UNIVERSITY OF COPENHAGEN

Msc thesis

Matin Nafar & Simon August Mørk

Implementing Property-Based Testing for Futhark

Advisor: Troels Henriksen

Handed in: May 31, 2026

Abstract

In this thesis, we design, implement, test, and evaluate FutPBT, a property-based testing tool for the high-performance and purely functional data-parallel programming language Futhark, that lets programmers specify general correctness properties, which are tested against many randomly generated inputs.

FutPBT adapts the usual components of property-based testing to Futhark's server-based execution model. Properties, generators and shrinkers are expressed as Futhark entry points, declared as test components with a comment and attribute design, while the external Haskell runner orchestrates generation, shrinking, validation, reporting, and error handling through the Futhark server protocol, and manages server-side values with an explicit freeing discipline. The runner also supports fallback mechanisms for generation and shrinking. The implementation integrates with the existing `Futhark test` execution model.

The implementation is tested through integration tests and evaluated qualitatively as a testing tool, focusing on robustness, usability, and use on realistic Futhark code. Overall, FutPBT shows that property-based testing can be successfully implemented for Futhark, by adapting it to the constraints of the language and the server protocol.

Contents

1 Introduction

In property-based testing users specify general correctness properties, and those properties are then tested on many generated inputs. Instead of writing individual test cases by hand, the user describes the behavior that should hold, while the tool is responsible for testing many randomly generated inputs against the property, and reporting any, potentially shrunk, counterexamples. Systems such as QuickCheck have shown this approach to be useful, with properties, generators, and shrinkers expressed directly in the host language [CH00; Hug16].

Futhark provides a different setting for property-based testing. Futhark is a high-performance, data-parallel functional programming language designed for pure array computations [Hen17], but is not designed for implementing complete systems inside the language itself [EHO18a], such as testing frameworks like property-based testing. Furthermore, Futhark does not provide type classes, general printing facilities or exception-style error handling. Consequently, orchestration in another language is required and the classical QuickCheck approach must be adapted to this setting.

This thesis describes the design, implementation, testing, and evaluation of FutPBT, a property-based testing extension for Futhark in Haskell. FutPBT allows users to express correctness properties as Futhark entry points, associate them with generators and shrinkers, and run them through the existing *Futhark test* workflow. The repeated testing loop is implemented in Haskell and communicates with the Futhark program through the Futhark server protocol. This division keeps the Futhark-side interface small, while allowing the runner to handle and orchestrate generation, shrinking, validation, reporting, and error handling outside the tested program. The central design problem is to adapt the usual components of property-based testing to this server-based setting, while preserving as much of the QuickCheck design, usability and feature range as possible within the constraints of Futhark.

Section 2 introduces the necessary background on Futhark, property-based testing and QuickCheck. Section 3 develops the FutPBT protocol step by step, starting from a simple Boolean property. Section 4 presents the seven central requirement goals and how these are achieved. Section 5 explains how the implementation is tested against its requirements. Section 6 evaluates the robustness and usability of the resulting tool, and Section 7 discusses possible future work.

FutPBT is part of release 0.26.3 of Futhark [Hen26b] and is available at:
`https://github.com/diku-dk/Futhark/pull/2405` (Futhark)
`https://github.com/Black-Speedy/futhark-pbt/tree/pbt` (standalone)

2 Background

This section introduces the background needed for the rest of the thesis. We first describe Futhark and the constraints it imposes on an external testing tool. We then introduce property-based testing, focusing on the roles of properties, generators, and shrinkers. Finally, we show established practice through QuickCheck, which serves as the main point of comparison for FutPBT.

2.1 Futhark

Futhark is a high-performance, statically typed, purely functional, data-parallel array language in the ML family. It is designed for programs that operate on arrays, where the programmer writes high-level functional code and the compiler generates efficient parallel code [Hen17]. In this way, Futhark combines the reasoning benefits of functional programming with the performance benefits of parallel hardware such as GPGPUs. The compiler is implemented in Haskell [Hen17].

Futhark is not intended to be used for complete applications, but rather for small, performance-sensitive parts of larger programs, typically by compiling Futhark code as a library that can be called from another language [EHO18b]. This specialization is reflected in the language interface: Futhark does not provide general input/output facilities or exception-style error handling inside the language. Futhark programs expose externally callable functions through *entry points*, which are declared with the keyword `entry` [The26c].

2.2 Property-based testing

Property-based testing (PBT) uses randomly generated data to verify general invariants, or "properties," of a program [CH00]. In this sense, it shares DNA with fuzz testing [MHC19], but is distinguished by focusing on logical properties and structural data rather than focusing on inducing program execution failure. Property-based testing can be split into three general sub-parts: a property, a generator, and a shrinker. These address the questions: (1) What should always hold? (2) Which inputs do we test? (3) Can we make it easier to understand?

2.2.1 Property

A property is a high-level requirement that must hold for all possible inputs within a given domain. Common strategies for defining properties include round-tripping (e.g. `reverse(reverse(xs))==xs`) or checking metamorphic properties where the relationship between inputs should be preserved in the output [CH00; Che+18]. The challenge of PBT lies in "specifying how to test" [Gol+24], shifting the developer's burden from writing test cases to defining the formal boundaries of their system.

2.2.2 Generator

Properties need input data to test against. The utility of properties is limited by the quality of its input data. A generator is a distribution or function responsible for producing random values of a specific type. Effective generators must balance randomness with coverage, ensuring that "edge cases" (such as empty lists, null values, or large integers) are tested frequently enough to catch bugs [Lam+19]. Modern frameworks often use a language of combinators to build complex generators for custom data types from simple primitives [MRH18].

2.2.3 Shrinker

When a property fails, the randomly generated input is often large and may contain "noise" that is irrelevant to the actual bug, making debugging difficult [CH00]. *Shrinking* is the process of simplifying a failing counterexample while preserving the failure [Vri23]. The goal is not necessarily to find a globally minimal input, but to produce a smaller and more understandable counterexample from the previous one. There are many options for shrinking [Cla26; Pik14; Vri23], but in this thesis we will mainly focus on the QuickCheck approach.

2.3 Established practice: QuickCheck

Modern property-based testing is commonly traced to QuickCheck, originally developed for Haskell by Claessen and Hughes [CH00]. The same basic idea has been adapted to many languages and programming environments [MHC19]. Other testing tools explore different points in the same design space [RNL08].

2.3.1 Design of QuickCheck

In QuickCheck, the programmer writes a property as a Haskell expression or function, and repeatedly tests that property on randomly generated inputs, by passing it to the function `QuickCheck`. QuickCheck does not require a separate specification language, as the specification is written in the same language as the program being tested [CH00]. A simple QuickCheck property can be a Boolean expression. For example, a property about list reversal can state that reversing a list twice gives back the original list.

```
1 prop_RevRev xs = reverse (reverse xs) == xs
```

Figure 1: QuickCheck property example of reverse taken from [CH00].

Since such a property returns either `True` or `False`, QuickCheck can run the property and report whether it succeeded. However, QuickCheck does not restrict properties to plain Booleans. It also has a `Property` type, which represents richer

testable statements, such as properties with conditions, labels, generated input, or other testing annotations. QuickCheck generalizes the notion of a property through the `Testable` type class. The purpose of `Testable` is to describe which Haskell values can be interpreted as testable properties. In simplified form, a value is testable if QuickCheck can convert it into a `Property`:

```
class Testable prop where
  property :: prop -> Property
```

This abstraction is important because it lets QuickCheck treat both boolean values and functions as properties. A value of type `Bool` can be tested directly. A function type such as `Int -> Bool` is interpreted as a universally quantified property over generated integers: QuickCheck generates an `Int`, applies the function to it, and checks the resulting Boolean. More generally, if a value of type `a` can be generated and `prop` is testable, a function of type `a -> prop` is also testable. This is how QuickCheck supports properties with several arguments, such as `Int -> [Int] -> Bool` [CH00].

The generator side of QuickCheck is represented by the abstract type `Gen a`, which denotes a generator for values of type `a`. Since the `Testable` instance for functions requires generated arguments, QuickCheck needs a systematic way to generate values of different types. This is provided by the `Arbitrary` type class and its method `arbitrary`, which produces a generator for values of the given type:

```
class Arbitrary a where
  arbitrary :: Gen a
```

A type is an instance of `Arbitrary` when QuickCheck can generate random values of that type. QuickCheck provides generators for many basic types, while programmers can define their own `Arbitrary` instances for user-defined data types. This is essential for testing realistic programs, since many properties are not about primitive values alone, but about structured inputs such as lists, trees, expressions, or domain-specific data structures.

Generators in QuickCheck are compositional. Simple generators can be combined into generators for more complex values, and generator combinators can be used to choose between alternatives, generate lists, restrict values, or build structured data. The original QuickCheck paper also discusses the importance of size control [CH00]. QuickCheck provides a size parameter that generators can use to control the complexity of the values they produce [CH00].

To define custom generators, QuickCheck provides combinators such as `elements`, which chooses from a fixed list, and `listOf`, which creates a generator for a list of values. For instance, if a programmer wants to generate a list containing colors, they could define the following:

```haskell
data Color = Red | Blue | Green
    deriving (Show, Eq)

-- A generator choosing from a set of alternatives
genColor :: Gen Color
genColor = elements [Red, Blue, Green]

-- A generator for a list of colors
genColorList :: Gen [Color]
genColorList = listOf genColor
```

Figure 2: Example of generator combinators adapted from [CH].

Some properties quantify not only over values, but also over functions. Claessen and Hughes show that generated functions can be represented extensionally, by mapping generated inputs to generated outputs. This makes it possible to test higher-order properties [CH00]. A classic example of a higher-order property is the associativity of function composition. Since functions cannot be compared directly by ordinary equality, Claessen and Hughes define an extensional equality operator, written ===, which compares two functions by applying them to a generated input. QuickCheck must then generate three functions and an input value to test whether the composition law holds [CH00].

```haskell
-- Extensional equality for generated inputs.
(f === g) x = f x == g x

-- Function composition is associative.
prop_CompAssoc :: (Int -> Int) -> (Int -> Int)
               -> (Int -> Int) -> Int -> Bool
prop_CompAssoc f g h =
  f . (g . h) === (f . g) . h
```

Figure 3: Higher-order property example from [CH00].

When this property is passed to `QuickCheck`, the framework automatically generates functions for `f`, `g`, and `h`. The property is tested extensionally: rather than comparing functions directly, QuickCheck compares the outputs produced by the two sides of the equation on generated inputs.

The original QuickCheck paper did not introduce shrinking as a central part of the testing algorithm, although it briefly mentions a function `smaller :: a -> [a]` in the context of pretty printing counterexamples [CH00]. Modern QuickCheck, however, includes shrinking as a standard part of the workflow [Cla26]. When a property fails, QuickCheck does not simply report the first randomly generated counterexample. Instead, it tries to replace the failing input with smaller candidate inputs and reruns the property on those candidates. If a smaller candidate still fails, QuickCheck continues shrinking from that value, to find a simpler counterexample that still produces the bug.

```haskell
-- A property that fails if any element is greater than 10
prop_small :: [Int] -> Bool
prop_small xs = all (<= 10) xs
```

```
Example Output without shrinking:
*** Failed! Falsifiable (after 3 tests):
[15, 2, 9, -34, 100, 5, 22, 11]
Example Output with shrinking:
*** Failed! Falsifiable (after 3 tests and 5 shrinks):
[11]
```

Figure 4: Illustrative comparison of a raw counterexample and a shrunk counterexample. The exact output depends on the generator, seed, and QuickCheck version.

As an extension, shrinking in modern QuickCheck is connected to the `Arbitrary` type class through the `shrink` method [Cla26].

```haskell
class Arbitrary a where
  arbitrary :: Gen a
  shrink    :: a -> [a]
```

The `arbitrary` method describes how to generate random values of type `a`, while `shrink` describes how to produce simpler candidate values from an existing value. For example, a shrinker for integers may try values closer to zero, while a shrinker for lists may try shorter lists or lists whose elements have themselves been shrunk.

The design of shrinking has become an important part of later property-based testing systems, including modern QuickCheck and newer systems, such as `falsify`, which revisits the relationship between generation and shrinking in Haskell [Vri23; Cla26; Pik14].

3 Design

This sections presents the user-facing design of FutPBT. The goal is to extend the existing Futhark testing workflow with property-based testing, keeping the Futhark-side interface small, while aiming to adapt the behavior, robustness, and user experience provided by QuickCheck. A user should be able to write the computational parts of a property-based test in Futhark, with the surrounding testing logic, such as repetition, generation control, shrinking control, and error handling, managed by an external runner. An overall goal of the design is to keep the interface simple and usable, in line with the central appeal of property-based testing: making testing easier for the user, summarized in the idea: Do not write tests, generate them [Hug16].

The design of the protocol is presented as an iterative, example-driven process, where each section incorporates a new step of the protocol. We begin with the simplest protocol: a property is a Futhark entry point that returns a boolean. From there, we identify the limitations of this design and extend the protocol with (1) Property inputs and how these should be handled (2) Generation, which defines how to produce test data while leaving the distribution under user control (3) Shrinking, where the main design challenge is to simplify counterexamples while keeping the Futhark-side protocol simple. When a design component reaches its final form, we state it explicitly as a definition.

