KOTLIN/NATIVE + CLANG, TRAVEL NOTES

NIKOLAY IGOTTI, JETBRAINS

KOTLIN IS...

NOT JUST AN ISLAND

```
fun generateCAdapter() {
    when {
        isFunction -> {
            val function : FunctionDescriptor = declaration as FunctionDescriptor
            cname = "_konan_function_${owner.nextFunctionIndex()}"
            val llvmFunction : LLVMValueRef = owner.codegen.llvmFunction(function)
            // If function is virtual, we need to resolve receiver properly.
            val bridge : LLVMValueRef = if (!DescriptorUtils.isTopLevelDeclaration(function) &&
                    !function.isExtension && function.isOverridable) {
                // We need LLVMGetElementType() as otherwise type is function pointer.
                generateFunction(owner.codegen, LLVMGetElementType(llvmFunction.type)!!, cname) { this: FunctionGenerationC
                    val receiver : LLVMValueRef = param( index: 0)
                    val numParams : Int = LLVMCountParams(llvmFunction)
                    val args : List<LLVMValueRef> = (0..numParams - 1).map { index -> param(index) }
                    val callee : LLVMValueRef = lookupVirtualImpl(receiver, function)
                    val result : LLVMValueRef = call(callee, args,
                            exceptionHandler = ExceptionHandler.Caller, verbatim = true)
                    ret(result)
                }
            } else {
                LLVMAddAlias(context.llvmModule, llvmFunction.type, llvmFunction, cname)!!
            }
            LLVMSetLinkage(bridge, LLVMLinkage.LLVMExternalLinkage)
        }
        isClass -> {
            // Produce type getter.
            cname = "_konan_function_${owner.nextFunctionIndex()}"
            val getTypeFunction : LLVMValueRef = LLVMAddFunction(context.llvmModule, cname, owner.kGetTypeFuncType)!!
            val builder : LLVMBuilderRef = LLVMCreateBuilder()!!
            val bb : LLVMBasicBlockRef = LLVMAppendBasicBlock(getTypeFunction, Name: "")!!
            LLVMPositionBuilderAtEnd(builder, bb)
            LLVMBuildRet(builder, (declaration as ClassDescriptor).typeInfoPtr.llvm)
            LLVMDisposeBuilder(builder)
        }
```

KOTLIN LANGUAGE

- FP and OOP language
- Type inference, smart casts, nullability checks
- Generics (erased, with reification and controlled variance)
- Rich standard library (collections, regexes, etc.), coroutines
- Unchecked runtime exceptions
- Transparent boxing (primitive types are formally objects)
- Automated memory management, need to collect cycles
- Transparent interoperability with the platform (JVM, JS, C, Objective-C)

KOTLIN/NATIVE

- Kotlin -> platform binaries (ELF, COFF, Mach-O, WASM)
- Targets iOS, macOS, Linux, Windows, WebAssembly and embedded (x86, ARM, MIPS)
- Currently uses LLVM 5.0, compiler written in Kotlin/JVM, runtime in Kotlin/Native and C++
- Provide runtime guarantees (exceptions, memory management) similar to JVM in VM-less environment
- Automated interoperability with C/Objective-C using libclang
- Broad set of platform libraries (POSIX, Apple frameworks, Win32, W3C DOM, etc.)

LLVM INTEGRATION

- Produce bitcode from Kotlin code with LLVM API
- Produce bitcode from the C++ runtime with clang
- Link and generate code with llvm-lto
- Kotlin LLVM API is autogenerated from LLVM C bindings with a generic interop tool
- Closed world, DCE and optimizations

KOTLIN/NATIVE COMPILER

- Shares frontend with Kotlin/JVM and Kotlin/JS
- Source code -> high level IR
- Multiple lowering passes on IR
- Devirtualization, escape analysis
- Bitcode generation from lowered IR
- LLVM (llc, ld) tools to generate final binaries
- Own library format: Kotlin metadata + bitcode
- Non-optimizing, most optimisations come from LLVM

