FortressCheck: Automatic Testing for Generic Properties

Seonghoon Kang*
KAIST
kang.seonghoon@mearie.org

Sukyoung Ryu*
KAIST
sryu@cs.kaist.ac.kr

ABSTRACT

QuickCheck is a random testing library designed for the purely functional programming language Haskell. Its main features include a descriptive yet embedded domain-specific testing language, a variety of test generators including a generator for functions, and a set of operations for monitoring generated inputs. QuickCheck is limited to ad-hoc testing, compared to more systematic testing methods such as full coverage testing. However, experiences showed that well-factored functions and properties make the QuickCheck approach as effective as systematic testing while maintaining its conciseness. QuickCheck and its variants are now available in dozens of programming languages.

We present a version of QuickCheck for the Fortress programming language in this paper. Fortress is an object-oriented language with extensive support for functional programming, with the strong emphasis on high-performance computing, parallelism by default, and growability of the language. While the main features of QuickCheck are straightforward to implement, we are extending them to support unique features of Fortress and to support seamless integration to Fortress. We observed that the prevalent uses of implicit parallelism in Fortress call for testing parallel language constructs especially those using side effects. Also, because Fortress provides both subtype polymorphism and parametric polymorphism unlike Haskell, testing both polymorphic properties becomes interesting. We propose FortressCheck to test implicit parallelism and to test parametric polymorphism via reflection, by generating first-class type objects and using QuickCheck's own implication checking as a safety mechanism.

Categories and Subject Descriptors

D.3 [**Programming Languages**]: Language Constructs and Features

* Supported in part by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST) / National Research Foundation of Korea(NRF) (Grant 2010-0001723)."

General Terms

Languages

Keywords

Fortress, automatic testing, QuickCheck

1. INTRODUCTION

Fortress [4] is a new programming language developed for quality-critical, high-performance computing, which provides extensive supports for functional, object-oriented, and parallel features. To support high-performance computing in the current multicore world, Fortress provides various levels of implicit parallelism to take advantage of the inherent parallelism underneath the multicore computers. To help scientists and engineers develop quality-critical softwares, Fortress provides both a static type system to check static properties and built-in language supports such as contracts, tests, and properties to check dynamic properties as a way of machine-checkable specifications. The Fortress language specification version 1.0 β [3] describes what properties such language features describe but it does not describe how to check or test the properties. An ideal way might be to provide a language support for verifying correctness of program properties, but it is a time-consuming and difficult task. As a stepping stone, we have developed an intensive testing tool that is both practical and easy to use, inspired by the QuickCheck [8] library in Haskell.

QuickCheck provides random and ad-hoc testing. By *random*, we mean that QuickCheck generates random test cases and uses them to find a counter-example of a given assertion, if any. Users should provide both a machine-readable specification of an assertion and an appropriate test generator for testing the assertion. QuickCheck takes full advantage of Haskell to simplify these tasks: (1) to provide machine-readable specifications, QuickCheck provides an embedded domain-specific language that adapts to Haskell, and (2) to provide appropriate test generators for given assertions, QuickCheck uses the strong type system of Haskell to determine a test generator from the type of the specification.

By *ad-hoc*, we mean that QuickCheck is not a systematic testing tool. Unlike ad-hoc testing tools, systematic testing tools provide a guarantee to find a counter-example if it exists. Such a guarantee is provided with "a test adequacy criterion"; a simple example criterion is that every single branch has to be reached during a test. While systematic testing is more powerful than ad-hoc testing in general, the authors of QuickCheck argued that ad-hoc tests at a finer granularity could provide a test coverage comparable to systematic tests; combined with a difficulty of adapting such heavyweight methods to a highly functional language, Haskell, their choice of ad-hoc testing has been shown reasonable. In fact, some of the trickiest bugs the authors have found have required test cases that exercise bits of the code several times–they are bugs that QuickCheck can find, but systematic approaches would not [12].

Since the initial release in 2000, QuickCheck has gained much interests from the various language communities, and has been ported to dozens of other languages; a partial list of them is available [7, 2]. While most of them capture the key ideas of QuickCheck, many of them reflect a feature set and characteristics of the target language, not Haskell. For example, most languages with dynamic type systems cannot infer the most adequate generator from the type of a property, and object-oriented languages have different notions for the adequacy of generators.

We have designed and developed *FortressCheck*, a version of QuickCheck in Fortress, which supports testing of unique features in Fortress such as subtype polymorphism and parametric polymorphism. Unlike other QuickCheck ports, FortressCheck introduces a new idea of testing generic properties using reflection. Our implementation of FortressCheck also gives us an opportunity to evaluate the expressiveness of Fortress.