3.1 Defining properties

To start off, we want the ability to state properties of functions as outlined in Section 2.2.1, for example `reverse` as the involution:

```
reverse (reverse xs) == xs
```

Section 2.3 described how QuickCheck treats several Haskell values as properties through the `Testable` type class. We want properties to be expressed purely in Futhark, with the same separation of concerns as QuickCheck, where properties check if a value violates the property. We begin with the simplest possible interface: Properties as functions returning a boolean.

```
1 reverse_involution : bool =
2   let xs = [1i32, 2i32, 3i32]
3   in and (map2 (==) (reverse (reverse xs)) xs)
```

This, however, would be equivalent of a unit-test, where the input must be written explicitly. This is also not how properties are defined in QuickCheck. Furthermore, if `xs` is a random value, we cannot see the value if the property fails, as there is no IO in Futhark. Being able to store the value outside of Futhark allows for inspecting it.

We now define a *property* as a unary Futhark entry point from some type *testType* to a Boolean.

Definition 3.1 (Property)**.** *A* property *is a Futhark entry point of type*

`testType -> bool`.

For example, the *property* that `reverse` is an involution can be written as follows:

```
entry reverse_involution (xs: []i32) : bool =
  and (map2 (==) (reverse (reverse xs)) xs)
```

The *property* now takes an input to validate with. We call such an input value a *candidate*.

Definition 3.2 (Candidate)**.** *A* candidate *is a value* `x : testType`.

Given a *property* and a *candidate*, evaluate the *property* on that *candidate*. We call this single evaluation a *test*.

Definition 3.3 (Test)**.** *A* test *is the evaluation of a* property *on a* candidate*. If the* property *returns* `true`*, the* test *is interpreted as a success. If the* property *returns* `false`*, the* test *is interpreted as a failure.*

A failing *test* shows that a concrete *candidate* does not hold for a *property*. We call such a *candidate* a *counterexample*.

Definition 3.4 (Counterexample)**.** *A* counterexample *is a* candidate *for which evaluating the* property *by performing a* test *returns* `false`.

With these definitions, we can devise a system called the *runner*. For the example of `reverse_involution`, the runner generates many random *candidates* of type `[]i32`, passes each candidate to the *property* entry point, and checks whether the result is `true` or `false`. If the property returns `false` for some *candidate*, that *candidate* is a *counterexample*. Before reporting the *counterexample*, the *runner* may try to simplify the *counterexample* through shrinking, and then report it to the user. However, using this simple definition of *property* has some limitations.

This is the first departure from the approach described in Section 2. QuickCheck can make the notion of a property implicit through `Testable`; FutPBT must instead make it explicit as an entry-point protocol. Our definition of a *property* is more restrictive than properties in QuickCheck. First, a *property* must be an entry point, such that the runner can call it, which means that it cannot be higher-order [The26a]. Furthermore, in QuickCheck, properties are not restricted to one fixed surface type such as `a -> Bool` as explained in Section 2.3.1. QuickCheck supports properties with different numbers of arguments, as well as richer constructs such as conditional properties and classification combinators [CH00].

Futhark does not have type classes [EHO18b]. This means that our *property* interface does not directly support QuickCheck-style properties with varying numbers of arguments, higher-order properties, or monitoring facilities in the same way as `classify` [CH]. The benefit is that the interface is simple and easy to check: the Futhark side only has to answer whether a *candidate* satisfies the *property*. More involved testing machinery, such as *candidate* generation, reporting, and classification, can then be placed outside the Futhark language boundary. Futhark is statically typed [EHO18b], so types must be defined before compilation. This does not by itself distinguish FutPBT from QuickCheck, since Haskell is also statically typed.

At this point, we have defined the shape of a *property* and the basic vocabulary used by the runner: *candidates*, *tests*, and *counterexamples*. However, the runner still has no way to evaluate which entry points in a Futhark program should be treated as a *property*. A Futhark file may contain many entry points, and only some of them are intended to be part of the property-based test suite. The next question to be addressed is how the user's intent should be represented.

3.1.1 Declaring properties

Since a *property* is an entry point of type `testType -> bool`, one possible design is to let the runner classify every entry point with an arity of one and a boolean output as a declared *property*.

This design has the advantage of being simple and does not require additional syntax from the programmer. It also resembles part of the QuickCheck experience: a function returning a boolean can be interpreted as a testable property, and QuickCheck's `Testable` class generalizes this idea to functions of several arguments whose final result is testable. As we do not have type classes in Futhark, the closest simple approximation is to use the server-visible type of an entry point: if it has type `testType -> bool`, then the runner may try to test it.

This solution, however, is too broad. A Futhark program may contain entry points of type `testType -> bool` that are not intended as *properties*. For example, an entry point may expose a normal predicate, a helper used for debugging, or a boolean-valued computation that is meant to be called directly. The type tells us that the function can be evaluated as a test, but it does not tell us that the programmer intended it to be part of the property-based test suite. Treating all such entry points as *properties* would therefore risk running tests that the user did not ask for. For further discussion of this see [Hen26c].

A solution would be to pass the *property* names on the command line. This would make discovery explicit, since the runner would only test entry points named by the user. However, this moves the test specification out of the Futhark program.

One of the goals of the design is that a test file should describe the tests it contains, rather than requiring the essential test declaration to live in an external invocation. We hence reject this solution.

An alternative design option is to use a naming convention.

```
entry prop_reverse_involution (xs: []i32) : bool =
  and (map2 (==) (reverse (reverse xs)) xs)
```

In this design, the runner treats entry points whose names begin with `prop_` as *properties*. This follows a common convention from QuickCheck examples, where properties are often named with a `prop_` prefix, such as `prop_reverse_involution`. This avoids treating all boolean-valued functions as tests, because the user must at least name the *property* as a property.

The naming convention is more precise than using the type alone, but it still has its issues. It makes evaluation depend on spelling rather than on an explicit declaration. If the user renames an entry point and forgets the prefix, the entry point silently stops being treated as a *property*. Conversely, any entry point whose name begins with `prop_` will be interpreted as a *property*, even if the prefix was used only as an informal name. This convention communicates intent, but only indirectly.

A solution is to place the *property* declaration in a comment block. For example:

```
-- ==
-- property: prop_RevRev

entry prop_RevRev (xs:[]i32) : bool = ...
```

This has the advantage that the test declaration is kept in the same comment-based testing format already used by Futhark, following the principle of least astonishment. It also makes it easy to choose which *properties* are part of a particular test block.

A drawback is that comments are separate from the entry points they describe. A comment can name a *property* that no longer exists, or it can fail to mention an entry point that is otherwise intended as part of the property-based test suite. The runner must therefore perform additional validation to detect mismatches between the comment-level declaration and the actual visible entry points.

A solution is to attach the declaration directly to the *property* entry point by adding it as an attribute:

```
#[prop]
entry prop_reverse_involution (xs: []i32) : bool =
  and (map2 (==) (reverse (reverse xs)) xs)
```

This makes the identification local to the *property*. The *property* entry point attribute declares that it is a *property*. Unlike the naming convention, this does not require the *property* to be named in any particular way. Unlike a comment-only solution, the declaration is attached directly to the entry point.

However, an attribute-only solution does not match the structure of *Futhark test*. Futhark tests are selected through test blocks, and it is useful for property-based tests to participate in the same mechanism. We will explain this in more detail in Section 4.3. We will use another option: a hybrid design.

```
-- ==
-- property: prop_reverse_involution

#[prop]
entry prop_reverse_involution (xs: []i32) : bool =
  and (map2 (==) (reverse (reverse xs)) xs)
```

In this hybrid design, the attribute indicates that a *property* is testable, while the comment block is the source of information about whether the *property* is included in a particular test specification. The runner must check that all property test blocks refer to an attributed *property*, and that all *property* attributes are connected to a corresponding test block. At last, we can define a *property test*:

Definition 3.5 (Property test)**.** *A* property test *is a* property *selected by* `property:` *in a test block and annotated with a* `#[prop]` *attribute.*

3.1.2 Invariants of properties

The definition of *property* also introduces another problem. A *property* over arrays of type `[]i32` may only be valid for sorted arrays, non-empty arrays, or arrays satisfying some relation to another value. These are not expressible in Futhark's type system. But our system allows for *property* to take all *candidates* of `testType`. Therefore, a runner that generates *candidates* only from the Futhark type may produce well-typed values that are invalid for the *property*. So while the runner can, as a default, generate *candidates*, we need another solution. We could demand that our user implements, for example, sorting in their *property*, but this would dilute the responsibility of the *property*, and also stray from how generation is handled in QuickCheck. This motivates the next extension of the protocol: a way to control how *candidates* are generated.

3.2 Defining generators

We now move from *testing candidates* to generating them. Since a *property* consumes values of type `testType`, the most direct interface is to define a generation entry point to produce values of exactly that type. In other words, the *generator* is responsible for producing *candidates*.

Definition 3.6 (Generator)**.** *A* generator *is a Futhark entry point of type* `i64 -> u64 -> testType`.

For example, the following *generator* produces an array of `i32`:

```
entry gen_i32_array (size: i64) (seed: u64) : []i32 =
  let n = i64.max 0 size
  let seed_i32 = i32.u64 seed
  in tabulate n (\i ->
    let x = seed_i32 + i32.i64 i
    in (x * 1103515245i32 + 12345i32) % i32.i64 (size + 1))
```

This definition allows the user to implement the *generator* in such a way that the runner can control and vary the generated *candidates*. By implementing a *generator* that utilizes the size variable, the runner can influence the magnitude or structural size of generated *candidates*, such as the range of integers or the length of arrays. By implementing a *generator* that utilizes the seed variable, generation will depend on a source of randomness, enabling repeated tests to explore different *candidates*. With this, generation remains under user control, since *candidate* generation can depend on semantic restrictions that are not expressible in Futhark's type system.

The purpose of our definition of a *generator* is not to infer the best possible test distribution automatically. Instead, the protocol gives the user a simple way to write executable generation logic in Futhark. This also means that questions of coverage and distribution are primarily the user's responsibility: the runner provides size and seed parameters, but the generation implementation determines how these parameters are used.

Our definition for a *generator* and *candidate* generation is a more explicit than the QuickCheck generation described in Section 2.3. QuickCheck also supports generation through combinators such as `forAll`, but these are still expressed inside QuickCheck's generation abstraction [CH00].

Futhark does not have an `Arbitrary`-style type class mechanism for attaching *generators* to types. The auto-generator handles primitive types and their composites generically, but users can override it when they need semantically meaningful test data, e.g. sorted arrays or constrained integers, that the generic *generator* cannot infer from the type alone. So the *generator* is a named entry point that the runner can call directly through the Futhark server. This keeps the boundary between Futhark and the runner explicit: the Futhark program can define how *candidates* are generated, and the runner supplies the size and seed values.

3.2.1 Declaring generators

Once *generators* are represented as entry points, the next question to address is how the runner should evaluate which *generator* belongs to a given *property*. One solution is to place the association in a test comment. For example, a property-based test could be declared in a comment block that names both the property and the generator.

```
-- ==
-- property: prop_RevRev
-- generator: gen_i32_array

entry prop_RevRev = ...
```

Or:

```
-- ==
-- property: prop_RevRev { generator=gen_i32_array }

entry prop_RevRev = ...
```

This has the advantage that the test declaration is kept in the same comment-based testing format already used by Futhark. This makes it easy to choose which *properties* are part of a particular test block.

A drawback of this design, however, is that it would introduce a new syntax for specifying the *generator*. *Futhark test* already has its own syntax for declaring entry points, inputs, and expected outputs. Extending that syntax with nested property configuration would make the test comment responsible not only for selecting tests, but also for describing how each *property* should be executed. This makes the comment block more expressive, but also more complicated.

There is also a locality problem. The association between the *property* and the *generator* would be written away from the *property* entry point itself. If the *property* is renamed, moved, or reused in another test block, the comment-level declaration may become inconsistent with the actual entry points in the program.

As a consequence, the runner would have to validate that the named *property* and *generator* both exist, and that the *generator* has the correct type for the *property*. These checks are necessary in any case, but a comment-only design makes the association more fragile when the protocol information is separated from the code it configures.

Another solution is to attach the association directly to the *property* entry point by adding it to the attribute:

```
entry gen_i32_array (size: i64) (seed: u64) : []i32 =
...

#[prop(gen(gen_i32_array))]
entry prop_reverse_involution (xs: []i32) : bool =
  and (map2 (==) (reverse (reverse xs)) xs)
```

This makes the association local to the *property*. The *property* entry point declares not only that it is a *property*, but also which *generator* should be used to produce its *candidates*. Unlike a comment solution, the declaration is attached directly to the entry point it configures, and adding sub-attributes to entry points is already implemented in Futhark. A declared *property* with an associated *generator* now looks like this:

```
-- ==
-- property: prop_reverse_involution

entry gen_reverse_involution (size: i64) (seed: u64) : []i32 =
...

#[prop(gen(gen_reverse_involution))]
entry prop_reverse_involution (xs: []i32) : bool =
  and (map2 (==) (reverse (reverse xs)) xs)
```

In this hybrid design, the attribute is the source of information about how the *property* should be executed, while the comment block is the source of information about whether the *property* is included in a particular test specification. Naturally, we would need to check if all comments have a connected attribute, and that all prop-attributes have a connected test block.

We can now extend our notion of a runner. It now also calls a *generator* if specified. It must connect two levels of information. The comment block determines which *property* tests are part of the test file, while the attribute attached to the *property* determines how that *property* test is executed. In particular, the runner must check that every property named in a comment block refers to an entry point with a corresponding `#[prop(...)]` attribute, and that every *generator* named by such an attribute exists and has the correct type.

