CHALMERS | UNIVERSITY OF GOTHENBURG

AGATA

Random generation of test data

Master of Science Thesis

JONAS ALMSTRÖM DUREGÅRD

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, December 2009

AGATA
Random generation of test data

JONAS ALMSTRÖM DUREGÅRD

Examiner: Koen Lindström Claessen

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden December 2009

Abstract

Agata Generates Algebraic Types Automatically. The generated data can be used to perform property based testing with the Haskell testing framework QuickCheck, or the alternative framework SmallCheck. Unlike regular QuickCheck generators, Agata generators are mechanically derivable from the definition of an algebraic data type. Agata moves all logic from the individual generators into a customizable wrapper function. This enables user-side reconfiguration of generators without rewriting their source code.

Agata uses a novel definition of size. This resolves the scalability issues of QuickCheck, associated with generating collection-type data-structures. Experimental results demonstrate the existence of properties falsifiable by Agata but not by QuickCheck nor SmallCheck

Automation and suitability for collection-type structures make Agata ideal for parser testing. Agata is implemented as an extension of the BNFC parser generator. Experimental results demonstrates the usability of this tool, discovering several errors in published software.

Contents

Chapter 1
Introduction

When testing software with a finite set of test cases, chosen from an infinite set of possible inputs, which test cases are the best ones to use? The ones that cause a failure of course! This bug-finding requirement makes it very difficult to evaluate the quality of a set of test cases. If the choice is made automatically by a test data generator, the quality of the generator is equally difficult to estimate.

Consider the design of a random test-case generator for a general-purpose data type. The generator is intended to be useable for any automatic tests involving the type. Somewhere in the design process, a choice needs to be made. The choice will impact the distribution of the values generated. An example of this would be a generator for a type of binary trees: one decision might skew distribution in favor of balanced trees, whereas another will typically create deeper unbalanced trees. Since no assumptions can be made as to the nature of the tests, any value may cause a failure. This implies that the design-choice made will always be the wrong one, at least for some test-situation.

One solution to this problem is to create several generators, each with a unique design. If the design-choice is binary, then two separate generators are created. If the choice is polyadic (e.g. the value of a numeric constant) then a parameter can be introduced to create an infinite number of generators. The tester must then choose a generator for each specific test.

Even for a specific test, it may be difficult to determine which of a set of generators to use. A meta-generator may then be used to randomly choose a generator from a finite set, or randomly generate a parameter to construct one from an infinite set. Just as randomness is trusted to discover a failing input from a large set, randomness is also trusted to choose a generator-design that is likely to find errors.

This approach suggests that design-choices shall not be hard-coded into general-purpose generators. This limitation transforms the process of constructing these generators into a completely mechanical task. As such it should be performed by a machine, rather than a human (since human labor is slower, more costly and more error prone).

Agata introduces a new method to define test data generators in Haskell. The generators are compatible with the property based testing frameworks QuickCheck[1] and SmallCheck[2]. The process of writing agata generators is quite mechanical, and a tool that automatically creates generators for all types in a module is available.

1.1 Background

"If I have seen further it is only by standing on the shoulders of giants"
-Isaac Newton

1.1.1 Automated testing

Automated testing is the use of software to control the execution of test-runs. This includes setting up preconditions for each test, running the test, and comparing actual outcome to expected outcome[3]. To do this, the testing software will typically need to have a set of test inputs and an oracle that given an input outputs the expected outcome.

Regression testing Automated testing is especially useful in combination with a technique called regression testing[4]. In regression testing, a large suite of tests are collected for each module (unit) of software and these tests are repeated each time the software is changed. This mitigates the problems related to iterative programming where seemingly beneficial changes actually causes previously correct software to malfunction.

Property based testing In property based testing the primary manual component of the testing process is writing a set of properties that hold for all (or a well defined subset) of the possible inputs. Properties are derived from specifications, if needed these are further formalized and possibly divided into simpler properties until each property is trivial to implement as executable code[5]. Typically, the properties require the same input as the expected tested function/functions. The general notion is that the property is universally quantified over any values that satisfies the requirements on input for this property (i.e. correctly typed and/or other requirements defined by the programmer). Using a strongly typed languages is preferable, since often there is no need to specify requirements on the input other than specifying its type.

Consider a sorting routine. The most obvious property is "for any input, the output of the routine should be sorted". This property is easy enough to code, without reimplementing the sorting routine in the property. If this property is combined with "for any input A, the output of the routine will be a permutation of A", any software that satisfies both is correctly sorting its input. When coding the second property care must be taken not to construct a circular argument i.e. use a sorting function to decide if output is a permutation. If a verified working sorting function is available, but the implementation under test is still useful, then the obvious property would be that for any input the resulting output will be the same as from the working sorting function.

Automatic test data generation One advantage of using a property as a test oracle, instead of writing expected input/output pairs, is the possibility of automatic test data generation[5]. Writing lots of test case inputs is tedious work for the testers and important corner cases can easily be missed by a biased human mind. In property based testing the tester can instead use software tools to write a generator that automatically creates test data in the input domain. The generators may choose values randomly, or they may enumerate all values of a type (or some well defined subset). This approach is especially rewarding when using a strongly typed language. Instead of writing generators for each test, a generator is associated with each type. The testing software will then simply have to determine the input types of a property to determine which generators to use.

1.1.2 QuickCheck

QuickCheck[1] is a software tool for property based testing of Haskell programs[1]. A property is typically coded as a Haskell function with a Boolean result, and the QuickCheck framework can automatically test properties with random input. There are predefined type-specific test data generators for a number of standard Haskell data types, including functions. There is also a library of combinators to create custom test data generators.

An embedded language for generators The combinators for writing generators provided by QuickCheck constitute an embedded language. As with any embedded language it inherits the features of the Host language (Haskell).

This generator specifying language has a powerful monadic interface A generator is essentially a function that maps random seeds to values. This gives a very high flexibility when constructing generators.

Defining size The intention of a generator is to randomly choose test data from the generated type. The choosing process determines the distribution, i.e. the probability of generating any specific value. Since the default behavior of QuickCheck is to generate finite values only, a difficult problem arises when infinite types are introduced (e.g. recursive types). QuickCheck ensures termination for generators of infinite types by dividing each type into an infinite number of finite subsets. Each such set is uniquely identified by a non negative number n. For type t, the set t_n contains all values from t of size no greater than n. Size in this context is not the exact number of constructors used in the value, nor is it any other general property of algebraic types. The definition of size is determined on a type-to-type basis, the only real requirement is that a sized value is guaranteed to be finite.

When generating test data, the QuickCheck framework will determine which subset of the type to generate from, by choosing a upper size bound (n). QuickCheck will start with a small n then gradually increase the size bound if the tested property holds.

These are the default generators for the types `(,)` and `[]`:

1 Tools based on the original Haskell QuickCheck exists for a multitude of programming languages.

Definition 1 - Built-in QuickCheck generators

```
instance (Arbitrary a, Arbitrary b) => Arbitrary (a,b)
where
  arbitrary = liftM2 (,) arbitrary arbitrary

instance Arbitrary a => Arbitrary [a] where
  arbitrary = sized $ \n →
    do k <- choose (0,n)
       sequence [ arbitrary | _ <- [1..k] ]
```

Points of interest here is that the size of a list (k) is chosen randomly between 0 and n, n being the upper size bound provided by the QuickCheck framework. The generator for pairs is the simplest possible one; generate both elements and pair them. The upper size bound is the same when generating both elements of a pair. For the list generator, each element in the list will have the same global size bound as the list itself. This implies the following definition of size for these data structures:

Example 1 - fictional size functions for (,) and []

```
size (a,b) = max (size a) (size b)

size [] = 0
size xs = max (length xs) (maximum $ map size xs)
```

This constitutes a guideline for general purpose data structures. If all relevant generators follow these guidelines, the following definition of size can be used:

- The size of a value v is $max(k,r)$ where k is the size used for the structure of v, and r is the size of the largest value in a non recursive field throughout the structure of v.

For the generator of type t, any value with size less than or equal to the size bound can be generated, i.e. for all v n: $v \in t \Leftrightarrow size(v) \leq n$. This implies the following property of the size bounded subset t_n: $t_n \subset t_{n+1}$.

Probabilities Given any generator for an infinite type t, it is possible to determine the probability $P_n(v)$ of generating any value v, using the size bound n. This determines the size-bounded subset t_n because, for all values v in type t, $v \notin t_n \Leftrightarrow P_n(v) > 0$.

For every generator, the average probability of generating any value $v \notin t_n$ using size bound n will be $1/|t_m|$. If the distribution is uniform across the set then this will be the actual probability for every value. For any other distribution, there will be some worst case value that has a lower probability of being generated.

Since $t(n)$ is a subset of $t(n+1)$, $|t(n)| \leq |t(n+1)|$ which means that as the size bound increases, the average probability of generating a specific value decreases. Any value v will thus have a probability peak at $n = size(v)$. If the size bound is lower, then v can not be generated. If it is higher, then the probability of generating v might decrease since values are generated from a larger set. This allows the definition of peak average probability $P(v) = 1/|t_{\text{size(v)}}|$.

For the type *`[Bool]`*, *`size(x)`* is the length of the list. The number of values in |*`[Bool]`*$_n$| is 2^{n+1}-1. The peak average probability of generating *`[True,True,True]`* (or any other combination of three Booleans) is thus:

Example 2 - Peak average probability

$$
\begin{aligned}
&P([True,True,True]) &=\\
&1/|[Bool]_{size([True,True,True])}| &=\\
&1/|[Bool]_3| &=\\
&1/(2^{3+1}-1) &=\\
&1/15
\end{aligned}
$$

This is the probability generating any specific list from [Bool]$_3$, if the list generator has a uniform distribution. The default list generator provided by QuickCheck does not have a uniform distribution. With the default generator, the probability *P_3`([True,True,True])`* is actually 1 / (3*2^3)=1/24, because there is a one in three chance to choose a list of length 3. The probability *P_3`([])`* on the other hand is 1/3.

Shrinking When QuickCheck has found a counterexample of a property it will "shrink" the value before presenting it to the user, by removing data irrelevant to the failure. In order for this to work a shrinking function must be added to the test data generator. A shrinking function for type `t` has type `t -> [t]` and all elements of the returned list should be strictly smaller than the parameter value. A wrapper function will then re-test the property for any values the shrinking function returns. If one of the smaller values falsify the property, it is adopted as the new counterexample. The shrinking mechanism is then executed iteratively on this value, until no smaller counterexample are found. Shrinking counterexamples often simplifies the debugging effort drastically.

1.1.3 SmallCheck/Lazy SmallCheck

Similar to QuickCheck, SmallCheck[2] uses automatic test data generators based on the Haskell type system to conduct property based testing. Instead of using randomly generated values, SmallCheck enumerates all values of a subset of the type. The subset is limited by a numeric depth value, determined by the tester. The depth of a nullary constructor is 0, the depth of any positive-arity constructor is one greater than the deepest component. If a counterexample is found by SmallCheck it will be a simplest one, eliminating the need for shrinking.

