduced by reducing values of the operands, y and an enumerator ϵ_x, and then constructing the resulting enumerator ϵ_y according to a valid tuple (e_x,v,e_z) defined by the combinator rule $C_@$. The applied combinator rule is determined by the specific instance of the combinator.

4.3 Combinators in SciFe

SciFe instantiates eight combinators over the two main classes of enumerator objects in the framework. We show inference rules that define semantics of combinators in terms of underlying formal enumerator objects, as well as their types (that correspond to types in the implementation). They define rules $C_@$ that determine the semantics of the combinator application in the enumerator fragment of the language.

First-order combinators. First-order combinators (shown in Figure 1) correspond to standard operations on sets, lifted to the corresponding bijections. The notation $\_ : A \xrightarrow{1:1} B$ denotes an unnamed bijection from A to B, with different occurrences denoting possibly different bijections. If we view the effect of operations on the range of the bijection, then we obtain natural set-theoretic operations of disjoint union for $\oplus$, Cartesian product for $\otimes$, intersection with a set defined by the predicate for $\succ$, and the function image operation for $\uparrow$.

Inference rules for these combinators are given in Figure 4: $\oplus$ (defined by the MERGE rule) yields an enumerator that enumerates a disjoint union of sets of elements of the underlying enumerators (note that the type of enumerated elements U is the most specific ancestor of T and V in the type hierarchy, *mgu(T,V)*); $\otimes$ (PRODUCT) enumerates Cartesian product of the underlying sets of elements, of the product type (T,V); $\uparrow$ (MAP) applies the given function f to the set of elements at the time of the enumeration (f determines the type of enumerated elements); $\succ$ (FILTER), accepts a predicate function f and enumerates only elements that satisfy the predicate.

Dependent combinators. Inference rules for dependent combinators (from Figure 2) are given in Figure 5. Conceptually, dependent $\oplus$ and $\otimes$ apply the first-order $\oplus$ and $\otimes$, respectively, pointwise to inner enumerators, for each parameter value. Specifically, dependent merge ($\oplus$, defined by the rule MERGE) takes

$$\text{MERGE}\ \frac{\begin{array}{c} e_1 = (S_1, \_ : \mathbb{N}_{0..|S_1|} \xrightarrow{1:1} S_1) \qquad \texttt{e1:Enum[T]} \\ e_2 = (S_2, \_ : \mathbb{N}_{0..|S_2|} \xrightarrow{1:1} S_2) \qquad \texttt{e2:Enum[V]} \\ S_1 \cap S_2 = \emptyset \qquad e = e_1 \oplus e_2 \end{array}}{\begin{array}{c} e = (S_1 \uplus S_2, \_ : \mathbb{N}_{0..|S_1|+|S_2|} \xrightarrow{1:1} S_1 \uplus S_2) \\ \texttt{e:Enum[U]} \qquad mgu(T,V) = U \end{array}}$$

$$\text{PRODUCT}\ \frac{\begin{array}{c} e_1 = (S_1, \_ : \mathbb{N}_{0..|S_1|} \xrightarrow{1:1} S_1) \qquad \texttt{e1:Enum[T]} \\ e_2 = (S_2, \_ : \mathbb{N}_{0..|S_2|} \xrightarrow{1:1} S_2) \qquad \texttt{e2:Enum[V]} \\ e = e_1 \otimes e_2 \end{array}}{e = (S_1 \times S_2, \_ : \mathbb{N}_{0..|S_1|\cdot|S_2|} \xrightarrow{1:1} S_1 \times S_2) \qquad \texttt{e:Enum[(T,V)]}}$$

$$\text{MAP}\ \frac{\begin{array}{c} e = (S, \_ : \mathbb{N}_{0..|S|} \xrightarrow{1:1} S) \qquad \texttt{e:Enum[T]} \\ f : S \xrightarrow{1:1} S' \qquad \texttt{f:T} \Rightarrow \texttt{V} \\ e' = e \uparrow f \end{array}}{e' = (S', \_ : \mathbb{N}_{0..|S|} \xrightarrow{1:1} S') \qquad \texttt{e':Enum[V]}}$$

$$\text{FILTER}\ \frac{\begin{array}{c} e = (S, \_ : \mathbb{N}_{0..|S|} \xrightarrow{1:1} S) \qquad \texttt{e:Enum[T]} \\ S' = \{s \in S \mid f(s)\} \qquad \texttt{f:T} \Rightarrow \texttt{Boolean} \\ e' = e \succ f \end{array}}{e' = (S', \_ : \mathbb{N}_{0..|S'|} \xrightarrow{1:1} S') \qquad \texttt{e':Enum[T]}}$$

Figure 4: Inference rules constraining first-order combinators.

$$\text{MERGE}\ \frac{\begin{array}{c} e_1 = (E_1, p_1) \qquad p_1 : D_1 \to E_1 \qquad \texttt{e1: Depend[I, T]} \\ e_2 = (E_2, p_2) \qquad p_2 : D_2 \to E_2 \qquad \texttt{e2: Depend[I, V]} \\ \forall i \in D_1 \cap D_2 . p_1(i) \cap p_2(i) = \emptyset \qquad e = e_1 \oplus e_2 \end{array}}{\begin{array}{c} e = (E, p) \qquad p : D \to E \qquad D = D_1 \cap D_2 \\ \forall i \in D . p(i) = p_1(i) \oplus p_2(i) \\ \texttt{e: Depend[I, U]} \qquad mgu(T,V) = U \end{array}}$$

$$\text{PRODUCT}\ \frac{\begin{array}{c} e_1 = (E_1, p_1) \qquad p_1 : D_1 \to E_1 \qquad \texttt{e1: Depend[I, T]} \\ e_2 = (E_2, p_2) \qquad p_2 : D_2 \to E_2 \qquad \texttt{e2: Depend[I, V]} \\ e = e_1 \otimes e_2 \end{array}}{\begin{array}{c} e = (E, p) \qquad p : D \to E \qquad D = D_1 \cap D_2 \\ \forall i \in D . p(i) = p_1(i) \otimes p_2(i) \\ \texttt{e: Depend[I, (T, V)]} \end{array}}$$

$$\text{INMAP}\ \frac{\begin{array}{c} e = (E, p) \qquad p : D \to E \qquad \texttt{e: Depend[I, T]} \\ f : D' \xrightarrow{1:1} D \qquad \texttt{f: I'} \Rightarrow \texttt{I} \\ e' = e \downarrow f \end{array}}{\begin{array}{c} e' = (E, p') \qquad p' : D' \to E \\ \forall i \in D' . p'(i) = p(f(i)) \qquad \texttt{e': Depend[I', T]} \end{array}}$$

$$\text{BIND}\ \frac{\begin{array}{c} e_1 = (S, \_ : \mathbb{N}_{0..|S|} \xrightarrow{1:1} S) \qquad \texttt{e1: Enum[T]} \\ e_2 = (E, p) \qquad p : D \to E \qquad \texttt{e2: Depend[T, V]} \\ e = e_1 \oslash e_2 \end{array}}{\begin{array}{c} S' = \{(i,t) \mid i \in S \cap D, t \in \pi_1(p(i))\} \\ e = (S', \_ : \mathbb{N}_{0..|S'|} \xrightarrow{1:1} S') \qquad \texttt{e: Enum[(T, V)]} \end{array}}$$

Figure 5: Inference rules constraining dependent combinators.

two dependent enumerators e_1 and e_2, and yields a dependent enumerator that, for a dependent parameter i, applies first-order merge to inner enumerators returned from indexing e_1 and e_2 with i. Analogously, PRODUCT defines $\otimes$. The type of inner enumerators in the resulting dependent enumerators is determined by the inner application of first-order combinators. Note that the domain of the dependent parameter becomes an intersection of domains of the underlying enumerators. Inmap ($\downarrow$, INMAP) is a function composition that changes the type of the dependent parameter of the given enumerator. It applies f to the parameter value before indexing and thus, effectively, adapts the dependent enumerator to the domain of f. (In § 3, left and right subtrees are enumerated in pairs with such an adaption). Bind ($\oslash$, BIND) generalizes a dual function composition and changes the type of the enumerated value in the resulting bijection. It provides a way for composing first-order e_1 with dependent enumerator e_2 and enumerating elements from inner enumerators of e_2, according to dependent parameter values enumerated from e_1. The result is a first-order enumerator that enumerates a union of elements S' encoded by inner enumerators of e_2, paired with corresponding dependent values. (This combinator is similar to the *monadic bind* operation in functional languages and `flatMap` in Scala). SciFe applies different enumeration algorithms to enumerate S' depending on the underlying enumerators e_1 and e_2; see § 5.2. This combinator is essential in our benchmarks, as it allows enumerating sets of structures that depend on different values (e.g. different sizes and ranges, as in enumerating subtrees in § 2).

Bijectivity. The enumerator algebra assumes that each atomic enumerator enumerates with a bijective mapping and ensures all combinators preserve bijectivity during composition. This in turn ensures that every enumerator in the algebra is bijective. The supplied function f for map and inmap is restricted to be bijective since otherwise, f might map to a smaller set S'. The chosen semantics of filter can only shrink the set of encoded elements and thus preserves bijectivity. However, SciFe's implementation does not restrict enumerator construction and function arguments to combinators that might break bijectivity; see § 7. Note that all our benchmarks construct bijective enumerators; see § 8.