The runner can now validate the complete test before execution. The *property* must be an entry point of type `testType -> bool`. The generator must be an entry point of type `i64 -> u64 -> testType`, where `testType` is the input type of the *property*. If these requirements are not met, the test is rejected before any *candidates* are generated. This gives the user an explicit interface while keeping errors close to the declarations that caused them.

We also extend the attribute with an optional user-defined size:

```
-- ==
-- property: prop_reverse_involution

entry gen_reverse_involution (size: i64) (seed: u64) : []i32 =
...

#[prop(gen(gen_reverse_involution), size(1000))]
entry prop_reverse_involution (xs: []i32) : bool =
  and (map2 (==) (reverse (reverse xs)) xs)
```

```
entry shrink_reverse_involution (xs: []i32) (random: u64) : []i32 =
...

#[prop(gen(gen_reverse_involution), shrink(shrink_reverse_involution))]
entry prop_reverse_involution (xs: []i32) : bool =
  and (map2 (==) (reverse (reverse xs)) xs)
```

To allow the user to define what size the *generator* should take.

3.2.2 Generator conclusion

We now have a protocol for producing *candidates* and associating those *candidates* with a *property*. A *generator* is a server-callable Futhark entry point that takes a size and a seed and returns a *candidate* of `testType`. The association between a *property* and its *generator* is declared by an attribute, while the decision to include the *property* in a run is declared in a test block with `property:`. The runner repeatedly generates *candidates* and evaluates the *properties*, until it finds a *counterexample* or the configured test bound is reached. If a *property* holds, it reports that the property passed.

We could stop here and report the found *counterexample* to the user. However, a generated *candidate* may be large or difficult to understand when printed. For example, a *counterexample* of array type may contain many elements, even though only a small part of it is relevant to the failure. This leads to the next part of the design: simplifying *counterexamples*.

3.3 Defining shrinking

We now introduce the notion of a *shrinker*, as presented in Section 2.2.3. The purpose of shrinking in FutPBT is not to guarantee a global minimal *counterexample*. In general, there is no single notion of minimality: for integers, smaller may mean closer to zero; for arrays, it may mean shorter; for records, it may depend on which fields are relevant to the *property*. Instead, our goal is to provide a protocol in which the user can define how *counterexamples* should be simplified, while the

runner guarantees a regular execution strategy. For example, it is the user's responsibility to implement shrinking that returns smaller *candidates*. Shrinking can be implemented arbitrarily bad, but as long as the runner semantics are not violated, this responsibility lies with the user. The main question with shrinking is how possible simplifications of a *counterexample* should be exposed to the runner. We need to define what shrinking should do.

3.3.1 Integrated shrinking: Auto-shrinking through the generator

One way to design shrinking, is to use the *generator*. If a generator already describes how values are constructed relative to a size parameter, it becomes possible to search for smaller *counterexamples* by rerunning the *generator* with smaller size values and a new seed drawn from the initial one. This gives a form of shrinking through the *generator* space rather than through a separate shrink function.

The attraction of this approach is that it reuses information already present in the *generator*. A user who has invested effort in writing a *generator* that produces *candidates* of increasing structural complexity as size grows has, in effect, already provided a notion of approximation to smaller *candidates*. Furthermore, if the *property* has some invariant, such as only being defined for sorted lists, the *generator* should already generate *candidates* that follow this. If the *generator* does not use size, automatic shrinking will end when size reaches zero. We will leave automatic shrinking as a default.

However, this style of shrinking only explores smaller *candidates* that remain reachable through the *generator* at smaller sizes, and depends heavily on how well the *generator* correlates size with simplicity. Furthermore, this design would not shrink actual found *counterexamples*, but present new ones after re-running the *generator*. This design works, but would not strictly be shrinking as described in Section 2.2.3. Nonetheless, it is attractive because it offers a useful fallback strategy when the user has not provided a dedicated shrinker. It improves usability while remaining compatible with the runner-driven architecture. We now extend the protocol with optional dedicated shrinking.

3.3.2 Designing a dedicated shrinker

Section 2.3 described QuickCheck shrinking as a function `shrink :: a -> [a]`, which returns a list of smaller *candidates*. A natural starting point is as such a list-based shrinking design. An example of an entry point to shrink *counterexamples* of type `i32` could be:

```
entry shrink_simple (x: i32) : []i32 =
  if x == 0 then
    []
  else
```

```
5     [0, x / 2]
```

The function shrink_simple is called again and again, each time shrinking the *candidate* incrementally.

In QuickCheck a single step of shrinking can be understood as returning a collection of immediately smaller *candidates*, calculated from a *counterexample* that falsifies the *property*. A *candidate* that falsifies the *property* is selected and then shrinking continues from this new *counterexample*. This has the advantage of making shrinking conceptually simple: the shrinking entry point proposes several possible next steps, and the runner selects one from among them.

Drawing inspiration from QuickCheck, we defined our first iteration of a shrinking entry point of type `testType -> []testType`.

3.3.2.1 return_val and return_len solution

To support list-based shrinking, the first design introduced two entry points to be written in Futhark, along with the *property*, *generator* and a shrinking entry point: **return_val** and **return_len**. **return_val** was an entry point function that given a list and an integer denoting index would return the value at that index of the list. **return_len** took a list and returned the length. At the time of implementing there was no way to get dimensions of lists through the Futhark Server protocol, but this changed during the project, making **return_len** obsolete. Using **return_val** worked for simple shrinking of non-composite types. But because Futhark changes arrays of composite to composite of arrays this complicated composite-types. Furthermore, we wanted a simpler protocol and interface, and forcing the user to implement an indexing function was not ideal. As a result, the indexing logic was moved to the *runner*.

3.3.2.2 Array slicing solution

To move the indexing to the *runner*, helper functions were introduced to support this. They existed to extract a *candidate* at a given outer index from the returned array of *counterexamples* and make it as a standalone server-side value that could be passed to the *property*. This however quickly introduced significant complexity, especially in slicing arrays and the server representation of tuples and records. As such this solution did still not support composite types, shrinking arrays along dimensions in an easy manner, and the solution was not trivial.

3.3.2.3 List-shrinking design limitations

The list-based shrinking design proved to have limitations in the setting of Futhark. In QuickCheck, shrinking is expressed as a function that returns a list of smaller

candidates, and Haskell's laziness allows the testing loop to consume this list incrementally [Cla26]. In contrast Futhark provides a different execution model. A shrinker returning `[]testType` would return a concrete Futhark array of *candidates*, which the external runner would then have to traverse.

Furthermore, in QuickCheck, the shrink representation and the testing loop are in the same host language. In Futhark each proposed *candidate* would have to be extracted, materialized, and passed back to the *property* before the runner could decide whether to accept it. The problem is not only that data must cross the boundary, but also that the runner would have to manage each *candidate* explicitly.

The array representation also creates a more fundamental restriction. Futhark arrays are regular, meaning that all inner arrays in an array of arrays must have the same size [The26c]. Thus, if `testType` itself is an array, a return-value of type `[]testType` cannot conveniently represent a collection of *candidates* where different array *candidates* have different lengths. This is precisely the kind of variation that shrinking often needs, for example, when trying both to remove elements from an array and to shrink the values of individual elements. List-based shrinking would be awkward both operationally, due to the server boundary, and semantically, because Futhark's regular arrays do not match the irregular structure of a shrink tree.

3.3.3 A new shrinker design

Prompted by the above, we instead adopt a design in which a *shrinker* returns a single *candidate* per call. As such, the shrinking entry point is of type `testType -> testType`. An example of this would be:

```
1 entry shrink_simple (x: i32) : i32 =
2   x / 2
```

The protocol favors repeated external orchestration over internal list production. Compared with the earlier list-based design, this reduces the need for indexing and materialization, keeps the protocol closer to the existing server execution model, and makes it easier to integrate shrinking into the same life cycle as *generation*, property evaluation, and reporting. While this loses the classical structure of the QuickCheck shrinking style, it is better suited to the operational constraints of an external runner for Futhark. However, this approach presents us with the question of when to stop shrinking. In QuickCheck, this happens when a list without a *counterexample* is returned. But only returning one *candidate* means it will often not be a *counterexample*, even though there may exist a smaller *counterexample*. The Futhark shrinking entry point does not track information indicating how many times it has been called, or what steps it has already performed. As such we extend the shrinking entry point with a status and a tactic.

3.3.3.1 Shrinking tactic and status

The tactic-based shrinking is inspired in part by the tactic-oriented shrinking style used in Theft for C [Vok14]. In this design, the shrinking entry point takes the current *counterexample* together with an `u64` tactic value and returns both a *candidate* and a status code. The shrinking entry point is of type `testType -> u64 -> (testType,i8)`. It could for example look like this:

```
entry shrink_simple_tactic (x: i32) (tactic: u64) : (i32, i8) =
  if tactic == 0 then
    -- Then try halving the value.
    (x / 2, i8.bool (x / 2 == x))
  else if tactic == 1 then
    -- Then try moving one step toward zero.
    if x > 0
      then (x-1, i8.bool (x-1 == x))
      else (x + 1, i8.bool (x+1 == x))
  else
    -- No more tactics.
    (x, 2)
```

The motivation is to let the user be explicit about control flow: whether the runner should return to the first tactic, advance to the next tactic, or stop shrinking completely. This shrinking design is expressive and gives the user direct control over when a tactic is exhausted, allowing it to communicate if a single iteration of shrinking did not change the *counterexample* and when shrinking should terminate. We also considered simplifying the status to a boolean and have the runner inspect whether the returned value had actually changed in order to distinguish between "continue shrinking", "advance", or "stop", which in practice pushed more implicit logic back into the runner, but we decided against it.

The status/tactic approach made the bridge between Futhark and the runner more complicated than necessary. The user would have to understand the meaning of status, the meaning of tactic, and both of these would be points where the user might make a mistake. Since one of the goals of our design is to keep the protocol as simple as possible, we again return to designing shrinking.

3.3.4 Multiple shrinkers

We considered splitting different shrinking strategies out as more entry points, which would then be specified either in the attribute or somewhere else, but decided against it. The orchestration would be unnecessarily complex, and there would be more potential points of error for the user.

3.3.5 Final shrinker definition

We now define the final *shrinker* interface, taking the form

Definition 3.7 (Shrinker)**.** *A* shrinker *is an entry point of type* `testType -> u64 -> testType`.

That is, the *shrinker* receives the current *counterexample* together with an auxiliary *random value*, and returns a single *candidate* value of the same type as the *counterexample*. It could, for example, take this form, where an auxiliary random value can be used to decide how to shrink, as per the tactic approach:

```
entry shrink_i32_array (xs: []i32) (random: u64) : []i32 =
  let n = length xs
  in if n == 0 then
       xs
     else if random % 2u64 == 0u64 then
       -- Try shrinking the structure by removing the last element.
       take (n - 1) xs
     else
       -- Try shrinking the values by moving every element toward zero.
       map (\x -> x / 2i32) xs
```

With this definition of a *shrinker* we now define a *shrink step* and a *shrinking session*:

Definition 3.8 (Shrink step)**.** *Given a* shrinker*, a* counterexample *and a* random seed*, compute a new* candidate*. Perform a* test *with the computed* candidate*. Accept it as the new* counterexample *if the* test *fails, otherwise, reject it.*

Definition 3.9 (Shrinking session)**.** *A* shrinking session *is a sequence of* Shrink steps *performed after a* counterexample *has been found. Beginning with an initial* counterexample*, repeatedly apply* Shrink steps*, with a random seed computed deterministically from the initial random seed. A* shrinking session *stops when a pre-defined amount of* Tests *in a row are successes.*

In this protocol, the *shrinker* only proposes a single *candidate*. The *shrinker* cannot communicate to the runner that a *candidate* is actually smaller, and it does not decide when the *shrinking session* should stop. Those decisions remain with the runner. The runner evaluates the proposed *counterexample* with the *property*; if the *counterexample* still fails the *test*, it becomes the new current *counterexample*, and if it succeeds, it is discarded.

The auxiliary `u64` input gives the runner a simple way to vary shrinking across repeated calls. Without this input, a deterministic *shrinker* of type `testType -> testType` would return the same *candidate* every time it was called with the same *counterexample*, and if the new *candidate* is in fact not a *counterexample*, the *shrinker* would not be able to explore other options. The auxiliary value therefore acts as a runner-controlled variation parameter. It may be interpreted by the *shrinker* as a tactic, a random value, or an index into a family of shrinking attempts. The protocol does not prescribe this interpretation.

Compared with the earlier status-based design, this interface deliberately removes control-flow information from the *shrinker*. The *shrinker* does not return a status code, boolean continuation flag, or a list of alternatives. This makes the interface less expressive, but it simplifies the interaction with the runner and there are fewer potential points of error for the user. The user only has to implement a function that proposes a *counterexample* of the correct type. They do not have to learn a separate status protocol, decide when the runner should restart or stop, or ensure that status codes remain consistent with the returned *counterexample*.

Keeping orchestration outside Futhark fits the overall design of the protocol, where the Futhark-side components remain entry points and the testing logic is handled by the runner. The resulting design is less declarative and deterministic than QuickCheck's list-based shrinking, since the *shrinker* does not expose an explicit collection of all immediate smaller *candidates*. It is, however, adapted to a setting with an outside runner. It avoids materializing whole collections of *counterexamples*, avoids indexing into arrays of values, and works naturally through repeated calls to Futhark entry points.