1.2 Problem description

"Testing shows the presence, not the absence of bugs"
-Edsger W. Dijkstra

This chapter describes the problems that motivates the development of Agata[1]. A few specific flaws that test data generators can exhibit are presented, as well as examples where existing generators exhibit these flaws.

1.2.1 Properties of types

This section will clarify a few concepts used in throughout the following chapters.

Set properties A type is a set of values (not all sets of values are types though, so the concepts are not interchangeable). All general properties of sets can be used to describe types. For instance a type can be infinite or finite[2], any set of values may be a subset of a type, a value may or may not be a member of a type etc.

Dimension Many examples in this thesis use the concept of dimension to classify generated types. The dimension of a type is usually the number of nested collection-type data-structures used to construct it. For instance *`dimension([()])`* is one, *`dimension([[()]])`* is two etc. A recursive type will have higher dimension than any non-recursive type in its recursive constructors. For any type *`t`*, *`dimension(t)`* is formally defined by:

- If *`t`* is an enumeration type (nullary constructors only), *`dimension(t)`* is 0.
- If *`t`* is a recursive type with no non-recursive fields, *`dimension(t)`* is 1.
- Every other type has two sets of component types:
  - fs is the set of component types occurring in non recursive constructors
  - gs is the set of non recursive types occurring in recursive constructors.
  - *`dimension(t)`* is $max(\{dimension(g) + 1, g \in gs\} \cup \{dimension(f), f \in fs\})$

1.2.2 General problems with test data generators

This section will describe a few problems that may arise when designing test data generators.

Tedious and error prone to write by hand In order to produce a generator for any type, a certain amount of Haskell code must be written by a programmer. While this problem applies to all areas of software development and not just generator construction, it is stated explicitly here because automation is an important motivation for the development of Agata.

1 True to the well established computing tradition of using recursive acronyms, AGATA spells out "AGATA Generates Algebraic Types Automatically". While acronyms are typically all upper case, Agata is instead used as a proper name, capitalizing only the first letter.

2 Some would use the term "infinite type" for types that have infinitely large definitions.

If possible, any task should of course be delegated to a computer rather than a person. This is especially true for critical tasks such as software testing, since a testing program can be proven correct, whereas a testing human can not.

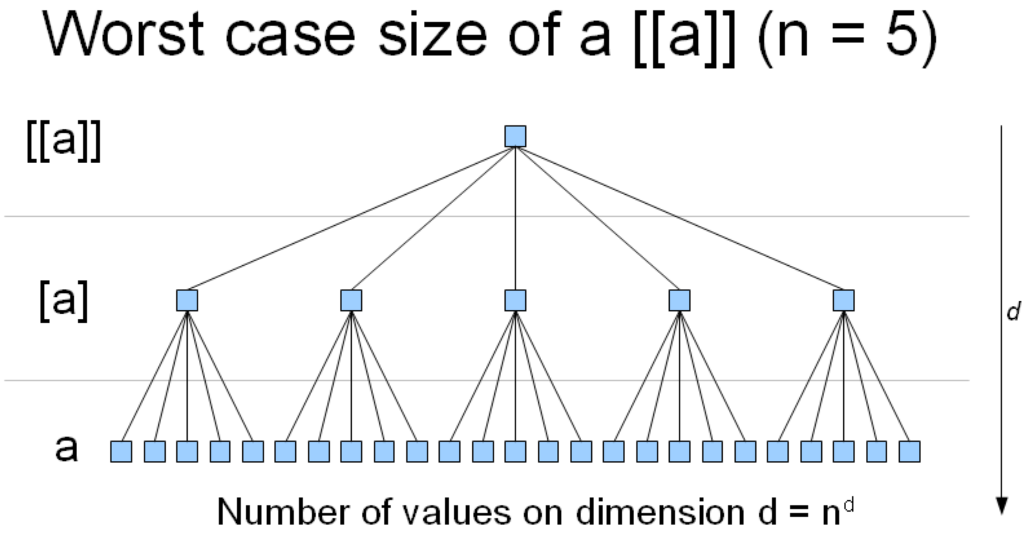
Scalability and termination When writing a generator, care must be taken to ensure that the generating process will terminate, i.e. that all generated values are finite. QuickCheck provides a size bound to assist generators in limiting the size of generated values[1]. Still, guaranteeing termination might cause a lot of extra work. This problem is particularly present when dealing with some specific mutually recursive types:

Example 3 - Parameter induced mutual recursion

```
data D = MkD [D]
d = arbitrary >>= MkD
```

The generator *d* will in most cases fail to terminate. The default list generator assumes that the type parameter of *[]* is not a mutually recursive type, and thus all generated *D* values are given the same size bound as the *D* value that contains them. This severely reduces the re-usability of the default list generator in these cases, essentially forcing the implementer of *d* to rewrite the list generator with the assumption that the type variable is recursive.

Even if a generator is guaranteed to terminate, it may generate values that are to large to process or even fit in memory. This scalability problem stems from the discrepancy between absolute size (i.e. the number of constructors used), and QuickCheck size (i.e. the largest sub-value generated by a single generator). For instance the value *(a,a)* will have the same QuickCheck size as the value *a*. The situation is worse for lists, if *size(a)* = *n* then *size(replicate n a)* = *n*. Consider the types of *d*-dimensional lists (where *[a]* is one-dimensional *[[a]]* is two-dimensional etc.). How does the worst case absolute size relate to the QuickCheck size? Since the standard list generator determined the size of the generated list randomly between 0 and the size bound *n*, each list of dimension *d+2* will contain on average *n*/2 lists of dimension *d+1* and each of these will also contain *n*/2 elements. This illustration demonstrates the growth for a two-dimensional list and *n=5*:


Worst case size of a [[a]] (n = 5)
[[a]]
[a]
a
d
Number of values on dimension d = n^d

The number of list elements will be exponential to the dimension of the list. Consider the five-dimensional list of integer values, *[[[[[Int]]]]]*. The default settings for QuickCheck is to use 100 as the highest value of *n*. On average a value of this size would contain 50^5 = 312500000 integers and in the worst case 100^5 = 10000000000.

Since the default behavior when writing generators is to use the same size bound for non recursive fields as for the value that contains them, this problem is not limited to nested lists. Any nested recursive types will exhibit the same growth.

Distribution and coverage Several problems with automated testing relates to coverage[6]. If a specific type constructor is absent throughout a test suite, parts of the implementation under test may never be executed and bugs in these sections will never be detected. In random testing, probability must be considered as well. Although there is a theoretical chance that a specific test case is used, if it will require millions of test runs before it is generated it is in reality not covered. As previously mentioned QuickCheck will divide the type t into finite size-bounded subsets t_n where n is the upper size bound of member values.

A natural consequence of using only finite subsets of an infinite type is that some values will not be possible, or probable, ever to be generated. As shown, the average probability of generating a value from a specific subset is reverse proportional the size of the subset[1]. Because $|t_n| \subset |t_{n+1}|$, subsets grow with the QuickCheck-size bound they represent. This implies that large values (as defined by QuickCheck) will have a lower probability of generation. When calculating probabilities the tester must be aware of the discrepancy between the absolute size (number of constructors) and the QuickCheck-size of a value. Consider the following value for instance:

Example 4 - A nested list

```
x = [[],[],[],[],[],[],[],[]] :: [[()]]
```

The value *x* is the list of length eight containing only empty lists. This is a small test case in absolute size (eight recursive steps, 17 constructors total). The set of values with equal or smaller absolute size is fairly small. Since the recursive constructors are concentrated to the outer list however, the minimal size bound required to generate this with the default list generator is eight. This places the value in a much larger subset of the type. The peak average probability $P(x)$ of generating *x* is $1/8^9$ = 1/134217728.

This discrepancy between the definition of sizes is not limited to nested recursive types. Consider the type *([()],[()])* for instance. The probability of generating *([(),()],[(),()])* is greater than the probability of generating *([()],[(),(),()])*, although their absolute sizes are equal. This is because the former value is in the subset $[()]_2$ whereas the latter is in the larger set $[()]_3$.

If a property fails for a specific isolated test case and the QuickCheck size of this test case is too large, the failure will never be detected. This implies that when writing or using a generator for any property the following must be considered:

- How fast does the subsets of the type grow using this generator?

1 A limited number of values may be excluded from this rule, having constant probability of generation independent of the size of the subset.

- At which point does the subsets become to large do find isolated test cases?
  - Is there a risk that there is an isolated test case in a higher subset that falsifies the property?

Each of these questions require careful consideration, and trying to answer them constitutes a significant amount of work. In most cases, a tester will instead use the tools QuickCheck provides to monitor distribution of values and determine at a glance if it is acceptable. A process that is more error prone.

1.2.3 Limitations

Implicit invariants Many data types have inherent implicit invariants. These invariants range from simple predicates such as numerical non-negativity to more complex ones such as type correctness of a programming language. Agata will not respect any such invariants when generating test data. The primary reason for this limitation is that there is no way to infer them by studying the definition of the type (hence the name implicit invariants). While it would be nice to have a means of specifying such invariants in a precise enough fashion to use it for automatic generation of test data, such a project is far out of scope for this thesis.

In some cases the invariant can be specified as a Boolean predicate and this predicate can be used as a precondition for test data in the definition of properties. This requires at least the following properties of the generator and invariant:

- There must be a reasonable probability that a random value will respect the invariant. If not, QuickCheck will not find any values that to use and give up.
- The probability that a random value will respect the invariant must not depend greatly on the size or nature of the value. If this is the case, distribution will inevitably be effected.

The probability that an integer value is non negative is ½ independently of the absolute size of the integer, and this is a reasonable precondition. If the same invariant is required on each element of a list of integers, the distribution will be skewed towards smaller lists. If the invariant is that a random C program initiates all variables used, the generated cases are unlikely to contain any variables at all.

Infinite values Agata will only generate fully defined finite values. If a type has only infinite values, Agata might not be able to generate it.

Chapter 2
Agata

2.1 The Generator-generating algorithm

The core of Agata is an algorithm that creates a test data generator for an algebraic type by studying its structure.

2.1.1 Inadequate algorithms

This section describes a few unsuccessful attempts at achieving the required algorithm, and explains their inadequacy.

The naive algorithm The simplest possible algorithm constructs generators similar to the generator for tuples (see Definition 1, p. 5):

1. Choose a constructor randomly.
2. Assign arbitrary values to each field.

The primary problem is termination. A recursive type is not guaranteed to terminate. This forces the generators to distinguish between recursive and non recursive fields, not forgetting mutual recursion. The QuickCheck manual describes this problem and offers the following example as a way of using size to solve it:

Definition 2 - QuickCheck generator for trees

```
data Tree = Leaf Int | Branch Tree Tree

tree = sized tree'
tree' 0 = liftM Leaf arbitrary
tree' n | n>0 =
     oneof [liftM Leaf arbitrary,
            liftM2 Branch subtree subtree]
  where subtree = tree' (n `div` 2)
```

Generalizing this solution yields an algorithm roughly like this:

- Parameter *n* is the upper size bound for the generated data, provided by QuickCheck.