This paper consists of three parts. In the first part, we discuss the issues that we have encountered and tried to solve via FortressCheck. We introduce FortressCheck in the second part and compare other ports and extensions of QuickCheck with Fortress in the third part.

2. MOTIVATION AND BACKGROUND

In this section, we present Fortress language features which call for automated testing, a QuickCheck-like solution in Fortress: side effects inside parallel evaluation and generic properties.

2.1 Side Effects inside Parallel Evaluation

Fortress programs using two seemingly conflicting features, implicit parallelism and side effects, often result in unexpected results. While most language constructs in Fortress are purely functional, Fortress provides also a set of imperative features which makes the results of parallel programs surprising to the programmers.

For example, while developing a prototype of FortressCheck, we found an unexpected result from a parallelly-evaluated expression:

```
generateℤ64(g: AnyRandomGen): ℤ64 =
    (widen(generateℤ32(g)) LSHIFT 32) ∨∨
    widen(generateℤ32(g))
```

The instance generator function, $generate\mathbb{Z}64$, generates one random 64-bit integer from two random 32-bit integers generated by a given random generator g. In Fortress, function arguments may be evaluated in parallel, thus, two operands of the BITOR operator ⩔ may be evaluated in any order. Because a random generator is inherently stateful, repeated evaluations of $generate\mathbb{Z}64$ may not produce the same random number, which breaks our need for reproducibility of the test data. A correct version, which makes the ordering of the random number generation explicitly sequential, is as follows:

```
generateℤ64(g: AnyRandomGen): ℤ64 = do
    hi = widen(generateℤ32(g))
    lo = widen(generateℤ32(g))
    (hi LSHIFT 32) ∨∨ lo
  end
```

As another example, the following code shows a bug we found in our own Fortress String library implementation, which involves a parallel evaluation of a `for` loop:

```
object SubString(basestring: String, range: Range)
  extends String
  . . .
  writeOn(stream: WriteStream): () =
    . . .
    for (start, str) ← basestring.splitWithOffsets() do
      subrange = (start # |str|) ∩ range
      substr = str.uncheckedSubstring(subrange)
      substr.writeOn stream
    end
  . . .
end
```

In Fortress, a `for` loop is a special kind of *generators*, which is not necessarily sequential. The loop body may be evaluated in parallel unless explicitly stated as sequential by the call of *seq* or *sequential*. If a programmer omits *seq* or *sequential* by accident, however, the execution order of the generator becomes nondeterministic. In this case, the order of writing each piece is nondeterministic and the entire string appears mangled.

This experience shows that implicit parallelism combined with side effects can be one of the major sources of bugs invisible to the programmers and a tool to detect such bugs would be greatly helpful. The first bug was actually caught by FortressCheck itself during an unrelated test, presenting an ability of FortressCheck to find such bugs.

Note that FortressCheck does not provide any specific mechanisms to test implicit parallelism; instead it highly depends upon the nondeterministic behavior of such bugs, which often invalidates expected results. While it is possible for a particular Fortress implementation to fix the execution order of parallel constructs to hide some bugs, FortressCheck ignores this issue to simplify the design.

2.2 Generic Properties

There are two kinds of polymorphisms in Fortress: *subtype polymorphism* and *parametric polymorphism*. Specifying and testing both kinds of polymorphic properties impose various challenges in FortressCheck.

In Fortress, the subtype hierarchy allows programmers to express subtype-polymorphic properties. For example, if we add a property *commutativeAddition* to a trait Number, every subtype of Number should satisfy the property:

```
trait Number
    opr +(self, other: Number): Number
    opr =(self, other: Number): Boolean
    . . .
    property commutativeAddition =
        ∀(x: Number, y: Number) (x + y = y + x)
  end
```

$$\texttt{property } \mathit{doubleReversal}[\![T]\!] = \\ \quad \forall\big(g\colon \text{Generator}[\![T]\!]\big)\ (\mathit{list}\ g.\mathit{reverse}.\mathit{reverse} = \mathit{list}\ g)$$

$$\texttt{property } \mathit{mapSizeInvariant}_1[\![\text{Key},\text{Val}]\!] = \\ \quad \forall\big(\mathit{map}\colon \text{Map}[\![\text{Key},\text{Val}]\!], k\colon \text{Key}, v\colon \text{Val}\big) \\ \qquad \big(0 \le \big|\mathit{map}.\mathit{add}(k,v)\big| - |\mathit{map}| \le 1\big)$$

$$\texttt{property } \mathit{mapSizeInvariant}_2[\![\text{Key},\text{Val},\text{Res}]\!] = \\ \quad \forall\big(\mathit{map}\colon \text{Map}[\![\text{Key},\text{Val}]\!], f\colon (\text{Key},\text{Val}) \to \text{Maybe}[\![\text{Res}]\!]\big) \\ \qquad \big(\big|\mathit{map}.\mathit{mapFilter}[\![\text{Res}]\!](f)\big| \le |\mathit{map}|\big)$$