Expressiveness of enumeration. The class of structures that can be enumerated is limited solely by the rules for combinator composition, which do not restrict the enumeration ordering and allow different implementations. Therefore, $C_@$ allows picking different implementations for combinator applications, according to the context (e.g. of parallel enumeration); see § 7.1. The algebra guarantees a valid resulting enumerator, according the semantics (while optimizations in SciFe preserve the semantics; see § 7.2). Moreover, the algebra does not impose any restrictions on the expressiveness of encoded values, since dependent enumeration and map combinators can be instantiated

with arbitrary functions (and for this reason, inverse indexing requires developers to provide additional function inverses; see § 6.2). However, the enumerator algebra is designed for efficient enumeration of data structures with invariants that can be expressed in terms of dependent enumeration, where the inductive property allows sharing of intermediate structures between enumerated elements (and thus enabling performance gains with memoization). With the exception of filter—which in order must use sequential testing in the general case—the instantiated combinators provide efficiently computable indexing functions. Our benchmarks show enumeration of a variety of data structures, where efficiency comes from both memoization and dependent enumeration alone (by avoiding backtracking); see § 8.

5. Enumeration Algorithms

Properties of the enumerator algebra necessitate an exhaustive and efficient indexing function over all elements encoded by enumerators. To satisfy these properties, the enumerator framework needs to incorporate mapping functions for each of its combinators, according to the types of enumerators they apply to and different desired enumeration orderings, including enumerations from both finite and infinite domains. SciFe leverages several mapping functions from combinatorics for different enumeration strategies, most notably Cantor's and Szudzik's pairing for infinite domains [39, 46]. We introduce a new mapping function that is used for parallel enumeration. To the best of our knowledge, this is the first work that applies these pairing functions for enumeration in general-purpose structure generation.

5.1 Indexing function requirements

The indexing function retrieves an element from the encoded set at a given index as long as it is within the domain of the enumerator's mapping. Thus, mapping functions for first-order enumerators form the backbone of the enumeration. (Since every enumerator expression reduces to a first-order enumerator before enumerating elements, as shown in § 4.2). Bijective mapping for first-order enumerators is achieved with two functions:

- *decompose*, $p_d : \mathbb{N} \to (\mathbb{N},...,\mathbb{N})$, which decomposes a given number into n component numbers
- *compose*, $p_c : (\mathbb{N},...,\mathbb{N}) \to \mathbb{N}$, which composes n component numbers into a single number

where, if $i \in dom(p_d)$ and $\overline{j} \in dom(p_c)$, where $\overline{j}$ is a tuple, then $p_d(i) = \overline{j}$ and $p_c(\overline{j}) = i$, and n depends on the type of the enumerator. (E.g. for application of $\otimes$ n=2, and for map n=1, i.e. p_d and p_c are identity functions). Together, decompose and compose represent a pairing function that realizes the mapping for an enumerator. To achieve exhaustive and efficient enumeration, the pairing function must be bijective and efficiently computable.

Decompose function p_d is used when indexing a composite enumerator $e@x = (S,p)$ with i, which computes $p_d(i) = \overline{j}$ and uses $\overline{j}$ to index underlying enumerators (and compute the result according to the given combinator @). Compose function p_c is used when inverting an enumerator; see § 6.2. In general, any functions p_d and p_c that achieve bijective mapping are applicable as long as the number of components and their respective domains (and ranges) correspond to the combinator semantics. Note that combinator semantics allows various orderings, and thus customization of enumeration strategies, by leveraging different pairing functions.

5.2 Pairing functions for enumeration

Combinators that compose first-order enumerators, namely merge ($\oplus$), product ($\otimes$), and bind ($\oslash$), require mapping to multiple component numbers. SciFe utilizes different pairing functions for these combinators according to different types of underlying enumerators, that guarantee properties of the algebra.

Enumeration from finite domains. In the case of finite enumerators, efficient pairing functions are constructed in an intuitive way. We assume e_1 and e_2 are finite, and the desired mapping is between the value i and components $\overline{j} = (x,y)$ (i.e. $n = 2$). Let $e = (S,p)$, $e_1 = (S_1,p_1)$, $e_2 = (S_2,p_2)$, represent different enumerators for each presented pairing function.

For $e = e_1 \oplus e_2$ the mapping p is defined by:

$$p_d(i) = \begin{cases} (0,i) & \text{if } i < |S_1| \\ (1,i-|S_1|) & \text{otherwise} \end{cases} \qquad p_c(x,y) = x \cdot |S_1| + y$$

The value of the first component, $x = 0$ or $x = 1$, designates whether e_1 or e_2, respectively, gets indexed with y. We denote such mapping by *linear*$_2$ since it performs a simple linear pass to exhaust one enumerator before indexing into another. Note that the domain of p is then naturally $dom(p) = |S_1| + |S_2|$.

For $e = e_1 \otimes e_2$ the mapping p is defined by:

$$p_d(i) = (i \bmod |S_1|, \lfloor i/|S_1| \rfloor) \qquad p_c(x,y) = y \cdot |S_1| + x$$

We denote this mapping by *div-mod*$_2$, since it corresponds to combining all elements from one of the two enumerators (here, e_1 specifically) with each element of the other enumerator, to exhaust all possible pairs of elements (much like the "round-robin" ordering). Here, $dom(p) = |S_1| \cdot |S_2|$.

Linear$_n$ and *div-mod*$_n$ mappings can be extended to greater cardinalities of components, i.e. $n > 2$. For a list of enumerators $\overline{e} = \langle (S_1,p_1),(S_2,p_2),...,(S_n,p_n) \rangle$, we can use an iterative process to calculate the list of components $p_d(i) = f_n(i)$:

$$f_k(i) = \langle g(i), f_{k-1}(h(i)) \rangle, \text{ if } k > 0$$

where we assume f_0 ends the process and $g(i)$, $h(i)$ are:

- $\{(1,i)$ if $i < |S_k|$ and $(0,0)$ otherwise$\}$, i-$|S_k|$ for linear$_n$, where the j-th component $(1,\ x)$ designates that j-th enumerator gets indexed with x
- $i \bmod |S_k|$, $\lfloor i/|S_k| \rfloor$ for div-mod$_n$

These generalized decompose functions have inverse compose functions which are computable with a reversed iterative process. It is straightforward to see that these functions achieve a bijective mapping, and how they straightforwardly map to an implementation. SciFe instantiates these higher-cardinality mappings when rewriting enumerator expressions that reduce to multiple applications of the same combinator; see § 7.2.

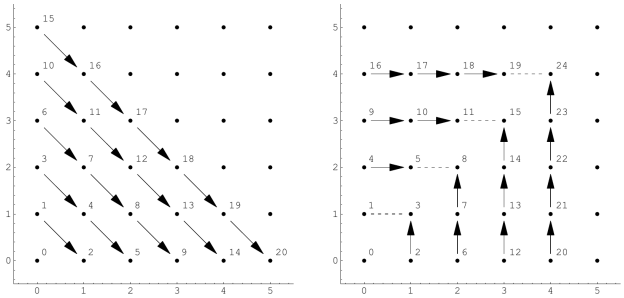

Figure 6: Cantor and Szudzik pairing functions

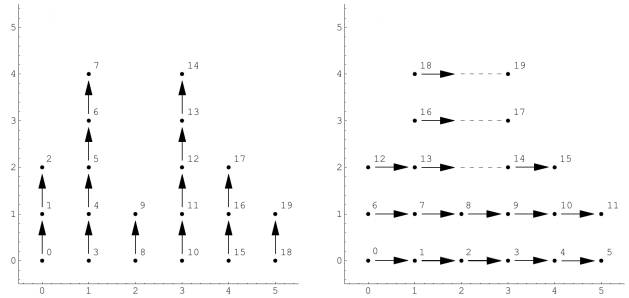

Figure 7: Linear and disperse pairing on a sample enumeration space

Enumeration from infinite domains. Pairing functions used in the case of infinite enumerators encode all natural numbers with pairs of natural numbers [39, 46].

Cantor's mapping is a classical example of a primitive recursive bijection that counts infinite sets. It lends itself perfectly for composing infinite enumerators:

$$p_d(i) = (i - \frac{w^2+w}{2}, \frac{w^2+3w}{2} - i), w = \left\lfloor \frac{\sqrt{8i+1}-1}{2} \right\rfloor$$

$$p_c(x,y) = \frac{1}{2}(x+y)(x+y+1)+x$$

For a pair of any two natural numbers x and y, this pairing function associates a unique natural number i with that pair. It assigns consecutive numbers to points along diagonals in the enumeration space (quadrant I of the plane, in Figure 6, left).

Cantor's mapping can generalize decomposition into tuples of greater arity ($n > 2$), but the decomposition becomes inefficient to compute [47]. On the other hand, *Szudzik's mapping* is a bijective function amenable for efficient generalization [46]:

$$p_d(i) = \begin{cases} (w, \left\lfloor \sqrt{i} \right\rfloor) & \text{if } w < \left\lfloor \sqrt{i} \right\rfloor \\ (\left\lfloor \sqrt{i} \right\rfloor, w - \left\lfloor \sqrt{i} \right\rfloor) & \text{otherwise} \end{cases}, w = i - \left\lfloor \sqrt{i} \right\rfloor^2$$

$$p_c(x,y) = \begin{cases} y^2 + x & \text{if } y > x \\ x^2 + x + y & \text{otherwise} \end{cases}$$

This pairing function assigns consecutive numbers to points along the edges of increasingly large squares (Figure 6, right).