1. If n is 0 choose a non recursive constructor[1]. Otherwise choose any constructor.

1 In some cases where there is mutual recursion, there will not be any non-recursive constructor to choose. In these cases the algorithm must find a recursive constructor that has a path to a non recursive type without returning to the current type. For simplicity, these cases will be ignored henceforth.

2. Assign arbitrary values to non recursive fields.
3. Divide (*n-1)* randomly across recursive fields. Use these new sizes as upper size bounds when generating each field.

The reason for dividing size randomly instead of evenly, is that evenly dividing it will limit the depth of the tree to $Log_2\ n$.

This algorithm will ensure termination, since each field will be generated from a strictly smaller subset of the type. It has some other problems however, notably the correlation between size of the data structure and size of each element remains. As shown this causes the size of generated data to grow exponentially with the dimension of the type. The most straightforward way to solve this problem is to limit the size of non recursive fields as well, making the upper size limit global for the test case.

- Parameter n is the upper bound for size of the generated data

1. If n is 0 choose a non recursive constructor. Otherwise; choose any constructor.
2. Divide (n-1) randomly across all fields. Use these new sizes as upper size bounds when generating each field[1].

Consider the following snippet of Haskell code:

Example 5

```
data Nat = Zero | S Nat
type NatList = [Nat]
```

The algorithm is applied both to *Nat* and *[]*. The subset with upper size bound *n=0* will now contain *[]*, *n=1* will contain *[]* and *[0]*. For *n=2*, *[0,0]* and *[1]* are added to the possible outcomes.

While this algorithm will have a much stronger control of the size of generated data, it introduces a flaw in the distribution of values. Even with random distribution of sizes, if *n=100* the average size of the first element of the list will be 50 and the average total size of the tail will be 50 as well. This skews distribution in favor of descending-ordered lists. The expected size of the first element will be $n/2$, the second $n/4$ and the m:th $n/2^m$. Furthermore, since the tail will be (on average) half the size of the first element, the expected size of the list will be $Log_2\ n$ instead of $n/2$ like the original list generator. This behavior is not unique to linked lists. Most collection data structures will suffer from some variant of this condition, for instance trees will have larger elements near the root.

The QuickCheck list generator With the generator QuickCheck uses to construct arbitrary linked lists (see Definition 1, p 4) the length of the list is determined separately from the generation of individual elements. Combining the algorithm so far with this approach yields a generator that first determines the length of the list, then divides size for each list element.

1 Arguably, fields with non recursive types should not be assigned sizes, sine they have no way of using it.

Example 6 - A new generator for lists

```
instance Arbitrary a => Arbitrary [a] where
  arbitrary = sized $ \n ->
    do k <- choose (0,n)
       ms <- piles k n
       sequence [ resize m arbitrary | m <- ms ]
```

The `piles` function randomly distributes an integer value into a number of piles. Formally, `piles k m` returns a random list of integers of length *k* where the sum of all elements equals *m*. The function `resize` is built in to QuickCheck, it will set the value of the upper size bound to a given value for a given generator.

One could argue that the second argument of piles should be *n-k* instead of *n*. Having *n* means that the actual maximum size increases with the depth of the type tree. However, while in the original generator the relation between depth *d* and max size *m* was $m \in O(n^d)$, it is now $m \in O(nd)$. The reason for not using *n-k* is that it would impact distribution. For instance when generating the type `[[()]]` and `n = 100`, the average number of `[[()]]` elements would be 50, the elements contained in this lists would have a total maximal size of 50. The average length of each of the lists will thus be one. The algorithm could be altered to choose *k* differently; for instance if the average case is that *k* is the square root of *n*, then the expected length of the outer list will be the same as the expected length of any inner list. This requires some assumptions to be made, and the impact these assumptions will have on distribution are difficult to predict. Using the full size (*n*) as the second argument of `piles` reduces the strictness of the upper size bound in favor of not having to make assumptions, maintaining genericity.

This adjustment of the list generator impacts the distribution in several ways. Most apparently it inverts the correlation between the length of the list and the size of the elements, long lists will tend to have small elements instead of large. Short lists with small elements will be generated whenever *n* is low. Short lists with large elements will occur when *n* is large and a small *k* is chosen. If a large *k* is chosen instead, long list with small elements will be generated. The only way to generate long lists with large elements is if *n* is very large, which is natural given that the purpose of the modified algorithm is to reduce the size of test data.

If this approach is to be generalized to any algebraic data type, several uncertainties must fist be decided. Consider the type `(Int,[Int])` for instance. How will size be distributed between the fist and the second value of this pair? What is the relation between the expected size of the single integer on the left and any integer in the list? If size is randomly divided between them, what will the distribution be? In the average case the size of the single integer in the pair will equal the sum of the sizes of all integers in the list. This means there is a correlation between a types position in the type structure and the expected size of its values. This becomes a major problem when dealing with many everyday situations. Consider the following type for the abstract syntax of an object oriented language:

Example 7 - Syntax of a computer language

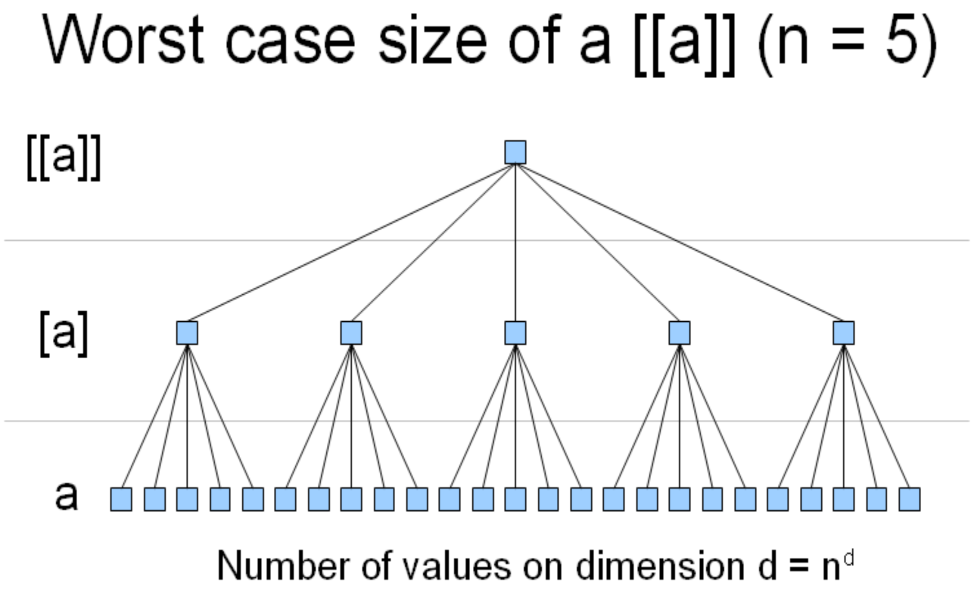
```
data Class = Class String [Function]
```

The String field is the name of a class. The other field is the contents of the class. The unreasonable outcome of using randomly divided size is that in the average case half the size of the program will be spent on the name of the module, the other half being spent on a list of functions and their bodies. Clearly, the algorithm must distinguish types by their complexity, to divide size fairly between fields.

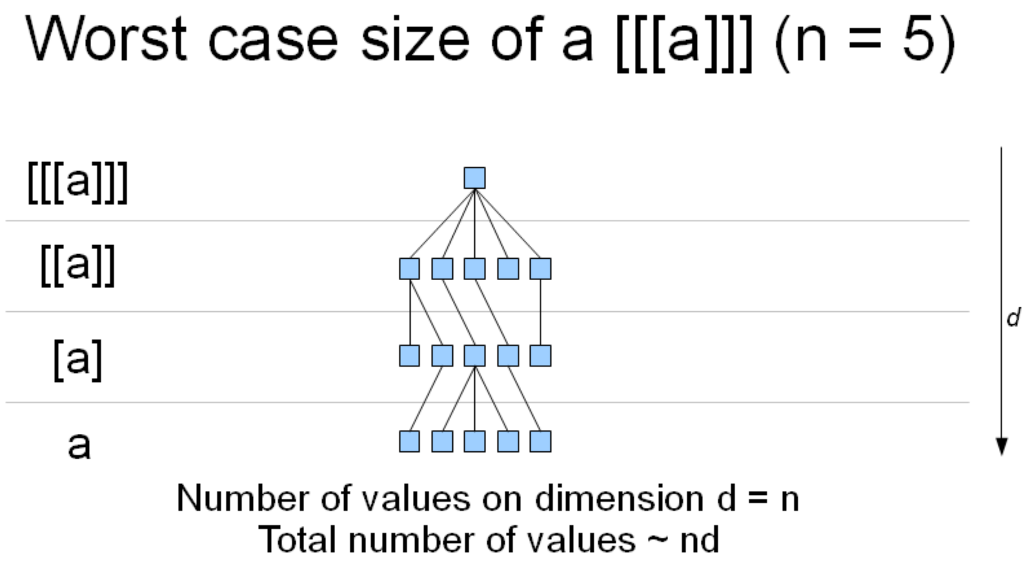
2.1.2 Division of size

When there was correlation between the position of an element in a data structure and the expected size of the element, it was solved by dividing size randomly between all elements. When expected values correlate to positions in a type structure, a similar solution is possible: divide size equally between all values of the same type. Since all elements of a list have the same type, this incorporates the mechanisms to eliminate correlation between list position and size. It also solves the problem with the unfair class generator (see Example 7). On average the string will be as large as any other string throughout the generated program.

Revisit the example with d-dimensional data structures. The problem with these types, is the exponential growth of the number of elements. This diagram illustrates the problem:


Worst case size of a [[a]] (n = 5)
[[a]]
[a]
a
d
Number of values on dimension d = n^d

Dividing size across all elements of the same type limits the number of elements of type a in a d-dimensional list of a to n, where n is the upper size bound (5 in these examples). The worst case is then of this form:


Worst case size of a [[[a]]] (n = 5)
[[[a]]]
[[a]]
[a]
a
d
Number of values on dimension d = n
Total number of values ~ nd

Having generalized from dividing size across data structure elements to dividing across types, there is still one more step of generalization possible. Consider the type *`(Int,Int,Integer)`*. If size is divided by type, the average case is that each of the *`Int`* values are only half as large as the *`Integer`* value. In the interest of limiting size, there is little difference between *`Integer`* and *`Int,`* hence the same size quota should be divided among all three values. This relaxes the size division paradigm from "divide size randomly across values of same type" to "divide size across values of types with similar structure".

There is natural limitation imposed on this similarity relation. In order to divide size across values in a group of similar types, the number of values in these types must be counted. Thus if the size of the values of type *`a`* determine the number of values of type *`b`*, *`a`* and *`b`* can not be similar. So *`[a]`* can never be similar to *`a`* etc. This corresponds well with the definition of dimension. The widest possible definition of "similar structure", useable in this context, is thus "of equal dimension".

