An OCL Map Type

Edward D. Willink

Willink Transformations Ltd, Reading, England,
ed_at_willink.me.uk

Lightning presentation at the 19th International Workshop in OCL and Textual Modeling, September 16, 2019, Munich, Germany.

1 An OCL Map Type - E.D.Willink

OCL's family of `Collection` types is well known, but a `Map(K,V)` type is missing. Some distinguished authors have suggested that the deficiency can be remedied by a `Set(Tuple(K, V))`, but this is clearly misguided since a `Set(Tuple)` can have many different-valued entries for the same key, whereas a map can only have one value per key.

The Java Map type is very familiar and might perhaps inspire an equally familiar library type for OCL, but this too is misguided since a Java Map is mutable while an OCL `Map` should be immutable.

The ordered `Collections` are therefore a better source of inspiration. The following have obvious functionality:

`=`, `<>`, `isEmpty()`, `notEmpty()`, `size()`.

New `keys()` and `values()` operations can access the two halves of the map.

Similarly obvious functionality with respect to the keys can be provided by:

`excludes(k)`, `excludesAll(c)`, `excluding(k)`,

`excludingAll(c)`, `includes(k)`, `includesAll(c)`

Further emulation of ordered Collections suggests that `at(k)` accesses the map at index `k`. `including(k,v)` creates a new map with an additional or replacement `k<-v` binding. It returns `null` for a null value and `invalid` for a missing value.

Richer support can be provided by:

`excludesMap(m)`, `excludesValue(v)`, `excludingMap(m)`,

`includesMap(m)`, `includesValue(v)`, `includingMap(m)`

Construction of a `Map` literal can re-use the `Tuple` literal syntax in conjunction with a new binding operator `<-`. Thus a map literal with two entries for two value to key bindings my be created by:

`Map{k1<-v1, k2<-v2}`

The above facilities were prototyped in Eclipse OCL 2015-06. They provide an adequate ability to use a `Map` but prove very inefficient for Map construction since creating a Map with `N` entries requires progressive construction of `N-1` intermediate maps; the execution performance is therefore at best quadratic. The Eclipse OCL 2019-03 release therefore extends the create/operation `Map` functionality with iteration support.

A `Map` is treated as a set of keys each with a bound co-value. All the standard `Collection` iterations apply to `Map`s using the set of keys as the iteration domain. Additionally a co-iterator may be bound to the iterator to avoid the need to invoke `at(k)` to obtain the value of each key. Thus a map can be checked to ensure that each `v` bound to its `k` iterator is equal to the squared value of the iterator.

```
let c : Map(Integer, Real) = ... in c->forAll(k<-v | v = k*k)
```

A new `collectBy` iterator, that may be used on `Collection`s or `Map`s, supports creation of a `Map` by collecting an expression value for each iterator key. A map from ten integer values to their squares may be built by:

```
Sequence{1..10}->collectBy(k | k*k)
```

Future work might generalize `k<-v` from special purpose punctuation to an expression operator. The downside of this generalization is the cost/complexity of a new `Entry(K,V)` type for the new expression result and of course many new operations to allow an `Entry` type to be used sensibly. The upside is that the `collectBy` body may use let variables and may compute both key and value. A map that binds an integer value to its string value could be built by:

```
Sequence{1..10}->collectBy(k | k.toString()<-k)
```

A consequence of permitting both key and value to be computed, is that, in general, uniqueness of the key values cannot be guaranteed. To avoid non-deterministic loss of colliding values, the `Map` parameterization would need to be a multimap: `Map(K,Bag(V))`.