

# halcheck

---

Anthony Vandikas

December 22, 2023

Overview

Summary

# Overview

---

## Why ha1check?

1. First-order(ish) API
2. Support for custom test-case generation strategies
3. Better space complexity

All PBT frameworks are direct ports or descendants of QuickCheck. These frameworks all consist of:

All PBT frameworks are direct ports or descendants of QuickCheck. These frameworks all consist of:

### A central generator data type:

```
--      Source of  
--      randomness ↓  
data Gen a = Gen (Random → a)
```

## halcheck — Overview — API

All PBT frameworks are direct ports or descendants of QuickCheck. These frameworks all consist of:

### A central generator data type:

```
--      Source of  
--      randomness ↓  
data Gen a = Gen (Random → a)
```

### A set of basic combinators:

```
choose      :: (Int, Int) → Gen Int  
suchThat    :: (a → Bool) → Gen a → Gen a  
frequency   :: [(Int, Gen a)] → Gen a  
...
```

All PBT frameworks are direct ports or descendants of QuickCheck. These frameworks all consist of:

### A central generator data type:

```
--      Source of  
--      randomness ↓  
data Gen a = Gen (Random → a)
```

### A set of basic combinators:

```
choose      :: (Int, Int) → Gen Int  
suchThat    :: (a → Bool) → Gen a → Gen a  
frequency   :: [(Int, Gen a)] → Gen a  
...
```

- Users must be comfortable reasoning about higher-order functions.
- Users must ensure generators are only invoked in the **correct context**.



**Example:** Write a generator combinator that produces `std::vectors` up to (but not including) a given length.

```
// RapidCheck
Gen<std::vector<int>> example(int N) {
    return gen::container<std::vector<int>>(
        *gen::inRange(0, N),
        gen::arbitrary<int>);
}

-- QuickCheck
example n = vectorOf (choose (0, n - 1)) arbitrary
```

**Example:** Write a generator combinator that produces `std::vectors` up to (but not including) a given length.

```
// RapidCheck
Gen<std::vector<int>> example(int N) {
    return gen::container<std::vector<int>>(
        *gen::inRange(0, N), // WRONG (no compiler error)
        gen::arbitrary<int>);
}

-- QuickCheck          ↓ WRONG (compiler error)
example n = vectorOf (choose (0, n - 1)) arbitrary
```

**Example:** Write a generator combinator that produces `std::vectors` up to (but not including) a given length.

```
// RapidCheck
Gen<std::vector<int>> example(int N) {
  // WRONG: n is computed once, before calling container
  auto n = *gen::inRange(0, N);
  return gen::container<std::vector<int>>(n, gen::arbitrary<int>());
}
```

```
-- QuickCheck           ↓ WRONG: choose :: (Int, Int) → Gen Int
example n = vectorOf (choose (0, n - 1)) arbitrary
--                       ↑ WRONG: vectorOf :: Int → Gen Int → Gen [Int]
```

**Example:** Write a generator combinator that produces `std::vectors` up to (but not including) a given length.

**Solution:** Delay computation of `*gen::inRange(0, N)` using `gen::exec`.

```
// RapidCheck
Gen<std::vector<int>> example(int N) {
    return gen::exec( [=] {
        return *gen::container<std::vector<int>>(
            *gen::inRange(0, N),
            gen::arbitrary<int>());
    });
}

-- QuickCheck
example n = do
    i <- choose (0, n - 1)
    vectorOf i arbitrary
```

- **Problem:** Need to ensure generators are only invoked in the correct context.
  - Haskell's type system ensures this always happens.
  - C++'s type system can provide no such guarantee!