A small correction For some odd types, counting the number of values on a specific dimension will require some values of that dimension to be defined. Consider this type for instance:

Example 8 - A peculiar type

```
Type A = Nil | Cons Bool A | Transform [()]
```

The type *`A`* is a list with the strange property that it might terminate in a list of units rather than in an empty list. This is essentially the same type as ([Bool],Maybe [()]). The problem with *`A`* is that if the list terminates with Nil there is one value of *`dimension(A)`*, but if the list terminates with a new list there are two (since the terminating list has the same dimension as *`A`*). Either the values are considered as a single list, and the generator will have to determine how to divide size between them, or "similar structure" is redefined so that *`A`* and *`[()]`* are no longer similar.

Since the counting becomes even harder for mutual recursive types, Agata uses a slightly modified concept of dimension to classify types. With this new definition, any recursive type will have higher dimension than its component types. Using Agata's definition *`dimension(A)`* will be 2, because it is a recursive type and it contain *`[()]`* which is dimension 1. Formally, dimension is defined in the following way:

1. Non-recursive types have dimension equal to that of the highest dimension occurring in fields of its constructors, 0 for no fields.
2. Recursive types have dimension one greater than that of the highest dimension type occurring in (non recursive) fields of its constructors and the constructors of mutually recursive types.

The following examples demonstrate how these rules are applied

Example 9

```
data A = MkA | MkAB B
data B = MkB
```

No recursion is present in these types, rule 1 applies. *`B`* has no fields, thus its dimension is zero. The highest dimension field of *`A`* is of type *`B`*. This implies that the *`dimension(A)`* will equal the *`dimension(B)`*, zero. Now consider these types of Peano numbers and lists and trees of *`()`*:

Example 10

```
data Nat = Zero | Succ Nat
type List = [()]
data Tree = Branch Tree Tree | Leaf ()
```

All of these are recursive types, and applying rule 2 reveals that they all have the same dimension. *`Nat`* have no non-recursive fields, subsequently the maximal dimension is zero and the final dimension for *`Nat`* is one. The highest dimension fields of *`List`* and *`Tree`* are of type *`()`* with dimension zero, so both these types are dimension one as well. Now consider these types:

Example 11

```
type ListList  = [[()]]
data Tree      = Leaf Nat | Branch Forest
data Forest    = Forest [Tree]
```

These types are all dimension two. *`ListList`* has *`[()]`* as its only non recursive field, *`[()]`* is dimension one so *`ListList`* is two. *`Tree`* and *`Forest`* are mutually recursive, so the fields of both types are considered when determining their dimension (mutually recursive types will always have the same dimension). The highest non recursive field is *`Nat`*, and *`dimension(Nat)`* is one, so *`dimension(Tree) = dimension(Forest) = two`*. Consider the following types:

Example 12

```
data NatOrList = Left Nat | Right ListList
data NatAndList = Pair Nat ListList
```

These types are both non recursive, so rule 1 applies. The respective unions of field-types in constructors are identical for both; *`Nat`* and *`ListList`*. *`Nat`* is dimension one and `ListList` is dimension two, meaning `NatOrList` and `NatAndList` are dimension two as well.

2.1.3 Counting entry point values

As mentioned, dividing a fixed size across all values of types with the same dimension requires some way of determining how many values are needed of each type. An entry point value is a value that is not in the recursive field of another value. In a list of natural numbers for instance, each natural number is an entry point value. The list itself may be an entry point value in a pair of a list and some other type. The tail of the list however, is not an entry point value.

Each entry point value is designated a size limit, which is used to generate the actual recursive structure value. The total number of values of a type will be proportional to the total size designated to entry point values of this type.

There are some restrictions on the order in which you can generate types. For instance the structure of all trees of natural numbers must be generated before all natural numbers, because the structure of the trees may affect the number of natural number entry point values. As mentioned, the definition of dimension imply the following important property: For any value v of a type with dimension d, to determine the number of entry point values with types of any dimension $n < d$ occurring throughout the structure of v it is only required to inspect values of dimension $n + 1$ or higher. This allows the following algorithm to be used to generate a value of type t on dimension d:

- Start with an undefined value, x, a size bound n and a current dimension $c = d$.

1. If $c = 0$ define all values inside x on this dimension (no recursion possible).
2. If $c > 0$:
    1. Count the number of entry point values (k) on dimension c.
    2. Divide a random fraction of n randomly into a set of k integer values (ns).
    3. Define all values of dimension c in x. Each entry point value is assigned a unique number from ns. The entry point value will contain exactly this number of recursions. Non-recursive fields (fields of lower dimension than c) are left undefined.
    4. Restart with $c := c - 1$ and the updated value of x.

This algorithm has the following properties:

- When the algorithm terminates all values will be defined.
- The maximum number of recursive constructors used is $n*l$.
- The subset defined by size bound n will contain all values where the number of constructors required on any specific dimension is less or equal to n.

2.1.4 A class of Buildable types

In order to perform a similar operation on a multitude of types in Haskell, such as generating random test data, type classes are used. In QuickCheck, a generator is defined directly in a type class (`Arbitrary`). These generators can then be combined to construct generators for new types. Since the algorithm described above requires some overview of the generating process as a whole, it is not desirable to build this logic directly into the this type class. Instead a type class that supplies a minimal set of operations required to implement the algorithm is used. A wrapper function is then used to construct a QuickCheck generator for any type in this class.

The algorithm needs to know the dimension of the type it will be generating. This calls for the following first definition of the class:

Example 13 - The dimension function

```
class Buildable a where
  ...
  dimension :: a -> Int
```

The value passed to *`dimension`* should not be inspected, because it may be undefined. Its only use is to indicate to the Haskell type system which instance declaration to use. This function may return a constant, or it may call the *`dimension`* function for the types of fields in its constructors to calculate the dimension at runtime. This is necessary for parametrized types. If the dimension is calculated at runtime, care must be taken to avoid mutually recursive types calculating the dimension of one another. Failing to do so will result in non-termination.

Build The second part of the minimal requirement is a description of the type. Specifically a list of constructors and for each constructor a descriptions for all fields. This description must be expressive enough to supply functions that perform at least the following operations:

- Determine which fields of the constructors are recursive.
- Given a list of sizes, create a value using this constructor. Each recursive field should be constructed with a unique size from the given list. Non-recursive fields should be left undefined.

Improve Because a defined value needs to be traversed to count and define undefined sub-values, the third component needed is a pattern matching function.

Prototype generators These requirements enable the definition of a few prototype generator that supplies exactly the information required:

Example 14 - Buildable instances for enumeration types

```
instance Buildable () where
  build = [use ()]

instance Buildable Bool where
  build = [use True, use False]
```

To generate an enumeration type, a list of values is the only component needed. The dimension will be calculated at run-time by looking at the dimension of fields of the constructors. Since there are no such fields, the dimension is determined to zero. Also, since there are no fields, these types will never have to be improved.

Here is another example of a generator:

Example 15 - Buildable instance for a type of pairs

```
data BoolPair = MkBP Bool Bool
instance Buildable BoolPair where
  build = [construct MkBP $ nonrec . nonrec]
  improve x = case x of
    MkBP a b -> rebuild MkBP $ rb a >=> rb b
```

This type has a single constructor *`MkBP`*. This constructor has two non-recursive fields. This information is encoded by passing the constructor and a twofold composition of the function *`nonrec`* to the *`construct`* function.

This information is sufficient to determine the dimension of *`BoolPair`* to zero. A small performance increase can be achieved by overriding the *`dimension`* function with a constant value, but this will impact modularity because it assumes that the dimension of *`Bool`* will never change.

The function *`improve`* is really not needed for this type, because it's dimension-zero and thus its fields will always be defined. Not including it will imply the same modularity disadvantage as overriding the *`dimension`* function.

Here is another example of a generator:

Example 16 - A list type

```
data BoolList = Nill | MkBL Bool BoolList
instance Buildable BoolList where
  build = [
    use Nill,
    construct MkBL $ rec . nonrec]
  improve x = case x of
    MkBL a b -> rebuild MkBL $ rb a >=> rb b
    _        -> return x
```

This linked list has a *`Nill`* value, and a constructor MkBL which has one non-recursive and one recursive field. The nature of the composition function forces the user to define field recursivity in reverse order. If the order of *`rec`* and *`nonrec`* is accidentally swapped, a type error will occur.

Once again the default *`dimension`* function will automatically calculate the dimension (in this case 1) of the type.

Here is another example of a generator:

Example 17 - A mutual recursive type

```
data BoolListTree   = Leaf [Bool] | Branch
BoolListForest
data BoolListForest = Forest [BoolListTree]

instance Buildable BoolListTree
  dimension _ = 2
  build = [
    construct Leaf $ nonrec,
    construct Branch $ mutrec ]
  improve x = case x of
    Leaf a   -> rebuild leaf $ rb a
    Branch a -> rebuild Branch $ rb a

instance Buildable BoolListForest where
  dimension _ = 2
  build = [construct Forest mutrec]
  improve x = case x of
    Forest a -> rebuild Forest $ rb a
```

For mutual recursive types, the dimension function will typically need to be overridden. This is because these is no obvious way to gather the dimensions of all non-recursive fields in a set of mutual recursive types. Mutually recursive fields should be designated by composing the *mutrec* function. Unfortunately the type system will not enforce correct usage (i.e. *mutrec* and *nonrec* have the same type).

Parametrized types These examples go to great lengths to avoid using type parameters (this also demonstrates how essential this feature is). Type parameters will complicate things because:

- The *dimension* function can not be expressed as a constant, this is a problem when dealing with mutually recursive types.
- It becomes more difficult to determine if a field is recursive or not, e.g. the element field of a list is typically not recursive, but it can be if it is promoted by some other data type (see Example 3, p 7).

For these reasons, parametrized types are discussed in a separate section.

Encapsulation Note that all functions in these examples can be considered primitives for the generator specification language. This means that the type of *build* can be changed without invalidating any instances, as long as the types of *construct* and *use* are changed as well. This encapsulation has several advantages, for instance it allows the user to choose between multiple possible wrapper functions by importing different modules. For those who still wish to know, these are the (current) type signatures of the interesting functions:

Example 18 - The types of some of the language primitives

```
data Application b a =
    Build (Improving (a,[Int]))
  | Collect [Int]
  ...

construct :: Buildable b =>
               a ->
               (Application b a -> Application b b) ->
               Builder b

nonrec :: Buildable a => Application c (a -> b) ->
```

```
                             Application c b

rec :: Buildable a => Application a (a -> b) ->
                      Application a b
```

An *Application* represents one of several predefined computations to be carried out. For instance *Collect* will calculate the dimension of all fields:

Example 19 - Collecting dimensions

```
rec x@(Collect xs) = Collect $ appDimension x : xs
```

2.1.5 The Base module

The module *Test.Agata.Base* defines type classes and functions used by generators created by Agata. Since Agata may be used to generate *Arbitrary* instances directly useable by QuickCheck, the user will typically not need to be familiar with this module. The module contains the type class Buildable:

Definition 3 - The Buildable class

```
class Buildable a where
  build :: [Builder a]
  improve :: a -> Improving a
  dimension :: a -> Int
```

The module also contains all the functions used to write these instances, e.g. *construct*, *use* and *nonrec*. As mentioned, the exact structure of the type *Builder* is encapsulated from the user, to allow a flexibility in the implementation of the wrapper function. Section 2.1.4 describes the information the type carries.

