

halcheck

Anthony Vandikas

December 22, 2023

Motivation

Design

Summary

Motivation

Why halcheck? (and why not RapidCheck)?

1. **Simpler API**
- 2.
- 3.

Most PBT frameworks are direct ports of QuickCheck or Hedgehog.

These frameworks all consist of:

Most PBT frameworks are direct ports of QuickCheck or Hedgehog.

These frameworks all consist of:

A central generator data type:

```
--           Source of  
--           ▽ randomness  
data Gen a = Gen (Random → Tree a)  
--   Tree of shrunk values ▴
```

Most PBT frameworks are direct ports of QuickCheck or Hedgehog.

These frameworks all consist of:

A central generator data type:

```
--           Source of
--           ▽ randomness
data Gen a = Gen (Random → Tree a)
--   Tree of shrunk values ▴
```

A set of basic combinators:

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

Most PBT frameworks are direct ports of QuickCheck or Hedgehog.

These frameworks all consist of:

A central generator data type:

```
--           Source of
--           ▽ randomness
data Gen a = Gen (Random → Tree a)
--   Tree of shrunk values ▴
```

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 know the distinction between *generator code* and *non-generator code*.

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

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

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

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

Problem: `*gen::inRange(0, N)` is evaluated *before* `example` returns, resulting in a generator that always produces `std::vectors` of the same length.

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

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

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

- Generators should only be invoked (**operator** *) in the correct context.
- Haskell's type system completely rules out this kind of error.
- C++'s type system is not powerful enough!
- Problem goes away if we simply *get rid of the generator type and just write first-order functions!*

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

Why ha1check? (and why not RapidCheck)?

1. Simpler API
2. **Support for test-case generation strategies**
- 3.

There are various desirable strategies for generating data:

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

Most PBT frameworks (including RapidCheck) use a **fixed strategy**.

halcheck provides combinators for specifying strategies:

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

halcheck provides combinators for specifying strategies:

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

//  limited(strategy, int) → strategy
// ▽ Executes at most 100 test cases.
test::limited(test::random(), 100)([] { run_test(); });
```


halcheck provides combinators for specifying strategies:

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

//  limited(strategy, int) → strategy
// ▽ Executes at most 100 test cases.
test::limited(test::random(), 100)([] { run_test(); });

//  shrinking(strategy) → strategy
// ▽ Performs test-case shrinking if a bug is found.
test::shrinking(test::random())([] { run_test(); });
```

Why halcheck? (and why not RapidCheck)?

1. Simpler API
2. Support for test-case generation strategies
3. **Better space complexity**

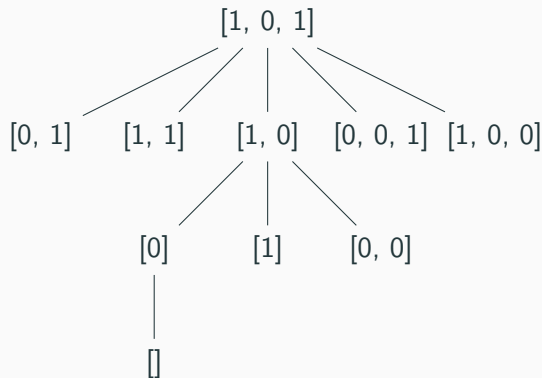
How does shrinking work?

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

Internally, every generator returns a "shrink tree" of values.

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

Shrink tree for a list:



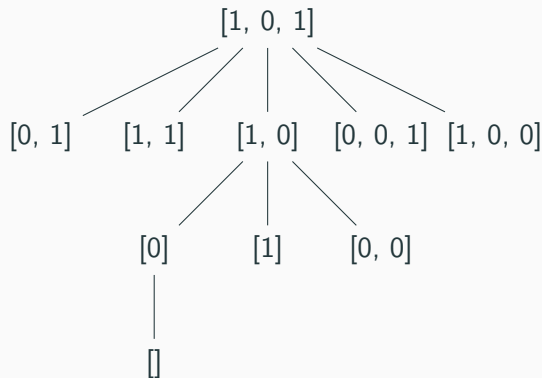
How does shrinking work?

