

halcheck

Anthony Vandikas

December 22, 2023

Overview

Design

Summary

Overview

Why halcheck?

1. Clearer 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:

A central generator data type:

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

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::vector`s shorter than 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::vector`s shorter than 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` shorter than a given length.

```
// RapidCheck
Gen<std::vector<int>> example(int N) {
  // WRONG: n is computed once, all generated values have the same size!
  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` shorter than 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!

- **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 halcheck?

1. Clearer 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**.

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 */ });
```

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

```
//  limit(strategy, int) → strategy
// ↓ Executes at most 100 random test cases.
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
// ↓ Executes all test cases in order, from smallest to largest.
test::ordered()([] { /* test code */ });
```

```
//  limit(strategy, int) → strategy
// ↓ Executes at most 100 random test cases.
test::limit(test::random(), 100)([] { /* test code */ });
```

(Intended for advanced users.)

Why halcheck?

1. Clearer API
2. Support for custom test-case generation strategies
3. Better space complexity

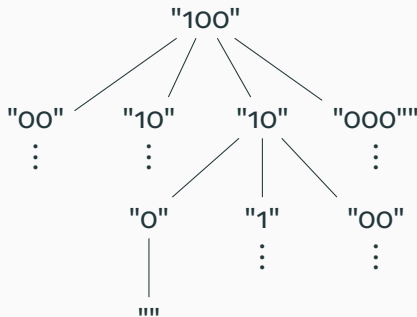
How does shrinking work?

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:



How does shrinking work?

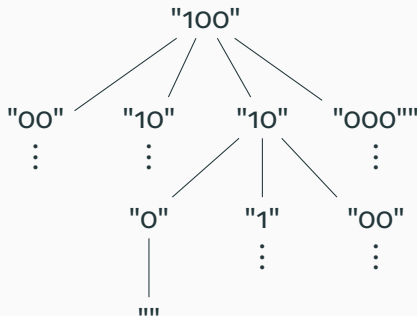
Internally, every generator is a function returning a “shrink tree” of values.

```
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:



Example:

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

Example:

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

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

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.

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.

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!

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?

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

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

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 halcheck?

1. Clearer API
2. Support for custom test-case generation strategies
3. Better space complexity

Design

Goal: Simple API with few higher-order functions and no generator type.

```
template<typename T>  
T gen::arbitrary();
```

```
template<typename T>  
T gen::range(T min, T max);
```

```
template<typename T>  
T &gen::element(std::vector<T> &container);
```

```
template<typename T, typename F>  
T gen::container<T>(size_t size, F gen);
```

Goal: Simple API with few higher-order functions and no generator type.

```
template<typename T>  
T gen::arbitrary();
```

```
template<typename T>  
T gen::range(T min, T max);
```

```
template<typename T>  
T &gen::element(std::vector<T> &container);
```

```
template<typename T, typename F>  
T gen::container<T>(size_t size, F gen);
```

Motto:

Just write functions!

Goal: Simple API with few higher-order functions and no generator type.

```
bool gen::next(int w0, int w1);
```

Goal: Simple API with few higher-order functions and no generator type.

```
bool gen::next(int w0, int w1);
```

- Returns `true` with probability $\frac{w_1}{w_0 + w_1}$ (w_0 and w_1 are relative weights).

Goal: Simple API with few higher-order functions and no generator type.

```
bool gen::next(int w0, int w1);
```

- Returns `true` with probability $\frac{w_1}{w_0 + w_1}$ (w_0 and w_1 are relative weights).
- This is the **only** source of randomness — all generators are built from this function!

Goal: Simple API with few higher-order functions and no generator type.