The most interesting function in Base is the wrapper function agata:

Definition 4 - The type signature of the agata function

```
agata :: Buildable a => Gen a
```

Since Agata will automatically generate instances of *Buildable* for all types it analyzes, this function allows the user to define arbitrary instances in a very simple way. For instance if the user wishes to use Agata to generate lists (this will typically cause a collision with the default list generator):

Example 20 - A new Arbitrary instance for lists

```
instance Buildable a => arbitrary [a] where
  arbitrary = agata
```

If the user wishes to use Agata on a case-to-case basis rather than as the default generator, the *forAll* quantifier function from the QuickCheck library may be used:

Example 21 - Using Agata for a specific property

```
prop_sort :: [()] -> Bool
prop_sort = isSorted . sort
```

```
prop_sort_Agata = forAll agata prop_sort
```

Agata uses the `Improving` monad to iteratively define values on each dimension. The `Improving` monad is a combination of a state monad and the `Gen` monad used by QuickCheck:

Definition 5 - The improving monad and its basic functions

```
type Improving a = StateT (Int, Int, [Int]) Gen a

getDimension :: Improving Int
getDimension = gets $ \(l,r,ss) -> l

request :: Improving ()
request = modify $ \(l,r,ss) -> (l,r+1,ss)

acquire :: Improving Int
acquire = do
  (l,r,s:ss) <- get
  put (l,r,ss)
  return s
```

The `Improving` monad carries an integer value representing the currently improved dimension, i.e. the dimension where values are in the process of being defined, and an integer representing the number of entry point values on the dimension below the current. It also carries a list of integers containing allocated sizes for each entry point value on the current dimension. The `getDimension` function simply returns the numerical value of the dimension that is currently being generated. The request function signals that an entry point value on the dimension below the current has been found. The acquire function is used when generating a new entry point value on the current dimension, the integer value returned will be the number of recursions (size) allowed in the new value.

The agata function is defined as follows:

Definition 6 - The agata function

```
agata :: Buildable a => Gen a
agata = sized (dummy undefined) where
  dummy :: Buildable a => a -> Int -> Gen a
  dummy a size = flip evalStateT (dimension a,1,[]) $
ii a
    where
      ii a = getDimension >>= \lvl -> case lvl of
        0 -> put (0,0,[]) >> realImp a
        _ -> do
          x <- distrib size >> realImp a
          dec
          ii x
```

The *dummy* function is required to use the *dimension* function on an undefined value of type *a*. The interesting part is the iteratively improving function, *ii*. It starts by distributing size to the single entry point value of type *a*, this will put a single integer between zero and *size* (the QuickCheck size bound) in the list in the state of the Improving monad. The *realImp* function defines the value using exactly the number of recursions designated, and stores the number of entry point values on the next dimension in the state. The function *dec* will decrease the current-dimension value stored in the state before the function is repeated with the improved value. This is the definition of *distrib* and *dec*:

Definition 7 - The distrib and dec functions

```
distrib :: Int -> Improving ()
distrib k = get >>= \(lvl,r,[]) -> case r of
    0 -> put (lvl,0,[])
    _ -> do
      ms <- lift $ piles r k
      put(lvl,0,ms)

dec :: Improving ()
dec = get >>= \(lvl,r,[]) -> put (lvl-1,r,[])
```

The function *piles* has already been explained (see Example 6, p 12). This leaves the *realImp* function. This function ensures that *improve* is only used on defined values, and that undefined values are either defined, entry-point counted or left undefined as required on the current dimension.

Example 22 - The realImp function

```
realImp :: Buildable a => a -> Improving a
realImp a = do
  cur <- getDimension
  case compare (dimension a) cur of
    GT -> improve a
    EQ -> if cur == 0 then
            realBuild 0 else
            acquire >>= realBuild
    LT -> if dimension a == cur - 1 then
            request >> return a else
            return a
```

The *improve* function always calls *realImp* for all field values. The *realBuild* function will construct a random recursive skeleton of a *Buildable* type, based on the information retrieved from the *build* function. The *Int* argument passed to *realBuild* will determine the number of recursive steps taken to build the skeleton.

2.1.6 Parametrized types

Generation of parametrized types introduces some additional difficulties. For non-parametrized types, the *dimension* function of a *Buildable* instance can always be a constant. For parametrized types this value will need to be computed at runtime (or possibly compile time if the compiler is smart enough). This is not difficult to implement, but just like in the agata function, a dummy value must be used to guide the type system. This is one possible *dimension* function for lists:

Example 23 - A dimension function for [a]

```
instance (Buildable a) => Buildable [a] where
  dimension x = dummy x undefined where
    dummy :: Buildable a => [a] -> a -> Int
    dummy _ a1 = 1+dimension a1
  …
```

Since this *dimension* function is a bit of an eyesore, it would be nice to automatically induce the level of this type instead.

Recursivity A more severe problem is that the "recursivity" of a field is no longer static. For instance the element field of a parametrized list is typically not recursive, but it might be in some cases. The functions *nonrec*, *rec*, and *mutrec* have been used in examples so far, but none of these can describe the recursivity of a field with a type parameter. A fourth recursivity primitive, *autorec*, is needed:

Definition 8 - A Buildable instance for [a]

```
instance (Buildable a) => Buildable [a] where
  improve x = case x of
    (a:b) -> rebuild (:) $ rb a >=> rb b
    _     -> return x
  build = [
      use         []
    , construct   (:)  $ rec . autorec
    ]
```

The improve function is identical to the definition of the non-parametrized *BoolList* (see Example 16, p 18), constructor names aside. The only essential difference in the build function is that the element field is now designated *autorec* instead of *nonrec*. The *autorec* function can be used on any field, provided it is safe to calculate the *dimension* of this field. E.g. if the *rec* function is replaced by *autorec* in the definition above, it will fail to terminate because the default *dimension* function will call itself.

This information is sufficient to automatically induce the *dimension* function for lists. Note that the function assumes that the type parameter is not mutually recursive. Consider the type *D*, discussed earlier, and a possible *dimension* function for the type:

Example 24 - Parameter induced mutual recursion

```
data D = MkD [D]

instance Buildable D where
  dimension x = dimension (undefined :: [D])
  ...
```

This `dimension` function will never terminate, since the `dimension` function for `D` calculates the dimension of `[D]` and vice versa. To avoid rewriting the list generator, the only viable alternative is to use a constant value for dimension i.e.:

Example 25 - A better dimension function

```
instance Buildable D where
  dimension x = 1
  build = [construct MkD $ mutrec]
  ...
```

This will lead to the result that the dimension of `D` is 1 and and the dimension of `[D]` is 2. These values are correct if the types are used in other types (i.e. for entry point values), but when `[D]` occurs inside a `D` value, it must be demoted to dimension 1. When generating `[D]`, the Agata wrapper will automatically perform this demotion.

There are of course numerous other and more complex examples of parameter induced recursion. For instance the type `D` could have type parameters of its own, disabling the use of a constant dimension. Consider this type:

Example 26 - Using parameters to promote (,) to []

```
data Lst a = MkLst (Maybe (a,Lst a))
```

This is essentially a linked list of `a`, implemented by inducing mutual recursion between `(,)` and `Lst`. Just like for `D`, the dimension of the field can not be calculated. It is safe to calculate the dimension of `a` however[1]:

Example 27 - The dimension of list

```
instance (Buildable a) => Buildable (Lst a) where
  dimension x = dummy x undefined where
    dummy :: Buildable a => Lst a -> a -> Int
    dummy _ a1 = 1+dimension a1
  ...
```

This is essentially the same definition as for `[]`, which is natural since they represent the same data structure. This method constitutes a general approach for defining dimension: use constants whenever possible, only the dimensions of parameter types should be calculated at runtime. This approach guarantees termination, and it is relatively efficient since runtime computations impacts performance. The drawback of this approach is low modularity. Even though the generators for `Maybe` and `(,)` are not reimplemented for in the generator `Lst`, intimate knowledge of these types is still required when implementing `Lst`. If the dimension of a type changes, then all types that contain this type may have to change their dimension as well.

1 This assumes of course that the dimension of `a` does not calculate the dimension of `Lst a`, i.e. that `Lst` is not used in an incorrect way by some other type.

2.2 Extensions

This section details a number of extension to Agata, as well as the advantages and disadvantages of using them.

2.2.1 Distribution control

By default, all distribution control in QuickCheck is delegated to the generators. This means that if a property requires an altered distribution, a test-specific generator must be written. In Agata, most of the distribution is determined by the wrapper function. This means that a property can take any Agata-compatible (*Buildable*) type and transform its distribution by using an alternative wrapper function. The *distrib* function determines distribution in the default wrapper. It determines the amount of recursion on each dimension, and distributes it across members on these dimension. A generic wrapper function *agataWith* can be defined:

Definition 9 - agataWith

```
type Distributor = Int -> Int -> Gen(Improving ())

agataWith::Buildable a => Distributor -> Gen a
```

The function takes a distribution strategy, of the type *Distributor*. The first argument of a *Distributor* function is the dimension of the wrapped type (the type variable *a*). The second argument is the size bound passed from QuickCheck. The precondition for the resulting *Improving* computation is that the state of the monad matches the pattern *(dim,r,[])*, where *dim* is the dimension about to be improved and *r* is the number of entry point values on this dimension. The postcondition is that the dimension is unchanged, the length of the list is *r* and the entry point count is reset to 0. The generator *agataWith (return . const distrib)* will be exactly the same as the default *agata* generator.

To demonstrate the flexibility of *agataWith*, consider the following *Distributor*:

Definition 10 - Default QuickCheck generators, provided by Agata

```
exponentialSize :: Distributor
exponentialSize _ s = return $ do
  (lvl,r,[]) <- get
  ns <- sequence $ replicate r $ lift $ choose (0,s)
  put (lvl,0,ns)
```

This changes the definition of size from Agata-size to the default QuickCheck-size. The generator (*agataWith exponentialSize :: Gen [[Bool]]*) is essentially the same generator as (*arbitrary :: Gen [[Bool]]*).