At this point, we have a complete protocol for our runner to execute. It can discover *properties*, generate *candidates*, *test candidates*, print *candidates*, and optionally attempt to shrink *counterexamples*. We can now define the final protocol and execution through the runner.

3.4 Final protocol

The final protocol consists of the prerequisites for participating and the mechanism for protocol execution.

3.4.1 Prerequisites for participating

To participate in this protocol a *property*, and any associated *generator* and *shrinker*, must satisfy the static requirements given in the definitions above. In addition, execution is parameterized by a test bound, a size parameter, and a seed parameter. The test bound determines the maximum number of *candidates* generated by a *generator* to *test* with a *property*. The size parameter is supplied to the *generator* to control generation, and the seed parameter determines the initial random seed used by the *generator* and the *shrinker*.

3.4.2 Protocol execution

Once a well-formed *property test* has been provided, execution begins by the runner either automatically generating a *candidate* or by invoking the associated *generator* to construct a *candidate*. The *candidate* is then supplied to the associated *property*, defined as a *test*. If the *test* succeeds, a new *candidate* is generated and

the process repeats until either a failure is found or the configured test bound is reached. If the test bound is reached without a failure, the *property* is reported as having passed.
If a *test* fails, the failing *candidate* becomes the initial *counterexample*. If no *shrinker* is provided, the runner performs automatic shrinking. If a *shrinker* is provided, the *runner* initiates a *shrinking session* in order to search for a smaller failing *counterexample*. Once execution has finished, the final *counterexample* is reported.

The overall protocol execution is shown in Figure 5. The figure shows the control flow of executing a single *property test* and runner / user-side boundary.

3.5 Design conclusion

This section has presented the design of FutPBT as a protocol between Futhark entry points and an external runner. The central design decision is to keep the Futhark-side interface small. The final protocol is by necessity different from QuickCheck. In QuickCheck, type classes such as `Testable` and `Arbitrary` allow the library to interpret many Haskell values as properties and to select generation behavior from the tested type. Futhark does not provide the same abstraction mechanisms, and FutPBT interacts with compiled Futhark programs through entry points. FutPBT can infer simple default generation from some Futhark types, but it cannot infer domain-specific generation or shrinking in the same way that QuickCheck can use type-class instances. These components have to be made explicit when the defaults are not sufficient. Test comments select which *properties* to run, while attributes on the *property* entry point describe the optional associated *generator*, *shrinker* and size override.

The design reflects the operational constraints of Futhark and the Futhark server protocol. In particular, the *shrinker* does not return a list of possible smaller *candidates*, as in QuickCheck. Instead, it returns a single *candidate* per call. This avoids materializing collections, avoids complicated indexing into server-side arrays, and gives the *runner* control of when to continue or stop shrinking. The cost is that shrinking becomes less declarative, but the benefit is a simpler and more robust server-mediated protocol.

The result is a design in which Futhark remains responsible for pure computations, while the external *runner* is responsible for the orchestration of those computations. The next chapter describes how this design is implemented and integrated into the existing Futhark testing infrastructure.

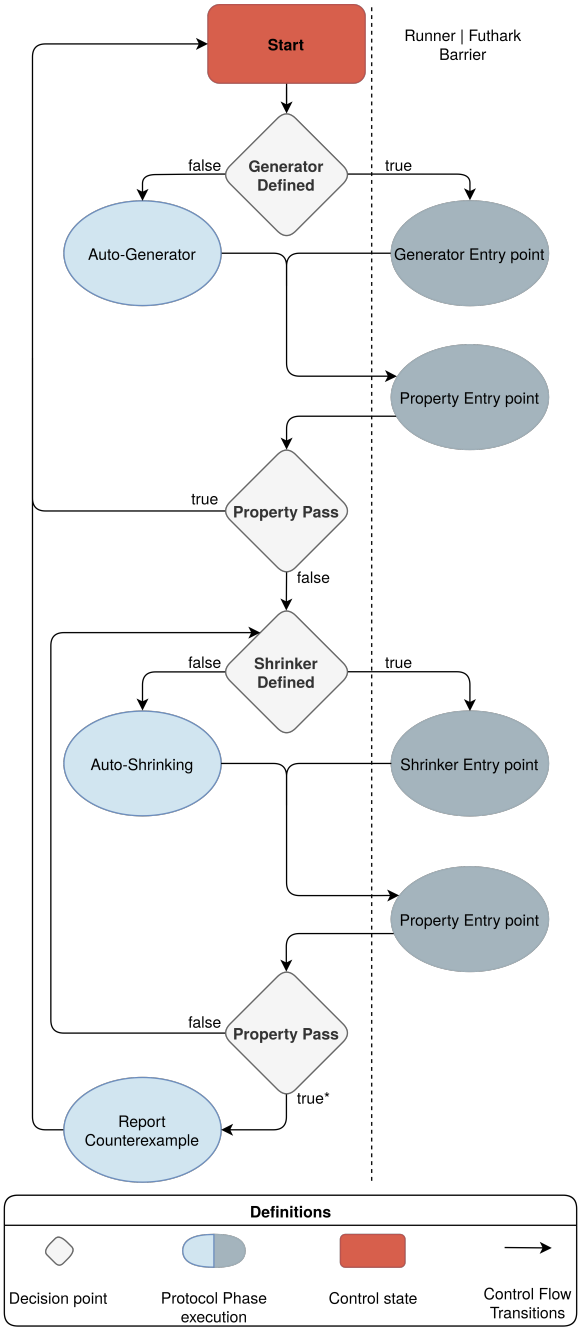

Figure 5: Control-flow diagram for executing a property-based test and runner / user barrier. Solid arrows show control-flow transitions between execution states. Please, note that `true*` indicates shrinking does not end if a single *candidate* that is not a *counterexample* is returned, but may employ specific stopping strategies, such as if a preconfigured number of *candidates* upholding the *property* are returned in a row.

4 Implementation

Our implementation is shaped by the protocol developed in Section 3 and by the decision to integrate property-based testing into the existing *Futhark test* infrastructure [The26b]. This section first describes the execution model of *Futhark test*: how each source file is treated as a test unit, how the compiled Futhark program is run as a server process, and how the Haskell runner described in Section 3 communicates with that process through the server API. It then states the seven most important implementation requirements and explains how the implementation integrates *property* tests into *Futhark test*, manages server-side variables, handles failures, validates protocol components, preserves counterexamples, and provides automatic fallback components.

The central runner logic is implemented in `Futhark.Test.Property`, while the surrounding integration extends the existing *Futhark test* machinery for parsing test blocks, discovering property specifications, and reporting results.

4.1 Execution model

The *Futhark test* command is a test runner for Futhark programs. Its execution model is file-oriented: each Futhark source file is parsed as a test specification, compiled or interpreted according to the selected test mode, and tested independently of the other files in the test run. The test runner may schedule several files concurrently, but each file is still treated as a separate test program with its own specification, and its own pass/fail result. The *Futhark test* command is implemented in Haskell, and is responsible for parsing test specifications, compiling the source file, starting the compiled program, and collecting test results. In compiled server mode, the Futhark program being tested runs as a separate server process. The runner communicates with the server process through the Futhark server protocol.

The Futhark server protocol is an interface for interacting with a compiled Futhark server program as a long-running process maintaining a Futhark context. Instead of executing the program once with a fixed input and then terminating, the compiled executable is started in server mode. An external process can send commands, such as storing or freeing values, to this server over standard input and receive responses over standard output [The26d]. In *Futhark test*, this external process is the test runner, while the server process is the compiled Futhark program being tested. This gives the test runner a degree of isolation: if the compiled Futhark program fails, crashes, or is aborted, the runner process may still continue and report the failure. However, this isolation also has a limitation. Values that exist only as server variables are owned by the server process. If the server becomes unavailable, those values can no longer be inspected or recovered through the protocol.

The server protocol also exposes entry points of the compiled Futhark program [The26d]. The runner can for example query which entry points exist, inspect their input and output types, provide input values, call entry points, retrieve results, store and load values to and from files, and free server-side variables. A typical interaction is not a single function call, but a sequence of protocol commands.

A complete interaction through the Haskell server API, corresponding to a sequence of server protocol commands, can have the following form. Given an entry point in a Futhark file like this:

```
entry inc (x: i32) : i32 =
  x + 1
```

Interaction through the server protocol could look like this:

```
-- Put the input value into the server.
FSV.putValue srv "x" val

-- Call the entry point.  This creates the server variable "y".
cmdCall srv "inc" "y" ["x"]

-- Store the result in Futhark's binary data format.
cmdStore srv "output.bin" ["y"]

-- Free the server variables that are no longer needed.
cmdFree srv ["x", "y"]

-- Restore the stored value into a new server variable.
cmdRestore srv "output.bin" [("z", "i32")]

-- Free the last variable
cmdFree srv ["z"]
```

Here, `x`, `y`, and `z` are server variable names. The first operation creates `x` inside the server. The call to `inc` uses `x` as input and creates `y` as output. The `cmdStore` command writes `y` to `output.bin`. The `cmdFree` frees `x` and `y`. The `cmdRestore` command reads that binary file back into the server, creating a new server variable `z` of type `i32`. Then `z` is freed.

This example illustrates the central idea of the server protocol. Values inside the running Futhark program are referred to by server variable names such as `x` and `y`. The external runner does not directly manipulate or inspect the internal representation of these values. Instead, it instructs the server to create, use, store, restore, and free them.

The runner can inspect values whose structures are exposed through the protocol: querying kinds, fields, element types, and shapes, and retrieving primitive values

directly into Haskell. However, it cannot decompose opaque types, since these have no protocol-visible structure.

Futhark test includes parsing of '`-- ==`' comments. For a file containing input-output tests, the runner first reads the embedded test specification and constructs a `ProgramTest`. This specification describes which entry points should be tested, which input datasets should be supplied to them, and which outputs or failures are expected [The26b]. For example, a unit test can be written as follows:

```
-- ==
-- entry: foo
-- input {1i32}
-- output {2i32}
```

The test runner then dispatches the file according to the selected mode. In compiled mode it compiles the file once and reuses the resulting executable to run all tests for that file. The compiled Futhark server is reused for the tests belonging to the same source file.

4.2 Requirements of the implementation

Given the design presented in Section 3 and the execution model, we define the overall implementation requirements as follows.

1. *Property tests* must run as part of *Futhark test*, rather than through a separate command or configuration format. Tests must be visible to the test manager before compilation, to be counted in the same progress reporting as unit-tests.

2. The runner described in Section 3 must use the Futhark server protocol to call user-written protocol components, while keeping orchestration, validation and failure handling separate. As a consequence, the implementation must manage server-resources explicitly. Temporary variables must be freed when they are no longer needed, while variables such as those containing the current *candidate* or *counterexample* must remain available for the duration of the execution, as later phases may need them.

3. The implementation must distinguish between recoverable failures and critical errors. For example, if a *generator*, *property*, or *shrinker* triggers a Futhark runtime error, the runner must report the failure as part of the corresponding *property test*, rather than terminating the entire runner.

4. Protocol entry points must be validated before the main test loop begins. This allows type errors in *properties*, *generators*, and *shrinkers* to be reported before executing *tests*. This means that validation failures must not occur after initial validation.

5. Once a *counterexample* has been found, later failures such as in shrinking must not be able to hide it. The runner must preserve enough information to report, store or reconstruct the *counterexample* even if a later phase fails.

6. The runner must handle non-terminating user code. Since a Futhark entry point may fail to return, the implementation must report which property-testing phase was active when execution timed out.

7. The runner must handle protocol-level non-termination, where the *shrinking session* does not terminate by either storing or reporting enough information to reconstruct the *counterexample*.

4.3 Integrating property-based tests into *Futhark test*

The implementation extends the existing *Futhark test* execution model in two places, following requirement one and two. First, the test specification parser is extended so that a test block may name one or more *properties* to run, using the existing comment '`-- ==`' introduced in Section 3.1.1. These property declarations are represented alongside the input-output tests in the internal test specification:

```
-- ==
-- property: prop_reverse_involution
```

Second, the test suite is expanded to include property-based testing immediately following the standard input-output tests. By using the same server process and compiled executable, these tests skip the need for a separate compilation step or independent command, functioning instead as an extension of the file's primary test cycle.

This integration preserves the file-oriented structure of *Futhark test*. A source file may contain input-output tests, property-based tests, or both. The file is still compiled once and the resulting server is reused for all tests belonging to that file. The property-based tests are then accumulated with the other test results for the file.

A property-based test is counted as one test run. If the property succeeds for the configured number of generated *candidates*, it contributes a passing result. If a *counterexample* is found, it contributes a failing result and reports additional information, including the property name, the size and seed used for generation, and the final *counterexample*. These failures are reported through the same overall mechanism as test failures, so a file containing a failing property test is treated as a failing test program.

The use of explicit property declarations in test blocks allows for *Futhark test* to count the number of test runs a file contains before executing it, in order to display progress correctly. If *properties* were discovered only from compiled entry-point

metadata, then the number of *property tests* would not be known until after compilation. This also supports the final choice from Section 3.1.1: test blocks serve to register which *properties* are being tested, whereas attributes specify the behavior for those *properties*.

4.4 Server resource management

The Futhark server protocol provides the runner with a convenient way to interact with compiled Futhark code, but it also introduces a boundary layer. When the runner creates a variable in the server, that variable remains there until it is explicitly freed. This creates the need for central resource-management expressed in the second requirement listed in Section 4.2: the runner must free variables when they are no longer needed, but it must not free variables while they are still part of the current active protocol call. If variables are not freed, the server accumulates state. If a variable name is reused before its previous binding has been freed, the server instance rejects the operation because the variable already exists.