Figure 1: Properties parameterized over types

The property *commutativeAddition* takes two value parameters x and y and specifies that addition of them are commutative. There are two ways to test this property: an obvious way is to have a single generator for Number, but it means that we have to update the generator whenever we add or remove a subtype in the Number hierarchy, which is generally not possible. Instead, we may automatically create a generator for Number from the current type hierarchy, provided that we can build and inspect types at run time.

Using generic types, Fortress programmers can express properties parameterized over types. Consider that we want to express three invariants on containers of any types as shown in Figure 1: reversing a container twice produces an identical container (*doubleReversal*), adding a key-value pair to a map increases the size of the map by at most one ($\mathit{mapSizeInvariant}_1$), and applying *mapFilter* on a map does not increase the size of the map ($\mathit{mapSizeInvariant}_2$). Since the containers are generic to the types of its elements, the corresponding properties are also generic to the types, and they can be described as the following (hypothetical) code: The property *doubleReversal* takes a type parameter T where white square brackets delimit the declaration of the type parameter.

Unlike the *commutativeAddition* property in the first example, the properties in the second example are parameterized over types, which are not provided by the current Fortress language. As a workaround, the *doubleReversal* property could be rewritten to a non-generic property by adding a non-generic AnyGenerator trait as a supertype of Generator$[\![T]\!]$:

$$\texttt{trait } \text{AnyGenerator} \\ \quad \texttt{getter } \mathit{reverse}()\colon \text{AnyGenerator} \\ \texttt{end}$$

$$\texttt{trait } \text{Generator}[\![T]\!]\ \texttt{extends}\ \text{AnyGenerator} \\ \quad \texttt{getter } \mathit{reverse}()\colon \text{Generator}[\![T]\!] \\ \quad \ldots \\ \texttt{end}$$

$$\texttt{property } \mathit{doubleReversal} = \\ \quad \forall(g\colon \text{AnyGenerator})\ (\mathit{list}\ g.\mathit{reverse}.\mathit{reverse} = \mathit{list}\ g)$$

However, this workaround cannot be applied to the other properties, because the other properties include some parameters whose types include the type parameters. For example, the two parameters of $\mathit{mapSizeInvariant}_1$, k and v, have types Key and Val, respectively. As another workaround, $\mathit{mapSizeInvariant}_1$ could be moved into the corresponding trait parameterized by the type parameters Key and Val:

$$\texttt{trait } \text{Map}[\![\text{Key},\text{Val}]\!]\ \texttt{extends}\ \text{Generator}\big[\!\big[(\text{Key},\text{Val})\big]\!\big] \\ \quad \texttt{property } \mathit{mapSizeInvariant}_1 = \\ \qquad \forall\big(\mathit{map}\colon \text{Map}[\![\text{Key},\text{Val}]\!], k\colon \text{Key}, v\colon \text{Val}\big) \\ \qquad\quad \big(0 \le \big|\mathit{map}.\mathit{add}(k,v)\big| - |\mathit{map}| \le 1\big) \\ \quad \ldots \\ \texttt{end}$$

Because Key and Val are now the type parameters of the enclosing trait of the property, the property does not need to be parameterized over types.

However, this workaround does not work well for the third property, with $\mathit{mapSizeInvariant}_2$:

$$\texttt{trait } \text{Map}'[\![\text{Key},\text{Val},\text{Res}]\!]\ \texttt{extends}\ \text{Map}[\![\text{Key},\text{Val}]\!] \\ \quad \texttt{property } \mathit{mapSizeInvariant}_2 = \\ \qquad \forall\big(\mathit{map}\colon \text{Map}[\![\text{Key},\text{Val}]\!], f\colon (\text{Key},\text{Val}) \to \text{Maybe}[\![\text{Res}]\!]\big) \\ \qquad\quad \big(\big|\mathit{map}.\mathit{mapFilter}[\![\text{Res}]\!](f)\big| \le |\mathit{map}|\big) \\ \texttt{end}$$

Adding a trait to include the extraneous type parameter Res produces an extraneous type in the type hierarchy.

