Programming MPC applications with FRESCO using Kotlin and Coroutines

January 24, 2018

1 Fresco-Logistic-Regression

Fresco-Logistic-Regression[1] introduces an *Expression* class and subclasses that enable to easily specify FRESCO computations. The idea is that an *Expression* captures a computation to be performed by FRESCO; by running *e.build()*, this expression is turned into a FRESCO *Computation* instance that can be executed by FRESCO.

Fresco-Logistic-Regression also includes a *DummyEvaluation* module providing an *evaluate* function that directly evaluates an *Expression* using FRESCO's dummy evaluation system. This does not perform an actual multi-party computation: instead, it performs the computation locally on one PC.

However, the above approach does not allow intermediate results in a computation to be opened. For instance, in logistic regression, the resulting model (a vector with one coefficient for each attribute) is determined in an iterative proceess. For efficiency reasons, one would like to open the current estimate of the resulting model after each iteration, so that it can be used in the plain in the next iteration. As a workaround, the code in the proof-of-concept that builds up the *Expression* instance for the logistic regression computation does the following:

```
val openBeta = evaluate(beta)
```

That is, the function contains a recursive call to the *evaluate* dummy evaluation – but this does not actually perform a multi-party computation. Instead, what we would like to happen here is that the line above temporarily suspends the function that builds up the Expression to execute the MPC protocol corresponding to the *beta* expression. This is enabled by the use of generators.

[1] https://github.com/Charterhouse/Fresco-Logistic-Regression, by Mark Spanbroek and Stefan van den Oord

2 Fresco-Logistic-Regression & Generators

Generators, for instance available in Python, "allow you to declare a function that behaves like an iterator, i.e. it can be used in a for loop"[2]. Kotlin does not support generators directly but it supports a more general concept of "coroutines" that can be used to support them[3]. For instance, consider one of the examples there:

```
// Simple example
fun idMaker() = generate<Int, Unit> {
    var index = 0
    while (index < 3)
        yield(index++)
}
fun main(args: Array<String>) {
    val gen = idMaker()
    println(gen.next(Unit)) // 0
    println(gen.next(Unit)) // 1
    println(gen.next(Unit)) // 2
    println(gen.next(Unit)) // null
}
```

In this example, *idMaker* is a generator that repeatedly "yields" values (in this case, 0, 1, and 2) that can be obtained by the caller (in this case *main*) by calling the function once and then repeatedly calling *next* to get the next yielded value. In this example, data only travels from the generator to the caller, but it is also possible to have data travel from the caller to the generator: the argument that the caller gives to *next* would be the return value of *yield* in the generator.

Intuitively, generators allow us to specify functions that can be temporarily suspended and then restarted later. This gives us a way to deal with the problem of opening intermediate results in a multi-party computation.[4] The idea is that we turn the function that builds up the *Expression* for the multi-party computation into a generator. At the point where it needs the value of an intermediate result, it can do something like:

```
val openBeta = yield(beta)
```

This would suspend the generator, giving the caller an *Expression* that the generator would like to see evaluated. The caller can evaluate this *Expression*, and provide it to the generator by providing it as the argument to *next*.

This is exactly what happens in the Fresco-Logistic-Regression fork[5]. The function to build the logistic regression model is turned into a coroutine that

[2] https://wiki.python.org/moin/Generators

[3] https://github.com/Kotlin/kotlin-coroutines/tree/master/examples/generator

[4] This idea was already known for the Twisted deferreds as "inline callbacks" and, as per my suggestion, has also made its way into the Python-based VIFF framework for multiparty computation that is used at Eindhoven University of Technology.

[5] https://github.com/meilof/Fresco-Logistic-Regression

yields several times to get the plaintext value of the current estimate for the model; and it finally yields to produce the eventual model (a coroutine cannot return values so this value needs to be yielded as well). Globally, it is structured as follows:

```
fun fitLogisticModelCoroutine(...) = generate<Any, Any> {
        // ... first estimate of beta ...
        for (i in 0 until numberOfIterations) {
            val openBeta = yield(beta) as plain.Vector
            // ... update estimate of beta
        }
        yield(beta)
    }
```

To enable this to work, some other small technical changes were needed in Fresco-Logistic-Regression that are also included in this fork.

3 Calling Evaluator generators

Given a generator producing *Expression*s above, we need code that "executes" the generator by repeatedly:

1. calling the generator to get *Expression*s;
2. calling *build()* on such an *Expression* to get a FRESCO *Computation* instance;
3. amending this *Computation* instance so that the value is not only computed, but also opened;
4. passing the anended instance on to the FRESCO engine for execution;
5. and using the result (i.e., the opened value corresponding to the yielded *Expression*) in the next call to the generator (step 1).

This is exactly what is demonstrated in the TestLogisticRegression project[6]. Specifically, the *CoroutineRunner* class in *Test.java* does this, for the case that the *Expression* yielded by the generator is a vector, making use of a *EvalOpenVectorApp* that opens such a vector to its plaintext values.

The above is sufficient to run the logistic regression example (the *Test* class demonstrates this by running logistic regression on a small example), but to make this code more generally useful, *CoroutineRunner* should be generalised to provide a matching opening protocol for any type of *Expression* that the framework in Fresco-Logistic-Regresion may return. Better yet, Fresco-Logistic-Regression *Expression*s should themselves have an *open()* function that provides a protocol to open them, such that the *CoroutineRunner* can be made much simpler.

[6] https://github.com/meilof/TestLogisticRegression