Our implementation addresses the need for explicit server resource management by assigning each server-side variable a role and a lifetime. A variable is either temporary and must be freed after the operation that created it has finished, or owned and must remain live across several calls. The main owned variable stores the current *candidate* before a failure has been found and the current best *counterexample* after a failure has been found. This variable must remain live while the runner evaluates the *property* and performs shrinking.

The runner uses a fixed set of server variable names for recurring protocol roles, as variables are not allocated freely during execution. Instead, each phase reuses known names after first freeing any previous binding of that name, if needed. The main names are shown in Table 1. The separate prefixes make the variable roles visible in the implementation and reduce the risk of temporary values from one phase accidentally overwriting variables that must remain live in another phase. The important invariant is that every server variable name has a well-defined role and lifetime.

The implementation uses a number of call patterns to preserve this discipline. We encode these patterns as higher-order Haskell functions that automate the necessary cleanup. See Figure 6 for a state chart diagram showcasing variable lifetimes.

Furthermore, for automatic generation and pretty printing of composite *candidates*, the runner may create several intermediate field variables. These variables are not part of the logical test state, and they must not survive beyond the construction that introduced them.

Variable name	Role	Lifetime
`candidate`	Current *candidate* or current *counterexample*.	Owned test-state variable
`size`	Size argument for a user-defined *generator*.	Temporary
`seed`	Seed argument for a user-defined *generator*.	Temporary
`property_ok`	Boolean result of the main *property* call.	Temporary
`shrink_candidate`	Temporary *candidate* proposed by the user-defined shrinker.	Temporary
`shrink_ok`	Boolean result of a *property* call during user-defined shrinking.	Temporary
`auto_shrink_candidate`	Temporary *candidate* proposed by the auto-shrinker.	Temporary
`auto_shrink_ok`	Boolean result of a *property* call during automatic shrinking.	Temporary
`gen_vars`	Temporary variable used to construct a *candidate*.	Temporary
`pretty_vars`	Temporary variables used to pretty-print a *candidate*.	Temporary

Table 1: Main Server variable names and naming schemes used by the runner. Variables with the `shrink_` prefix are used by the shrinking loop, and variables with `auto_shrink_` are variables used by the auto-shrinker.

The resulting invariant is that every server-side variable created by the runner is either a scoped temporary variable or an owned part of the current test state. Temporary variables are freed after use, while owned variables are preserved until they are explicitly replaced or until the test finishes.

The execution model and resource handling discipline described above gives the runner a useful interface to the compiled Futhark program: entry points can be called repeatedly, intermediate values can be kept inside the server, and values can be stored to files when they must survive beyond the current server state. However, this means that failures may occur across a process boundary. A Futhark entry point may fail at runtime, the server may time out or become unavailable, or the runner may violate its own assumptions about server variables. The implementation therefore distinguishes between failures that can be reported as part of a property test and errors that invalidate the current execution state.

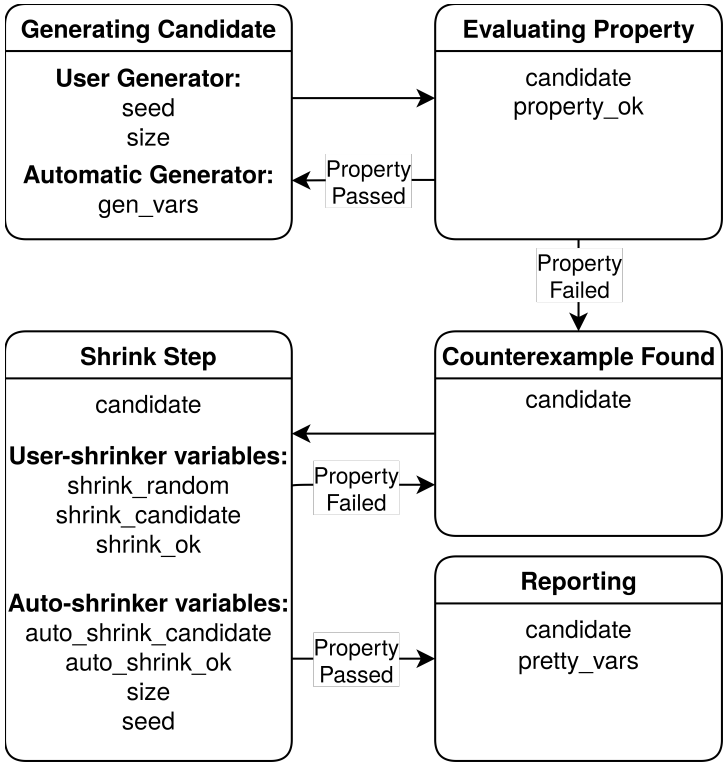

Figure 6: State chart diagram for the property-based testing runner. Each state shows the server variables that are live in that state. The variable `candidate` denotes both *candidates* and *counterexamples* if the property fails.

4.5 Recoverable failures and critical errors

Because the runner executes user-defined Futhark entry points through a separate server process, failures do not all have the same severity. To handle errors gracefully, where possible, the implementation distinguishes between *recoverable failures* and *critical errors*, in accordance with the third requirement listed in Section 4.2. A recoverable failure is a problem that can be attributed to a particular *property test* or one of its associated entry points, while the server remains usable. Such a failure can be reported as part of the test result. Examples include naming a *property* that does not exist, specifying a *generator* with the wrong type, using a *shrinker* whose input type does not match the *property* input type, or triggering a Futhark runtime error, such as division by zero.

A *critical error* is a failure after which the runner can no longer rely on the current server state. Such an error can often be associated with a specific command or entry point. The problem is that the data stored inside the server may no longer be accessible. For example, if the runner sends a command to the server and the server crashes, any *candidate* or *counterexample* that existed only as a server variable may be lost. Critical errors are therefore handled more conservatively than recoverable failures, because continuing the current test file may no longer be possible.

We make this distinction because the runner is responsible for producing useful diagnostics. Recoverable failures should mention the relevant *property*, entry point, expected type, actual type, or runtime error, but still count as a failing *test*. Critical errors should report as much externally recorded context as possible, such as the active *property*, the command or phase being executed, and the size, seed, or shrinker auxiliary value, where available, and should halt execution. The purpose is not only to explain which operation failed, but also to compensate for the fact that server-side values are no longer available, once the server has become unavailable.

4.5.1 Type validation

A category of recoverable errors can be type-mismatches with the protocol, the fourth requirement listed in Section 4.2 is that all protocol entry point types must be validated before generating *candidates*. This means that if the types are checked before the runner proceeds, these errors will not show up in later execution. The *property* must therefore satisfy the protocol types established earlier. If the *property* has type

```
testType -> bool,
```

then an associated *generator* must have type

```
i64 -> u64 -> testType.
```

and if a *shrinker* is provided, it must have type

```
testType -> u64 -> testType.
```

Validation also includes checking that all referenced entry points exist. The *property* configuration is local to the *property* entry point, because the runner reads the `#[prop(...)]` attribute attached to that entry point. This makes the association between *property*, *generator* and *shrinker* explicit, but it also means that misspelled attribute names are detected only by the runner's validation logic. The implementation handles each check independently. A malformed *property test*, for example, one with a type mismatch, is reported as a failure of that specific test case rather than as an immediate failure of the entire test runner.

4.6 Counterexample preservation

The fifth requirement listed in Section 4.2 is that once a *counterexample* has been found, later parts of the execution should not cause it to be lost, as a *counterexample* represents a found bug.

Under the assumption that the server remains available, the runner preserves *counterexamples* by maintaining a simple invariant: after a *counterexample* has been found, the current *counterexample* variable always contains a value that has been checked to falsify the *property*. This invariant is especially important during shrinking. The *shrinker* is allowed to propose new *candidates*, but the runner does not trust these proposals blindly and performs a *test* before accepting it.

This means that a poorly written *shrinker* cannot make the test result incorrect. It may fail to make progress, return *candidates* that are not smaller, return values that no longer fail, or crash. In those cases, the final report may be less useful because the *counterexample* is larger than necessary or not shrunk at all. However, the runner should still report a value that is known to violate the *property*. Thus, shrinking is treated as an optional improvement to reporting, not as part of the logical decision of whether the *property* failed. Just as a *shrinker* cannot change the result of a *test*, recoverable failures also maintains the result of the *test*, by reporting the *counterexample* in the error message, if possible.

The implementation must also handle the critical error of non-termination. An example of non-termination is an entry point that does not terminate. From the runner's perspective, this is similar to a server crash: the current server call never produces a response, and server-side variables are no longer accessible. The sixth requirement listed in Section 4.2 is therefore that the runner must handle non-terminating user code. A *generator* may loop forever while constructing a *candidate*, a *property* may loop forever while *testing*, and the same issue may occur in a *shrinker*.

Another very different example of non-termination is protocol-level non-termination. This problem is different from an entry point not terminating, as the runner can still communicate with the server. The testing loop has a fixed upper bound: the runner generates at most the configured number of *candidates*, and each *candidate* is tested once by calling the *property*. Thus, if no *counterexample* is found, the loop terminates when the test bound is reached. The *shrinking session* is different. Once a *counterexample* has been found, the runner repeatedly calls the *shrinker* to propose a new *candidate*, and then calls the *property* again to check whether that *candidate* is a *counterexample*. Unlike the main testing loop, this process is not naturally bounded. A poorly written *shrinker* may keep proposing accepted *counterexamples*, including the same *counterexample* or *counterexamples* that are larger. This means that protocol-level non-termination is mainly a shrinking-loop

problem. Our implementation must handle this in accordance with the seventh requirement listed in Section 4.2.

The Futhark server protocol provides a way to handle these two kinds of non-termination: values with known Futhark types can be written to disk with `cmdStore` and later reconstructed with `cmdRestore` [The26d]. This gives the runner an escape hatch when a value must survive outside the current server process. However, to do this, the values must be serialized and de-serialized, which for large values can be slow. We discuss two solutions for handling non-termination of these two kinds in Section 7.2, by storing values to disk and restoring them later. But for now, we present a simple solution.

One defense against non-termination critical errors is a timeout. If execution of a test file does not finish within the configured time limit, the runner aborts the current test-file execution and reports a timeout rather than waiting indefinitely. During execution, the runner tracks which *property* is active, which protocol component is being evaluated, and the relevant parameters such as size, seed, or *shrinker* auxiliary value, where applicable. If a timeout occurs, this information can still be reported by the runner, even though the server cannot provide additional information. This provides the user the information necessary to reproduce the error.

Reproducibility is therefore a form of *counterexample* preservation. The runner constructs a deterministic random-number generator from the configured initial seed and draws the concrete seeds and auxiliary random values used during generation and shrinking from this generator. Rerunning with the same initial seed and the same implementation reproduces the same random sequence. This is weaker than storing the concrete value, as it depends on replaying the same execution path, but it is faster than writing every *candidate* to disk. The implementation combines both approaches: it keeps values in the server during normal execution, reports the information needed to reproduce the run where possible, and stores concrete values when a failure report requires a value to survive beyond the current server state.

4.7 Automatic candidate generation

A requirement not stated as a general requirement is that the runner should support a limited form of automatic *candidate* generation. This is used when a *property* does not specify an explicit *generator*. In this case, the runner queries the Futhark server for the input type of the *property* and attempts to construct a *candidate* of that type directly. The generated *candidate* is inserted into the running server as the current *candidate* and can be passed to the *property* in the same way as a *candidate* produced by a user-defined Futhark *generator*. This feature is the closest FutPBT comes to the automatic generation described in Section 2.3, where QuickCheck uses the `Arbitrary` instance of a property argument type to select a generator.

This mechanism reduces boilerplate for simple properties. A user can write a *property* over primitive values, arrays of primitives, or simple composites of automatically generatable types, without also writing a *generator* entry point. The automatic generation does not include domain-specific invariants, such as sortedness, non-empty arrays or valid indices. For such inputs, the user still has to provide an explicit *generator*, or handle it inside of a *property* as a pre-processing step. The purpose of Haskell-side generation is to provide a useful default.

4.8 Conclusion of implementation

The implementation integrates property-based testing into the existing *Futhark test* workflow. Futhark source files remain the unit of testing, and *property tests* are registered through the same test-file mechanism as input-output tests. The runner uses the Futhark server protocol to interact with the compiled program, so that *properties*, *generators* and *shrinkers* can be called as Futhark entry points.

Values live inside the Futhark server and are accessed through named server variables. The runner has to manage these variables explicitly: temporary values are freed after use, while the current *candidate* or *counterexample* remains live across later phases. This resource discipline is what allows the implementation to run many tests without repeatedly moving values between Futhark and Haskell.

Robustness is handled by separating recoverable failures from critical errors. Malformed *property tests*, missing entry points, type mismatches, and Futhark runtime errors are reported as failures of the relevant *property test*, where possible. Non-termination and server-level failures are handled more conservatively, with progress information used to explain where execution was interrupted.

The resulting implementation follows the protocol from Section 3: the user writes the test components in Futhark, while the runner handles orchestration, validation, repeated execution, reporting, and preservation of *counterexamples*.

5 Testing

The implementation has been tested through integration tests. Each test is a Futhark source file executed through the modified *Futhark test* workflow. The testing method is to compare the produced output with an expected output. This means that the tests check the behavior visible to a user of the tool: whether tests pass or fail, *counterexamples* are reported, validation errors are shown and failures are attributed to the correct *property test*. The results are given in Section 5.3.