## halcheck — Overview — API

- **Problem:** Need to ensure generators are only invoked in the correct context.
  - Haskell's type system ensures this always happens.
  - C++'s type system can provide no such guarantee!
- **Solution:** Get rid of the generator type!
  - All code is written in the generator context.
  - Bonus: fewer higher-order functions.

```
// halcheck
std::vector<int> example(int N) {
    return gen::container<std::vector<int>>>(
        gen::range(0, N),
        gen::arbitrary<int>);
}
```

## Why ha1check?

1. First-order(ish) API
2. Support for custom test-case generation strategies
3. Better space complexity

There are various desirable strategies for generating data:

- Random (almost everything)
- Enumerative (SmallCheck/LeanCheck)



There are various desirable strategies for generating data:

- Random (almost everything)
- Enumerative (SmallCheck/LeanCheck)
- Learning-based (RLCheck)
- Coverage-guided (FuzzTest)

There are various desirable strategies for generating data:

- Random (almost everything)
- Enumerative (SmallCheck/LeanCheck)
- Learning-based (RLCheck)
- Coverage-guided (FuzzTest)

Most PBT frameworks (and all C++ PBT frameworks) use a **fixed strategy**.

## ha|check — Overview — Strategies

ha|check provides combinators for specifying strategies:

```
//    random(int) → strategy  
// ↓ Executes random test cases forever or until a bug is found.  
test::random(seed)([] { /* test code */ });
```

## halcheck — Overview — Strategies

halcheck provides combinators for specifying strategies:

```
//    random(int) → strategy
// ↓ Executes random test cases forever or until a bug is found.
test::random(seed)([] { /* test code */ });

//    ordered(strategy) → strategy
// ↓ Executes all test cases in order, from smallest to largest.
test::ordered()([] { /* test code */ });
```

## halcheck — Overview — Strategies

halcheck provides combinators for specifying strategies:

```
// random(int) → strategy
// ↓ Executes random test cases forever or until a bug is found.
test::random(seed)([] { /* test code */ });

// ordered(strategy) → strategy
// ↓ Executes all test cases in order, from smallest to largest.
test::ordered()([] { /* test code */ });

// limit(strategy, int) → strategy
// shrink(strategy) → strategy
// Executes at most 100 random test cases.
// ↓ Performs shrinking if an exception is thrown.
test::shrink(test::limit(test::random(), 100))([] { /* test code */ });
```

## halcheck — Overview — Strategies

halcheck provides combinators for specifying strategies:

```
// random(int) → strategy
// ↓ Executes random test cases forever or until a bug is found.
test::random(seed)([] { /* test code */ });

// ordered(strategy) → strategy
// ↓ Executes all test cases in order, from smallest to largest.
test::ordered()([] { /* test code */ });

// limit(strategy, int) → strategy
// shrink(strategy) → strategy
// Executes at most 100 random test cases.
// ↓ Performs shrinking if an exception is thrown.
test::shrink(test::limit(test::random(), 100))([] { /* test code */ });
```

(Intended for advanced users.)

## Why ha<sub>l</sub>check?

1. First-order(ish) API
2. Support for custom test-case generation strategies
3. Better space complexity

# halcheck — Overview — Space Complexity

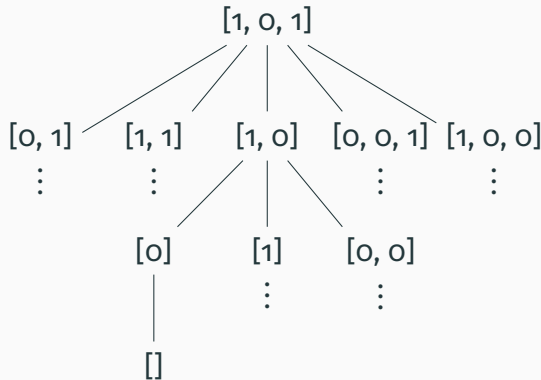
## How does shrinking work?

Internally, every generator is a function returning a “shrink tree” of values.

```
data Gen a = Gen (Random → Tree a)
```

Shrink trees can be **very large** so they must be computed lazily.

## Shrink tree for a list:





# halcheck — Overview — Space Complexity

## How does shrinking work?

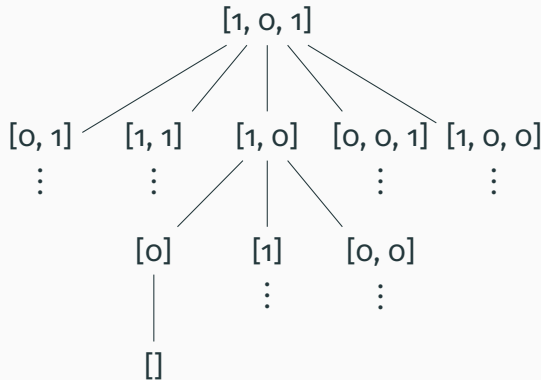
Internally, every generator is a function returning a “shrink tree” of values.

```
data Gen a = Gen (Random → Tree a)
```

Shrink trees can be **very large** so they must be computed lazily.