2.2.2 Distribution strategies by worst case

Each distribution strategy defines a relation between a size bound and worst case absolute size of generated data. Both in the default QuickCheck notion of size and the default Agata notion, this relation can be defined using dimension. For QuickSize-size, the worst case is exponential to the dimension. For Agata-size, the worst case is linear to the dimension. As shown, the exponential growth of QuickCheck-size can be implemented as a distribution strategy. Quadratic and constant growth can be implemented as well:

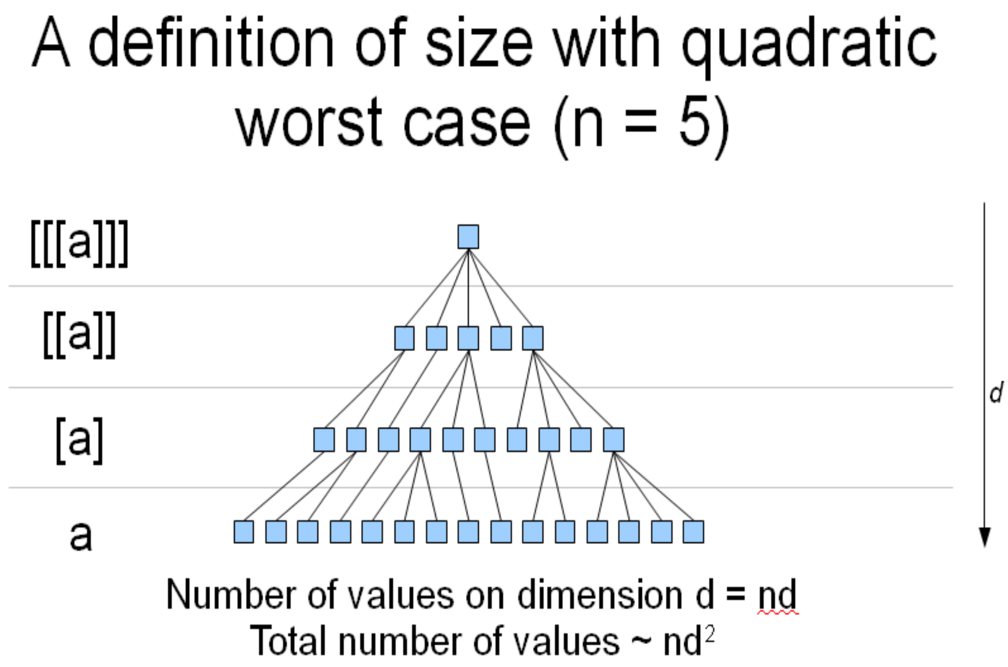
Definition 11 - quadratic and constant distribution strategy

```
quadraticSize,fixedSize :: Distributor

quadraticSize lev0 size = return $ get >>= x where
  x (lvl,r,[]) = case r of
    0 -> put (lvl,0,[])
    _ -> do
      k <- lift $ choose (0,size*d)
      ms <- lift $ piles r k
      put(lvl,0,ms)
    where d = (lev0 - lvl) + 1

fixedSize = listDistributor $ \l s -> case l of
  0 -> return []
  _ -> piles l s
```

The worst/average case of the default linear size strategy used by Agata will have many small or empty values in lower dimensions. The quadratic size strategy is more generous to these low dimension elements, without exponential growth of default QuickCheck generators. The constant size strategy can be implemented in many different ways. The solution above is very simple, and the distribution of generated values may not make sense for many types. This strategy will also produce very small values. The illustrations on page 13 clearly demonstrates the difference between exponential and linear strategies. Illustrated in this manner, the worst case of the linear strategy looks like a box, with equal number of elements on each dimension. The worst case of the exponential strategy looks like a funnel. The worst case of the quadratic strategy looks like a pyramid, with a constant number of elements added on each dimension:


A definition of size with quadratic worst case (n = 5)
[[[a]]]
[[a]]
[a]
a
d
Number of values on dimension d = nd
Total number of values ~ nd2

The "worst case" of the constant size strategy contains at most n recursive constructors, where n is the size bound. It may use all recursive constructors in the top-level list, or it may use some on lower dimensions instead.

When writing properties, testers can choose a strategy by using the *forAll* function from the QuickCheck library. In most cases there is no obvious advantage of using any particular strategy. This opens for the definition of an additional strategy; the random strategy. Consider this function:

Definition 12 - The random strategy

```
randomStrategy :: Distributor
randomStrategy l s = oneof $ map (\f -> f l s)
 [linearSize,exponentialSize,fixedSize,quadraticSize,pa
rtitions]
```

When generating values with this strategy, Agata will first randomly select a "real" strategy. Then it will use the chosen strategy for the actual generation of the value.

2.2.3 Partitions

This extension uses distribution control to change the way a recursive type is divided into finite sets by QuickCheck. For type t, the size-bounded set t_n of values allowed with size bound n is defined as all values v where $size(v) \leq n$. This means that $t_n \subseteq t_{n+1}$. If the alternative definition: $size(v) = n$ is used instead, then for $m \neq n$, $t_n \cap t_m = \varnothing$, i.e. all sets are disjoint. Since the union of all sets t_x will constitute the entire type t, each set is a partition of t. This has a number of beneficent effects:

- Fewer repeated tests: If 100 tests are performed, each on a unique size bound, then each generated value will be unique.

- Smaller size-bounded subset: Each subset t_{n+1} will be smaller since it does no longer contain the values in t_n. This means that the probability of generating any specific value in this set is greater.
- More even distribution: Typically, values of smaller size will have a larger chance of being generated. For instance for *[Bool]* there is a 1/6 probability of generating *[True]* from $[Bool]_3$, but only 1/24 chance of generating *[True,True,True]* from the same set. If *partitions* is used the distribution from $[Bool]_n$ is completely even, the length of the list will be exactly *n* and each specific value will have a $1/2^n$ chance of being generated.

This would be difficult to implement directly in QuickCheck, and still maintain compatibility. The requirement imposed on a generated value would be that somewhere inside the value a sub-value must be generated by using the maximal size allowed. This would require an overview of the generating process that can not be achieved using only the *Arbitrary* class. Fortunately, Agata has exactly this kind of overview in the wrapper function.

The *agata* function decides randomly how many recursions to use on each dimension, limited by the size bound. Since Agata is aware of the dimension of the generated type, it could determine the number of recursions on each dimension before generation is initiated. Consider these functions:

Example 28 - A more specific agata function

```
listDistributor :: (Int -> Int -> Gen [Int]) ->
Distributor

partitions :: Distributor
partitions = listDistributor $ \d s -> do
      xs <- sequence $ replicate (d-1) $ choose (0,s)
      permute (s:xs)
```

The function *listDistributor* will create a distributor which, instead of randomly determining the number of recursions on each dimension, uses numbers from a pre-generated list. If one of the elements in the list is always the size bound and all others are random values lesser than or equal to the size bound, each size bound will *almost* be a partition.

If the dimension of the generated type is 1, then this distribution strategy will divide the type correctly into partitions. This means that a standard test run on the type *[()]* will enumerate all values of this type up to length 100. It also means that the distribution in $[Bool]_n$ is completely uniform. For the type *[[()]]* however, if the size bound is 10 and the distribution-list *[0,10]* is used then *[]* is always generated. In general the element *[]* will intersect all "partitions" of multi-dimensional lists. This can be fixed by having only non-zero sizes in the prefix of the list that does not contain the size bound.

The type *[Maybe [()]]* is trickier to fix. Any repetitions of *Nothing* will intersect all partitions. Although there is sufficient information in the *Buildable* class to overcome this problem, it requires massive changes to the wrapper function.

2.2.4 Agata SmallCheck

As demonstrated, Agata can be adapted to mimic default QuickCheck generators. A very nice addition is a feature that mimics SmallCheck in a similar way. Since the overloaded function *build* can perform any predefined function on all fields of all constructors, it can be used to enumerate all values to a certain depth. The following function is from the module *Base*:

Definition 13 - agataSC

```
agataSC :: Buildable a => Int -> [a]
```

This function can be used as a generator for SmallCheck. To define a default SmallCheck generator for any type *T* the following code snippet can be used

Example 29 - A SsmallCheck generatot provided by Agata

```
data T = ...
instance Buildable t where
  ...
instance Serial T where
  series = agataSC
```

2.3 The shrinking algorithm

A shrinking function, as used by QuickCheck, for type *a* has type *a -> [a]*. The important property being that all elements in the resulting list are strictly smaller than the original value. Optimally (i.e. to find the smallest possible counterexample) the function should try and shrink the input in as many ways as possible. If the testing framework finds a counterexample to a property, it will apply the appropriate shrinking function to the input and test the property for each of the smaller inputs. If any of the smaller inputs causes a failure as well, it will be adopted as the new counterexample. The shrinking function is then reapplied to this value, and the process is repeated until there are no smaller inputs that cause the property to fail. The resulting minimal failing input is reported as a counter example. Consider this type of linked lists of integers and its shrinking function:

```
data IntList = Nil | Cons Int IntList

shrinkList Nil         = []
shrinkList (Cons x xs) = [ xs ]
                      ++ [ Cons x xs' | xs' <- shrinkList xs ]
                      ++ [ x':xs | x'  <- shrinkInt x ]
```

In this function, shrinking is done in three steps. First the function tries to remove the head of the list. If this does not yield a failing input the function tries to recursively shrink the tail of the list, possibly removing elements from it. Finally if nothing else works, it will maintain the structure of the list and instead try to shrink the first integer element. Since the function is applied iteratively to the input, it will combine these three steps in any fashion to reach a minimal input. Thus when the shrinking framework is finished, no element can be removed from the list and no element can be made any smaller. This algorithm could possibly be made faster by removing larger chunks of the list (e.g. half the list) at once, reducing the number of repeated shrinks needed. Since this code is only executed when a bug is present and detected, performance is a minor issue.

Shrinking is not part of the core functionality of Agata, i.e. there is no way to derive a shrinking function like this from a *Buildable* instance. It is however an extra feature of the code generating tool. The algorithm used by Agata derives a shrinking function for any algebraic type. Like the example with the integer list, three consecutive steps are required to shrink a value. The following section will describe each of them in detail.

Attempt to replace the value by a value occurring in a field This step tries to find a value of the same type as the original value somewhere inside it. For non-recursive types this step is omitted, since a value of type *a* can not be contained in a value of type *a* unless *a* is a recursive type. For self-recursive types this step is simply a matter of listing the values of all recursive fields. This implements the first step of the shrinking function for integer list: The *Nil* constructor has no recursive fields, so the resulting list is empty. The *Cons* constructor has one recursive field, the tail of the list, the value of this field is thus the only result of this part of the shrinking process. For a binary tree both the left and right branch fields would be recursive, and the list would contain both.

For mutually recursive types the process is slightly more complicated. In this case the process involves recursively gathering relevant values from all fields with types that may contain them. Consider the following mutually recursive types:

```
data IntTree = Leaf Int | Branch IntForest
data IntForest = Nil | Cons IntTree IntForest
```

The shrinking function for the *IntTree* type should return all *IntTree* values contained in the *IntForest* field of a branch constructor. Agata does this for general types in the following way: For each group of mutually recursive types, a type class is established. The type class contains a function that collects the values of all recursive fields in a register, and recursively collects values from these values as well. The function must also be aware of which type it is collecting because when it encounter a value of the sought type, it should only record it – not gather from its recursive fields. This is a performance issue. Any such values will eventually be tested by the iterating framework, including them in every iteration might cause an unacceptable performance cost (i.e. exponential time complexity).

Attempt to construct a value using any constructor with field values from a strict subset of the shrinked constructors field values This is the most tricky step in the shrinking process because careless implementation will lead to a combinatorial explosion. Specifically the problem lies in choosing which combinations of fields to choose if there are several fields of the same type. The current implementation ensures only that each possible field is included at least once in the new list of values.