Pairing for infinite domains in SciFe. For an enumerator composed of infinite enumerators $e_1,...,e_n$, SciFe uses:

- div-mod$_n$ pairing (defined previously) for merge ($\oplus$): for index i, $i \bmod n$ chooses the enumerator to index with i/n
- Szudzik's pairing for product ($\otimes$)

Then, div-mod$_n$ enumerates from each of the underlying enumerators at the same pace, while Szudzik's pairing enumerates unique combinations of elements from underlying enumerators. SciFe rewrites multiple chained applications of $\oplus$ and $\otimes$, so that the mapping uses these generalized pairing functions; see § 7.2.

Pairing for the bind combinator. To achieve exhaustive enumeration of corresponding inner enumerators of $d = (E,t)$ in $e \oslash d$, where $e = (S,p)$, the mapping uses $p_d(i) = (x,y)$ to enumerate $\pi_2(t(p(x)))(y)$, where $\pi_2(E,t) = t$. The pairing function for bind reduces to enumerating from a list of $k = |S \cap D|$ inner enumerators (§ 4.3). SciFe, for bind, uses:

- linear$_k$, if all inner enumerators of d are finite, while e may be infinite, and disperse (defined below), only if e is finite, for parallel enumeration (Figure 7)
- div-mod$_k$, if all inner enumerators of d are infinite, but e is finite (enumerating inner enumerators with "round robin")
- Cantor's mapping (of cardinality $n = 2$), if all inner enumerators of d are infinite and e is infinite (chosen over Szudzik's mapping, for performance reasons)

Disperse pairing. We introduce a new pairing function *disperse* that assigns numbers to points from multiple finite groups ("columns" in Figure 7), similarly to linear$_n$ (Figure 7, left), but in an order that disperses a sequence of assigned points over different groups (Figure 7, right). The idea is to assign points from all groups fairly, with respect to the potential variety in their sizes. The decompose function of disperse is given in Algorithm 1.

Algorithm 1 disperse pairing algorithm

Require: S, i; $\forall(x,s) \in$ S.$s > 0$, $0 \leq$ i $<$ sum($\{s|(x,s) \in$ S$\}$)

1: $\langle (s_j, X_j) \rangle_j$ = group $(x,s) \in S$ by s, sort ascending by s, where $X_j = [x_{j1},...,x_{jc_j}]$, $x_{j1} < ... < x_{jc_j}$, $(x_{ji}, s_j) \in S$
2: j= 1, rest= $|S|$, y_0= 0, lo= hi= 0
3: **loop** ▷ iterate through partitions
4: lo= hi; hi += rest · (s_j - y_0) ▷ set partition bounds
5: **if** i < hi **then** ▷ if in the right partition
6: i_w = i − lo ▷ index within partition
7: x' = $X_j[i_w \bmod$ rest$]$; y' = $y_0 + \lfloor i_w /$rest$\rfloor$
8: **return** (x', y') ▷ div-mod$_{\text{rest}}$ pairing within partition
9: **else**
10: rest -= c_j, y_0 += s_j ▷ note $c_j = |X_j|$
11: j += 1 ▷ move to next partition

The input is a list of integer pairs S and an index value i. In the enumeration space, the value s in $(x,s) \in S$ determines the height of the x-th column; i.e. points (z,y) where $z = x$ (Figure 7). The algorithm groups pairs with the same height in the list $\langle (s_j, X_j) \rangle$, sorted in the ascending order, where X_j represents all columns of height s_j. Groups determine "partitions" that form successive rectangles in the enumeration space. The algorithm assigns points partition-by-partition. It maintains the

bounds for the current partition j in *lo* and *hi*. The index i belongs to the current partition if $lo \le i < hi$. If i belongs to the j-th partition (s_j, X_j), then the partition has $hi - lo = \text{rest} \cdot (s_j - y_0)$ points. Thus, $s_j - y_0$ points can be assigned from each of the remaining rest columns; effectively forming a rectangle. (Initially, for the first partition, s_1 points are assigned from each column in S). Within the found partition, points are assigned with div-mod$_{\text{rest}}$. The algorithm keeps track of discarded partitions (i.e. exhausted columns) and remaining points with y_0 and rest to correctly compute offsets for the resulting coordinates (x', y').

SciFe uses disperse for parallel enumeration of bind applied to finite enumerators (where columns become inner enumerators, possibly of various sizes). It can be applied to other combinators as well, e.g. merge. Disperse assigns sequential indexes, and thus workers, to distinct (inner) enumerators to start with and proceeds in "lock-step", rather than exhausting enumerators one-by-one (which lowers contention overheads; see § 8.4).

6. Features of Enumerators

The enumerator algebra supports extensions of the behavior of the existing operations and defining new ones that go beyond generation of structures. A key property of the algebra is that it allows introducing new behaviors of enumerators as extensions, while preserving existing behaviors and their compositions.

6.1 Lazy enumeration

Lazy evaluation, which delays computation until its results are needed, is a commonly used technique for avoiding unnecessary computation and many BET approaches use some forms of it to prune the search space [4, 18, 42]. *Lazy enumeration* in SciFe allows controlling the process of enumeration according to the usage of enumerated values—it allows enumeration to skip certain elements, while preserving the effect of exhaustive enumeration. SciFe allows enabling lazy enumeration selectively for existing enumerator definitions, as shown in § 2.

Enumerators can utilize the feedback information from the operation under test. Instead of just enumerating structures and passing them to the code that uses them, the enumerator gets bound to a field of the enumerated structure for which it enumerates values for. Effectively, an enumerator tracks accesses to fields of enumerated data structures and the trace is used to determine how many structures to skip in the enumeration. (Implementation of the tracking is described in § 7.3). Enumerators for lazy enumeration are designated with:

```
trait LazyEnum[T] extends Enum[T] {
  def skip(ind: Int): Int              // computes an index value
  def reset; def isTouched: Boolean }
```

An enumerator is touched when its bound field in the structure is accessed. Method skip is called before an element is enumerated and it computes a new index in the enumeration that might skip structures, according to the trace. It takes an index value i and computes the first next index i', such that $e(i')$ enumerates a structure with new values only for the previously accessed fields (which may exhibit new behavior of the underlying code), under the assumption accesses were tracked for the structure at $e(i)$. Skipping applies to combinators that combine elements of multiple enumerators, where some combinations might be skipped.

Algorithm 2 Lazy exhaustive enumeration

Require: e:`LazyEnum[T]`, e=(S,p), i$\in\{0..|S|-1\}$, C=$\emptyset$

```
1:  procedure SKIP(e, i) ▷ returns next index for tracked e(i)
2:    switch (e,p) do                          ▷ case analysis on the enumerator
3:      case (f ⊘ d)                                          ▷ d is dependent
4:        find j, k, t s.t. d(t)(k) = e(i) and t = f(j)
5:        if d(t) is touched then return [d(t)] + SKIP(d(t), k)
6:          if j = f.size-1 then return |S|                  ▷ skip to end
7:          else return [d(f(j+1))]                ▷ skip to next inner enum.
8:      case (e1 ⊗ e2,(pd,pc))                          ▷ e1 and e2 are finite
9:        if e1,e2 are touched then return i+1
10:       else if e1,e2 are not touched then return |S|
11:       else if e2 is touched then (i/|S1|+1)|S1|
12:       else C = C ∪ {i mod |S1|}                    ▷ skip current column
13:         for j = i; y ∈ C, where (x,y) ← pd(j); j=j+1 do
14:           return j                          ▷ next non-skippable element
15:     case e@f, @ is ↑ or ≻
16:       return SKIP(e)                  ▷ skip in underlying enumerator
17:     case (S,p)                                          ▷ atomic enumerator
18:       return i+1                                        ▷ nothing to skip
19: procedure RESET(e)                          ▷ shows only for ⊘ and ⊗
20:   e.touched ← false                                     ▷ reset the flag
21:   switch e do
22:     case (e1 ⊘ d,p)                                   ▷ recursively call reset
23:       for all i ∈ {0..|S1|−1}, e' = d(i). RESET(e')
24:     case (e1 ⊗ e2,p)                    ▷ sets C=∅ in e2 if needed
25:       RESET(e1); RESET(e2)
```

Lazy enumeration is described in Algorithm 2. The algorithm traverses and matches on the enumerator tree expression (which has combinator applications as nodes) and performs a case analysis at its nodes. With $[e]$ we denote the index value that enumerates the first element of the inner enumerator e. For first-order enumerators, the algorithm just increments i (since they have to be exhaustively enumerated). In the case of bind, the algorithm finds the inner enumerator d(t) that enumerates e(i), by maintaining pointers, and values of d(t) and [d(t)], from the last call to skip (for efficiency). If d(t) is not touched, the algorithm returns an index that starts at the next inner enumerator (thus skipping the rest of elements in d(t)), or $|S|$ to mark the end of enumeration, if d(t) was the last inner enumerator in d. Otherwise, the next index is in d(t), which is queried recursively to skip elements within d(t). In the case of product, the algorithm relies on finite pairing to determine which elements to skip in the enumeration space. It currently considers only div-mod$_2$ pairing: if both e_1 and e_2 are touched, no element can be skipped; if both are untouched, all elements in e can be skipped; in the interesting case, when only e_2 is touched, the algorithm skips the rest of the current "row" of elements (jumps to the next row); if only e_1 is touched, the algorithm tracks which "columns" can be skipped in the variable

C and loops to find the first next element that cannot be skipped. The RESET procedure traverses the enumerator expression and resets the field access indicator. In addition, it resets C to $\emptyset$ in e_2, only if e_1 enumerated a new element (needed to avoid invalid C).

