Introduction to `halcheck`

Anthony Vandikas

`halcheck` is a property-based testing framework for C++. Property-based testing is a form of testing that focuses on executing *properties* instead of specific test-cases. For example, a typical unit test for `std::sort` might check that the output of `std::sort` is exactly the sorted version of some specific input:

```cpp
HALCHECK_TEST("example") {
  using T = std::pair<int, int>;

  std::vector<T> xs = {{3, 1}, {3, 2}, {1, 1}};

  auto cmp = [](T x, T y) { return x.first < y.first; };
  std::sort(xs.begin(), xs.end(), cmp);

  REQUIRE_EQ(xs, std::vector<int>{{1, 1}, {3, 1}, {3, 2}});
}
```

Unfortunately, this test is brittle: `std::sort` is not guaranteed to be a stable sort, so a future version of this function could return the values `{{1, 1}, {3, 1}, {3, 2}}` instead. A less brittle version of this test would check a more generic *property*, such as the sortedness of the output.

```cpp
  // REQUIRE_EQ(xs, std::vector<int>{{1, 1}, {3, 1}, {3, 2}});
  REQUIRE(std::is_sorted(xs.begin(), xs.end(), cmp));
```

Since the expected result in this new test is not hard-coded, it will require no future modifications. Furthermore, the test no longer depends on the specific value assigned to `xs`. This allows us to easily write a *parameterized test* (using `SUBCASE` in `doctest`).

```cpp
  std::vector<T> xs;
  SUBCASE("empty")    { xs = {}; }
  SUBCASE("sorted")   { xs = {{1, 1}, {3, 1}, {3, 2}}; }
  SUBCASE("unsorted") { xs = {{3, 1}, {3, 2}, {1, 1}}; }
```

But why write test cases when you can generate them? With `halcheck`, you can randomly generate appropriate inputs for your tests and achieve greater test coverage with less code.

```
4    using namespace halcheck;
5    auto xs = gen::arbitrary<std::vector<T>>();
```

The final example is as follows:

```
1  using namespace halcheck;
2
3  HALCHECK_TEST("example") {
4    using T = std::pair<int, int>;
5
6    auto xs = gen::arbitrary<std::vector<T>>();
7
8    auto cmp = [](T x, T y) { return x.first < y.first; };
9    std::sort(xs.begin(), xs.end(), cmp);
10
11   REQUIRE(std::is_sorted(xs.begin(), xs.end(), cmp));
12 }
```

By default, `halcheck` will generate random test-cases until a failure is found or 100 test-cases have passed.

A property-based test can be decomposed into two components: a **property** to test, and **generators** that provide test-cases. The property in the previous example is expressed by lines 7-10: after sorting, `xs` should be sorted. The generator in the previous example appears on line 5: we use `halcheck`'s built-in generator for producing arbitrary `std::vector`s.

`halcheck` is unopinionated about how users express properties. Instead, `halcheck` provides tools for test-case generation, including a library of data-generation functions and a library of *strategies* for controlling the quantity and order of test-cases. To illustrate the latter feature, the upper limit on successful test-cases can be removed by changing the third line, causing the test to repeat endlessly:

```
HALCHECK_TEST("example", test::random()) {
  ...
}
```

(The default behaviour is explicitly written as `test::limit(test::random())`.)