This step is not needed in the *IntList* example but it will affect it, in each iteration of the shrinking process it will try to replace any *Cons* constructed values with *Nil* values. This is because `Nil` has no fields, which is a strict subset of `Cons` two fields. A more useful example is a language grammar like this:

Example 30 - A grammar for a computer langauge

```
data Stmt = SIfElse Exp Stmt Stmt
          | SIf Exp Stmt
          | SExp Exp
          | ...
```

This part of the shrinking algorithm would try to replace a if-then-else clause with an if-else, both by preserving the first statement and by preserving the second. It would also try to shrink the value by using the guard as a standalone expression.

If shrinking a ternary-or-binary-tree this part of the algorithm will try to replace a ternary branching with a binary one in two different ways: by using the first and second branches or the second and third branches.

Shrink any field If the shrinked value can not be replaced by a contained sub-value and the constructor used can not be swapped to a less "expensive" one, then the only remaining option is to shrink one of the constructor fields. This represents the second and third step of the `IntList` example and is pretty straightforward: Make a list of values equal to the original with exception that the value of the first field is altered in every possible way its shrinking function suggests. Concatenate this to the list of values where only the second field value is shrunk etc. Since the process will be iterated, only one field should be shrunk in each new value.

Chapter 3
Examples

"Example isn't another way to teach. It is the only way to teach."
- Albert Einstein

This section contains several examples as well as comparisons between Agata, the default QuickCheck generators and SmallCheck.

3.1 Scalability

When comparing scalability between the default QuickCheck generators and Agata, a good measurement is the worst case absolute size of a generated value given a type and a size bound.

To measure the scalability of SmallCheck, the size of the values generated are not that interesting, since these will always be very small (as per the name). The scalability concern of SmallCheck is instead the number of tests required in relation to depth of verification.

3.1.1 Abstract Syntax Trees

Motivation This example is tailored to benefit Agata. Agata will have a scalability advantage over the default QuickCheck generators for types with high dimension. Types in Abstract Syntax Trees (models of the grammar of a computer language) typically have high dimensions. For instance a file may contain several classes, each class several functions, each function several statements, each statement may contain several variable declarations and each declaration may include a function call with several arguments.

Setup The abstract syntax of a subset of a language with functions, Boolean values and variables can be defined as follows using standard Haskell data types:

Example 31 - A simple abstract syntax tree

```
type File      = ([Char], [Class])
type Class     = ([Char], [Function])
type Function  = ([Char], [Stmt])
type Stmt      = [((Type,Var),Exp)]
type Var       = [Char]
type Type      = [Char]
type Exp       = Either Bool (FName,[Either Var Bool])
  type FName     = ([Char],[Char])
```

The dimension of the type File is 6. A dummy property is used that calculates the length of all Strings and inspects all `Bool` values.

Outcome Using the default QuickCheck generator, GHCi crashed after around 30 tests. At the time of crash, GHCi was using 1.9 GB of RAM.

Using SmallCheck, the property could be verified to depth 4 in a few hours of testing. The number of test cases on depth 4 was 57424705. Given the exponential growth of the number of test cases, it is unlikely that the property could ever be verified to depth 5.

Using Agata, a standard 100-test test run is executed in 30 seconds. Using the quadratic distribution, the corresponding value is 42 seconds.

3.1.2 Results

Typically, the expected absolute size of values generated by QuickCheck grow exponentially with the dimension of the generated type. By default, Agata-generated values will grow linearly instead. There are examples of types where the default size settings are not feasible to use even for simple properties.

3.2 Coverage and Distribution

This section measures the various testing frameworks abilities to falsify a few properties.

3.2.1 Optimized Quick-sort

Motivation This example is tailored to benefit Agata. The primary advantage of Agata distribution-wise is the closer correspondence between Agata-size and absolute size, as compared to QuickCheck-size. If a property fails for input that has small absolute size but large QuickCheck-size, Agata will find it faster than the default generators.

Setup Consider this implementation of the Quick-sort algorithm with some optimizations:

Example 32 - A broken sorting function

```
prop_qsort xs  = sort xs == qsort xs

type Nat = [Bool]
qsort :: [Nat] -> [Nat]
qsort l
  | length l < 10  = sort l
  | otherwise     = qsort' l where
  qsort' (x:xs) = case (filter (x >) xs, filter (x <=)
xs) of
    ([],big)    -> [x] ++ qsort' big
    (small,[])  -> qsort' small ++ [x]
    (small,big) -> qsort small ++ [x] ++ qsort big
```

To avoid unfairness due to different definitions of the size of an integer value, binary sequences (coded as lists of `Bool`) are used instead. The optimizations in place are:

- If the list is shorter than 10, some simpler sorting function is used.
- If the pivot element is the smallest or the (uniquely) largest element, the size of the remaining list is not re-checked. Instead the `qsort'` function is used directly to sort the rest of the list.

The error is in the second optimization, since `qsort`' assumes that the list passed to it is non-empty. This property will fail for any list longer than 10 if the list is sorted, reverse sorted or if a prefix/suffix of the list is sorted and the rest of the list is reverse sorted. It will also fail for any list that is reduced to a list like this when `qsort`' is applied. Examples of failing lists:

```
[0,0,0,0,0,0,0,0,0,0]
[1,2,3,4,5,6,7,8,9,10]
[1,2,3,4,5,20,19,18,17,16]
[20,19,18,17,16,1,2,3,4,5]
[1,0,2,0,3,0,4,0,5,0,6,0,7,0,8,0,9,0,10,0,11]
```

In absolute size, the smallest failing input is `[[],[],[],[],[],[],[],[],[], []]`. This counter-example has 10 recursive constructors, and the Agata- and QuickCheck-sizes are both 10.

Outcome Using the default generators, QuickCheck failed to falsify the property in 100 consecutive test-runs with the default settings, each test run executes 100 tests.

SmallCheck verified the property to depth 7 in little over two hours on the test system. The number of test runs on depth 7 was $1.8*10^8$. Given the exponential growth of the number of test runs, it seems unlikely that SmallCheck would ever attempt to verify at depth 10, which would falsify the property.

Agata found a counterexample on the first test-run, on the 11:th test case. In 100 consecutive test-runs, Agata falsified the property every time.

3.2.2 Abstract Syntax Trees

Motivation This example is tailored to benefit Agata. As previously shown, Agata has a scalability advantage when generating types of higher dimensions. How does the dimension of test data impact the probability of finding errors?

Setup The type `FName` (see Example 31) is a pair of a class name and a function name. A pre-processing function for the type `File` replaces all empty class names with the class name used in the previous function call, if no prior function calls exist then the class of the next function call will be used instead.

The bug in the pre-processing function is that if all function calls have no class name, then the program crashes. The smallest failing input is thus:

```
("",[("",[("",[[(("",""),Right (("",""),[]))]])])])
```

The QuickCheck-/ Agata-sizes of this counterexample are both 1.

Outcome SmallCheck did not find this failure, even when left testing overnight. The reason is that it never verifies deep enough to construct a function call. In order to find this counterexample, SmallCheck would need to verify to depth 6.

Thanks to lazy evaluation, the default QuickCheck generators could be used without crashing. A counterexample was not found on the first test-run. In 100 consecutive test-runs, the property was falsified 13 times.

Agata found a counterexample on the first test run. In 100 consecutive test-runs, Agata falsified the property every time.

3.2.3 Results

Some properties fail only when all members of a data structure share a certain property, e.g. all empty/non-empty, all equal/non-equal. QuickCheck might fail to find such examples under any of these circumstances:

- The fault only presents itself if a data structure is large enough.
- The dimension of the generated type is too high.

3.3 Parser testing

Parser testing is a domain where the benefits of Agata are well utilized, and the drawbacks are mitigated. For instance:

- There invariants of language grammar types can typically be expressed in the Haskell type system. Complex invariants of languages such as type correctness are ignored when testing parsers.
- The dimensions of language grammar types are often high, and Agata addresses the size and distribution problems relating to this.
- Often large groups of mutually recursive types, such as various forms of expressions, are present. This would make hand writing QuickCheck generators rather difficult and extremely tedious.
- Parsers are often generated automatically from a detailed specification by a parser generator. These specification can be used to construct Agata generators automatically.

The BNF Converter (BNFC) is a parser generator that uses a LBNF-specification of a context free language to automatically constructs a lexer, a parser, a pretty-printer and a few other useful tools for the specified language. BNFC is written in Haskell and supports several back-end languages including Haskell. As a part of the Agata project, BNFC is extended to generate a testing module as well. This module enables the tester to write properties of Strings and have them tested for a random subset of the language, with Agata as the test data generator. The testing is performed with the QuickCheck testing framework, and counterexamples are shrinked before they are presented.

3.3.1 Properties of languages

A number of interesting general properties can be tested for language parsers, including:

- A specification can be tested as specifying a subset of the language parsed by an existing parser implementation.
- Two specifications can be tested as specifying the same language (a generally undecidable problem).
- Two parser implementations can be compared as parsing the same subset of a specified language, i.e. what one can parse, the other can too.
- A pretty-printer/parser combination can be tested to preserve the abstract syntax in a parse-print cycle.

3.3.2 C parsers

The largest example of a language specified in the LBNF formalism is a grammar of ANSI C. This grammar was written by Ulf Persson as part of BSc thesis. The example is mentioned in the BNFC technical report.

Since C is a widely used language, its easy enough to test that the specification parses a superset of C, just test it on a few (or lots of) existing C programs. In order to call this a real specification of C, the reverse needs to be tested as well: the specified language must be a Subset of C. Since Agata can generate random strings in the specified language, these can be tested against a trusted C parser implementation (or possibly several not-so-trusted ones). If any string is not parsable, then there is an error either in the specification or in the parser implementation.

Testing against Language.C Since Agata only supports the Haskell back-end of BNFC, the easiest property to test is that all all strings in the specified language should be parsable by Language.C.Parser. This yields the following setup:

Example 33 - testing Language.C.Patser

```
langC = CheckC.parseProgram
parses :: String -> Either String CtranslUnit
parses = ...
main = quickCheck $ subset langC parses
```

`CheckC` is the testing module generated by Agata, `parseProgram` is a combined parser, pretty-printer, generator and shrinking function for the non-terminal `Program`. The `subset` function a predefined property that takes a BNFC-language and a parser function and returns a property that tests if the parser function can parse all strings in the language.

The result of running main is:

Example 34 - A counterexample

```
*** Failed! Exception: 'static ;' (after 1 test and 3
shrinks):
ConcProgramProgr (ListConcExternal_declarationSingle
(ConcExternal_declarationGlobal (ConcDecNoDeclarator
(ListConcDeclaration_specifierSingle
(ConcDeclaration_specifierStorage
ConcStorage_class_specifierLocalProgram)))))
```

The reported failing program is “static ;”. This program is indeed parsed by the BNFC-generated parser, but not by Language.C.Parser. The rest of the output is a representation of which BNF rules where used to construct the values. There is obviously a bug in either the grammar or the library. Running the program through the GNU C Compiler does not raise any parse error, just a warning about a useless storage class specifier. The error would thus appear to be in the Haskell C library.