Therefore, we propose to use reflection for testing generic properties. Because generic properties may have arbitrary type parameters, we should be able to test any type parameters in addition to any value parameters. We describe how FortressCheck tests generic properties in the next section.

3. FORTRESSCHECK

To address the issues we discussed in Section 2, we have implemented FortressCheck, a version of QuickCheck for Fortress. At the moment, it runs only on the Fortress interpreter because the Fortress compiler is not yet fully developed. A notable characteristic of FortressCheck is that it heavily uses reflection, or a run-time type inspection.

3.1 Gen$[\![T]\!]$ Trait

A test instance generator Gen$[\![T]\!]$ provides three methods:

$$\texttt{trait } \text{Gen}[\![T]\!] \\ \quad \mathit{generate}(c\colon \text{TestContext})\colon T \\ \quad \mathit{perturb}(\mathit{obj}: T, g\colon \text{AnySeededRandomGen}): \\ \qquad\qquad \text{AnySeededRandomGen} \\ \quad \mathit{shrink}(\mathit{obj}: T)\colon \text{Generator}[\![T]\!] \\ \texttt{end}$$

The *generate* method generates a test instance of type T from a random generator included in a given TestContext. The TestContext type contains a random number generator and various utility functions. To support test generation of functions in a similar way to the `Coarbitrary` type class in QuickCheck, the *perturb* method returns a random number generator which depends only on its two parameters. The *shrink* method returns similar instances but smaller than a given test instance and it is used to "shrink" failing instances. For example, the following generator:

$$\texttt{object } \mathit{genBoolean}\ \texttt{extends}\ \text{Gen}[\![\text{Boolean}]\!] \\ \quad \mathit{generate}(c\colon \text{TestContext})\colon \text{Boolean} = \\ \qquad c.\mathit{oneOf}[\![\text{Boolean}]\!]\big(\langle\, \mathit{false}, \mathit{true}\,\rangle\big) \\ \quad \mathit{perturb}(\mathit{obj}\colon \text{Boolean}, g\colon \text{AnySeededRandomGen}) = \\ \qquad \texttt{if } \mathit{obj}\ \texttt{then}\ g.\mathit{perturbed}(1)\ \texttt{else}\ g.\mathit{perturbed}(2)\ \texttt{end} \\ \quad \mathit{shrink}(\mathit{obj}: \text{Boolean}) = \text{Nothing}[\![\text{Boolean}]\!] \\ \texttt{end}$$

shows an instance of Gen⟦T⟧, which generates test instances of type Boolean. Because Boolean values can have only one of two values, *true* and *false*, the *shrink* method does not generate any smaller instances.

Since most of the collection libraries in Fortress are subtypes of the Generator trait, a single test instance generator, GenGenerator, serves for most collection libraries:

```
trait GenGenerator⟦E, T extends Generator⟦E⟧⟧
      extends Gen⟦T⟧
  genE: Gen⟦E⟧
  abstract fromGenerator(obj: Generator⟦E⟧): T
  generate(c: TestContext): Generator⟦E⟧ =
      fromGenerator RandomGenerator⟦E⟧(self.genE, c)
  shrink(obj : T): Generator⟦T⟧ =
      ShrinkingGenerator⟦E, T⟧(obj, self.genE,
                                   fromGenerator)
end
```

and the job for writing a test instance generator for each collection library requires only modest cost. For example, a test generator for String, which is a generator for Char, is implemented as follows:

```
object genString extends GenGenerator⟦Char, String⟧
  genE: Gen⟦Char⟧ = genChar
  fromGenerator(obj: Generator⟦Char⟧): String = || obj
  perturb(obj: String, g: AnySeededRandomGen) =
      g.perturbed(|obj|).perturbed(obj.indices.
          map⟦ℤ32⟧(fn i ⇒ obj_i.codePoint))
  end
```

The *perturb* method reduces the number of *perturb* calls, which can be expensive for some random number generators.

3.2 Choosing Test Generator

The Gen⟦T⟧ trait is analogous to the `Arbitrary` type class in the original QuickCheck, but, unlike `Arbitrary`, defining Gen⟦T⟧ does not immediately make its test functions available. While QuickCheck chooses the most appropriate instance of a given type from its current scope, FortressCheck defines the Arbitrary trait which knows how to choose the best Gen⟦T⟧ instance from a given type T:

```
trait Arbitrary
    gen⟦T⟧(): Gen⟦T⟧
end
```

FortressCheck also provides the DefaultArbitrary trait and the *defaultArbitrary* instance to map from types to their default generators. Programmers can add more generators by extending DefaultArbitrary:

```
object myArbitrary extends DefaultArbitrary
  gen⟦T⟧(): Gen⟦T⟧ = do
      f = fn (_ : T) : T ⇒ throw FobiddenException
      typecase f of
          MyType → MyType ⇒ genMyType
          else ⇒ (self asif DefaultArbitrary).gen⟦T⟧()
      end
  end
end
```

Note that while the `typecase` expression in Fortress selects the first clause that its type is a supertype of the type of a given expression, the body of the *gen*⟦T⟧ method uses a function expression to get an exact match for a given type T. The FortressCheck library uses an exact matching because Gen⟦T⟧ is not covariant: $T <: U$ does not imply that Gen⟦T⟧ is usable in place of Gen⟦U⟧.

