ToiletPaper #123

jambit
WHERE INNOVATION WORKS

λf. (λx. f (x x))(λx. f (x x))

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

✗ Problem

In der jüngeren Vergangenheit wurden C++ (ab C++11), Java (ab Version 8) und viele andere Sprachen um “Lambdas” erweitert. In der Praxis handelt es sich dabei um eine verkürzte Notation, um eine anonyme Klasse mit einem funktionalen Interface zu definieren und gleichzeitig ein Objekt dieser Klasse zu instanziieren. Aber wie soll man eine anonyme Methode rekursiv aufrufen?!?

✓ Lösung

In C++ kann man zwar Lambdas à la std::function<void()> f = [f]() { /* ... */ f(); }; definieren, aber nicht inline (z.B. als Parameter), und auch nur mit einer zusätzlichen Indirektion (z.B. via std::function). Für Java sieht's nicht besser aus. Eleganter geht es mit dem “Y-Kombinator” (wobei man im Internet besser nach “fixed-point combinator" sucht): Man ersetzt die rekursive Funktion durch eine Funktion höherer Ordnung, die statt sich selbst ihren Parameter aufruft. Diese Funktion füttert man dann in besagten Kombinator, welcher die Funktion immer wieder mit sich selbst als Parameter aufruft.
Und wie sieht dieser mysteriöse Kombinator jetzt aus? In C++ (für beliebige Parameter) bzw. Java (mit Currying für den ersten Parameter, die Funktion) zum Beispiel so:

C++ (für beliebige Parameter beliebigen Typs)

```
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 (mit Currying für die Funktion, und einem weiteren Parameter)

```
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);
  }
}
```

➔ Beispiel

Schauen wir uns das Standardbeispiel für Rekursion an, nämlich die Fakultät. Als Funktion (in C++), sowie als Lambda in C++17 und Java sieht das so aus:

C++ Funktion

```
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
)
```

Den trailing return type braucht man in C++ leider, weil der Compiler sonst “auto type deduction“ triggert, was zu einer zyklischen Abhängigkeit führt.

✚ Weiterführende Aspekte

- Es gab schon mal den Vorschlag, den Y-Kombinator zur C++ Standardbibliothek hinzuzufügen: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0200r0.html
- Wer sich um Performance Gedanken macht, sei unbesorgt: Ein optimierender C++ Compiler erzeugt keinen Overhead. Siehe z.B. https://godbolt.org/ (beim Experimentieren volatile nicht vergessen, sonst optimiert der Compiler am Ende noch die ganze Funktion weg)