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

Internally, every generator returns a "shrink tree" of values.

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

Shrink tree for a list:



This implementation strategy does not work for C++!

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);
```

- Shrinking x requires picking a different element of xs.

Example:

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

- Shrinking x requires picking 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);
```

- Shrinking `x` requires picking 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);
```

- Shrinking `x` requires picking 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 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);
```

```
// Example: assign a  
// random element to 0.  
*referenceOf(xs) = 0;
```

What type should referenceOf have?

Generators cannot return references:

```
// Generates a random reference  
// to an element of xs.  
rc::Gen<int &> referenceOf(??? xs);
```

```
// Example: assign a  
// random element to 0.  
*referenceOf(xs) = 0;
```

What type should referenceOf have?

exec arguments must capture by value:

```
auto bound = *gen::arbitrary<int>();  
auto ranges = *gen::container<...>(  
  gen::exec([&] {  
    // bound does not exist  
    // during shrinking!  
    auto min = *gen::inRange(0, bound);  
    auto max = *gen::inRange(min, bound);  
    return std::make_pair(min, max);  
  }));
```

- halcheck **never** makes copies of generated data.
- halcheck **only** records decision made during the generation process.
- All the previous examples work in halcheck without any unnecessary copying!

Why ha1check? (and why not RapidCheck)?

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

Why ha1check? (and why not RapidCheck)?

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

Design

Goals:

- Support user-defined generation strategies
- First-order API (as much as possible)

Goals:

- Support user-defined generation strategies
- First-order API (as much as possible)
- C++ 11 support
- Thread safety

Goals:

- Support user-defined generation strategies
- First-order API (as much as possible)

To support these goals, every generator needs to be built from a set of **overridable core functions**:

Goals:

- Support user-defined generation strategies
- First-order API (as much as possible)

To support these goals, every generator needs to be built from a set of **overridable core functions**:

- `gen::next`
- `gen::discard`
- `gen::shrink`
- `gen::group`

```
gen::next(int x = 1, int y = 1) → bool
```

Intuition: `gen::next(x, y)` performs a biased coin flip with $p = \frac{y}{x+y}$.

- This is our *only* source of randomness.
- Is this enough to define all generators?

```
gen::next(int x = 1, int y = 1) → bool
```

Example 1: *generate a number in the range $[a, b)$.*

`gen::next(int x = 1, int y = 1) → bool`

Example 1: *generate a number in the range $[a, b)$.*

- Solution: use binary search, replace comparison with biased coin flip.

```
int range(int a, int b) {  
    if (a + 1 >= b)  
        return a;  
  
    int c = std::midpoint(a, b);  
    if (gen::next(c - a, b - c))  
        return range(mid, b);  
    else  
        return range(a, mid);  
}
```

```
gen::next(int x = 1, int y = 1) → bool
```

Example 2: *generate a `enum dim`.*

```
enum class dim {  
    x = 0,  
    y = 3,  
    z = 8  
};
```

```
gen::next(int x = 1, int y = 1) → bool
```

Example 2: *generate a **enum dim**.*

- Solution 1: consult `gen::range` to determine which value to return.

```
enum class dim {
    x = 0,
    y = 3,
    z = 8
};

dim arbitrary(gen::tag<dim>) {
    switch (gen::range(0, 3)) {
        case 0: return dim::x;
        case 1: return dim::y;
        case 2: return dim::z;
    } }
}
```



```
gen::next(int x = 1, int y = 1) → bool
```

Example 2: *generate a **enum dim**.*

- Solution 1: consult gen::range to determine which value to return.
- Solution 2: define gen::element (implemented via gen::range).

```
enum class dim {
    x = 0,
    y = 3,
    z = 8
};
dim arbitrary(gen::tag<dim>) {
    return gen::element({x, y, z});
}
```

```
gen::next(int x = 1, int y = 1) → bool
```