This implementation strategy **does not work for C++!**

## Shrink tree for a list:



## halcheck — Overview — Space Complexity

### Example:

```
auto xs = *gen::arbitrary<std::vector<int>>>();  
auto x  = *gen::elementOf(xs);
```

## halcheck — Overview — Space Complexity

### Example:

```
auto xs = *gen::arbitrary<std::vector<int>>();  
auto x  = *gen::elementOf(xs);
```

- To shrink  $x$ , RapidCheck must pick a different element of  $xs$ .

## halcheck — Overview — Space Complexity

### Example:

```
auto xs = *gen::arbitrary<std::vector<int>>();  
auto x  = *gen::elementOf(xs);
```

- To shrink  $x$ , RapidCheck must pick a different element of  $xs$ .
- Shrinking is performed *after* the test case has finished ( **$xs$  no longer exists**)!
  - Not a problem in languages with automatic memory management.

## halcheck — Overview — Space Complexity

### Example:

```
auto xs = *gen::arbitrary<std::vector<int>>();  
auto x  = *gen::elementOf(xs);
```

- To shrink  $x$ , RapidCheck must pick a different element of  $xs$ .
- Shrinking is performed *after* the test case has finished ( **$xs$  no longer exists**)!
  - Not a problem in languages with automatic memory management.
- `gen::elementOf` must store a copy of  $xs$  in order to avoid creating a dangling reference.

## halcheck — Overview — Space Complexity

### Example:

```
auto xs = *gen::arbitrary<std::vector<int>>();  
auto x  = *gen::elementOf(xs);
```

- To shrink  $x$ , RapidCheck must pick a different element of  $xs$ .
- Shrinking is performed *after* the test case has finished ( **$xs$  no longer exists**)!
  - Not a problem in languages with automatic memory management.
- `gen::elementOf` must store a copy of  $xs$  in order to avoid creating a dangling reference.

**Conclusion:** all combinators (with shrinking behaviour) must make copies of their arguments!

# halcheck — Overview — Space Complexity

## Example:

```
auto xs = *gen::arbitrary<std::vector<int>>();  
auto x  = *gen::elementOf(xs);
```

- To shrink  $x$ , RapidCheck must pick a different element of  $xs$ .
- Shrinking is performed *after* the test case has finished ( **$xs$  no longer exists**)!
  - Not a problem in languages with automatic memory management.
- `gen::elementOf` must store a copy of  $xs$  in order to avoid creating a dangling reference.

**Conclusion:** all combinators (with shrinking behaviour) must make copies of their arguments!

**Problem:** by default, copies in C++ are deep ( $\mathcal{O}(n)$  instead of  $\mathcal{O}(1)$ ).

### Generators cannot return references:

```
// Generates a random reference  
// to an element of xs.  
rc::Gen<int &> referenceOf(??? xs);  
//           What goes here? ↑  
  
// Example: assign a  
// random element to 0.  
*referenceOf(xs) = 0;
```

What type should `referenceOf` have?



## ha<sub>l</sub>check — Overview — Space Complexity

ha<sub>l</sub>check is inspired by work on **internal shrinking**.

- Motto: shrink inputs, not outputs!
- Data is recomputed, never copied.

## halcheck — Overview — Space Complexity

halcheck is inspired by work on **internal shrinking**.

- Motto: shrink inputs, not outputs!
- Data is recomputed, never copied.

**Note:** halcheck does **not** use internal shrinking.

- Users have full control over shrinking.

## Why ha<sub>l</sub>check?

1. First-order(ish) API
2. Support for custom test-case generation strategies
3. Better space complexity

## Summary

---

## New strategies:

- ordered (SmallCheck/LeanCheck)
- Coverage-guided (fuzztest)
  - Requires support for mutations.
- Learning-based (RLCheck)
- Reproducing test-cases

## Test framework integration:

- Google Test
- CUnit
- doctest (partially done)