Kotlin - fold expressions

Samstag, 17. August 2024 17:14

Author: Damir Ljubic
e-mail: damirlj@yahoo.com

Fold expression in C++

In C++ folding expressions are way to **repeatedly, at compile-time**, apply one of the predefined ***binary operators*** on the each and every argument in variadic **arguments pack**, where the arguments are of the same type (or convertible to the common type)

One typical example is compile-time summing of arbitrary number of arguments

```
template <typename…Args>
constexpr auto sum(Args&&…args) noexcept
{
    return (0 + … + args); // left fold expression (((init op arg1) op arg2) op … argn)
}
```

The constraint is that it works only for “addable” types, or types that implement the *operator* +.

Beside arithmetic operators, there is ***comma operator*** to apply the custom specific callable object that takes a single argument - consecutively, on the given argument pack

```
template <typename Func, typename…Args>
constexpr auto apply(Func&&func, Args&&…args)
{
    return ( std::invoke(std::forward<Func>(func), std::forward<Args>(args)), ...  );
}
```

We can decorate the given callable with additional functionality - as kind of Decorator Pattern, independent on the callable itself
@note We define here a "closure" - an anonymous function object (type) - rather than local function, where the call operator takes the templatized argument (auto): in C++, we can't have the function template as local function, while this is completely legal in Kotlin

```
namespace decorator
{
    template <typename Func, typename...Args>
    constexpr auto apply(Func&&func, Args&&...args)
    {
        // This is not necessary, but yet demonstrates the wrapper
        // around the given function - in this case, to log the arguments
        // It's kind of Decorator Pattern: extending the existing API with additional functionality
        auto callable = [f =std::forward<Func>(func)](auto&& arg) mutable
        {
            std::cout << arg << '\n';
            return std::invoke(f, std::forward<decltype(arg)>(arg));
        };

        return ( callable(std::forward<Args>(args)), ... );
    }
}
```

We can re-implement the compile-time sum as

```
// sum with comma operator
long sum = 0;
apply([&sum](auto&& el) mutable{sum += el;}, // func
      1,2,3,4,5 // args
     );
```

The entire code
https://godbolt.org/z/6Kjqs4sbe

Fold expressions in Kotlin

There is a similar concept in Kotlin, with limitation that it's available only for **Collections**.
It has obvious resemblance with *comma operator* in C++, with one exception.
In Kotlin - there is no compile-time code execution: the bytecode will be loaded to the JVM at runtime.

```
val l = listOf(1,2,4,17,5)
val r = l.fold(0){sum, el -> sum + el}
println("result: $r")
```

This should give the sum of all elements in the list, applying repeatedly the given lambda, starting with initial value of 0 as accumulator.

Well, the sum is not the very best example of fold expressions in Kotlin - only the fairly simple one, since there is already embedded *sum* (extension) function to List<T> interface

```
println("result: ${l.sum()}")
```

@note In the same way, for the runtime sum calculation on the vector in C++, we would use iterative algorithm - *std::accumulate*

Interesting enough, similar with C++, the Kotlin offers the *right fold expressions* as well

```
val r = l.foldRight(0){sum, el -> sum + el}
```

Here, it doesn't make any difference, in which order arguments are summed up.

Kotlin operators

Going one step back - we need to introduce the operators in Kotlin.
The Kotlin supports the predefined set of operators - there is no new/delete operators in Kotlin due to Garbage Collection involved, but for any custom type we can overload the arithmetic operators, or Comparable<T> interface (Java compatibility) which covers the <=> (spaceship operator), etc.

We usually overload operators using Extension Functions

```
// Overloading the operator+
data class Num {val value: Int }
operator fun Num.plus(other :Number) = Num(value + other.value)
```

Or, in case that we need to access private members of the enclosing class - we define it as a member functions

In C++ we have *call operator*, that turns any class into function object.
Additionally, we can use std::invoke method to call any visible member function, or even property of the class.

<Example>
https://godbolt.org/z/s7Yqq4K1M

In Kotlin, we have the *invoke operator*, with arbitrary number (and type) of arguments - to accomplish the same: turning the customer type into the function object

```
class Square(private val value: Int) {
    operator fun invoke() = value * value
}
```

And call it as

```
val square = Square(10)
square()
```

Links

#Atomic Kotlin: https://www.atomickotlin.com
#Fold expressions in C++17: https://en.cppreference.com/w/cpp/language/fold