Axiom based testing

Ali Alnajjar
Supervisor: Magne Haveraaen

Axiom-Based Testing

Axiom-based testing from concepts has two main parts:

- **axioms**, in the form of conditional equations, and
- suitable **test data points**.

Axioms

Multiple equations combined by logical *and*

The sides of the equations are full
C++ expressions

```
concept Monoid<typename T>
    : Semigroup<T> {
  T op(T, T);
  T identity_element();
  axiom Identity(T x) {
    op(x, identity_element()) == x;
    op(identity_element(), x) == x;
} }
```

The Oracle Problem

- equality operator used in a concept axiom is **not implemented**

- **behavioural equivalence**

  two values are considered equal if they cannot be distinguished by any operation in the system.

From Axioms to Test Code

There are two steps involved in generating tests from concepts.

- generate a **test oracle** (function having the same parameters)for each axiom.
- generate **test cases** for each type

From Axiom to Test Oracle

```
  axiom ArrayEqual(A a, A b, I i) {
    if (a == b)
      a[i] == b[i];
  }
}
```

```
template <typename A,typename I,typename E>
requires Indexable<A, I, E>
struct Indexable_oracle
{
  static bool ArrayEqual(A a, A b, I i)
  {
    if (a == b)
      if (!(a[i] == b[i]))
        return false;
    return true;
  }
};
```

```
template <int size,typename E>
requires Indexable<ArrayFI<size, E>, FiniteInt<size>, E>
struct Indexable_testCase<ArrayFI<size, E>, FiniteInt<size>, E>
{
  static void ArrayEqual() {
    typedef HasDataSet<ArrayFI<size, E>>::dataset_type dt_0;
    dt_0 b_0 = HasDataSet<ArrayFI<size, E>>::get_dataset();
    for (DataSet<dt_0>::iterator_type a_0 = DataSet<dt_0>:: begin(b_0)
       ; a_0 != DataSet<dt_0>::end(b_0); ++a_0) {
      typedef HasDataSet<ArrayFI<size, E>>::dataset_type dt_1;
      dt_1 d_0 = HasDataSet<ArrayFI<size, E>>::get_dataset();
      for (DataSet<dt_1>::iterator_type c_0 = DataSet<dt_1>:: begin(d_0)
         ; c_0 != DataSet<dt1>::end(d_0); ++c_0) {
        typedef HasDataSet<FiniteInt<size>>::dataset_type dt_2;
        dt_2 f_0 = HasDataSet<FiniteInt<size>>::get_dataset();
        for (DataSet<dt_2>::iterator_type e_0 = DataSet<dt_2>:: begin(f_0)
           ; e_0 != DataSet<dt_2>::end(f_0); ++e_0)
        check(Indexable_oracle<ArrayFI<size, E>,
                               FiniteInt<size>, E>::ArrayEqual(*a_0,*c_0, *e_0),
            "Indexable", "ArrayEqual");
} } } };
```

Provide Data for a Free Variable

- The parameter has a known, primitive C++ type.
- The parameter has a known, user-defined type.
- The parameter type is a template argument to the test oracle.

Provide Data for a Free Variable

- The parameter has a known, primitive C++ type.
  (random generator library)
- The parameter has a known, user-defined type.
  (provided a test data generation interface
- The parameter type is a template argument to the test oracle.
  (concept maps)

Provide Data for a Free Variable

1. User selected data sets.
2. Randomly chosen generator terms.
3. Randomly chosen data structure values.

```
if (a == b && b == c) a == c;
```