Example 3: *generate a **struct** **vec**.*

```
struct vec {  
    int x;  
    int y;  
};
```

```
gen::next(int x = 1, int y = 1) → bool
```

Example 3: *generate a struct vec.*

- Trivial.

```
struct vec {  
    int x;  
    int y;  
};  
  
vec arbitrary(gen::tag<vec>) {  
    vec output;  
    output.x = gen::arbitrary<int>();  
    output.y = gen::arbitrary<int>();  
    return output;  
}
```

```
gen::next(int x = 1, int y = 1) → bool
```

Example 4: *generate a std::string.*

```
gen::next(int x = 1, int y = 1) → bool
```

Example 4: *generate a `std::string`.*

- Attempt: generate random size n , then generate n elements.

```
std::string gen_string() {  
    std::string xs;  
    auto size = gen::arbitrary<size_t>();  
    while (size-- > 0)  
        xs.push_back(gen::arbitrary<char>());  
    return xs;  
}
```

```
gen::next(int x = 1, int y = 1) → bool
```

Example 4: *generate a `std::string`.*

- Attempt: generate random size n , then generate n elements.
- Problem:
`gen::arbitrary<size_t>` is uniformly distributed — high chance of generating *billions of GBs*.

```
std::string gen_string() {  
    std::string xs;  
    auto size = gen::arbitrary<size_t>();  
    while (size-- > 0)  
        xs.push_back(gen::arbitrary<char>());  
    return xs;  
}
```

```
gen::next(int x = 1, int y = 1) → bool
```

Example 4: *generate a `std::string`.*

- Attempt: add elements as long as `gen::next` returns `true`.

```
std::string gen_string() {  
    std::string xs;  
    while (gen::next())  
        xs.push_back(gen::arbitrary<char>());  
    return xs;  
}
```

```
gen::next(int x = 1, int y = 1) → bool
```

Example 4: *generate a `std::string`.*

- Attempt: add elements as long as `gen::next` returns `true`.
- Problem: distribution biased towards shorter lists.

```
std::string gen_string() {  
    std::string xs;  
    while (gen::next())  
        xs.push_back(gen::arbitrary<char>());  
    return xs;  
}
```


Problem: how do we define `gen_string` with a sensible distribution?

Problem: how do we define `gen_string` with a sensible distribution?

Standard Solution: define an incrementally increasing size parameter and choose lengths uniformly up to the current size.

```
std::string gen_string() {  
    std::string xs;  
    auto size = gen::range(0, gen::size());  
    while (size-- > 0)  
        xs.push_back(gen::arbitrary<char>());  
    return xs;  
}
```

Problem: how do we define `gen_string` with a sensible distribution?

Standard Solution: define an incrementally increasing size parameter and choose lengths uniformly up to the current size.

```
std::string gen_string() {  
    std::string xs;  
    auto size = gen::range(0, gen::size());  
    while (size-- > 0)  
        xs.push_back(gen::arbitrary<char>());  
    return xs;  
}
```

- `gen::size` cannot be defined in terms of `gen::next`.
- Do we really need an extra combinator?

Problem: how do we define gen_string with a sensible distribution?

Standard Solution: define an incrementally increasing size parameter and choose lengths uniformly up to the current size.

```
std::string gen_string() {
    std::string xs;
    auto size = gen::range(0, gen::size());
    while (size-- > 0)
        xs.push_back(gen::arbitrary<char>());
    return xs;
}
```

- gen::size cannot be defined in terms of gen::next.
- Do we really need an extra combinator? **NO!**

```
gen::next(gen::weight x = 1, gen::weight y = 1) → bool
```

- Actual definition of `gen::next` takes `gen::weight` parameters.
- A `gen::weight` is a symbolic value parameterized by an unknown size.
- A `gen::weight` is semantically a function in $\mathbb{N} \rightarrow \mathbb{N}$.
- **Bonus:** generators automatically behave uniformly with respect to size!

```
std::string gen_string() {  
    std::string xs;  
    auto size = gen::weight::current; // current is the symbolic variable  
    while (gen::next(1, size--))  
        xs.push_back(gen::arbitrary<char>());  
    return xs;  
}
```

- Proof of correctness left as an exercise to the reader 😊

Summary: `gen::next` alone allows us to write generators for

- weighted distributions,
- **enums** and **unions** (i.e., sum types),
- **structs** (i.e., product types), and
- recursive data structures via sized generation.