5.1 Feature coverage

The tests cover primitive numeric values, arrays, records, tuples, and nested composite values across the main features of FutPBT, including both successful and failing *property tests*. The feature tests also include edge cases and failure-oriented examples. These tests are intended to exercise the runner on inputs and declarations that are likely to expose protocol errors.

5.2 Implementation requirements

We also test the implementation requirements described in Section 4. Each of the following subsections presents a requirement listed in Section 4.2 and describes how it is tested, what behavior is expected, and which limitations remain.

5.2.1 Integration with *Futhark test*

This is tested with Futhark files that contain input-output tests, *property tests*, and combinations of both. The expected behavior is that all tests are discovered from the same file, run through the same *Futhark test* command, and summarized in the same pass/fail output.

The tests also check the hybrid comment-and-attribute design, with *property tests* comments and `#[prop(...)]` attributes. The test suite includes cases where one side is missing: a *property test* comment without a corresponding attribute, and an attribute-backed *property* without a corresponding test comment. The expected behavior is that the runner reports a clear diagnostic rather than silently ignoring the mismatch.

This also tests the requirement that declared *property tests* can be counted before execution. Since the selected *properties* are listed in test comments, the test runner can include them in the total number of tests to be reported, even though the full attribute information is only available after compilation and server startup.

5.2.2 Server resource management

This is tested indirectly by running property tests that perform many iterations and create many server-side variables for generated *candidates*, size and seed arguments, boolean results and shrink attempts. The expected behavior is that repeated execution does not fail because of stale variable bindings or name collisions. In particular, the runner should be able to reuse fixed variable names across many generated *candidates* by freeing previous bindings before creating new ones. The shrinking tests also exercise ownership transfer: a proposed shrink result should remain temporary unless it still falsifies the *property*, in which case it is renamed into the current-*counterexample* role. This also tests that output variable names can be reused safely, because the runner frees old bindings before creating new ones.

5.2.3 Recoverable failures and critical errors

These tests check that the resulting report identifies the active *property*, the failing entry point, and the relevant execution context. The test suite includes entry points that deliberately trigger Futhark runtime errors. The expected behavior is that such failures increase the failed-test count and produce diagnostic messages, while still allowing later *property test*s to run when the server remains usable.

5.2.4 Type validation

This is tested with protocol entry points that violate the expected protocol types. The validation tests include incorrect number of inputs, wrong argument types, wrong result types, and mismatches between the *property* input type and the types used by any associated *generators* or *shrinkers*. The expected behavior is that the runner reports a validation failure for the specific malformed *property test*, rather than beginning execution and failing later during generation, shrinking, or printing.

The implementation requirement that *property tests* should be handled independently was tested by placing several *property* declarations in the same file. Some are well-formed, while others are malformed. The expected behavior is that each extracted property specification is validated and reported as its own test result.

5.2.5 Counterexample preservation

This tests that concrete *counterexamples* are stored when a failure report requires the value to survive outside the server. In particular, tests with failing *shrinkers* check that the runner can still report or store a *counterexample* and report the error.

Reproducibility is tested by checking that failure reports after a timeout contains the contextual information needed to rerun a failing case, where possible. The tests check that reports include the active *property*, size, and relevant shrinker auxiliary value, where available.

5.2.6 Automatic candidate generation

This is tested with *properties* over primitive values, arrays of primitive values, records, tuples, and simple nested composites. The expected behavior is that the runner queries the server for the *property* input type, constructs a value of that type on the Haskell side, inserts it into the server, and passes it to the *property*.

5.3 Results

The tests support that the implementation satisfies the main requirements from Section 4. *Property tests* are discovered and run through the modified *Futhark test* workflow, contribute to the same pass/fail result as input-output tests, and support the hybrid design where test comments select *properties* and attributes describe how they should be run. Missing *property test* comments or missing attributes are reported as diagnostics rather than being silently ignored.

The tests also support the server-side execution discipline. Repeated runs, shrinking, and automatic generation do not fail due to stale server variables or name collisions, which gives confidence that temporary variables are freed and that owned variables such as the current *candidate* or *counterexample* remain live when needed.

Failure-oriented tests show that malformed or failing *property tests* are generally reported as recoverable failures. Missing entry points, type mismatches, wrong arities, and runtime errors in user-provided *generators*, *properties* and *shrinkers* produce diagnostics for the relevant test rather than silently succeeding. Timeout tests show that non-terminating entry points are interrupted and reported with phase information when available.

The resource-management tests do not prove that every possible exceptional path frees every temporary variable. They do, however, exercise the main execution paths: generation, property evaluation, shrinking, automatic generation, and runner-side printing. The absence of server variable reuse errors in these tests gives confidence that the implemented variable discipline is sufficient for the tested protocol paths.

The tests support the *counterexample*-preservation requirement. Failing *properties* report *counterexamples*, shrink *candidates* are only accepted after re-running the *property*, and failures in the optional *shrinker* do not hide an already discovered *counterexample*. Automatic generation and shrinking are also exercised for primitive values, arrays of primitives, records, tuples, and simple nested composites.

Overall, the tests do not constitute a full empirical evaluation of bug-finding ability, performance, or usability. They do, however, validate the implementation against its stated requirements. Within that scope, the results are positive.

6 Evaluation

This chapter aims to evaluate FutPBT qualitatively. The goal is not to measure how many bugs the tool finds, nor to show that property-based testing is useful in general. Instead, we evaluate FutPBT as a testing tool for Futhark. We choose two aspects for this. First, the tool must be robust: it must handle malformed test files, failing entry points, non-termination, and optional components that themselves contain errors. Second, the tool must be useful: it should make it practical to write property-based tests for Futhark programs. The evaluation therefore considers robustness as a testing tool, usability of the interface with comparison to QuickCheck, and use on a sample of realistic Futhark code.

6.1 Robustness as a testing tool

Evaluating a testing tool is different from evaluating an ordinary program. FutPBT is expected to execute code that may be erroneous, incomplete, non-terminating, or badly specified. This means that robustness is part of the tool's usability. In this context, robustness means that FutPBT should fail gracefully when encountered with such code. It is not enough for the runner to handle successful properties. It must also provide useful diagnostics when the test file itself is the source of the failure.

This is also how *Futhark test* already treats input-output tests. A test file is not assumed to contain only well-behaved code. Compilation failures, unexpected runtime failures, wrong outputs, missing expected failures, and timeouts are all reported as test results. FutPBT extends this situation by adding more user-provided components that may fail: not only the function being tested, but also *generators* and *shrinkers*.

The server protocol gives the runner a degree of separation from the Futhark program under test. The compiled Futhark program runs as a separate server process, while the runner remains a Haskell process that controls the test execution. If a Futhark entry point reports a runtime error, the runner can often continue and report the error as part of the relevant test. If the server fails and becomes unavailable, the runner may still be able to report which property-testing phase was active, even though the server-side values themselves may no longer be accessible. Future work for increasing robustness is outlined in Section 7.2. The test suite gives some confidence in the robustness of the tool.

The implementation also addresses the robustness by distinguishing recoverable failures from critical errors. Recoverable failures, such as missing entry points, type mismatches, or runtime errors in user-provided entry points, are reported as failures of the relevant *property test*, where possible. Critical errors, such as timeouts or loss of communication with the server, are handled more conservatively

since the current server state may no longer be usable. This distinction is also important for usability: mistakes in the test file should produce actionable diagnostics, while failures that invalidate the server state should not be disguised as ordinary test failures.

The implementation is also robust in its *counterexample* preservation. Once a *counterexample* has been found, FutPBT treats the *property test* as failed independently of whether later optional phases succeed. A user-defined *shrinker* may fail or propose *candidates* that are not *counterexamples*. In this case, the implementation preserves the already discovered *counterexample* and reports it with whatever diagnostic information is still available. Shrinking is therefore robustly isolated from the logical result of the test: it may improve the quality of the report, but it cannot turn a failed *property* into a successful one or hide that a *counterexample* was found.

6.2 Usability of the interface

The main advantage strengthening the usability of FutPBT is that the interface is explicit and simple. In QuickCheck, the user works inside a rich Haskell library. FutPBT has a smaller protocol. This makes the basic model easy to explain and learn.

The default mechanisms make it possible to start writing *property test*s with relatively little supporting code. If no explicit *generator* is provided, the runner can automatically generate *candidates*. If no explicit *shrinker* is provided, the runner can attempt automatic shrinking. These defaults are limited, but they reduce the amount of code required for simple *properties*. Thus, FutPBT requires less library-specific knowledge than QuickCheck.

The cost of this explicit interface appears when the user needs more control. QuickCheck can often infer generation behavior through the `Arbitrary` type class, and users can build on a mature library of generators and shrinkers. As mentioned previously, FutPBT cannot rely on the same mechanisms. Many useful properties require domain-specific inputs, such as sorted arrays, non-empty arrays, valid indices, or related values that satisfy some invariant. In those cases, automatic generation is not enough, and the user must write a custom Futhark *generator*. Furthermore, automatic generation is limited for arrays with more than one dimension. The runner does not recover all source-level size relationships through the server protocol. The auto generator currently generates simple regular shapes, such as square arrays. This is useful as a default, but it is not always representative of the input space. Users who need related dimensions or domain-specific shape constraints must provide an explicit *generator*. Currently, the runner cannot infer sizes of arrays from their type e.g. [5]i32 becomes []i32 in type validation of entry points. This makes sophisticated test data generation more cumbersome than in

QuickCheck. However, the protocol does offer a solution if domain-specific generation is needed, in the form of the *generator*.

Shrinking is another usability difference. QuickCheck shrinkers return a lazy collection of smaller *candidates*, which the testing library can explore incrementally. When a list without any values violating the property is found, shrinking stops. FutPBT instead uses a *shrinker* that proposes one *candidate* per call. This is less expressive, but it fits the server-based execution model better, because values live inside the Futhark server and moving many *candidates* across the boundary would be expensive. The drawback is that the runner has little information about whether progress has actually been made. It checks that every proposed *candidate* still falsifies the *property* before accepting it, but it does not evaluate whether the proposed *candidate* is smaller or even different.

The current stopping condition for user-defined shrinking is a compromise rather than an ideal solution. The runner stops after a configurable number of consecutive *shrink steps* that no longer falsify the *property*. This prevents the *shrinker* from continuing indefinitely once it appears to have stopped finding useful *candidates*. However, it can also stop too early if a useful *counterexample* would have appeared later. Conversely, it does not prevent non-progress when the shrinker keeps producing accepted *counterexamples* that are not actually smaller. A richer or different design might give the runner better control. Such a design would have to be balanced against its complexity and usability.

Counterexample reporting is also less visible in the user interface than in QuickCheck. QuickCheck generally requires tested values to be printable, for example through `Show`, in order to display counterexamples. FutPBT instead lets the runner print values whose structure is exposed through the server protocol, so simple tests do not require the user to define an additional printing mechanism to inspect failing values.

FutPBT also lacks several convenience features found in QuickCheck. It does not yet have a mature library of generators and shrinkers, it does not provide QuickCheck-style classification, coverage, or labelling combinators, and it does not support rich property combinators in the same style as QuickCheck. These limitations do not prevent the core property-based testing loop, but they mean that FutPBT is currently less expressive and less convenient for advanced use.

Overall, FutPBT is usable for simple properties because the core model is small and the runner provides defaults for generation and shrinking. The interface becomes less convenient when the test requires domain-specific generation or custom shrinking, but still allows users to implement their own *generators* and *shrinkers*, if necessary. Compared with QuickCheck, FutPBT trades library-level convenience for an explicit protocol that fits Futhark's server-based execution model.

6.3 Use on realistic Futhark code

Ideally, FutPBT would have been evaluated on a larger collection of existing Futhark programs. Due to time constraints, this was not possible. Most of the tests described in Section 5 are smaller programs designed to exercise specific parts of the protocol. This limits the strength of the evaluation: the test suite gives confidence that the implementation behaves as intended on representative examples, but it does not show how well the tool works across a broad range of existing Futhark code. Usability problems might also not be clear yet, as FutPBT is a new tool, and so far has not been used extensively.

The implementation has nevertheless had some practical use. In a Futhark blog post on counting trailing zeros [Hen26a], the in-progress property-based testing work was used to test a divide-and-conquer function over arrays. The example defines a generator for arrays of `i32` values, attaches it to a *property* with `#[prop(gen(...))]`, and declares the *property* in a test block so it can be run with *Futhark test*. The *property* checks that the computed number of trailing zeros is within bounds, that the corresponding suffix contains only zeros, and that the preceding element, when present, is nonzero.

This example shows that the interface can express a meaningful correctness condition for Futhark functions and existing Futhark programs. It also reveals a usability issue: FutPBT currently does not provide a log or summary of tested *candidates*. A user can see whether a *property test* passed or failed, but not whether the generator explored useful inputs or repeatedly produced similar values. This suggests that *candidate* statistics or optional *candidate* logging would be useful future extensions.

FutPBT was also used on an experimental hashmap implementation[1]. Examples of tested *properties* can be seen in Appendix A. The examples indicate how the tool can be applied to existing Futhark code. The generated *candidates* have the following type:

```
type~ candidate = ?[n][m].{
  keys: [n]i64,
  vals: [n]i32,
  queries: [m]i64,
}
```

A *candidate* consists of key-value pairs to insert into a map, together with query keys used to test membership and lookup behavior. The *generator* constructs unique inserted keys, assigns each key a deterministic value, and creates a query array whose first half contains inserted keys and whose second half contains keys that should be absent from the map. This makes the expected behaviour simple to

[1] https://github.com/diku-dk/futhark-hashmap-experiments

state: inserted keys should be members, looking up an inserted key should return the corresponding value, absent keys should not be members, and looking up an absent key should return `#none`.