Supporting subtyping instead of exact matching in the *gen*⟦T⟧ method may have the following issues (We write $U \backslash T$ for types that are subtypes of U but not of T.):

1. An automatic lifting from Gen⟦T⟧ to Gen⟦U⟧ may unintentionally omit generation of values of type U\T, if any. Therefore, any lifting of Gen⟦T⟧ should be explicit.

2. The *shrink* method in Gen⟦T⟧ cannot handle values of type $U \backslash T$, so it is not even type-compatible.

3. Supporting subtyping requires both covariant and contravariant matchings because of arrow types, which requires significant duplication of code.

Note that QuickCheck does not have this problem because Haskell does not provide subtype polymorphism. While exact matching implies that we cannot easily make test generators for open types, we discuss how we alleviate this restriction in later sections.

3.3 Property Specification

As in QuickCheck, FortressCheck uses an embedded domain-specific language to specify properties. Every property is represented as an instance of the Testable⟦T⟧ trait, which is paired with a Gen⟦T⟧ test generator from the Arbitrary trait in order to perform actual testing:

```
trait Testable⟦T⟧
    run(arg : T): TestResult
end
```

The FortressCheck library also provides a number of operations that generate Testable⟦T⟧ instances. It also allows filtering test data by a certain condition (“tagging”), categorizing test data by a given criterion (“classifying”) and collecting test data for the later inspection (“collecting”) as supported by QuickCheck. Some example operations are shown in Figure 2.

Given a Testable⟦T⟧ instance, a *checkResult* function actually performs testing:

```
checkResult⟦T⟧(t: Testable⟦T⟧, g: Gen⟦T⟧, c: TestContext) :
    TestResult
```

The *checkResult* method repeatedly runs the given property t with the arguments generated by the test generator g and the test context c, and returns its result as another test result. Because default generators and contexts work reasonably for most cases, the FortressCheck library provides various wrapper functions, all of which named *check*, for convenience. Therefore, the actual testing is as simple as follows:

```
test runTests(): () =
    p = forAll (fn (a: ℤ32, b: ℤ32) ⇒ (a + b = b + a))
    check(p)
end
```

```
(* Make a property from a given function *)
p = forAll (fn (a: ℤ32, b: ℤ32) ⇒ (a + b = b + a))
q = forAll (fn (a: ℤ32, b: ℤ32) ⇒ (a − b = b − a))

(* Conjunction *)
pandq = forAll (fn (a: ℤ32, b: ℤ32) ⇒ p(a, b) ∧ q(a, b))
(* Conjunction with tagging *)
pandq′ = forAll (fn (a: ℤ32, b: ℤ32) ⇒
        ("add" |: p(a, b)) ∧ ("subtract" |: q(a, b)))
(* Disjunction *)
porq = forAll (fn (a: ℤ32, b: ℤ32) ⇒ p(a, b) ∨ q(a, b))
(* Implication *)
ptheng = forAll (fn (a: ℤ32, b: ℤ32) ⇒ p(a, b) → q(a, b))

(* Collecting the test data *)
p′ = forAll (fn (a: ℤ32, b: ℤ32) ⇒
        collect(⌊ log(a b)/ log(10) ⌋) p(a, b))
(* Classifying the test data *)
q′ = forAll (fn (a: ℤ32, b: ℤ32) ⇒
        classify(a < b, "a<b") classify(a = b, "a=b")
        classify(a > b, "a>b") p(a, b))
```

Figure 2: Example operations generating testable instances of type Testable⟦ℤ32⟧

As with QuickCheck, successful results do not necessarily mean that the test is indeed true; it just shows an inability to find a counterexample in a given limit.

3.4 Reflection in FortressCheck

A major difference between QuickCheck and FortressCheck is the use of reflection. FortressCheck uses reflection to solve two problems: testing subtype polymorphism by constructing new generators from existing generators, and testing parametric polymorphism by desugaring generic properties. We chose the reflection technique over other metaprogramming techniques because both problems involve generation of first-class type objects, for which the reflection technique is very well suited.