```
bool gen::next(int w0, int w1);
```

Example:

```
int gen::range(int min, int max) {  
    while (min + 1 < max) {  
        auto mid = std::midpoint(min, max);  
        if (gen::next(mid - min, max - mid))  
            max = mid;  
        else  
            min = mid; }  
    return min; }
```

Users expect more!

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

Users expect more!

- Discards

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

Users expect more!

- Discards
- Sized generation

```
data Gen a = Gen
  ( Random
  → Size
  → Maybe a
  )
```

Users expect more!

- Discards
- Sized generation
- Distribution logging

```
data Gen a = Gen
  ( Random
  → Size
  → Maybe (a, [Label])
  )
```

Users expect more!

- Discards
- Sized generation
- Distribution logging
- Shrinking

```
data Gen a = Gen
  ( Random
  → Size
  → Maybe (Tree a, [Label])
  )
```

Users expect more!

- Discards
- Sized generation
- Distribution logging
- Shrinking

```
data Gen a = Gen
  ( Random
  → Size
  → Maybe (Tree a, [Label])
  )
```

What **functions** do we need to provide to support these features?

Users expect more!

- Discards
- Sized generation
- Distribution logging
- Shrinking

```
data Gen a = Gen
  ( Random
  → Size
  → Maybe (Tree a, [Label])
  )
```

What **functions** do we need to provide to support these features?

- `void gen::discard()`
(throws exception)

Users expect more!

- Discards
- Sized generation
- Distribution logging
- Shrinking

```
data Gen a = Gen
  ( Random
  → Size
  → Maybe (Tree a, [Label])
  )
```

What **functions** do we need to provide to support these features?

- `void gen::discard()`
- `int gen::size()`

Users expect more!

- Discards
- Sized generation
- Distribution logging
- Shrinking

```
data Gen a = Gen
  ( Random
  → Size
  → Maybe (Tree a, [Label])
  )
```

What **functions** do we need to provide to support these features?

- `void gen::discard()`
- `int gen::size()`
- `void gen::label(std::string)`

Users expect more!

- Discards
- Sized generation
- Distribution logging
- Shrinking

```
data Gen a = Gen
  ( Random
  → Size
  → Maybe (Tree a, [Label])
  )
```

What **functions** do we need to provide to support these features?

- | | |
|--|---|
| • <code>void</code> <code>gen::discard()</code> | • <code>int</code> <code>gen::size()</code> |
| • <code>void</code> <code>gen::label(std::string)</code> | • ??? |

```
std::optional<int> gen::shrink(int size = 1)
```

- `gen::shrink(n)` returns an `int i` if shrinking should occur at the call site, and `std::nullopt` otherwise.
- Intuitively, `i` is the *index* of the child element to shrink to.

```
std::optional<int> gen::shrink(int size = 1)
```


Example:

```
std::string gen_string() {  
    auto xs = /* random string */;  
    for (size_t i = 0; i < xs.size();)  
        if (auto c = gen::shrink(2))  
            if (*c == 0) xs.erase(i);  
            else xs[i] = '\0';  
    else  
        i++;  
    return xs; }
```

Explanation:

```
std::optional<int> gen::shrink(int size = 1)
```

Example:

```
std::string gen_string() {
    auto xs = /* random string */;
    for (size_t i = 0; i < xs.size(); )
        if (auto c = gen::shrink(2))
            if (*c == 0) xs.erase(i);
            else xs[i] = '\0';
        else 
            i++;
    return xs; }
```

Explanation:

1. Normally, `gen::shrink(n)` always returns `nullopt`, so `xs` is unmodified.

```
std::optional<int> gen::shrink(int size = 1)
```

Example:

```
std::string gen_string() {
    auto xs = /* random string */;
    for (size_t i = 0; i < xs.size(); i++)
        if (auto c = gen::shrink(2)) {
            if (*c == 0) xs.erase(i);
            else xs[i] = *c;
        }
    return xs; }
```

Explanation:

1. Normally, `gen::shrink(n)` always returns `nullopt`, so `xs` is unmodified.
2. When shrinking, `gen::shrink(n)` returns an integer $c < n$.