`gen::discard()`

Intuition: `gen::discard()` terminates the current test case (no failure).

- Used to implement precondition checking (`RC_PRE`).
- Also useful for implementing rejection sampling (`gen::suchThat`).

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

Intuition: `gen::shrink(n)` returns an integer if shrinking should occur at the call site, or `std::nullopt` otherwise.

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

Example:

```
std::string gen_string_shrink() {  
    auto xs = gen_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;  
}
```

Behaviour:

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

Example:

```
std::string gen_string_shrink() {  
    auto xs = gen_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;  
}
```

Behaviour:

1. `gen::shrink(n)` returns `std::nullopt` until a test case failure occurs.

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

Example:

```
std::string gen_string_shrink() {  
    auto xs = gen_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;  
}
```

Behaviour:

1. `gen::shrink(n)` returns `std::nullopt` until a test case failure occurs.
2. On subsequent test cases, `gen::shrink(n)` returns an integer $< n$ indicating which value to shrink to.

`gen::group()`

Example:

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

- Suppose a test case failure occurs with `a == b == true` and `c == false`.
- Then `!gen::shrink()` subsequently evaluates to `false` and `b` is not evaluated.
- We expect that `c` should still evaluate to `false`.

gen::group()

Example:

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

- **Problem:** halmcheck is just a library. It cannot inspect the internal structure of a program!
- Internally, halmcheck only saves and replays the results of gen::next.
- Sequence 1:
{a: true, b: true, c: false}
- Sequence 2:
{a: true, c: true, [unused]: false}

`gen::group()`**Example:**

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

Intuition: `gen::group` informs `halcheck` that all subsequent calls to `gen::next` (and `gen::shrink`) should be treated as a single "step".

- Sequence 1:
{{a: true, b: true}, c: false}
- Sequence 2:
{{a: true, [unused]: true}, c: false}

- All generators are implemented using the core functions.

- All generators are implemented using the core functions.
- How are the core functions themselves implemented?

- All generators are implemented using the core functions.
- How are the core functions themselves implemented? **It depends!**
- Core functions can be **overridden**.

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
```

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

- `gen::next`, `gen::discard`, etc. are *effects*.

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

- `gen::next`, `gen::discard`, etc. are *effects*.
- Effects are *invoked* via the call operator (`effect()`).

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

- `gen::next`, `gen::discard`, etc. are *effects*.
- Effects are *invoked* via the call operator (`effect()`).
- A *handler* can be *installed* using `auto _ = effect.handle(my_handler)` is *uninstalled* when the return value (`_`) goes out of scope.

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

- `gen::next`, `gen::discard`, etc. are *effects*.
- Effects are *invoked* via the call operator (`effect()`).
- A *handler* can be *installed* using `auto _ = effect.handle(my_handler)` is *uninstalled* when the return value (`_`) goes out of scope.
- Effects are **lexically scoped**: any effects invoked within an effect handler behave as if they were invoked *just before the effect handler was installed*.
 - This property is what makes strategies composable.

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

- `gen::next`, `gen::discard`, etc. are *effects*.
- Effects are *invoked* via the call operator (`effect()`).
- A *handler* can be *installed* using `auto _ = effect.handle(my_handler)` is *uninstalled* when the return value (`_`) goes out of scope.
- Effects are **lexically scoped**: any effects invoked within an effect handler behave as if they were invoked *just before the effect handler was installed*.
 - This property is what makes strategies composable.
- Effects are **thread local**.

Summary

- Every generator in `halcheck` is built from four first-order core functions, enabling a "direct-style" API.
- User-defined generation strategies are supported: every core function can be overridden.
- Shrinking is performed by changing generator inputs (`gen::shrink`) instead of generator outputs, resulting in less memory usage.