Skipping works only for combinator mappings for finite enumerators (such as div-mod$_2$ for $\otimes$) to be able to compute the number of elements to skip. We omitted merge ($\oplus$), which can be used by propagating skipping to the right underlying enumerator.

6.2 Inversion of enumeration

Efficient bijective mapping allows not just efficient indexing function but also efficient *inverse indexing*. Inverse indexing represents an inverse of the indexing function: for a given value it computes an index with which the value is enumerated. Inverse indexing can be achieved with the compose function of the mapping, as long as it is bijective and the enumerator does not contain applications of $\succ$ (otherwise, returning a single valid inverse index might not be efficiently computable).

Inverse indexing allows querying an enumerator $e=(S,p)$, with a given element x for its index, so that if the element is encoded by e, i.e. $x \in S$, the resulting value i satisfies $p(i)=x$. SciFe declares inverse enumerators with the following trait:

```
trait Reversible[T] extends Enum[T] { def inverse(x: T): Int }
```

More specifically, given a reversible enumerator $e_r=(S,p)$ and an element x, e_r.inverse(x) == i only if $p(i)=x$.

Inverse enumeration algorithm. For a composite enumerator, inverse indexing needs to: 1) transform the given element into components and their indexes in the underlying enumerators; 2) use the compose function with the indexes from the underlying enumerators to compute the index in the composite enumerator. Being bijective is itself not enough for inverse indexing; the algorithm therefore imposes additional constraints on enumerators and their combinators. For every atomic enumerator, the algorithm needs to find the right index value for the given element, and for every application of the map combinator, the algorithm needs to know how to invert the resulting output into the right input. Note that SciFe has the information to invert each atomic enumerator automatically, according to how it is constructed, the developers need only to provide an inverse function f^{-1} for each map combinator when the enumerator is constructed; see § 7.1. Since the implementation does not enforce bijectivity, as discussed in § 4.3, it only guarantees that inversion returns one valid result.

Inverse indexing, given in Algorithm 3, takes an enumerator with a bijective mapping and an element t and assumes existence of i such that e(i)==t. (In the implementation, if such i does not exist, some enumerator fails the inverse and throws an exception.) Note that if the inversion is invoked on a bind, the given element is a pair t=(t_x,t_y): the value t_x is initially provided as a dependent parameter when invoking inversion (as shown in § 2, developers provide 15, 1 to 15 in bst(15, 1 to 15).member(...)). Similarly to SKIP (Algorithm 2), INV traverses the enumerator expression and performs a case analysis. For each considered case, it analyzes the current subexpression, recursively computes values of inverted components and uses the appropriate compose function (as defined in § 5) to compute the result. The matched enumerator can either be a first-order atomic enumerator or a combinator application (§ 4.2). With (p_d,p_c) we denote matching on the decompose and compose function of the enumerator mapping. For the map combinator, the algorithm uses the externally provided inverse function f^{-1}, which computes a value that is then recursively inverted. For bind and other combinators (matched with @), the algorithm recursively inverts values in the pair (t_x,t_y) and composes them into the result. In the case of bind, t_y is enumerated from the inner enumerator d(t_x). As mentioned, each atomic enumerator is capable of performing the inverse operation directly.

Algorithm 3 Inverse enumeration

Require: e: `Reversible[T]`, element `t:T`, or `t:(I, T)` if e=$e' \oslash d$, and d:`Depend[I,T]`, such that $\exists i$. e(i) = t

```
1:  procedure INV(e, t)                          ▷ e.inverse(t), in code
2:    switch (e,p) do                            ▷ case analysis on the enumerator
3:      case (e' ↑ f,(p_d,p_c))                  ▷ map combinator
4:        return INV(e',f^-1(t))                 ▷ f^-1 is provided
5:      case (e' ⊘ d,(p_d,p_c))                  ▷ d is dependent
6:        (t_x,t_y) ← t                          ▷ t is a tuple, where t_y is in d(t_x)
7:        x = INV(e',t_x); y = INV(d(t_x), t_y)
8:        return p_c(x,y)                        ▷ apply compose function
9:      case (e_1@e_2,(p_d,p_c))                 ▷ binary combinator (⊕ or ⊗)
10:       (t_x,t_y) ← t
11:       x = INV(e_1,t_x); y = INV(e_2,t_y)
12:       return p_c(x,y)
13:     case (S,p)                               ▷ atomic enumerator
14:       assert(t ∈ S)                          ▷ value must be encoded
15:       return i, where p(i) = t               ▷ knows i by construction
```

6.3 Membership checking

Membership checking allows querying an enumerator $e=(S,p)$, with an arbitrary element t, to ask whether t is encoded by e, i.e. $x \in S$. It returns true only if $\exists i.e(i)=t$. This operation uses an algorithm similar to Algorithm 3 but merely checks whether the given value can be recognized (it does not compute the index). Membership, which is demonstrated in § 2, is defined in:

```
trait Member[T] extends Enum[T] {
  def member(x: T): Boolean }      // if x can be enumerated
```

For practical purposes it usually suffices to have an efficient structure recognition in order to check whether a structure satisfies properties of elements encoded by an enumerator (such as the search tree invariant in § 2). This operation sacrifices the information of the precise index with which the element is enumerated to achieve better performance—namely, the algorithm does not incur the overhead of the decompose computation (which may be significant for pairings such as linear$_n$).

6.4 Fast-Forwarding

In some scenarios, instead of enumerating in the ascending order of indexes, starting from the first element, it might be

useful to *fast-forward* an enumerator to a given element. The given element, if encoded by the enumerator, then becomes the first element in the current enumeration and the enumeration can proceed by enumerating all subsequent elements, without considering all elements that came before. For enumerators that enumerate values in the order of their increasing complexity (as would e.g. bst from § 2, if chained with an enumerator of increasing sizes and ranges), this procedure effectively allows "jumping" to an arbitrarily complex element and proceeding from there.

Fast-forwarding an enumerator $e = (S,p)$, given an element $t \in S$ such that $p(k) = t$, returns a new enumerator $e_f = (S_f,p_f)$, where $S_f \subseteq S$ and $\forall i \in [0,...,l].\exists j \in [k,...,|S|].p_f(i) = p(j)$ (where $l = |S| - k - 1$ if e is finite, otherwise $l = \infty$). Then e_f starts sequentially enumerating from t, thus effectively "fast-forwarding" k elements. Note that the order of enumeration from e_f might differ from the order defined by e, to allow flexibility in supporting different mappings. As when comparing membership and inversion, sequential enumeration after fast-forwarding is much more efficient than calculating the index of the element (with inverse) and then enumerating with indexes, in the general case (due to the fact that certain compose functions require more complex computations, such as linear_n).

Fast-forwarding constructs a sequential enumerator that does not support efficient indexing in the general case:

```
trait Forwardable[A] { def fforward(a: A): SeqEnum[A] }
```

The resulting enumerator supports efficient sequential enumeration, by enumerating next or previous element from the current, while the indexing function may still be used, albeit potentially being much less efficient.

Fast-forward uses a similar algorithm to INVERSE (Algorithm 3). It recursively fast-forwards enumerators to the appropriate position k, according to the given element, and transforms them into SeqEnum. The algorithm relies on mappings that can sequentially advance enumerators to enumerate $S_f \subseteq S$ without unnecessarily enumerating the prefix (of encoded elements up to k). Note that efficiently invertible mappings allow such transformations to be trivial: sequential traversal just increments the inverted index. Additionally, sequential enumeration is not restricted to the forward direction. With appropriate mappings, the resulting enumerator can enumerate elements of the prefix of elements $S \setminus S_f$, by sequentially enumerating backwards.

7. Implementation

SciFe provides a DSL for writing short and concise specifications of enumerators by implementing the enumerator algebra and language presented in § 4. It follows the algebra rules, ensures its semantics, and implements extensions to support the described features. In addition, SciFe employs techniques for improving performance of enumerators, through enumerator specialization and optimizations. SciFe is implemented in Scala and relies on its object-oriented and functional features such as class and trait hierarchies, partial functions, Scala collections, generics, implicit types and inference, to achieve concise and type-safe enumerator specifications [37]. We only present a short overview of the implemented facilities in the framework. For more information on the implementation details, the reader can visit the official website of the tool and its open source code repository [1].

