ToiletPaper #123

λf. (λx. f (x x))(λx. f (x x))

Author: Andreas Swoboda / Software Engineer / Business Division Banking & Insurance

✗ Problem

In the recent past, C++ (since C++11), Java (since version 8) and many other languages have been extended by "lambdas". In practice, this is a shortened notation to define an anonymous class with a functional interface and at the same time instantiate an object of this class. But how to call an anonymous method recursively?!?

✓ Solution

Even though it is possible to create lambdas à la std::function<void()> f = [f]() { /* ... */ f(); }; in C++, it only works with an additional indirection (e.g. via std::function) and not inline (e.g. as parameter). For Java, it does not look any better. A more elegant way is to use the "Y combinator" (you might search the internet for "fixed-point combinator"): You replace the recursive function with a higher-order function that calls its parameter instead of itself. You put this function into the Y combinator, which repeatedly calls the function with itself as parameter.
And what does this mysterious combinator look like? Here are possible implementations in C++ (for any number of parameters of any type) and Java (with currying for the function and an additional parameter):

C++

```
template <typename F>
class Y {
  F f;
public:
  constexpr Y(F f) : f(std::forward<F>(f)) {}
  template <typename...Ts>
  constexpr decltype(auto) operator()(Ts&&...ts) {
    return f(*this,
        std::forward<Ts>(ts)...);
  }
};
```

Java

```
class Y<T, R> implements Function<T, R> {
  private Function<Function<T, R>,
          Function<T, R>> f;
  public Y(Function<Function<T, R>,
           Function<T, R>> f) {
    this.f = f;
  }
  public R apply(T t) {
    return f.apply(this).apply(t);
  }
}
```

→ Example

Let's have a look at the standard example for recursion, the factorial. As a function (in C++), and as a lambda in C++17/Java, it looks like this:

C++ Function

```
int f(int x) {
  return x>0 ? x*f(x-1)
             : 1;
}
```

C++17 Lambda

```
Y([](auto f, int x) -> int {
  return x>0 ? x*f(x-1)
             : 1;
})
```

Java 8 Lambda

```
new Y<Integer, Integer>(
  f -> x ->
    x>0 ? x*f.apply(x-1)
        : 1
)
```

The trailing return type is unfortunately needed in C++, because otherwise the compiler triggers "auto type deduction", which leads to a cyclic dependency.

+ Further Aspects

- Proposal to add the Y combinator to the C++ standard library: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0200r0.html
- And do not worry about performance – an optimizing C++ compiler does not generate any overhead. See for example: https://godbolt.org/ (do not forget “volatile” when experimenting, or the compiler might optimize out the entire function)