A NEW LOOK AT MULTIMAPS

Robbert van Dalen
robbert.van.dalen@gmail.com

Abstract

Ordinary maps must contain only single occurrences of their domain values, while multimaps may contain multiple occurrences of both domain and range values.
In this short article we relax the occurrence type to be of any type - if such type obeys some kind of number law.
Because of this relaxation and because multimaps behave like numbers, multimaps can be the occurrence type of other multimaps.
We explore this idea with some sketchy code snippets and finish with a recursive multiple of nothing.

Definitions

We consider taking the intersection, union and difference between two multimaps. We define these operations on top of an abstract **number type** which is defined by the following scala trait:

```
trait Num[N <: Num[N]] {
 def add(o: N): N
 def mul(o: N): N
 def max(o: N): N
 def min(o: N): N
}
```

Each pair in a multimap also holds an associated number of occurrences. The following scala trait defines such **triple**:

```
trait Triple[A,B,N <: Num[N]] {
 def domain: A
 def range: B
 def occurrence: N
 def create(d: A, r: B, n: N): Triple[A,B,N]
}
```

A triple is the basis to create a **multimap**. Two disjoint multimaps can be joined together to create bigger multimaps. Triples of a multimap can be iterated through- and matched with other triples. The following scala trait sketches a multimap interface:

```
trait MMap[A,B,N <: Num[N]] {
 def join(o: MMap[A,B,N]): MMap[A,B,N]
 def match(d: A, r: B): Triple[A,B,N]
 def iterator: Iterator[Tripe[A,B,N]]
 def empty: MMap[A,B,N]
 def create(p: Triple[A,B,N]): MMap[A,B,N]
}
```

When two multimaps are combined, all their triples are matched one-to-one and combined in order to create new multimaps. During iteration, triples have their respective occurrences combined using abstract number operations.

Intersect

We demonstrate the **intersect** operation on multimaps given the `Int` occurrence type. The intersect operation iterates through two argument multimaps, taking the minimum occurrence of two matching triples.
Here is a naive and sketchy scala implementation:

```
type M[A,B] = MMap[A,B,Int]

def intersect(m1: M[A,B], m2: M[A,B]): M[A,B] =
{
 val r1 = intersec2(m1,m2) // guarantee full
 val r2 = intersec2(m2,m1) // coverage
 intersect2(r1,r2)
}

def intersec2(m1: M[A,B], m2: M[A,B]): M[A,B] =
{
 var result = m1.empty
 var mi = m1.iterator

 while (mi.hasNext)
 {
  val t1 = mi.next
  val t2 = m2.match(t1.domain,t1.range)

  val o1 = t1.occurrence
  val o2 = t2.occurrence

  val nc = o1.min(o2)

  val nt = t1.create(t1.domain,t1.range,nc)

  val nr = m1.create(nt)
  result = result.join(nr)
 }
 result
}
```

Note that by replacing `o1.min(o2)` with **`o1.max(o2)`** we get the **union** of two multimaps.

Union as maximum

We rename union to be the maximum and intersect to be the minimum of two multimaps. Maximum is denoted by `|` and minimum by `&`.

Additionally, we define the multiplication`(*)` of two multimaps to be the **multiplication** of the occurrences of their matching triples. Similarly, addition`(+)` is done by the **addition** of occurrences.

Syntax

We start with triples that have the following syntax:

```
occurrence:domain=range
```

When a triple's domain and range values are equal, one of them is dropped (together with the equals symbol). In case of a single occurrence, the occurrence prefix is dropped.

```
2:3=3  =>  2:3
1:4=5  =>  4=5
1:6=6  =>  6
```

A multimap is a list of triples, syntactically separated by commas and enclosed within square brackets:

```
[1:1=1,2:3=3,4:5=6]  =>  [1,2:3,4:5=6]
```

As a further syntactical convenience, all triples in a multimap that share the same occurrence are replaced by a single occurrence. After that, the shared occurrence is prefixed to the multimap.

```
[5:1=1,5:3=3,5:6=7]  =>  5:[1,3,6=7]
```

The empty multimap is denoted by `0`.

Canonical representation

A canonical syntactic representation has all its triples fully written out. Next to that, triples that match the same domain-range pairs, are consolidated into a single occurrence.

```
5:[1,1,6:4=5]  =>  [10:1=1,30:4=5]
```

Arithmetics

Following our defined semantics and syntax we are ready to do some multimap arithmetics.
For convenience however, we fix the domain and range values to be equal so that our examples collapse into **multiset** arithmetics.

```
0 + 6:[1,2]          =>   6:[1,2]
0 * 6:[1,2]          =>   0
0 | 6:[1,2]          =>   6:[1,2]
0 & 6:[1,2]          =>   0

4:[1,2] + 6:[1,2]    =>   10:[1,2]
4:[1,2] * 6:[1,2]    =>   24:[1,2]
4:[1,2] | 6:[1,2]    =>   6:[1,2]
4:[1,2] & 6:[1,2]    =>   4:[1,2,3]

4:[1,2] + 6:[2,3]    =>   4:[1] + 10:[2] + 6:[3]
```

Multimaps as occurrences

Because multimaps implement all arithmetic operations, we may choose them to be occurrences themselves. Of course, such pervasive recursion can quickly become unwieldy. We believe however that such recursion adds more semantic power to multimaps.
Nevertheless, we are not entirely sure what are the use-cases. We only show two examples to give an idea:

```
[1]:[3,4] + [2]:[3,4] => [1,2]:[3,4]
[1]:[3,4] + [2]:[4,5] => [[1]:3,[1,2]:4,[2]:5]
```

Numbers as multimaps

We allow recursion not to stop at numbers, because we define numbers to be multimaps that only contain multiple pairs of empty multimaps.

```
0       ==   0
[0]     ==   1
[0,0]   ==   2
[3:0]   ==   3
5:[0]   ==   5
```

Higher-order multimaps

Because range and domain values are also allowed to be multimaps, we can create higher-ordered multimaps. Even if we only allow `0` as an atom (which has type **nothing**) we can still built very sophisticated multimaps with just multiples of `0`, recursively.
We are even free to intermix numbers or any other higher-ordered multimaps which are solely built on top of nothing. Here are two examples:

```
[1=2,3,4] + 3           =>  [3:0,1=2,3,4]
[1,2]:[3,4] + 5:[3,4]   =>  [5:0,1,2]:[3,4]
```

Negative occurrences

Note that we deliberately did not consider occurrences to have **sign** - until now. Of course, nothing stops us from doing so. We might even consider **rational** occurrences, but we leave that venture open for future research.

Related work

"A new look at multisets" [Wildberger, 2003] was very influential. In this article, Wildberger's ideas on multisets are extended to multimaps - and more.

Discussion

A new look at multimaps was discussed but the implementation was rather poorly done.
A better implementation can be realized with the use of uniquely represented, confluently persistent treaps. The benefit of such implementation is that all multimap operations can be made efficient [van Dalen, 2012]

References

N J Wildberger, 2003. *A new look at multisets*
School of Mathematics UNSW Sydney 2052 Australia

R. van Dalen, 2012. *Finite higher-ordered binary relations*
http://github.com/rapido/spread/blob/master/binary_relations_2012.pdf