7.1 DSL interface

SciFe is implemented as a *library* that provides the enumerator (embedded) DSL. It is built solely from existing language constructs and does not require any compilation passes. The developers need merely to import SciFe's appropriate DSL definitions to construct enumerators and use their operations. Two main components of the enumerator DSL are: *combinator methods* in basic enumerator traits Enum and Depend (which can be used with an infix notation), and *implicit enumerator conversions* that create, transform, and optimize enumerators according to the specific context. Contexts are designated by importing specialized DSL constructs (e.g. enum.lzy for lazy enumeration).

Atomic enumerators. Atomic enumerators are basic building blocks in the framework and are indivisible. Other enumerators are composed with combinators. SciFe builds atomic enumerators with wrappers around standard constructs in Scala, such as collections and functions. The underlying construct determines indexing and inversion functions. (E.g. for indexed collections, SciFe uses the collection directly for indexing, but also computes a reverse mapping from elements to indexes for inversion.)

Recursive enumerators. For cases when an enumerator needs to refer to itself in its definition, the DSL provides a factory method rec (which is a convenience construct, due to the eager evaluation in Scala [37]). It takes a high-order function that constructs a dependent enumerator and takes a parameter that is bound to the enumerator being constructed. This allows recursive enumerator definitions by using the enumerator that is being constructed within its definition (as used in § 2).

Combinators. Combinators are themselves instances of enumerators. They are implemented as infix operators and have concise unicode identifiers, as presented before. Each combinator application invokes a factory method that constructs a composite enumerator with the appropriate mapping and features, based on the arguments and context. (E.g. for inversion, $\uparrow$ takes two functions, one used for inverting enumerated elements.)

Bijectivity. As a consequence of directly using host language constructs (and allowing more flexible and efficient implementation), SciFe does not prevent enumerator definitions that break bijectivity: $\uparrow$ and $\downarrow$ can map arbitrary (and possibly non-injective) functions, while enumerators can be constructed from arbitrary collections and functions (that might enumerate equal elements). (E.g. atomic enumerator constructed from Scala arrays, provides efficient indexing but is not checked for duplicates).

Implicit conversions. SciFe's DSL provides a wide range of implicit conversions between Scala constructs and enumerators. This allows their interchangeable use, e.g. needed for use with for-comprehensions, and more concise enumerator expressions. Note that implicit conversions might cause expensive transformations (to collections that store their elements, e.g. List[T] for

finite enumerators by default) and thus negate efficient operations otherwise provided by combinators (as mentioned in § 3).

7.2 Optimizations

Enumerators can use various optimizations that are enabled explicitly (e.g. memoization) or implicitly (e.g. transforming enumerators during enumeration). We will describe a few of the categories of optimization techniques used in SciFe.

Memoization. SciFe allows enabling memoization for both first-order and dependent enumerators, which stores elements enumerated for a given indexes and first-order enumerators for given dependent parameters, respectively. It provides several traits that use standard Scala collections (e.g. hash maps) and can be mixed into enumerators to enable memoization, while allowing other memoization methods to be easily added. For example, memoization traits for parallel enumeration handle concurrent accesses (with or without blocking) and try to minimize lock contention (e.g. with disperse mapping, once the memoization store is constructed, it is changed to a lock-free structure).

Specialized enumerator types. SciFe encodes different enumerator classes (e.g. according to finiteness and supported features) with separate traits. This distinction is made on the level of first-order enumerators, as well as dependent ones. Enumerators use generic types to designate the type of enumerated elements. This allows choosing efficient implementations, specialized for the given given enumerator classes and the context, during enumerator construction and dynamic optimizations at runtime (as e.g. the right mapping for $\oslash$, depending on the arguments).

Rewriting. SciFe defines rewrites as factory methods that construct specialized enumerators based on the specific types of underlying enumerators and combinators. Rewrite rules include generalizing pairing functions to multiple underlying enumerators (as defined in § 5), optimizing (e.g. binary search implements linear_n for larger n), preventing construction of empty composite enumerators, and evaluating sub-expressions eagerly (e.g. product applied to a singleton enumerator or functions of multiple chained map combinators).

7.3 Implementing enumeration

By allowing transformation of enumerated values with arbitrary functions, together with efficient indexing SciFe offers great flexibility in encoding various classes of data structures and achieving different enumeration strategies.

Enumeration strategies with indexing. SciFe implements various enumeration strategies solely by choosing appropriate integer values to apply to the indexing function, including lazy, parallel and random enumeration (§ 2, § 6).

Note that the encoded element at a given index, i.e. the enumeration ordering, is completely determined by the mapping used by the given enumerator and applied combinators. This allows not only strategies that do not rely on the knowledge of the mapping, like straightforward enumeration of disjunctive subsets of encoded elements (with disjunctive integer values), but also ones that do. In the latter case, changing the enumerator's mapping, by changing a combinator variant due to the context or optimizing, might thus break the existing strategy. SciFe enables usage of abstractions that hide the underlying use of specific index values and operations, such as for-comprehensions (for exhaustive and lazy enumeration, which incorporates calls to skip) and operations like member.

Lazy enumeration. The proxy interface for data structures that allow tracking field accessed (used in § 6.1) is defined as:

```
trait StructureField[F, D] {
  def setTrack(e: Touchable[D])
  def getValue: F } // invokes enumeration
```

Accesses to the fields of such structures invoke indexing in the enumerator that enumerates the field values. By using this proxy, the existing data structure interface remains the same and the enumerator gets embedded into the field evaluation. SciFe implements such data structures with Scala lazy values where field evaluation represents a function closure that invokes indexing in the appropriate enumerator (which may incur great overheads, both in time and memory) [37].

Graph enumeration. SciFe enumerates graph-like structures by implementing techniques for inductive algebraic representation of graphs and its conversion to other (possibly mutable) representations. The basic representation of graphs is the following inductive definition (which based on the one defined in [16]):

```
type Node = Int; type Adj[Lab] = List[(Lab, Node)]
type Context[T, Lab] = (Adj[Lab], Node, T, Adj[Lab])
case object Empty extends Graph                  // inductive graph
case class &:(ctx: Context, g: Graph) extends Graph
```

This definition is then instantiated with particular types for T and Lab that represent values associated with nodes and edges, respectively. Bidirectional edges are represented by encoding incoming and outgoing edges at each node. The definition allows SciFe to encode and reuse subgraphs simply by modifying integer values at each node with the map combinator. Moreover, this allows conversion to mutable graph definitions that only expose methods for addition of nodes to avoid more costly conversions of complete enumerated graphs.

8. Experimental Evaluation

We evaluated SciFe's exhaustive enumeration, features and optimizations with the goal of assessing its *performance*, *scalability*, and *usability* in practical (testing) scenarios.

Besides exhaustive enumeration (for test input generation), we evaluated SciFe in the context of different enumeration strategies (useful for parallel and random testing), as well as structure recognition (for bounded verification). Our performance evaluation aims to *exhaustively cover related work*, either by measuring related techniques directly or measuring techniques that reportedly subsume them. We developed a suite of benchmarks for SciFe and extended the existing BET benchmarks with new scenarios. In the interest of space, we omitted some results from this presentation. All benchmarks and full evaluation are publicly available [1].

Size	Binary search trees			Red-black trees								Heap arrays			B-trees		RIFF Images	
	SciFe/e	CLP	Korat	SciFe	CLP	Korat	HyTek	Udita	InSynth	sChk	CLP/o	SciFe	CLP	Korat	SciFe	CLP	SciFe	CLP
1	0/0	0	0.12	0	0	0.14	0	<1	0.01	0.15	0	0	0	0.1	0/0	0/0	0	0
2	0/0	0	0.12	0	0	0.15	0	<1	0.01	0.16	0	0	0	0.1	0/0	0/0	0	0
3	0/0	0	0.12	0	0	0.15	0.37	<1	0.02	0.18	0	0	0	0.1	0/0	0/0	0	0
4	0/0	0	0.14	0.01	0	0.19	0.38	<1	0.33	0.2	0	0	0	0.11	0/0	0/0.02	0.01	0
5	0/0	0	0.18	0.02	0	0.23	0.42	<1	2.17	0.17	0	0	0	0.16	0/0.01	0/0.13	0.01	0
6	0/0	0	0.23	0.03	0	0.28	0.52	1	t/o	32.9	0	0	0	0.34	0.01/0.01	0/0.76	0.03	0.02
7	0/0	0	0.42	0.07	0	0.38	0.83	2	t/o	t/o	0	0.01	0.01	0.45	0.01/0.04	0/2.49	0.06	0.08
8	0/0	0.01	1.6	0.11	0.01	0.62	1.86	6	t/o	t/o	0	0.06	0.08	1.31	0.01/0.06	0.01/7.59	0.1	0.53
9	0/0	0.03	11.8	0.17	0.04	1.8	5.61	28	t/o	t/o	0	0.52	0.81	12.19	0.01/0.12	0.04/20.79	0.15	4.43
10	0.01/0	0.12	94.91	0.25	0.14	8	14.46	130	t/o	t/o	0	5.86	8.59	140.53	0.02/0.22	0.19/55.5	0.24	24.84
11	0.03/0.01	0.47	t/o	0.35	0.51	42.62	23.73	t/o	t/o	t/o	0.01	28.21	110.25	t/o	0.02/0.43	1.12/147.21	0.88	187.57
12	0.06/0.03	1.84	t/o	0.48	1.94	t/o	37.41	t/o	t/o	t/o	0.02	-	-	-	0.02/0.7	2.79/ t/o	1.17	t/o
13	0.21/0.11	7.31	t/o	0.65	7.26	t/o	104.46	t/o	t/o	t/o	0.05	-	-	-	0.04/1.58	8/ t/o	4.88	t/o
14	0.82/0.44	29.42	t/o	0.85	27.42	t/o	t/o	t/o	t/o	t/o	0.1	-	-	-	0.04/3.43	20.93/ t/o	6.21	t/o
15	2.57/1.44	109.11	t/o	1.46	t/o	t/o	t/o	t/o	t/o	t/o	0.22	-	-	-	0.05/8.26	56.17/ t/o	39.77	t/o
#	9694845 (9.7M)		16k	10.4k	5.4k	586	586	1.2k	14	20	10.4k	111M	111M	10.3M	659k/9.3M	659k/1.73M	127.6M	331k
loc	29/46	20	100	25	32	209	>2M	250	34	24	34	27	19	53	91	85	36	80

Table 3: Performance comparison for exhaustive generation of data structures. Execution times are given in seconds (timeouts with t/o). Last two rows denote the largest number of generated structures and specification length, respectively. Unfeasible experiments are marked with -.