The *properties* were implemented for several map implementations: the two-level hashmap, the `u32`-based two-level hashmap, the linear hashmap, and the Eytzinger layout map. In addition to the basic membership and lookup *properties*, the test file checks operations such as `not_member`, `from_array_rep`, `from_array_hist`, `from_array_rep_nodup`, and `adjust`. For example, one *property* checks that `not_member` agrees with negating `member`, while another checks that `adjust` updates existing keys without inserting absent keys. The test file also contains a deliberately false *property*, used only to inspect the generated *candidate*s and confirm what kind of values the *generator* produced.

The *properties* did not reveal any unexpected *counterexamples*. The deliberately false *property* failed as expected, which was useful for inspecting the generated *candidates*. The main result of this case study is not that the hashmap implementation was fully validated. The *generator* is deliberately simple, and the *properties* cover only part of the map interface. Rather, the case study shows that FutPBT can be applied with relatively little effort to an existing Futhark program, and that its interface can express meaningful semantic properties about a realistic data structure.

The overall evaluation is still limited. More use on existing Futhark programs would likely reveal further problems, especially with the default mechanisms. Automatic generation is useful as a starting point, but it cannot infer domain-specific invariants, and its usefulness for realistic programs is uncertain. Automatic shrinking may not find good *counterexamples*. The main mitigation is that the protocol gives the user control. When the defaults are insufficient, the user can write an explicit *generator*, provide a custom *shrinker*, adjust the number of shrinking attempts, or omit shrinking. FutPBT is currently best considered a working foundation, but not yet a mature testing ecosystem comparable to QuickCheck.

7 Future work

The implementation was designed to keep the Futhark-side protocol small while placing orchestration in the runner. This also makes the implementation relatively easy to extend: new runner behavior can often be added without changing the shape of *properties*, *generators*, or *shrinkers*. Several of the limitations identified in Section 6 can therefore be addressed as extensions to the runner rather than as fundamental changes to the protocol. Only the future work proposed in Section 7.6 and Section 7.7 would actually require changing the protocol.

7.1 Shrinker library

A library of reusable *shrinkers*, or automatic shrinking that is not tied to the *generator*, as described in Section 3.3.1, could provide default shrinking for primitive integers, arrays, tuples, records, and common array patterns, such as reducing array length or shrinking individual elements. This would reduce boilerplate and make it easier to write useful *property tests*.

Implementing this in the Haskell runner is difficult as shrinking is less purely type-directed than generation. The runner would have to inspect an existing *counterexample*, construct smaller related *candidates*, and rebuild values inside the server. This is manageable for primitive values, but becomes complicated for arrays and composites. As outlined in Section 3.3, it is not always clear what "smaller" means: shrinking an integer, an array, and a record require different strategies, and domain-specific invariants may have to be preserved.

A Futhark-side library would give users direct access to the structure of their values and could provide reusable shrinking combinators. However, as Futhark does not have type classes or generic programming in the same sense as Haskell, the library could not automatically select a shrinker from the type alone. Users would still have to compose the relevant shrinkers explicitly, and the library would have to fit FutPBT's protocol of returning one proposed *candidate* per shrinker call.

A practical design could combine both approaches: simple runner-side defaults for basic types, and a Futhark library of reusable building blocks for user-defined *shrinkers*.

7.2 Defensive runner

As discussed in Section 4.6, the current implementation keeps *candidates* and *counterexamples* in the Futhark server during normal execution. The runner therefore stores concrete values only when they must survive outside the server, for example when a failure report needs to preserve a *candidate* or *counterexample*.

A possible future extension is a more defensive execution mode. In such a mode, the runner could store every accepted *counterexample* immediately using the server's value-store mechanism. This would make the latest known *counterexample* available outside the server, even if a later *shrinker*, *property*, or server call fails critically. Since the Haskell runner is separate from the Futhark server process, it may still be able to report the last stored *counterexample* even when the server state itself can no longer be inspected. The defensive mode could also attempt to continue investigation from the last *candidate* or *counterexample* stored.

The main drawback is performance. Storing values to disk can be expensive, especially during shrinking, where many accepted *counterexamples* may be produced in sequence. A defensive mode should therefore be optional rather than part of the normal execution strategy. The normal mode favors efficiency by keeping intermediate values in the server and relying on phase information, seeds, sizes, and auxiliary shrinker values for reproducibility when concrete storage is unnecessary. The defensive mode would instead trade performance for stronger preservation guarantees.

The same idea could also be used to guard against protocol-level non-termination in user-defined shrinking. As described in Section 4.6, the problem is not that a single Futhark entry point fails to return, but that the runner may keep accepting *counterexamples* forever. A defensive variant could activate after a time limit or a number of shrink steps. Once active, the runner could store or hash accepted *counterexamples* and compare them with previously seen *counterexamples*. If the same *counterexample* is encountered again, the runner could stop shrinking and report the current *counterexamples* as the final *counterexample* together with the information that the *shrinking session* seems to have encountered a loop.

This would handle *shrinkers* that cycle or repeatedly return the same accepted *counterexample*. It would not prove that shrinking makes progress in general. A *shrinker* may still produce *candidates* that are larger according to the user's intended notion of simplicity while still being *counterexamples*. A defensive runner mode should therefore be understood as a robustness extension, not as a complete solution to all non-progressing *shrinkers*.

7.3 Candidate statistics

Another extension would be to generate test-data statistics. QuickCheck allows users to classify generated inputs or collect information about the distribution of test data. This is useful because a *property* may pass many *tests* when tested on trivial *candidates*. QuickCheck supports this through combinators such as classification and collection, which let the user label test cases and inspect the resulting distribution. FutPBT currently reports whether a *property* passed or failed, but not what kinds of *candidates* were tested. Adding statistics would make it easier to

detect poor *generators*, for example *generators* that rarely produce empty arrays, large arrays, negative numbers, or other relevant edge cases.

One possible implementation would be runner-side inspection. The optional variant of the runner from Section 7.2, which stores or retrieves every generated *candidate*, could be used to compute generic statistics such as array sizes or scalar ranges. It would make the common case slower because every *candidate* would have to be moved out of the server, but could be optional.

Another possibility would be to add an optional classifier entry point. Such an entry point would map a *candidate* to a label or bucket, and the runner would collect a histogram of these labels during testing. This would be closer to QuickCheck's classification mechanism, while still fitting FutPBT's entry-point based design. It would also let the user define domain-specific statistics, such as whether an array is empty, sorted, contains duplicates, or satisfies some precondition. This would require a small extension of the protocol, but would avoid forcing the Haskell runner to understand the structure and meaning of every *testType*.

Future work could also investigate richer *counterexample* explanations. Tools such as Extrapolate show that counterexamples from property-based testing can be generalized to describe a broader class of failing inputs [BR17].

7.4 Stateful property-based testing

The current protocol tests pure Futhark entry points by generating individual *candidates*. This fits functions where a single input can be evaluated independently. Even the stateful hashmap example discussed in Section 6.3 is treated this way: the generated *candidate* contains key-value pairs and queries, and the *property* checks the result of applying a fixed collection of operations to that input. The *candidates* are single generated values, not an interactive sequence of commands.

Stateful property-based testing would instead generate sequences of commands. These commands would describe operations on some stateful object, such as inserting into, deleting from, or querying a data structure. The test would then check whether the implementation behaves like a reference model over the same sequence of commands. Such a protocol would require more components than the current one. First, the runner would need a *command generator*, corresponding to something like `Gen.action`, which generates valid operations rather than ordinary input values. Second, the test would need an initializer for the reference state, or *oracle*. Third, it would need an initializer for the Futhark implementation being tested. The runner would then repeatedly apply each generated command to both states: once to the reference model and once to the Futhark implementation. After each command, or after the whole sequence, the runner would compare the observable results.

The main complication is deciding where the state should live. One possibility is that the Haskell runner stores the reference model and drives the Futhark implementation through repeated server calls, applying one command at a time. This would make the runner responsible for sequencing, state management, and comparison. Another possibility is to encode more of the command execution inside Futhark, but this would require representing commands, states, and observations as Futhark values. Either way, the protocol becomes substantially larger than the one used for pure *property test*s.

Shrinking would also become more difficult. Instead of shrinking a single *counterexample*, the runner would need to shrink a failing command sequence. This could mean removing commands, simplifying individual commands, or shrinking the initial state. As with ordinary shrinking, the runner would have to re-run the sequence after each proposed *candidate* to check that it still exposes the failure. The current shrinker interface is not designed for this kind of structured, sequence-level shrinking. Stateful property-based testing is therefore best understood as a separate extension of FutPBT, rather than a small addition to the existing protocol.

7.5 Conditional properties

QuickCheck supports conditional properties through an implication operator, where a generated input is only tested when it satisfies a precondition [CH00]. This is useful when a property only makes sense for part of the input domain. For example, a property about inserting into an ordered list may require the input list to already be ordered. Inputs that do not satisfy the condition are discarded, and the tool continues searching for valid inputs.

This design is useful, but it also introduces a generation problem. If the precondition is rare, random generation may spend most of its time producing *candidates* that are discarded before the *property* is tested. In such cases, the test may give weak evidence, not because the property was tested and passed, but because the runner failed to find enough relevant inputs. QuickCheck reports this situation when it exhausts too many generated inputs without finding enough valid test cases [CH00].

For FutPBT, one possible design would be to add an optional condition entry point. Such an entry point could have type `testType -> bool`. The runner would first generate a *candidate*, then call the condition entry point, and only call the *property* if the condition returned `true`. This would keep the condition inside Futhark, while leaving the discard logic in the Haskell runner.

The design raises a usability question: in many cases, a specialized *generator* is better than a condition, because it can generate valid *candidates* directly instead

of relying on rejection. However, conditional properties could make automatic generation useful even when the *property* only applies to inputs satisfying certain invariants. Conditional properties are therefore a natural extension, but not a priority.

7.6 Shrinker protocol design

A possible improvement would be to replace the current stopping rule for *shrinkers* with a time budget, for example shrinking for a fixed number of seconds before reporting the best *counterexample* found so far. This could be implemented in the runner without changing the protocol.

A larger change would be to redesign the *shrinker* interface itself. For example, a *shrinker* could return both a proposed *candidate* and a status value, which we already addressed in Section 3. This would however be a change to the protocol.

7.7 Coverage-guided generation

Coverage-guided property-based testing extends the original property-based testing model by using execution feedback to guide input generation. Lampropoulos *et al.* introduce this idea in FuzzChick, where property-based testing is combined with coverage-guided fuzzing [LHP19]. Instead of only generating fresh random inputs, the tool records inputs that reach new parts of the program and mutates them to produce further candidates. This can make testing more effective when interesting behavior is hidden behind narrow input conditions.

For FutPBT, coverage guidance would require a substantially different execution model. The runner would need access to coverage information from the Futhark program or generated backend code, and this information would need to be connected back to *candidate* generation. The *generator* protocol would also need to change. A *generator* that only receives a size and seed cannot react to previous coverage results. A coverage-guided design would instead need a feedback loop where the runner records interesting *candidates* and uses them to influence later generation.

Futhark programs often contain data-dependent control flow, and randomly generated arrays may fail to exercise important paths. Coverage-guided generation could therefore complement user-defined *generators* by helping the runner explore more of the program behavior. The main challenge is that FutPBT currently treats the Futhark program as a server process with entry points, not as a program that exposes coverage information. Adding coverage guidance would require support from the compiler, the server protocol, or the generated backend code.

8 Conclusion

This thesis has investigated how property-based testing can be added to Futhark. The result is FutPBT, a property-based testing tool that allows users to write *properties* and optional *generators* and *shrinkers* as Futhark entry points, while an external Haskell runner handles orchestration and printing. FutPBT is integrated into the existing *Futhark test* workflow and utilizes the Futhark server protocol to execute the user-written components.

The main design is a small protocol adapted to Futhark's execution model. A *property* is a Futhark entry point from a *test type* to `bool`. A *generator* produces *candidates* from a size and seed. A *shrinker* proposes smaller *candidates*. FutPBT connects these components to *Futhark test* through test blocks and attributes. The implementation also validates protocol entry points before execution, manages server-side variables explicitly, distinguishes recoverable failures from critical errors, records phase information for critical errors, and preserves *counterexamples* throughout execution. These mechanisms are important because FutPBT is a testing tool, and must therefore be robust even when user-provided test components are malformed, failing, or non-terminating. Our tests support that the implementation satisfies the main requirements.

The qualitative evaluation suggests that FutPBT is practical for simple and moderately structured Futhark properties. The core interface is explicit and relatively easy to understand, and the runner provides defaults through automatic generation, automatic shrinking, and reporting. However, the evaluation also identifies some limitations. Domain-specific inputs still require explicit Futhark *generators*. The current shrinking protocol is simple, but gives the runner little information about whether progress is being made.

The main conclusion is that property-based testing can be successfully integrated into Futhark, but the approach cannot simply be copied from QuickCheck. Futhark's server-based execution model, explicit entry points, regular arrays, and limited internal control over printing and errors require a different division of responsibility. FutPBT keeps Futhark responsible for pure computations, while the external runner handles orchestration, validation, shrinking, reporting, and robustness around those computations. This gives a workable tool for property-based testing in Futhark, and points to future work: better shrinker support, candidate statistics, defensive execution modes, and broader evaluation on real Futhark programs.