```
std::optional<int> gen::shrink(int size = 1)
```

Example:

```
std::string gen_string() {
    auto xs = /* random string */;
    for (size_t i = 0; i < xs.size(); i++)
        if (auto c = gen::shrink(2))
            if (*c == 0) xs.erase(i); ←
            else xs[i] = '\0';
        else
            i++;
    return xs; }
```

Explanation:

1. Normally, `gen::shrink(n)` always returns `nullopt`, so `xs` is unmodified.
2. When shrinking, `gen::shrink(n)` returns an integer $c < n$.
 - 2.1 If $c = 0$: remove current element.

```
std::optional<int> gen::shrink(int size = 1)
```

Example:

```
std::string gen_string() {
    auto xs = /* random string */;
    for (size_t i = 0; i < xs.size(); i++)
        if (auto c = gen::shrink(2))
            if (*c == 0) xs.erase(i);
            else xs[i] = '\0';
    return xs; }
```

Explanation:

1. Normally, `gen::shrink(n)` always returns `nullopt`, so `xs` is unmodified.
2. When shrinking, `gen::shrink(n)` returns an integer $c < n$.
 - 2.1 If $c = 0$: remove current element.
 - 2.2 If $c = 1$: shrink current element to 0.

- `gen::shrink` provides a minimal interface for defining shrinking behaviour.
- Not meant to be used directly!
- Other combinators (e.g., Hedgehog's `shrink`) can be built using `gen::shrink`.

But `gen::shrink` has a problem!

Example:

gen::next → true, false, true

```
int x = 0;  
if (gen::next() && !gen::shrink())  
    x += gen::next() ? 1 : 0;  
x += gen::next() ? 1 : 0;
```

Explanation:

Example:

gen::next → true, false, true

```
int x = 0;  
if (true && !gen::shrink())  
    x += false ? 1 : 0;  
x += true ? 1 : 0;
```

Explanation:

- Suppose a test case failure occurs.

Example:

gen::next → true, false, true

```
int x = 0;  
if (true && false)  
    x += false ? 1 : 0;  
x += true ? 1 : 0;
```

Explanation:

- Suppose a test case failure occurs.
- Then !gen::shrink() subsequently evaluates to false.

Example:

gen::next → true, false, true

```
int x = 0;  
if (true && false)  
    x += false ? 1 : 0;  
x += true ? 1 : 0;
```

Explanation:

- Suppose a test case failure occurs.
- Then !gen::shrink() subsequently evaluates to false.
- Final call to gen::next should return the same value (true).

Example:

gen::next → true, false, true

```
int x = 0;  
if (true && false)  
    x += false ? 1 : 0;  
x += true ? 1 : 0;
```

Problem:

- halcheck is just a library. It can't inspect the internal structure of a program!
- Internally, halcheck only saves and replays the results of gen::next.

Example:

gen::next → true, false, true

```
int x = 0;  
if (true && false)  
    x += gen::next() ? 1 : 0;  
x += true ? 1 : 0;
```

Problem:

- halcheck is just a library. It can't inspect the internal structure of a program!
- Internally, halcheck only saves and replays the results of gen::next.
- Second call to gen::next is never evaluated.

Example:

gen::next → true, false, true

```
int x = 0;  
if (true && false)  
  x += gen::next() ? 1 : 0;  
x += false ? 1 : 0;
```

Problem:

- halcheck is just a library. It can't inspect the internal structure of a program!
- Internally, halcheck only saves and replays the results of gen::next.
- Second call to gen::next is never evaluated.
- Final call to gen::next returns next available value (false)!

Example:

gen::next → [true, false], true

```
int x = 0;
{
  auto _ = gen::group();
  if (gen::next() && !gen::shrink())
    x += gen::next() ? 1 : 0;
}
x += gen::next() ? 1 : 0;
```

Solution: gen::group

- gen::group informs halcheck that all subsequent calls to gen::next (and gen::shrink) should be treated as a single “step”.

Example:

gen::next → [true, false], true

```
int x = 0;
{
  auto _ = gen::group();
  if (true && !gen::shrink())
    x += true ? 1 : 0;
}
x += true ? 1 : 0;
```

Solution: gen::group

- gen::group informs halcheck that all subsequent calls to gen::next (and gen::shrink) should be treated as a single “step”.
- Evaluation before shrinking is unchanged.

Example:

gen::next → [true, false], true

