Polyglot programming for science

Alexander Nozik

for BAT meeting 2018_11

NPM The problem

- Different people prefer different languages.
- Different languages have limited interoperability.
- Different languages have different runtimes which further complicates the problem.

Solution 1: C API

C API allows to make calls with common low-level language

Limitations:

- Rather limited in terms of data structures and functions
- Memory management problems
- Function references only inside single process

Solution for BAT in talk by Alexey Khudyakov

NPM Solution 2: remote procedure call

Each request transformed into tree-like structure and transferred via socket/websocket connection. The response is transferred in a same way.

Problems:

- Overhead on each call from transport and parsing
- No obvious way to pass function

NPM Remote procedure call

- TCP round trip latency for local calls is 6 ms. Comparable with local cross-language calls.
- Parsing time could be dramatically reduced by using binary tree representation (CBOR, ProtoBuf, FlatBuffers).
- Non-blocking IO allows effective processing of TCP connections.
- A lot of support libraries for that.

Remote procedure call: functions

Example

```
{
    "action": "integrate.rieman",
    "parameters": {
        "from": 0,
        "to": 1,
        "function": {
            "name": "test:myFunction",
            "type": "remote",
            "connection": {
                "address": "192.168.11.11",
                "security": "security options here"
            }
        }
    }
}
```

Function handler

Server

Request with data handler and/or function handler

response

Data

Client

Function server

Function libraries

```
"function": {
        "name": "test:mySum",
        "type": "sum",
        "functions": [
            {
                "name": “sin“
            },
            {
                "name": "test:myRemoteFunction",
                "type": "remote“
            }
        ]
    }
```

Function from server local repository

Custom remote function

Function API (preliminary)

```kotlin
interface Function{
    operator fun invoke(buffer: ByteBuffer): ByteBuffer
}

interface UnivariateRealFunction: Function{
    operator fun invoke(value: Double): Double {
        val buffer = ByteBuffer.allocate(8)
        buffer.putDouble(value)
        return this(buffer).getDouble()
    }
}
```

+	-
• Different types • Different numbers of parameters • Custom argument types • Possible memory sharing	• Additional allocation • No control of actual layout of buffer

Solution 3: common runtime

Run all programs in the common runtime, compiling them to IR.

Current platforms:

- CLR
  - Supports its own languages (C#, F#, Basic, etc)
  - Could use native libraries code and could compile existing C++/Fortran code
  - Supports compile to LLVM via LLILC (Windows only, seems to be abandoned)
- GraalVM (RC)
  - Good support for JVM languages (currently JDK 8 compatible)
  - Powerful AST engine Truffle
  - Features its own implementation of Python, JavaScript and R
  - Runs LLVM IR (Julia as well?) in polyglot mode
  - Interface between native code and VM code without JNI (at last)
  - AOT compilation with SubstrateVM (no Windows yet ☹)

Solution 3.5: common code base

Compile the same code to be used later in different platforms. Code must be optimized for platform specifics and libraries.

- Good JVM support in Kotlin-JVM (all Java libraries)
- JavaScript support in Kotlin-JS (full interop with JS)
- Native compilation in Kotlin-Native (LLVM backend)
  - Full interop with C API, Objective C and Swift
  - No bridge between Native and JVM yet, but planned

Currently focus on mobile development and web back-end, but scientific community grows.

Kotlin for science

Kotlin vs Java

Java

```java
public class User {
    private final String firstName;
    private final String lastName;
    private final int age;

    public User(String firstName, String lastName, int age) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public int getAge() {
        return age;
    }

    public String toString() {
        return firstName + " " + lastName + ", age " + age;
    }
}
```

```java
class Main {
    public static void main(String[] args) {
        System.out.println(new User("John", "Doe", 30));
    }
}
```

Kotlin

```kotlin
public class User(val firstName: String,
                  val lastName: String,
                  val age: Int) {

    fun toString() = "$firstName $lastName, age $age"

}
```

```kotlin
fun main(args : Array<String>) {
    println(User("John", "Doe", 30))
}
```

https://hype.codes/kotlin-vs-java

Kotlin: example

```kotlin
// Extension function
fun Int.isOdd() = this % 2 != 0

// Result with type inference
val result = (10..20)
    .filter{ it.isOdd() } // filtering odds
    .associateWith{it.toString().last()} // associating to map
    .map {entry-> "${entry.value}: ${entry.key}" } // string interpolation

// Functional style consume
result.forEach { println(it) }
```

```
1: 11
3: 13
5: 15
7: 17
9: 19
```

Kmath: ndarray operations

```
array:
  [[1,  2.2],
   [3.1, -5]]

val function: (Double) -> Double = { x -> x.pow(2) + 2 * x + 1 }
val result = function(array) + 1.0
```

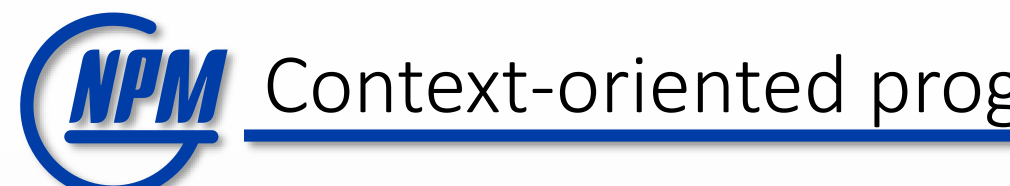
Function on numbers

Applied to array

Error?

```
/**
 * Element by element application of any operation on elements to the whole array. Just like in numpy
 */
operator fun <T> Function1<T, T>.invoke(ndArray: NDArray<T>): NDArray<T> = ndArray.transform { _, value -> this(value) }
```

```
operator fun <T> NDArray<T>.plus(arg: T): NDArray<T> = transform { _, value ->
    with(context.field) {
        arg + value
    }
}
```



Context-oriented programming

Real numbers

```
val context = FieldExpressionContext(DoubleField)
val expression = with(context) {
    val x = variable("x", 2.0)
    x * x + 2 * x + 1.0
}
assertEquals(expression("x" to 1.0), 4.0)
assertEquals(expression(), 9.0)
```

Complex numbers

```
val context = FieldExpressionContext(ComplexField)
val expression = with(context) {
    val x = variable("x", Complex(2.0, 0.0))
    x * x + 2 * x + 1.0
}
assertEquals(expression("x" to Complex(1.0, 0.0)), Complex(4.0, 0.0))
assertEquals(expression(), Complex(9.0, 0.0))
```

In-place context

```
fun <T> FieldExpressionContext<T>.expression(): Expression<T>{
    val x = variable("x")
    return x * x + 2 * x + 1.0
}

val expression = FieldExpressionContext(DoubleField).expression()
assertEquals(expression("x" to 1.0), 4.0)
```