Bibliography

[BR17] BRAQUEHAIS, RUDY and COLIN RUNCIMAN. "Extrapolate: generalizing counterexamples of functional test properties". In: *Proceedings of the 29th Symposium on the Implementation and Application of Functional Programming Languages*. IFL '17. Bristol, United Kingdom: Association for Computing Machinery, 2017. ISBN: 9781450363433. DOI: 10.1145/3205368.3205371. URL: https://doi.org/10.1145/3205368.3205371.

[CH] CLAESSEN, KOEN and JOHN HUGHES. *QuickCheck Manual*. URL: https://www.cse.chalmers.se/~rjmh/QuickCheck/manual.html (visited on 05/21/2026).

[CH00] CLAESSEN, KOEN and JOHN HUGHES. "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. ICFP '00. New York, NY, USA: Association for Computing Machinery, 2000, pp. 268–279. ISBN: 1581132026. DOI: 10.1145/351240.351266. URL: https://doi.org/10.1145/351240.351266.

[Che+18] CHEN, TSONG YUEH, FEI-CHING KUO, HUAI LIU, PAK-LOK POON, DAVE TOWEY, T. H. TSE, and ZHI QUAN ZHOU. "Metamorphic Testing: A Review of Challenges and Opportunities". In: *ACM Comput. Surv.* 51.1 (Jan. 2018). ISSN: 0360-0300. DOI: 10.1145/3143561. URL: https://doi.org/10.1145/3143561.

[Cla26] CLAESSEN, KOEN. *Test.QuickCheck.Arbitrary*. Generated documentation for QuickCheck 2.18.0.0. 2026. URL: https://hackage-content.haskell.org/package/QuickCheck-2.18.0.0/docs/Test-QuickCheck-Arbitrary.html (visited on 05/05/2026).

[EHO18a] ELSMAN, MARTIN, TROELS HENRIKSEN, and COSMIN E. OANCEA. *Parallel Programming in Futhark*. 2018. URL: https://futhark-book.readthedocs.io/en/latest/language.html (visited on 04/27/2026).

[EHO18b] ELSMAN, MARTIN, TROELS HENRIKSEN, and COSMIN E. OANCEA. *Parallel Programming in Futhark*. 2018. URL: https://futhark-book.readthedocs.io/en/latest/index.html.

[Gol+24] GOLDSTEIN, HARRISON, JOSEPH W. CUTLER, DANIEL DICKSTEIN, BENJAMIN C. PIERCE, and ANDREW HEAD. "Property-Based Testing in Practice". In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE '24. Lisbon, Portugal: Association for Computing Machinery, 2024. ISBN: 9798400702174. DOI: 10.1145/3597503.3639581. URL: https://doi.org/10.1145/3597503.3639581.

[Hen17] HENRIKSEN, TROELS. "Design and Implementation of the Futhark Programming Language". PhD thesis. Universitetsparken 5, DK-2100 Copenhagen: University of Copenhagen, Nov. 2017.

[Hen26a] HENRIKSEN, TROELS. *Count trailing zeros*. 2026. URL: https://futhark-lang.org/blog/2026-05-06-count-trailing-zeros.html (visited on 05/10/2026).

[Hen26b] HENRIKSEN, TROELS. *Futhark 0.26.3 released - now with property-based testing*. 2026. URL: https://futhark-lang.org/blog/2026-05-27-futhark-0.26.3-released.html (visited on 05/27/2026).

[Hen26c] HENRIKSEN, TROELS. *How should property-based tests be defined in Futhark?* 2026. URL: https://futhark-lang.org/blog/2026-03-25-property-based-testing.html (visited on 04/29/2026).

[Hug16] HUGHES, JOHN. "Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane". In: *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. Ed. by SAM LINDLEY, CONOR MCBRIDE, PHIL TRINDER, and DON SANNELLA. Cham: Springer International Publishing, 2016, pp. 169–186. ISBN: 978-3-319-30936-1. DOI: 10.1007/978-3-319-30936-1_9. URL: https://doi.org/10.1007/978-3-319-30936-1_9.

[Lam+19] LAMPROPOULOS, LEONIDAS, DIANE GALLOIS-WONG, CATALIN HRITCU, JOHN HUGHES, BENJAMIN C. PIERCE, and LI-YAO XIA. *Beginner's Luck: A Language for Property-Based Generators*. 2019. arXiv: 1607.05443 [cs.PL]. URL: https://arxiv.org/abs/1607.05443.

[LHP19] LAMPROPOULOS, LEONIDAS, MICHAEL HICKS, and BENJAMIN C. PIERCE. "Coverage guided, property based testing". In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019). DOI: 10.1145/3360607. URL: https://doi.org/10.1145/3360607.

[MHC19] MACIVER, DAVID, ZAC HATFIELD-DODDS, and MANY CONTRIBUTORS. "Hypothesis: A new approach to property-based testing". In: *Journal of Open Source Software* 4.43 (Nov. 21, 2019), p. 1891. ISSN: 2475-9066. DOI: 10.21105/joss.01891. URL: http://dx.doi.org/10.21105/joss.01891.

[MRH18] MISTA, AGUSTÍN, ALEJANDRO RUSSO, and JOHN HUGHES. "Branching processes for QuickCheck generators". In: *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*. Haskell 2018. St. Louis, MO, USA: Association for Computing Machinery, 2018, pp. 1–13. ISBN: 9781450358354. DOI: 10.1145/3242744.3242747. URL: https://doi.org/10.1145/3242744.3242747.

[Pik14] PIKE, LEE. "SmartCheck: automatic and efficient counterexample reduction and generalization". In: *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*. Haskell '14. Gothenburg, Sweden: Association for Computing Machinery, 2014, pp. 53–64. ISBN: 9781450330411. DOI: 10.1145/2633357.2633365. URL: https://doi.org/10.1145/2633357.2633365.

[RNL08] RUNCIMAN, COLIN, MATTHEW NAYLOR, and FREDRIK LINDBLAD. "Smallcheck and lazy smallcheck: automatic exhaustive testing for small values". In: *Proceedings of the First ACM SIGPLAN Symposium on Haskell*. Haskell '08. Victoria, BC, Canada: Association for Computing Machinery, 2008, pp. 37–48. ISBN: 9781605580647. DOI: 10.1145/1411286.1411292. URL: https://doi.org/10.1145/1411286.1411292.

[The26a] THE FUTHARK HACKERS. *Futhark 0.26.3 documentation: Compiler Error Index*. DIKU. 2026. URL: https://futhark.readthedocs.io/en/v0.26.3/error-index.html (visited on 05/27/2026).

[The26b] THE FUTHARK HACKERS. *Futhark 0.26.3 documentation:* `futhark-test`. DIKU. 2026. URL: https://futhark.readthedocs.io/en/v0.26.3/man/futhark-test.html (visited on 05/27/2026).

[The26c] THE FUTHARK HACKERS. *Futhark 0.26.3 documentation: Language Reference*. DIKU. 2026. URL: https://futhark.readthedocs.io/en/v0.26.3/language-reference.html (visited on 05/27/2026).

[The26d] THE FUTHARK HACKERS. *Futhark 0.26.3 documentation: Server Protocol — Futhark Documentation*. DIKU. 2026. URL: https://futhark.readthedocs.io/en/v0.26.3/server-protocol.html (visited on 05/27/2026).

[Vok14] VOKES, SCOTT. *Introducing theft: Property-Based Testing for C*. Sept. 2014. URL: https://spin.atomicobject.com/property-based-testing-c/ (visited on 05/16/2026).

[Vri23] VRIES, EDSKO de. "falsify: Internal Shrinking Reimagined for Haskell". In: *Proceedings of the 16th ACM SIGPLAN International Haskell Symposium*. Haskell 2023. Seattle, WA, USA: Association for Computing Machinery, 2023, pp. 97–109. ISBN: 9798400702983. DOI: 10.1145/3609026.3609733. URL: https://doi.org/10.1145/3609026.3609733.

A Hashmap property examples

This appendix shows a representative subset of the property tests used for the experimental hashmap implementation discussed in Section 6.3. The full test file contains the same kinds of properties for several map implementations, including two-level hashmaps, u32-based two-level hashmaps, linear hashmaps, and Eytzinger-layout maps. The examples below show the generated candidate type, the candidate generator, and selected properties for the two-level hashmap implementation.

```
import "lib/github.com/diku-dk/containers/hashmap"
import "lib/github.com/diku-dk/containers/key"
import "lib/github.com/diku-dk/containers/opt"

module two_level_hashmap = mk_hashmap i64key

type~ two_level 't = ?[n].two_level_hashmap.map [n] t

type~ candidate = ?[n][m].{
  keys: [n]i64,
  vals: [n]i32,
  queries: [m]i64,
}

entry gen_candidate (size: i64) (seed: i32) : candidate =
  let n = if size <= 0 then 1 else size
  let m = 2 * n

  let base = i64.i32 seed
  let stride = 2 * (i64.i32 (i32.abs seed % 17) + 1) + 1

  -- Unique inserted keys with seed-dependent spacing.
  let keys =
    tabulate n (\i -> base + stride * i)

  let vals =
    tabulate n (\i ->
      seed + i32.i64 (3 * i + 1))

  -- First half are present keys.
  -- Second half are absent keys between generated key positions.
  let queries =
    tabulate m (\i ->
      if i < n then
        base + stride * i
      else
        base + stride * (i - n) + 1)

  in {keys = keys, vals = vals, queries = queries}
```

Listing 1: Candidate type and generator for hashmap properties.

The generator creates unique inserted keys, assigns each key a deterministic value, and constructs query keys where the first half are present in the map and the second half are absent. This makes it possible to state both positive and negative lookup properties.

```
let all [n] (xs: [n]bool) : bool =
  reduce (&&) true xs

let absent_queries [n][m]
    (c: {keys: [n]i64, vals: [n]i32, queries: [m]i64}) =
  drop n c.queries

let is_none_i32 (x: opt i32) : bool =
  match x
  case #some _ -> false
  case #none -> true

let opt_i32_eq_val (x: opt i32) (v: i32) : bool =
  match x
  case #some x' -> x' == v
  case #none -> false
```

Listing 2: Shared helper functions for hashmap properties.

```
-- ==
-- property: prop_two_level_inserted_keys_are_members

#[prop(gen(gen_candidate))]
entry prop_two_level_inserted_keys_are_members (c: candidate) : bool =
  let hm : two_level i32 =
    two_level_hashmap.from_array_nodup () (zip c.keys c.vals)

  in c.keys
     |> map (\k -> two_level_hashmap.member () k hm)
     |> all
```

Listing 3: Inserted keys are members of the constructed hashmap.

```
-- ==
-- property: prop_two_level_lookup_inserted_keys

#[prop(gen(gen_candidate))]
entry prop_two_level_lookup_inserted_keys (c: candidate) : bool =
  let hm : two_level i32 =
    two_level_hashmap.from_array_nodup () (zip c.keys c.vals)

  let results =
    map (\k -> two_level_hashmap.lookup () k hm) c.keys

  in map2 opt_i32_eq_val results c.vals
     |> all
```

Listing 4: Looking up inserted keys returns their inserted values.

```
-- ==
-- property: prop_two_level_absent_keys_are_not_members

#[prop(gen(gen_candidate))]
entry prop_two_level_absent_keys_are_not_members (c: candidate) : bool =
  let hm : two_level i32 =
    two_level_hashmap.from_array_nodup () (zip c.keys c.vals)

  let absent = absent_queries c

  in absent
     |> map (\q -> !(two_level_hashmap.member () q hm))
     |> all
```

Listing 5: Absent query keys are not members of the hashmap.

```
-- ==
-- property: prop_two_level_lookup_absent_keys

#[prop(gen(gen_candidate))]
entry prop_two_level_lookup_absent_keys (c: candidate) : bool =
  let hm : two_level i32 =
    two_level_hashmap.from_array_nodup () (zip c.keys c.vals)

  let absent = absent_queries c

  in absent
     |> map (\q -> is_none_i32 (two_level_hashmap.lookup () q hm))
     |> all
```

Listing 6: Looking up absent query keys returns #none.

```
-- ==
-- property: prop_two_level_not_member_agrees_with_member

#[prop(gen(gen_candidate))]
entry prop_two_level_not_member_agrees_with_member (c: candidate) : bool =
  let hm : two_level i32 =
    two_level_hashmap.from_array_nodup () (zip c.keys c.vals)

  in c.queries
     |> map (\q ->
          two_level_hashmap.not_member () q hm
          == !(two_level_hashmap.member () q hm))
     |> all
```

Listing 7: The `not_member` operation agrees with negating `member`.

```
-- ==
-- property: prop_two_level_adjust_updates_inserted_keys

#[prop(gen(gen_candidate))]
entry prop_two_level_adjust_updates_inserted_keys (c: candidate) : bool =
  let hm : two_level i32 =
    two_level_hashmap.from_array_nodup () (zip c.keys c.vals)

  let increments =
    map (\_ -> 1i32) c.vals

  let hm_adjusted : two_level i32 =
    two_level_hashmap.adjust (+) 0i32 hm (zip c.keys increments)

  let results =
    map (\k -> two_level_hashmap.lookup () k hm_adjusted) c.keys

  let expected =
    map (+1i32) c.vals

  in map2 opt_i32_eq_val results expected
     |> all
```

Listing 8: The `adjust` operation updates existing keys without changing the key set.