Methodology. The evaluation used per-tool specific benchmarks whenever provided (alongside their respective timers). SciFe benchmarks use a framework that spawns and warms new virtual machines before each measurement. The experiment configuration consisted of: CPU with 3.5Ghz clock, 15MB (L3) cache; 1600Mhz DDR3 memory; 64b linux, JVM 1.7, and Scala 2.11; with imposed limits: 24GB for memory and 200s for running time (for more details see [1]). For parallel benchmarks, to enable utilizing more CPU cores, we used a configuration with a 10-core, 2.6Ghz clock, 25MB (shared L3) cache CPU, and other parameters same as above. All results represent the mean of at least 3 measurements with standard error within 10% of the mean.

8.1 Exhaustive Generation of Data Structures

The goal of this evaluation was to cover a broad set of existing techniques, as well as generated data structures, with different structural properties and use cases. Here, we report results only for representative data structures; full benchmark report is available at [1]. Our benchmarks cover data structures from different categories: prototypical trees, sorted (from § 2) and red-black trees; heap arrays (to examine upper bounds on enumerating very large sets of structures); B-Trees (which do not have a priori fixed structure); and image representations (based on the RIFF format which stores "data chunks" in a complex recursive structure).

Table 3 reports times for exhaustive generation of data structures with, in order: SciFe, the constraint logic programming (CLP) approach [43], Korat [4], HyTek [40] (which subsumes TestEra [28] and Alloy [25], that are not included), Udita [18], InSynth [22], and SmallCheck [42]. The table tabulates the running time needed for generating all valid data structures, per particular size. For ordered data structures, given ranges for the structure keys were determined by the size. Row # reports the number of generated structures for the largest size that succeeded within the timeout (where large values are rounded down). For brevity, we did not evaluate all tools on all benchmarks; instead we give an overview of the state-of-the-art tools on the prototypical example and focus on comparison with the fastest respective techniques on others. Performance of SciFe is reported in the *SciFe* column; in all measurements we enabled memoization (up to the given memory limit). CLP does not support arbitrary data structures, thus it generates only their encodings—to that end, to examine potential impact where only encodings are generated, *SciFe/enc* represents encoding trees with lists, similarly as in CLP. *CLP/opt* stands for highly (manually) optimized version of red black tree specification which carefully avoids backtracking (given in [43]). InSynth and SmallCheck only support generation within a *search depth* bound and may generate bigger structures before exhausting structures of the given size; thus, the table reports shortest executions that exhaustively generated a given size. Note that, unlike Korat, HyTek and CLP, which generate structures of exactly the given size, SciFe implicitly generates (and memoizes) all structures *up to the size*; thus subsequent enumerations should perform much faster. (These were not measured.)

In *all experiments*, SciFe *outperformed all evaluated tools*, up to by a few orders of magnitude. SciFe scaled better *both in terms of structure size* and their number; up to more than 127 million enumerated structures in less than a minute. Generators in SciFe are much shorter than those for imperative-backtracking tools (like Korat, UDITA) while comparable and/or shorter than property-based (SmallCheck) and logical (CLP) specifications.

InSynth uses fast type-directed generation but only generate-and-test for complex structures, thus serving as a naive implementation "baseline". Together with SmallCheck, which leverages lazy evaluation, they showed limits in scalability of generation with coarse grain depth bounds. While Korat and UDITA use similar backtracking mechanism, UDITA's nondeterminism based on JPF scaled worse. Although CLP's solver engine brings it on top of the other tools, it clearly suffers from extensive backtracking, especially in the Image benchmark (where,

unlike data structures with fixed structure, the invariant constraints both metadata and sizes of substructures [1]). Aggressive optimizations in the specification (and smaller constant overhead compared to JVM) made CLP faster than SciFe for sizes up to 15 (column *CLP/o* in the table). Interestingly, for sizes 16 to 18 (not shown in the table), running times for SciFe and *CLP/o* were: 1.96, 1.81; 2.49, 3.93; 3.22, 7.78 (respectively, in s). These results suggest that highly optimized specifications in CLP have advantage only for smaller sizes and do not exhibit better scalability in general. Moreover, *SciFe/e* demonstrates an interesting property of *enumerating structures as encodings*: a further performance gain may be obtained by splitting the invariant into parts, enumerate values individually, and then compose the parts.

Additionally, the results show that SciFe was *not bound by the processing power*. On most of the benchmarks the bottleneck was the working memory (due to memoization, which effectively trades space for runtime performance). This suggest that increasing the working memory might enable exhaustive enumerations of larger sizes. Note that in case of heap arrays of size above 10 (and sorted lists [1]), run times include garbage collection overheads[5]. In those benchmarks we selectively turned off memoization for bigger data structure sizes (by providing an alternative memoization trait to an existing enumerator). Heap arrays of size >11 were unfeasible to measure since the number of structures exceeds the range of 32b integers.

8.2 Generation of graphs

Table 4 tabulates the running time needed to generate valid graphs of given sizes. SciFe enumerated benchmarks for all shown sizes representable with 32b integers. (*SciFe/d* used a larger timeout and frequent garbage collection for size 6, and enumerated more than 10^9.) Directed and undirected graphs are denoted with */d* and */u*, respectively, and have only structural constraints. SciFe enumerated graphs using inductive representations, as described in § 7, while *Korat/u* used a simple encoding with boolean arrays instead of node pointers, to measure its performance without backtracking. For undirected graphs, SciFe scaled worse but *comparingly* to Korat that employed *no backtracking* (and just enumerated combinations of boolean values). Enumerating DAGs, shows that SciFe performs much better when one additional constraint for acyclicity is added (to digraphs). The last column represents Java class hierarchies, where t types can be either classes or interfaces and m methods may be overloaded and sealed. SciFe did not use memoization in this benchmark. The results suggest that, in the presence of complex constraints, the *dependent enumeration alone* can achieve favorable performance, by avoiding unnecessary search space.

8.3 Feature breakdown

One of the goals of our evaluation was to characterize the impact of different features of enumerators on generation and testing activities in general. We present only the case of sorted tree data structure given in § 2; the given specifications are reused in different contexts in the following benchmarks.

[5] 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	Simple graphs			DAGs		Bounds	Class Hierarchy	
	SciFe/u	Korat/u	SciFe/d	SciFe	Korat	(t, m)	SciFe	Korat
1	0	0.13	0	0	0.11	1,1	0	0.18
2	0	0.13	0	0	0.11	1,2	0	0.19
3	0	0.13	0	0	0.11	2,1	0.02	0.22
4	0	0.14	0.34	0	0.12	2,2	0.03	0.49
5	0	0.16	0.84	0	0.17	3,1	0.07	0.82
6	0.02	0.23	444.49	0.01	0.3	3,2	0.78	10.45
7	1.01	1.16	-	0.91	8.33	4,1	1.99	165.46
8	189.04	145.2	-	126.69	t/o	4,2	48.48	t/o
9	-	-	-	-	-	5,1	86.35	t/o
loc	268.4M		1.07B	268.4M	2.09M		900.3k	21320

Table 4: Performance of generating graph structures. Times are given in seconds; notation is as in Table 3.

Strategies of enumeration. The first part of Table 5 summarizes results of different ways of using enumerator of binary search trees: *enum/pl* (plain) serves as the baseline, sequential, exhaustive enumeration without memoization; *enum/m* is same as the previous, with memoization turned on; *enum/en* enumerates encodings of trees as lists of values, with memoization (as in Table 3); *enum/rn* performs random, non-exhaustive, enumeration where indexing is done with randomly generated index, e.size times; *enum/mrn* is same as the previous, with memoization. Results show drastic improvements when memoization is used—the inductive tree structure entails many intermediate subtrees being shared and their recomputation is avoided. Therefore, *memoization is an important factor for both performance and scalability* to large structures, in addition to avoiding search space with dependent enumeration. (With higher time limit of 10m, plain enumeration succeeded only up to size 13.) Randomized indexing demonstrates similar performance as plain enumeration, as it is expected since both strategies enumerate similar structures same number of times. However, when memoization is turned on, random indexing incurs performance penalties resulting in being up to 1.75x slower than sequential enumeration; this suggests that randomly accessing structures might miss potential opportunities for substructure sharing and demonstrates that memoization can lead to *favorable caching patterns* (this effect is magnified in parallel enumeration, as shown below).