```
int x = 0;
{
    auto _ = gen::group();
    if (true && false)
        x += gen::next() ? 1 : 0;
}
x += true ? 1 : 0;
```

Solution: gen::group

- gen::group informs halcheck that all subsequent calls to gen::next (and gen::shrink) should be treated as a single “step”.
- Evaluation before shrinking is unchanged.
- Final call to gen::next returns the proper value even after shrinking!

When to use `gen::group`?

- Number of calls to `gen::group`, `gen::next`, and `gen::shrink` (excluding calls within a `gen::group`) must be **constant**.
- Simple heuristic: wrap all conditionals and loops in a `gen::group`.
 - Future work: automatic `gen::group` insertion via instrumentation.
- Incorrect grouping will *not* break tests, only cause inefficient shrinking.

- All generators are implemented using the six (and counting) core functions:
 - `gen::next`
 - `gen::discard`
 - `gen::size`
 - `gen::label`
 - `gen::shrink`
 - `gen::group`
- How are the core functions themselves implemented?

- All generators are implemented using the six (and counting) core functions:
 - `gen::next`
 - `gen::discard`
 - `gen::size`
 - `gen::label`
 - `gen::shrink`
 - `gen::group`
- How are the core functions themselves implemented? **It depends!**
- Behaviour differs depending on strategy (e.g., random vs exhaustive).
- Sometimes users need to override behaviour (e.g., disable shrinking).
- `halcheck` lets you **override** core functions!

halcheck's core functions are implemented using a primitive system of **effect handlers**.

halcheck's core functions are implemented using a primitive system of **effect handlers**.

- Every core function f is an *effect*.

halcheck's core functions are implemented using a primitive system of **effect handlers**.

- Every core function f is an *effect*.
- A *handler* h for f can be *installed* using `auto _ = f.handle(h)`.

halcheck's core functions are implemented using a primitive system of **effect handlers**.

- Every core function f is an *effect*.
- A *handler* h for f can be *installed* using `auto _ = f.handle(h)`.
- Handlers are *uninstalled* when the return value (`_`) goes out of scope.

halcheck's core functions are implemented using a primitive system of **effect handlers**.

- Every core function f is an *effect*.
- A *handler* h for f can be *installed* using `auto _ = f.handle(h)`.
- Handlers are *uninstalled* when the return value (`_`) goes out of scope.
- Effects are *invoked* via the call operator ($f()$) and behave according to the last handler that was installed.

halcheck's core functions are implemented using a primitive system of **effect handlers**.

- Every core function f is an *effect*.
- A *handler* h for f can be *installed* using `auto _ = f.handle(h)`.
- Handlers are *uninstalled* when the return value (`_`) goes out of scope.
- Effects are *invoked* via the call operator ($f()$) and behave according to the last handler that was installed.
- Effects are **lexically scoped**: any effects invoked within an effect handler behave as if they were invoked *just before the effect handler was installed*.

Example: Overriding `gen::next`

```
CHECK_THROWS(gen::next()); // Throws by default
{
  //   ↓ gen::next calls the lambda as long as this is in scope
  auto _ = gen::next.handle([](int x, int y) {
    CHECK_THROWS(gen::next()); // Calling gen::next within the handler
                                // invokes the previous behaviour.

    return x == 0 && y > 0;
  });
  CHECK_EQ(gen::next(0, 1), true);
}
CHECK_THROWS(gen::next()); // Original behaviour restored
```

Summary

- Every generator in `halcheck` is built from a small set of first-order core functions, resulting in a “direct-style” interface.
- Every core function can be overridden.
 - Enables custom generation strategies (e.g., random vs exhaustive).
 - Enables local modifications to generator behaviour (e.g., disabling shrinking).
- Users must sometimes annotate functions with `gen : : group` to get proper shrinking behaviour.

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)