Testing language subsets Language grammars are inherently very modular, and different parts of the specification can be tested independently. Specifically, C-statements and C-expressions are essentially languages of their own, and Agata creates generators for these as well. The following setup is used:

Example 35 - Testing statements and expressions

```
langCStm = CheckC.parseStm
langCExp = CheckC.parseExp

cExpEnclosed s = cStmEnclosed $ "a = "++s++";"
cStmEnclosed s = "A {"++s++"}"

test_Stm = quickCheck $ subset langCStm (parses .
cStmEnclosed)
test_Exp = quickCheck $ subset langCExp (parses .
CexpEnclosed)
```

As could be expected, `test_Stm` fails in the same way as testing complete programs, with a storage class specifier. The function test_Exp produces a very interesting counterexample:

Example 36 - A failing expression

```
*Main> test_Exp
*** Failed! Exception: '( 06470063754347u ..66'
Exception: '08' (after 83 tests and 7 shrinks):
...
```

Apparently, Language.C.Parser rejects the expression 08. This is an error in the LBNF specification. In C, a number with a leading 0 is always a octal numeral. Thus any number with a leading zero containing an eight or a nine should not parse.

Testing against ANTLR Another parser generator is ANTLR. Like BNFC it supports a multitude of target languages (not Haskell though), and like BNFC it has a grammar of ANSI C on its web page. The parser produced by ANTLR can be compared directly with the BNFC grammar, as was done with the Language.C parser, and it can be tested against the Language.C parser using the LBNF grammar only to generate test data.

The target language for the ANTLR parser is Java. Properties for testing programs, statements, and expressions are written. A property that compares parsing results for Language.C.Parser and the ANTLR parser is also written.

Testing against the BNFC parser yields no counterexample in the first run. This indicates that the ANTLR grammar is identical to the BNFC one. Re-running the tests yields a difference in the expression syntax:

Example 37 - A failing expression

```
*** Failed! Exception: '! ( "P" | 03714222..\"'
Exception: ''''' (after 30 tests and 2 shrinks):
...
```

This is another error in the BNFC grammar, where it parses the string “''” as a valid expression. Testing against the Language.C parser yields this error:

Example 38 - A failing program

```
*** Failed! Exception: 'volatile ;' (after 1 test):
```

Once again the storage class specifier causes an error. Apparently the ANTLR grammar agrees with GCC and BNFC, that using the empty declaration is allowed in this case.

Testing parse-print cycles Agata is not limited to testing parsability. An example of a more advanced property is: for every generated string, parsing it with Language.C.Parser and printing the result with Language.C.Pretty should yield a new string that will parse to the same abstract syntax as the original string, when parsing with the BNFC parser.

This property yielded an error, namely that the expression “*`0ul`*” will parse-print to the string “*`0uL`*” which is not accepted by BNFC. The regular expression in the BNFC grammar will accept “*`0UL`*” or “*`0ul`*” but not “*`0Ul`*” or “*`0uL`*”. This is an error in the BNFC grammar.

Unfortunately, the usefulness of this method is limited because the “abstract” syntax of regular expressions in BNFC are just strings. This means that normal equivalence can not be expected from the abstract syntax after a parse-print cycle.

3.3.3 Results

Testing a few simple properties immediately yielded several errors both in the Haskell library and in the grammar.

Testing that a grammar specifies all of a language is often easy, provided there is a sufficiently large code base to test the generated parser on. Testing that the grammar does not specify a to large language is harder. One way of explaining this is that the domain of incorrect programs is very large. Look at a single correct program, how many incorrect programs can be made by changing a single character in the text? Clearly it would be very difficult to write a library of test cases and claim that it gives any coverage of the total set of incorrect programs.

Chapter 4
Future work

Invariants Arguably, the most important shortcoming of Agata is the lack of invariants. Since values on lower dimensions can not be inspected when generating a value, there is an inherent collision between the size control of Agata and specifying invariants in generators.

A possible alternative is to specify invariants as post-processing functions, i.e. functionz in `a → a` or `a → Gen a`. These functions should be injective into the set of values that respect the invariant.

Type specific distribution control Currently, the precision of distribution control is limited to changing all types on a particular dimension. This could possibly be sharpened to allow special treatment of specific types. There is probably no completely type-safe way of doing this, but perhaps further study will reveal some trick to solve this.

Tester-Tester library This thesis can only show the existence of properties that are falsified by Agata but not by QuickCheck or SmallCheck. For an estimation of the extent of this class of properties, and for finding classes of properties where Agata is lacking, a library of falsifiable properties could be useful. Preferably the properties should have a limited set of common input-types, so every generator can be tested for as many problems as possible. This would demonstrate how a generator tailored to falsify one property may be inadequate for testing others.

A new testing framework The QuickCheck testing framework could be rewritten to support more direct testing with Agata types, rather than using the `forAll` function or defining an `Arbitrary` instance. This is especially compelling since Agata incorporates a SmallCheck style enumeration strategy. The new framework can start by enumerating a predefined number of small values, and then continue with randomly testing a few larger ones. Preferably the random value generator should only generate values that are not in the initial enumeration.

Automatic shrinking The shrinking algorithm described in this thesis can be used to automatically construct shrinking functions for algebraic types. Since the `build` and `improve` functions provide most of the information needed to shrink a value (e.g. a pattern matching function which recursively shrink field values) a function like this would be nice:

Example 39 - An easier way to shrink

```
agataShrink :: Buildable a => a -> [a]
```

The rebuild function might need some help with identifying recursive fields (e.g. use *rbRec* instead of *rb* for those fields) and even this might not be enough for the complete shrinking algorithm. Mutual recursion will probably cause problems.

Function types Writing a Buildable instance for function types should be possible. Functions could be generated using the *coarbitrary* function, as done by default in QuickCheck.

Partitions The idea of changing the type division strategy from dividing a recursive type into growing subsets to dividing it into partitions is appealing. Further research might reveal more about the pros and cons of this division paradigm. The partitioning wrapper function in Agata is limited. If this strategy is deemed useful, the partitioning wrapper function for Agata could be improved to successfully partition any type.

Lazy SmallCheck Many properties will only evaluate a small part of their input. Compared to regular SmallCheck, Lazy SmallCheck excels at testing this type of properties, drastically reducing the number of tests on each depth-level. When testing a language grammar for instance, SmallCheck would generate millions of test cases where the only difference is the names of functions. Lazy SmallCheck would conclude in a few tests that the names of functions are not relevant and move on to a more interesting test. Unfortunately the enumeration generators for Lazy SmallCheck are a different from the generators for regular SmallCheck. If possible Agata should support this type of enumerations as well.

Using language extensions Many of the features in Agata could be made simpler or more effective by using language-extensions to Haskell. For instance it might be possible to define default functions for build and improve using Generic classes. In most cases the user would only need to override the *dimension* function, in some cases maybe not even that.

Chapter 5
Conclusions

Agata is a new strategy for writing QuickCheck generators. It is also a tool that automatically constructs generators for types in a Haskell module, and an extension to BNFC for testing parsers. Agata is best suited for types and properties without type invariants, other than those enforced by the Haskell type system.

Outsourcing logic Agata reduces the implementation of a generator to a mechanical description of the type, moving all logic into a wrapper function.

The primary advantage of the powerful monadic default interface for defining QuickCheck generators is the ingenuity it allows when writing them. When the process of writing generators is automated, no such ingenuity can be utilized. Maintaining a flexible interface for writing generators in this case is wasteful, because a more narrow interface gives better reflective capabilities at no additional cost.

Agata sacrifices the ability to encode type invariants into generators, in favor of easing automation and increasing the flexibility of use. Most new features of Agata stem from this outsourcing of generator logic.

Size does matter Agata offers a precise control of the expected absolute size of generated data. This ensures scalability even when generating complex types.

When generating collection-type data-structures, the default QuickCheck generator will tend to generate small structures with small elements and large structures with large elements, but rarely large structures with small elements. This "blind-spot" reduces test-coverage of these types.

When generating multi-dimensional collection type data-structures, the absolute size of QuickCheck test cases will typically grow exponential to the dimension of the generated type. The corresponding growth for a default Agata generators is linear. Experimental results demonstrate the improved scalability of Agata compared to default QuickCheck generators in these cases.

By default, QuickCheck generates all values independently[1]. Agata introduces a dependency between values of the same type. If the size of a data structure is atypically large, then the average size of each element in the structure will be smaller. This will ensure scalability, but it will also guarantee that corner cases such as "all elements are

1 The values are independent in the sense that the actual size of one element does not influence the actual size of another. The values are also dependent in the sense that they all have the same upper size bound, and thus the same expected value.

size zero” or “one element is large, the rest are small” have a probability of generating even if the size of the data structure is large.

Distribution control Agata can be used to control many features of a generator, without altering its source code. This is a modularity advantage compared to the default strategy for writing generators. The mechanism is powerful enough to mimic the size-paradigm of the default QuickCheck generators. A library of generator strategies is defined, and any one of these strategies (or a custom made strategy) can be applied to a generator to imbue it with specific properties.

When randomly testing a property, the test data generator will determine the distribution of values. A generator may favor corner cases or mainstream cases, large cases or small cases etc. Each distribution has its own advantages and disadvantages, and testing a property with several strategies will give better coverage than testing with only one. When using Agata, testers do not only have a new distribution strategy, they have many new distribution strategies!

Test outcomes Experimental results show that some properties are falsified by Agata, but not by default QuickCheck testing nor by SmallCheck testing. The extent of this class of errors and its occurrence in real applications is yet unknown.

The Agata extension to BNFC can be used to generate random sentences in any context free language. This is ideal for testing a parser against a grammar or vice versa. This application is especially well suited for Agata; BNFC is already used to generate code, so no extra manual work is needed to generate the testing module. Experimental results demonstrate the usefulness of this tool, Agata discovered several previously unknown bugs in published software.

Chapter 6
Bibliography

Bibliography

1: K. Claessen, J. Hughes, *QuickCheck: A Leightweight Tool for Random Testing of Haskell Programs* , ICFP '00, 2000
2: C. Runciman, M. Naylor, F. Lindblad, *SmallCheck and Lazy SmallCheck - automatic exhaustive testing for small values*, Haskell Symposium '08, 2008
3: A. Kolawa, D. Huizinga,*Automated Defect Prevention: Best Practices in Software Management*, Wiley-IEEE Computer Society Press, 2007
4: Lee J. White, *Software testing and verification*, excerpt from *Encyclopedia of Computer Science and Technology*, 1995
5: G. Fink, M. Bishop, *Property-Based Testing; A New Approachto Testing for Assurance*, ACM SIGSOFT Software Engineering Notes, 22(4), 1997
6: P Godefroid, N. Klarlund, K. Sen, *DART: directed automated random testing*, ACM SIGPLAN Proceedings of the Conference on Programming Language Design and Implementation, 2005