Membership checking. In the second part, Table 5 shows the performance of membership checking when used after insertions into search trees, as shown in § 2: *inv/ex* runs **ensuring** clause after insertion, while *inv/m* invokes the enumerator's member method. For size s, results show run time averages for inserting all s elements into trees of size $s-1$ (to measure the impact of member feature alone, only correctness checking was measured). Results show that even for simple structures like sorted trees, speedups of efficient matching of memoized substructures are substantial even for smaller sizes and scale with the size of the structure, reaching more than 4x for size 15. Note that this technique depends *solely on structure matching* and its performance is not worse for more complex invariants that involve expensive

size	8	9	10	11	12	13	14	15
en/pl	2.6k	16.1k	97.6k	t/o	t/o	t/o	t/o	t/o
en/m	2.69	4.97	13.88	27.09	61.49	205.27	819.21	2574.7
en/en	1.12	2.07	3.25	9.41	30.23	109.42	436.11	1437.93
en/rn	2.8k	16.8k	101.9k	t/o	t/o	t/o	t/o	t/o
en/mrn	17.92	36.69	39.21	60.2	122.61	341.65	1196.68	4513.64
inv/ex	0.16	0.54	1.82	6.66	25.03	95.7	368.07	1416.54
inv/m	0.06	0.15	0.5	2.32	7.47	26.25	91.08	311.33
x	2.6	3.53	3.63	2.87	3.35	3.65	4.04	4.55
en/inv	3k	11.44k	43.76k	167.96k	646.65k	2.5M	9.66M	37.44M
en/lzy	1.33k	4.87k	18.05k	67.7k	255.92k	973.21k	3.72M	14.26M
%	55.67	57.47	58.75	59.69	60.42	61.01	61.5	61.9

Table 5: Different features in generation and testing for sorted trees. Times are given in seconds; notation is as in Table 3.

computations (it can only be improved). This opens opportunities for even better performance, e.g. with hash-consing [20].

Pruning search space with lazy enumeration. The third part of Table 5 shows how many data structures were avoided by utilizing lazy enumeration in a manner as shown in § 2: *en/inv* exhaustively enumerated s keys (in the range [0..s-1]) and performed the insertion on all enumerated trees of size $s-1$, while *en/lzy* used lazy enumeration in the same scenario. Both rows report the number of key-tree pairs that were enumerated, while % gives the percentage of all possible pairs that were *avoided* during lazy enumeration. Results show that nearly 62% of all possible key-tree pairs were discarded for inserting all of 15 keys into trees of size 14, which means that lazy enumeration executed only 38% of all insertions. This adheres to the fact that starting from avoiding only small fractions of the search space for small sizes (for size 3, 12 out of 15 trees are enumerated), the technique proceeds to *prune exponentially larger amounts* of the search space. Although the current implementation of the technique incurs significant overheads, great potential lies in testing scenarios where demanding computations might be avoided.

8.4 Parallel enumeration

The results of evaluating parallel enumeration are shown in Figure 8. We performed evaluation on two benchmarks: 1) binary search trees (with the definition shown in § 2), to examine performance benefits when enumerating simple structures (which scales well when executed on a single thread; see Table 3); and 2) RIFF images, enumerating structures based on the widely-used Resource Interchange File Format, to demonstrate scaling of parallel enumeration in the case of complex data structures, which can distribute more computation to individual worker threads. The graph plots running times for exhaustive enumeration of structures of size 15, for different number of worker threads (i.e., utilized cores). Run times of benchmarks are shown scaled to their maximum running times: 227.76s for the case of RIFF Image and 2.94s for the search tree benchmark.

The results confirm our hypothesis that enumeration, based on the indexing function, is *immediately amenable to parallelization*, modulo high deviation for smaller sizes (where the JVM executor initialization incurs relatively big overhead). Both benchmark scaled well: for 8 cores, average speedup was 4.03x (719ms/2.9s) in case of search trees and 9.75x (20.82s/203.07s) in case of RIFF images. Interestingly, results for RIFF enumeration suggest that due to higher complexity of enumerated structures, benefits of *constructive sharing* alleviate the amount of work when divided. Parallel enumeration, with disperse mapping (which tries to balance accesses over multiple inner enumerators), achieves positive effects on *cache locality* due to access patterns in which different threads access and use the same intermediate results as part of their working sets. (Note that the results differ from ones given in § 8.1 for `numThr==1`, since a different benchmarking configuration, slower per single core, was used.)

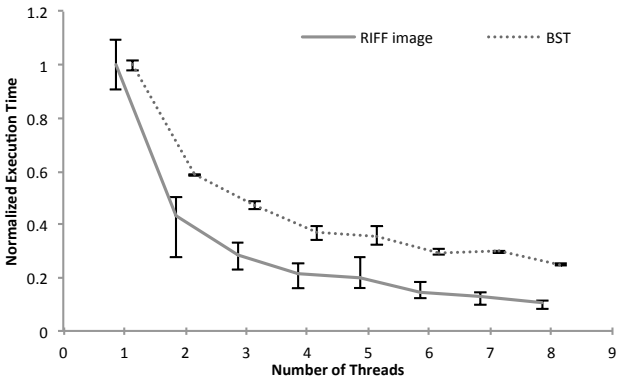

Figure 8: The chart plots scaled execution time of parallel enumeration for different number of threads (i.e. CPU cores).

9. Related Work

This work is partly inspired by (and improves upon) bounded-exhaustive enumeration applied for generation of well-typed terms in the domain of program synthesis [22, 30, 32, 49, 50]. A comparison and analysis of insufficiency of previously employed techniques can be found in [32]. We presented the main ideas for bounded-exhaustive enumeration and SciFe as a proof of concept in [33]. This work enriches and completes the combinator framework, generalizes the application of pairing functions for different enumeration strategies, and introduces features that go beyond exhaustive enumeration.

Enumerators in SciFe follow the common approach for designing a framework with combinators often adopted not just in testing tools [10], but also in tools for tasks such as term generation [17, 49, 50], and serialization [34]. Overviews of various techniques for generation of test inputs, particularly those for bounded exhaustive testing, can be found in [19, 44, 45, 48]. (We discussed and evaluated several such related techniques in § 1 and § 8). SciFe aims at unifying features that are supported in isolation in other tools for generation of test inputs, such as exhaustive generation [42], memoization [14], and lazy evaluation [18].

Constructive generators. SciFe is similar in spirit to other tools that follow the constructive approach for test input generation [7, 11, 14, 18, 42]. These tools aim at providing simple generators of values together with means for their composition, either through tool-specific interfaces [11], higher-order functional (and random access) combinators [14, 42], or

low-level non-deterministic primitives [7, 18]. However, these tools lack general facilities that avoid generate-and-test. SciFe is expressive for efficient constructive generation of complex structures, together with fine-grained control [11, 14], direct access [14], and laziness [42] of constructive approaches.

Declarative generators. We extensively related SciFe to many successful declarative approaches for BET [4, 13, 18, 40, 43]. Declarative techniques tend to be very efficient but highly sensitive to the relation between the generator specification, and the search process and its performance. Instead, SciFe provides expressive generation, without sacrificing conciseness and modularity of definition, (deterministic) control over the generation process, and efficiency (by avoiding unnecessary search). By solely relying on structure constructors, SciFe's enumerators do not require structure definitions, and support mutable and graph structures. For a class of structures, SciFe is capable of composing enumerators of values with orthogonal constraints, much like the high-level composition of constraints in declarative approaches. In contrast to performing search, enumerators can efficiently count the number of encoded structures without actually enumerating them.

Property-based and random testing. The idea of generation according to data types was popularized by QuickCheck [10] and has been replicated in a number of languages and systems [6, 12]. For complex structure generation, defining an adequate distribution and avoiding generate-and-test for such, may be challenging [24]. Some automated random testing tools use specific techniques, such as *feedback-directed testing* [38] or *adaptive random testing* [9], to control the distribution and increase coverage [8]. SciFe pursues exhaustive generation for expressive constraints, sidestepping problems of simple random generators, while enabling high-coverage random testing and parallelization thanks to efficient random access support.

Program inversion. The idea of inverting programs has been explored in the general context [2, 36, 41] and applied to specific format transformations [5, 26, 27]. *Relation-driven programming* [5, 27] focuses on bidirectional transformations between different structures given with textual representation. While SciFe can invert enumeration, unlike [26, 27, 36], it focuses on avoiding unnecessary search with expressiveness of dependent enumeration. Moreover, its combinators impose a structure that is leveraged to invert a given value deterministically (to a unique index), without backtracking (as in [36]). Ambiguity in SciFe is only allowed explicitly, rather than with ambiguous grammars [5, 27]: a dependent enumerator can be parametrized to encode, and thus recognize, different sets of structures.