INTERESTING COMPILER ASPECTS

- Simple top down codegen from AST-like lowered HIR
- Complex stuff (coroutines, lambda capturing) is performed in lowerings
- Memory management requires specific LLVM function organisation
- Basic block termination is hard to get right (no DCE pass)
- Kotlin null safety helps in codegen
- Optimise very specific operations (virtual dispatch, memory management)
- Library format uses serialized Kotlin metadata for linking

MEMORY MANAGEMENT

- ARC, with the cycle collector
- Compiler does not know about RC, just maintains root set
- Disjoint object graph for different threads
- Object subgraphs can be transferred between threads
- Immutable objects can be shared (object freezing)
- Root set is maintained per call frame
- Leak detection mechanism, abort on leaked memory
- C sees raw pointers to data, Objective-C sees its objects

EXCEPTION HANDLING

- Relies on landing pads mechanism
- Structure matches AST/HIR
- Uses C++ personality function
- Throw using C++ *'throw'* keyword
- Exception object memory managed by C++ wrapper
- Transparently interleaves with C++/Objective-C frames
- Unsupported for some targets (WebAssembly)

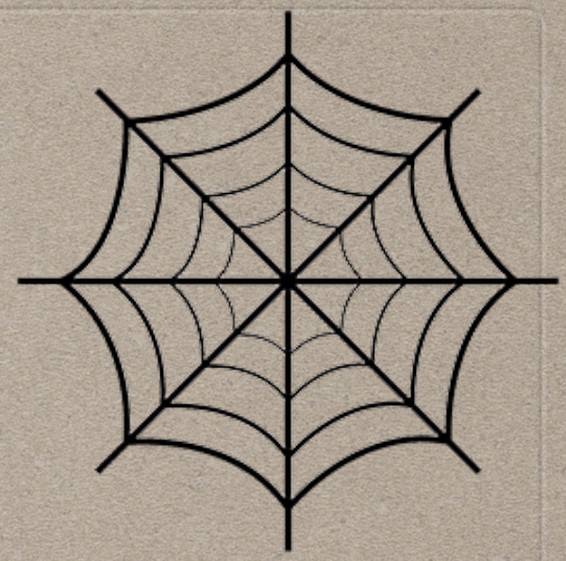
INTEROPERABILITY

- Mostly transparent interoperability with C, Objective-C (and thus Swift)
- Kotlin calls C/Objective-C, C/Objective-C calls Kotlin (in OOP manner)
- Kotlin extends Objective-C classes and vice versa
- Numbers passed as is, strings converted, collections and classes wrapped
- Memory manager aware of Objective-C runtime, and accounts properly
- For C Kotlin declaration wrapping C entities (functions, structs, unions, macroses, typedefs, etc.) are autogenerated
- For Objective-C OOP concepts (classes, protocols, blocks) are represented as matching Kotlin entities (classes, interfaces, lambdas)
- For Objective-C Kotlin code can be compiled to the framework

DEBUGGABILITY

- Use LLVM C++ debugging APIs
- Wrapped to C for interop sake
- Produce DWARF/dSYM in final binaries
- Breakpoints/single stepping works
- Evaluation works partially
- XCode has issues setting breakpoints in Kotlin code
- Verifier helps

WEBASSEMBLY



- Nothing but CPU, RAM and JS calls
- No libc, dlmalloc for memory allocator
- Not in standard LLVM builds
- No exceptions
- No debugging
- No JS object memory management integration
- Works!

PROBLEMS WITH LLVM

- API is not ideally documented
- Not all APIs available from C bindings
- Mysterious crashes (LLVMVerifyModule() helps)
- Debugger API had to be reversed from clang
- Missing public LLDB plugin API
- Exception handling API sometimes convoluted
- Artificially incompatible bitcode for different architectures
- Slow codegeneration and linking

NICE THINGS ABOUT LLVM

- Great LIR
- libclang allows interoperability without much ado
- Well tested, few code generator and optimiser bugs
- Wide range of supported platforms
- High quality code is produced
- Natural API

Your questions!

– *igotti@gmail.com*