We have implemented a reflection library for the Fortress interpreter, Reflect. The Reflect library provides ways to inspect static types of expressions, dynamic types of values, and fields and methods of traits and objects. It also provides ways to (partially) manipulate generic types. Due to the current status of the Fortress interpreter, the Reflect library supports only the types whose subtypes are known at compile time, that is, object types and trait types with `comprises` clauses. This limitation is not inherent in the FortressCheck design but a limitation of the current implementation; this limitation will go away when the interpreter has an ability to inspect the list of subtypes of a given dynamic type.

3.4.1 Making Generators from Other Generators

The *commutativeAddition* property described in Section 2.2 is an example of subtype-polymorphic properties:

```
property commutativeAddition =
    ∀(x: Number, y: Number) (x + y = y + x)
```

because the Number trait has many subtypes including $\mathbb{R}64$, $\mathbb{R}32$, $\mathbb{Q}$, $\mathbb{Z}$, $\mathbb{Z}64$ and $\mathbb{Z}32$. Assuming that we have generators for those subtypes but not for Number itself, we can define a non-polymorphic property using *commutativeAddition* as follows:

```
property commutativeAddition′ =
    ∀(xtype: Type, ytype: Type)
        ((xtype SUBTYPEOF theType⟦Number⟧()) ∧
         (ytype SUBTYPEOF theType⟦Number⟧())) →
        commutativeAddition(
            genFromType(xtype).generate(),
            genFromType(ytype).generate())
```

The `SUBTYPEOF` operator checks whether the first argument is a subtype of the second argument, and *theType*⟦Number⟧() returns a type object for the Number trait. If two given types to the property *commutativeAddition′* are subtypes of Number, we can look up the generators for the types using *genFromType* and feed the resulting test instances to the original property, *commutativeAddition*. In the actual implementation, instead of testing *commutativeAddition′* for arbitrary two types, Type, we test it only for subtypes of the Number trait, *theType*⟦Number⟧(), to reduce the number of ignored tests.

Another way to describe the property is to use a generic property:

```
property commutativeAddition″
        ⟦X extends Number, Y extends Number⟧ =
    ∀(x : X, y : Y) (x + y = y + x)
```

In this property, type variables X and Y in the type parameter list denote the exact types of X and Y instead of their subtypes. Applying the desugaring process of generic properties described in Section 3.4.2 to *commutativeAddition″* yields an equivalent result to *commutativeAddition′*.

3.4.2 Desugaring Generic Properties

As an example of parametric-polymorphic properties, consider $mapLengthInvariant_2$ described in Section 2.2:

```
property mapLengthInvariant₂⟦Key, Val, Res⟧ =
    ∀(map: Map⟦Key, Val⟧, f: (Key, Val) → Maybe⟦Res⟧)
        (|map.mapFilter⟦Res⟧(f)| ≤ |map|)
```

Similarly to the subtype-polymorphic properties, we can define a non-polymorphic property by generating corresponding type parameters and applying them to the parametric-polymorphic property:

```
property mapLengthInvariant′₂ =
    ∀(keytype: Type, valtype: Type, restype: Type) (do
        prop = applyStaticParams(mapLengthInvariant₂,
                                 (keytype, valtype, restype))
        keygen = genFromType(keytype)
        valgen = genFromType(valtype)
        resgen = genFromType(restype)
        prop(genMap(keygen, valgen).generate(),
            genArrow(genTuple₂(keygen, valgen),
                     genMaybe(resgen)).generate())
    end)
```

Assuming that we have a function *applyStaticParams*, which applies given type parameters to a generic property to obtain a non-generic property, we get a non-generic property *prop* from the generic property $mapLengthInvariant_2$.

This approach also applies to generic properties with bounded type parameters. The *commutativeAddition''* property, a generic version of *commutativeAddition*, has such type parameters and can be desugared as follows:

$$
\begin{array}{l}
\texttt{property}\ \mathit{commutativeAddition}''' = \\
\quad \forall(\mathit{xtype}\colon \mathrm{Type}, \mathit{ytype}\colon \mathrm{Type}) \\
\quad\quad \big(\big(\mathit{xtype}\ \texttt{SUBTYPEOF}\ \mathit{theType}[\![\mathrm{Number}]\!]()\big) \wedge \\
\quad\quad\ \big(\mathit{ytype}\ \texttt{SUBTYPEOF}\ \mathit{theType}[\![\mathrm{Number}]\!]()\big)\big) \rightarrow (\texttt{do} \\
\quad\quad\quad \mathit{prop} = \mathit{applyStaticParams}(\mathit{commutativeAddition}'', \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (\mathit{xtype}, \mathit{ytype})) \\
\quad\quad\quad \mathit{prop}(\mathit{genFromType}(\mathit{xtype}).\mathit{generate}(), \\
\quad\quad\quad\quad\quad \mathit{genFromType}(\mathit{ytype}).\mathit{generate}()) \\
\quad\quad \texttt{end})
\end{array}
$$

Note that inlining the call to *applyStaticParams* produces the same result as one by the subtype-polymorphic property, *commutativeAddition'*, described in Section 3.4.1.