Goal-directed programming. The idea of bringing non-determinism into programming languages has appeared in works that focus on goal-directed programming [21], functional and logic programming [7, 29] and recently, specifically testing [18]. These approaches introduced additional facilities into the existing programming model to allow operations analogous to enumeration in SciFe, such as querying for multiple results [18, 21] and fair enumeration of composite results [7, 29]. Instead of relying on non-determinism, SciFe lacks the generality of such approaches and offers specialized enumerators for exhaustive fine-grained controlled generation that avoids backtracking, for both simple and complex constraints.

Common knowledge in data structure generation. Our work explores the hypothesis that an approach to generating structures based on constructive definitions can be not only more efficient than approaches based on external solvers, but can also support an equally expressive and concise form of specification [13]. It suggests that search-based approaches might not always be adequate from a performance perspective, because their specifications frequently force unnecessary searching during generation. This work shows that an expressive constructive framework is achievable: it avoids the non-determinism and performance intricacies of search procedures, and allows compositional and modular definitions of robust generators that have more predictable and favorable performance.

10. Conclusions

We have presented a new framework for defining enumerations of values, which supports not only efficient sequential generation, but also random access to an element in the encoded sequence, testing whether an element belongs to the enumerable set, as well as lazy and parallel enumeration. We have demonstrated these features and overall advantage of the framework on an example development scenario. The framework is equipped with expressive combinators that enable enumerator composition and achieve efficient generation of complex structures that avoids unnecessary search space, and recomputation with memoization. We have shown that generation in our framework can be achieved with concise specifications, while exhibiting performance and scalability that exceed those in existing frameworks, as well as various features that might aid in practical testing scenarios.

The key insight of this work lies in the enumerator algebra that, with dependent enumeration, supports the idea of decomposing data structure invariants and propagating the appropriate constraints to corresponding enumerators to avoid enumerating superfluous values. Such an algebra can be expressed with a simple language that constructs enumerators in modular fashion and enables attractive properties and features that go beyond capabilities of existing approaches. Giving a precise semantics of the enumerator algebra and its language makes the approach amenable to various implementations, while making the algebra open for extensions, new features and optimizations. Due to a variety of supported features, the enumerator framework demonstrated superior performance and scalability in enumeration, with limitations governed by the amount of available memory. We believe the overall framework might have significant practical value since its modularity allows composable and reusable definitions that are easier to reason about and allow fine-tuning of the enumeration process. Although we showed that efficient generation (and recognition) can be concisely expressed in

a constructive manner, there remains the question about the relation, and potential (automatic) transformations, between enumerators and purely declarative, high-level specifications.

Acknowledgments

We thank Rastislav Bodík, Tobias Nipkow, Aleksandar Milićević, Djordje Jevdjić, Saša Misailović, and Ravichandhran Madhavan for insightful discussions and useful comments, and the anonymous reviewers for feedback on a paper draft. This work was supported by the European Research Council project *'Implicit Programming'*, the National Science Foundation grant number *CCF-1438969*, and Swiss National Science Foundation grant *'Constraint Solving Infrastructure for Program Analysis'*.

References

[1] SciFe website and repository. `http://kaptoxic.github.io/SciFe`.

[2] S. Abramov and R. Glück. Principles of Inverse Computation and the Universal Resolving Algorithm. *The Essence of Computation*, 2002.

[3] R. Blanc, V. Kuncak, E. Kneuss, and P. Suter. An overview of the Leon verification system. In *Scala*, 2013.

[4] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated Testing Based on Java Predicates. In *ISSTA*, 2002.

[5] C. Brabrand, A. Møller, and M. I. Schwartzbach. Dual syntax for XML languages. In *DBLP*, 2005.

[6] L. Bulwahn. The New Quickcheck for Isabelle: Random, Exhaustive and Symbolic Testing under One Roof. In *CPP*, 2012.

[7] J. Christiansen and S. Fischer. EasyCheck – Test data for free. In *FLOPS*, 2008.

[8] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Experimental assessment of random testing for object-oriented software. In *ISSTA*, 2007.

[9] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. ARTOO: adaptive random testing for object-oriented software. In *ICSE*, 2008.

[10] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP*, 2000.

[11] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *FSE*, 2007.

[12] M. Dénès, C. Hritcu, L. Lampropoulos, Z. Paraskevopoulou, and B. C. Pierce. QuickChick: Property-based testing for Coq. In *The Coq Workshop*, 2014.

[13] K. Dewey, L. Nichols, and B. Hardekopf. Automated Data Structure Generation: Refuting Common Wisdom. Unpublished manuscript, at: http://www.cs.ucsb.edu/ benh/research/papers.html, 2015.

[14] J. Duregard, P. Jansson, and M. Wang. Feat: Functional Enumeration of Algebraic Types. In *Haskell*, 2012.

[15] B. Elkarablieh, D. Marinov, and S. Khurshid. Efficient solving of structural constraints. In *ISSTA*, 2008.

[16] M. Erwig. Inductive graphs and functional graph algorithms. *Journal of Functional Programming*, 2001.

[17] P. Flajolet and B. Salvy. Computer Algebra Libraries for Combinatorial Structures. *Journal of Symbolic Computation*, 1995.

[18] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov. Test generation through programming in UDITA. In *ICSE*, 2010.

[19] J. Goodenough and S. Gerhart. Toward a Theory of Test Data Selection. *IEEE Transactions on Software Engineering*, 1975.

[20] J. Goubault. Implementing functional languages with fast equality, sets and maps: an exercise in hash consing. *JFLA*, 1994.

[21] R. E. Griswold, D. R. Hanson, and J. T. Korb. Generators in ICON. *TOPLAS*, 1981.

[22] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac. Complete completion using types and weights. In *PLDI*, 2013.

[23] J. Hughes. Generalizing monads to arrows. *Science of Computer Programming*, 2000.

[24] J. Hughes. QuickCheck testing for fun and profit. In *PADL*, 2007.

[25] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 2002.

[26] A. Kanade and R. Alur. Representation dependence testing using program inversion. In *FSE*, 2010.

[27] S. Kawanaka and H. Hosoya. biXid: a bidirectional transformation language for XML. In *ICFP*, 2006.

[28] S. Khurshid and D. Marinov. TestEra: Specification-based testing of java programs using SAT. In *ASE*, 2004.

[29] O. Kiselyov, C.-c. Shan, D. P. Friedman, and A. Sabry. Backtracking, interleaving, and terminating monad transformers. In *ICFP*, 2005.

[30] E. Kneuss, I. Kuraj, V. Kuncak, and P. Suter. Synthesis modulo recursive functions. In *OOPSLA*, 2013.

[31] A. S. Köksal, V. Kuncak, and P. Suter. Constraints as control. In *POPL*, 2012.

[32] I. Kuraj. Interactive Code Generation. Master's thesis, EPFL, Lausanne, 2013.

[33] I. Kuraj and V. Kuncak. SciFe: Scala Framework for Efficient Enumeration of Data Structures with Invariants. In *Scala*, 2014.

[34] H. Miller, P. Haller, E. Burmako, and M. Odersky. Instant Pickles: Generating Object-oriented Pickler Combinators for Fast and Extensible Serialization. In *OOPSLA*, 2013.

[35] S. Misailovic, A. Milicevic, N. Petrovic, S. Khurshid, and D. Marinov. Parallel Test Generation and Execution with Korat. In *FSE*, 2007.

[36] S.-C. Mu, Z. Hu, and M. Takeichi. An Injective Language for Reversible Computation. In *MPC*, 2004.

[37] M. Odersky et al. An Overview of the Scala Programming Language (2nd Edition). Technical report, EPFL, 2006.

[38] C. Pacheco and M. D. Ernst. Randoop: Feedback-Directed Random Testing for Java. In *OOPSLA*, 2007.

[39] A. Rosenberg. Efficient pairing functions–and why you should care. In *IPDPS*, 2003.

[40] N. Rosner, V. Bengolea, P. Ponzio, S. A. Khalek, N. Aguirre, M. F. Frias, and S. Khurshid. Bounded exhaustive test input generation from hybrid invariants. In *OOPSLA*, 2014.

[41] B. J. Ross. Running programs backwards: The logical inversion of imperative computation. *Formal Aspects of Computing*, 1997.

[42] C. Runciman, M. Naylor, and F. Lindblad. Smallcheck and lazy smallcheck. In *Haskell*, 2008.

[43] V. Senni and F. Fioravanti. Generation of test data structures using constraint logic programming. *TAP*, 2012.

[44] R. Sharma, M. Gligoric, V. Jagannath, and D. Marinov. A Comparison of Constraint-Based and Sequence-Based Generation of Complex Input Data Structures. In *ICSTW*, 2010.

[45] K. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson. Software assurance by bounded exhaustive testing. In *ISSTA*, 2004.

[46] M. Szudzik. An elegant pairing function. In *NKS*, 2006.

[47] P. Tarau. Deriving a Fast Inverse of the Generalized Cantor N-tupling Bijection. In *ICLP*, 2012.

[48] W. Visser, C. S. Păsăreanu, and R. Pelánek. Test input generation for red-black trees using abstraction. In *ASE*, 2005.

[49] J. Wells and B. Yakobowski. Graph-based proof counting and enumeration with applications for program fragment synthesis. In *LOPSTR*, 2005.

[50] A. Yakushev and J. Jeuring. Enumerating well-typed terms generically. *AAIP*, 2010.