Modification Theory

by Sven Nilsen, 2017

When you have two functions like these:

$$f : A \to \text{bool}$$
$$g : B \times A \to A$$

One function `f` describes a constraint of an object. The function `g` modifies objects of type `A` according to some information `B` (e.g. inserting a zero at location `b`).

You can construct a function that modifies objects only if the modified object satisfies the constraint:

```
modif := \(f : A → bool, g : B × A → A) = \(b : B, a : A) = if f(g(b, a)) { g(b, a) } else { a }
```

The `modif` function has the property that:

```
modif(f, modif(f, g)) <=> modif(f, g)
```

Often you pose the extra constraint on `g` such that it has the property of the `modif` function:

```
modif(f, g) <=> g
```

A such function is cheaper to compute than constructing it with `modif`, so this is almost always assumed when using modification theory. Finding cheap modification functions is a sub-problem of the field of modification theory.

The following is true for all such functions:

```
∀ a, b { f(a) => f(g(b, a)) }
```

Naturally, the space of modifications that are possible among all functions is very large. Still, the idea that you can modify objects while preserving some property is very interesting. The concept is very general, since it can be constructed for any two functions of the corresponding types, so you find some related idea whenever there is a problem containing such functions. Basically, it means that modification theory can be applied almost everywhere. One particular study of modification theory is the distinguishing of further properties that modification functions have. Either they got the property, or they do not, and this way the problems can be classified according to the view of modification theory.

Very often, the type of `B` is supporting a notion of iterative modifications, such that one does not have to write a loop when reasoning about modifications. Alternatively, it can be constructed with `moditer`:

```
moditer := \(g : B × A → A) = \(bs: [B], a : A) = {
    ga := a
    for i { ga = g(bs[i], ga) }
    ga
}
```

A modification function has “teeth” if the following is true:

```
teeth := \(g : B × A → A) = ∃ a₀, a₁ { ∀ b { g(b, a₀) == a₀ ∧ g(b, a₁) == a₁ ∧ a ¬= b } }
```

For example, the following function lacks teeth:

```
moditer(g) : [teeth] false

g := \(k : (> 1), x) = if (x % k) == 0 { x / k } else { x }
```

The reason is that if you can divide by anything larger than 1, then you eventually will end up with the number 1. All numbers can be modified until any modification leads to a change, and then you will end up with the same object. So, this kind of modification as no teeth.

Here is another example of a modification function that got teeth:

```
moditer(modif(f, g)) : [teeth] true

f := \(a : [nat]) = ∑ i { a[i] } > 0
g := \(i : nat, a : [nat]) = {
    a[i] = 0
    a
}
```

If you have a list of 3 numbers, e.g. `[a : (> 0), b : (> 0), c : (> 0)]`, it can be modified to:

```
[a, 0, 0]
[0, b, 0]
[0, 0, c]
```

These lists can not be modified further without breaking the constraint, but they are not equal to each other. If we changed the constraint to allow a sum of 0, it looses the teeth:

```
moditer(modif(f, g)) : [teeth] false

f := \(a : [nat]) = ∑ i { a[i] } >= 0
g := \(i : nat, a : [nat]) = {
    a[i] = 0
    a
}
```