Moreover, this approach is general enough to allow more complex type parameters. For example, we can test the LexicographicOrder⟦T, E⟧ trait defined in the Fortress standard library, which has a type parameter whose bound is the trait being defined:

(* If x is lexicographically less than y and x is not shorter than y, there is at least one pair of elements a and b at the same position such that $a < b$. *)

$$
\begin{array}{l}
\texttt{property}\ \mathit{lexico}[\![T\ \texttt{extends}\ \mathrm{LexicographicOrder}[\![T, E]\!], E]\!] = \\
\quad \forall(x : T, y : T)\ \big((x < y) \wedge (|x| \geq |y|)\big) \rightarrow \\
\quad\quad\quad\quad \left(\bigvee_{(a,b) \leftarrow x.zip[\![E]\!](y)} a < b \right)
\end{array}
$$

Assuming that *genericLO* is a type object for the generic type LexicographicOrder⟦T, E⟧ and its method *applyArgs* applies given types as its type arguments, the corresponding non-generic property would be simply like the following:

$$
\begin{array}{l}
\texttt{property}\ \mathit{lexico}' = \forall(\mathit{ttype}\colon \mathrm{Type}, \mathit{etype}\colon \mathrm{Type}) \\
\quad \big(\mathit{ttype}\ \texttt{SUBTYPEOF}\ \mathit{genericLO}.\mathit{applyArgs}(\mathit{ttype}, \mathit{etype})\big) \rightarrow \\
\quad\quad \ldots
\end{array}
$$

In the actual implementation, we implemented the generic properties as generic methods in a dedicated object, so that we can inspect the object to determine generators for individual properties. While top-level generic functions can be also used, the current implementation does not support them due to the current status of the Fortress interpreter.

3.5 Implementation

The current FortressCheck implementation is available from the Fortress source repository: `http://projectfortress.sun.com`. It consists of two components, QuickCheck and ReflectiveQuickCheck, to simplify a future port to the Fortress compiler, which needs additional support for reflection.

4. RELATED WORK

The QuickCheck library has been ported to dozens of programming languages, and the random testing technique has been integrated with systematic testing in various ways.

4.1 Comparison with Other QuickCheck Ports

While dozens of languages have ported QuickCheck, most of them merely implemented the basic features of QuickCheck. We divided the QuickCheck ports into a few (possibly overlapping) categories to compare:

Dynamically-typed languages:.

One of the biggest limitations of the QuickCheck ports in dynamically-typed languages is that they cannot infer generators from the types of given properties. Instead, programmers should specify the generators manually, as shown in the following Python code written for the `qc` library [6]:

```
from qc import *

@forall(x=an_integer(low=-999, high=999),
        y=an_integer(low=-999, high=999))
def test_commutative_addition(x, y):
    assert x + y == y + x
```

where the code explicitly specifies the generator `an_integer`.

Due to this limitation, QuickCheck is often used in conjunction with an existing unit testing facility. For example, `qc` extensively uses the "functional decorator" of Python such as the `@forall` decorator in the above example to automatically generate test functions, and it uses naming conventions for test functions such as the `test_` prefix, which makes it compatible to other unit testing libraries looking for such names. Besides this limitation, most ports implement the full features of QuickCheck; generation of random functions, for example, is supported by RushCheck [13] for Ruby and Scheme-Check [17] for Scheme.

Statically-typed languages:.

Even with QuickCheck ports in statically-typed languages, some port does not offer automatic inference of generators, and some port does not offer any concise specification language; the examples include `quickcheck` for Java™ [14] for the former and QuickCheck++ [1] for C++ for the latter. The latter, verbose interface, is mostly due to the lack of concise syntax for anonymous functions.

QuickCheck ports in functional programming languages provide better functionality. Moreover, some ports provide limited uses of reflection such as FsCheck [18] for F# and ScalaCheck [15] for Scala. Both ports use reflection as a mechanism of retrieving lists of properties, but FsCheck also uses it for constructing generators for compound types; the generator for record types in FsCheck, for example, is almost impossible to construct without metaprogramming. Indeed, the overall design of FortressCheck has been heavily inspired by them. However, their use of reflection is much simpler than that of FortressCheck. While both F# and Scala provide both subtype polymorphism and parametric polymorphism, they do not use reflection for testing them.

Parallel languages:.

Among the various QuickCheck ports, Quviq QuickCheck [9] for Erlang is the only one which specifically tests its parallel feature. It once used a linear temporal logic to specify parallel behaviors of programs without requiring any extension or modification to the QuickCheck library. However, due to the difficulties in specifying concurrent programs, Quviq QuickCheck now describes the behavior of a concurrent program as a sequential specification. It then tests the atomicity of the program by checking the equivalence between the program and its specification with the help of PULSE, a user-level scheduler for Erlang.

4.2 Other Extensions to QuickCheck

Recent research have integrated random testing and systematic testing in two ways: one way is to use random testing primarily while altering the distribution of test cases systematically, and the other way is to use systematic testing primarily while using random testing for the initial direction or for secondary choices. RANDOOP [16] falls into the former, and DART [11] and its successor, CUTE [19], are representative examples of the latter. These approaches require heavyweight instrumentations and inspection facilities, but they are considerably more powerful than undirected random testing.

JCrasher [10] is an automatic random testing tool for Java, which uses reflection to inspect the methods and their parameter types declared by a given class. It also shows a practical implementation of testing imperative features.

5. CONCLUSION AND FUTURE WORK

We presented FortressCheck, a version of QuickCheck random testing tool for the Fortress programming language, which was extended to support unique features of Fortress. Unlike Haskell, Fortress provides both subtype polymorphism and parametric polymorphism, and we proposed to use reflection for testing such polymorphic properties. We described our approach to handle polymorphic properties and implemented the proposed approach in the Fortress interpreter.

To improve the conciseness of the generators and properties in FortressCheck, we plan to support an embedded testing language using the extensible syntax system described in [5]. In addition to the basic testing facilities in FortressCheck, we plan to provide better supports for testing implicit parallelism, one of the main features of Fortress, more thoroughly.

6. REFERENCES

[1] QuickCheck++. `http://software.legiasoft.com/quickcheck/`.

[2] QuickCheck — Wikipedia, The Free Encyclopedia. `http://en.wikipedia.org/w/index.php?title=QuickCheck`.

[3] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr., and S. Tobin-Hochstadt. The Fortress Language Specification Version 1.0 β, March 2007.

[4] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr., and S. Tobin-Hochstadt. The Fortress Language Specification Version 1.0, March 2008.

[5] E. Allen, R. Culpepper, J. D. Nielsen, J. Rafkind, and S. Ryu. Growing a syntax. In *Foundations of Object-Oriented Languages, 2009*, Jan. 2009.

[6] D. Bravender. qc: A Quickcheck implementation for Python. `http://github.com/dbravender/qc`.

[7] K. Claessen and J. Hughes. QuickCheck: An Automatic Testing Tool for Haskell. `http://www.cse.chalmers.se/~rjmh/QuickCheck/`.

[8] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, New York, NY, USA, 2000. ACM.

[9] K. Claessen, M. Palka, N. Smallbone, J. Hughes, H. Svensson, T. Arts, and U. Wiger. Finding race conditions in Erlang with QuickCheck and PULSE. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 149–160, New York, NY, USA, 2009. ACM.

[10] C. Csallner and Y. Smaragdakis. Jcrasher: an automatic robustness tester for java. *Softw. Pract. Exper.*, 34(11):1025–1050, 2004.

[11] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, 2005.

[12] J. Hughes. personal communication, Nov. 2010.

[13] D. Ikegami. RushCheck: a lightweight random testing tool for Ruby. `http://rushcheck.rubyforge.org/`.

[14] T. Jung. quickcheck: Java implementation of QuickCheck. `https://quickcheck.dev.java.net/`.

[15] R. Nilsson. ScalaCheck. `http://code.google.com/p/scalacheck/`.

[16] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society.

[17] C. E. Scheidegger. Scheme-Check: Randomized Unit Testing for PLT Scheme. `http://www.inf.ufrgs.br/~carlossch/scheme-check/`. Currently only available via Wayback Machine: `http://web.archive.org/web/20050212183945/www.inf.ufrgs.br/~carlossch/scheme-check/`.

[18] K. Schelfthout. FsCheck: A random testing framework. `http://fscheck.codeplex.com/`.

[19] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 263–272, New York, NY, USA, 2005. ACM.