Kotlin Compiler Reading Notes

Yuxiang Jiang *

April 6, 2020

Contents

*yxjiang@linkedin.com

1 Methodology

It's often daunting to read through huge and complex codebase like the Kotlin compiler! Fortunately we have great tools at hand to deal with such complexity. In particular, IntelliJ Idea provides many priceless code navigation tools:

- Type Hierarchy (under Navigate)
- Structure (under View - Tool Windows)
- Find Usage
- Breakpoints

With these we'll be able to understand the compiler architecture and internals bit by bit...

1.1 Building From Source

Once you have the repo checked out, run `./gradlew dist` to build everything. This can take 10-20 minutes. Then you will be able to run tests, compiler.cli.cli-runner (i.e. the script runner), and compiler.preloader (i.e. the CLI compiler loader).

1.2 Tracing Compiler Execution

It's often useful to use the debugger to understand how the compiler pipeline works. This requires the ability to run custom code and to attach a debugger to the compiler process. One trick that I used was to add a local file to the cli-runner package (or compiler.preloader). This allows me to run or debug arbitrary code there while being able to link the code with the compiler. And since these these packages don't actually have source dependency on the whole compiler (only link to the dist jar), rebuilding is quite fast.

Another way is to link your code with a prebuilt compiler (e.g. add dependency `"org.jetbrains.kotlin:kotlin-compiler:1.3.70"`). I can't seem to fetch its sourceJar though...

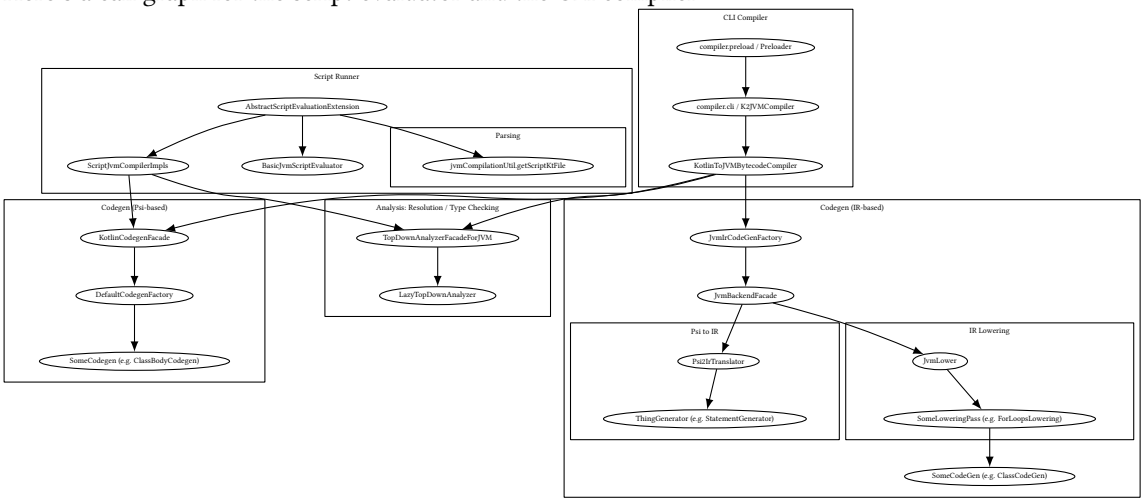
2 Tl;dr: Pipeline

The are couple key types in the compiler pipeline:

- KtElement (Psi): concrete syntax tree, backed by JetBrain's Psi system for wonderful IDE integration.
- core.descriptor, BindingContext, and friends: results of various frontend analysis: type checking, method binding, etc.
- FIR (FirElement etc): "Frontend IR", seems to be a new IR that's converible from Psi.

- IR (IrSymbol, IrDeclaration): Experimental backend IR.

Here's a call graph for the script evaluator and the CLI compiler:



3 KtElement (Psi)

This is the concrete syntax tree of Kotlin.

- See class KtVisitor for an overview of many Kotlin PsiElements
- KtFile and KtScript for toplevel containers
- KtClass for `class Foo`, KtNamedFunction for `fun foo()`, KtProperty for `val foo`.
- compiler.psi defines KtElement and Stubs, also provides parser
  - Stubs represents the interface parts of a kt compilation unit (.h, .mli, .hi etc).
  - There's also a LighterASTNNode (KotlinLightParser) – what is it? (Flyweight pattern is like hash consing)
  - Some KtElements are related to types: e.g. KtTypeReference (some of them are not KtElements but stubs!)
- compiler.psi.KtPsiFactory is the entrypoint for creating KtFile (a PsiElement) from source text.

4 DeclarationDescriptor

See core.descriptors.DeclarationDescriptorVisitor for an overview.

DeclarationDescriptorVisitor sounds like an type-instantiated/abstracted wrapper of an element. Also has a bunch of annotations (AnnotationDescriptor) and a name. And is also a tree node (has parent: getContainingDecl)

4.1 Survey of Class Hierarchy

Looks that descriptors are something that's used throughout the whole compilation pipeline. They are more often used in frontend, but even in backend I can see some usage of them.

4.1.1 CallableD: VisD & NonRootD & Subst

Receiver types (dispatch / extension), arg types, return types, type params;
Parameter names, names may be unstable/synthesized (e.g. from JVM object code)
Parameter values (See 4.1.6)
Cross ref to overridden methods.
UserDataKey<A>: stores typed user data

4.1.2 MemberD: VisD & NonRootD

Has member modifiers: expect / actual / external. And modality: final /sealed / open / abstract.

4.1.3 CallableMemberD: CallableD & MemberD

Kind: decl / delegation / fakeOverride / synthesized (what's the last two?)

4.1.4 ValueD: CallableD

Has a KotlinType.

4.1.5 VarD: ValueD

Has isVar (wat), isLateinit, isConst, and an optional compileTimeInitializer.

4.1.6 ParamD: ValueD

Represents a parameter that can be supplied to a callableD.

ReceiverParamD: ParamD Has a ReceiverValue.

ValueParamD: ParamD & VarD Has a index, hasDefaultValue, varargElementType, isCrossinline / Noinline (why on param?)

4.1.7 VarDWithAccessors

Has optional getter / setter typed VarAccessorD.

4.1.8 FunD: CallableMemberD

initialSignatureD: the initial D before renaming (didn't find SimpleFunctionD.rename)

hiddenToOvercomeSignatureClash: hack to handle corner case signature clash (said see nio.CharBuffer); also hiddenEverywhereBesideSupercalls: see 4.1.9.

Function modifiers: infix/inline/operator/suspend/tailrec

4.1.9 FunDImpl: NonRootDImpl & FunD

Base impl for function modifiers. Setters set the local modifier (mostly happen during conversion from KtElement), while some getters (infix, operator) respect super class methods.

Base impl for substitution (doSubstitute), and substituted value param. Worth reading.

Base impl for initialize.

Only here documents hiddenToOvercomeSignatureClash and hiddenEverywhere-BesideSupercalls: former makes the function completely hidden (even in super-call), latter permits super-call and propagates to overriden methods

4.1.10 ConD: FunD

containingD: ClassifierDWithTypeParams (what is this?)

constructedClass: ClassD

4.1.11 ClassConD: ConD

Just a bunch of return type specializations

4.1.12 ClassConDImpl: FunDImpl & ClassConD

Default (`<init>`) or synthesized.

Has a way to calculate dispatchReceiverParam. If inner, init's receiver is outer class instance (Whereas Java's outer class instance is passed as a param. Though in compiled code there's no difference, just at descriptor level they are different.); else null.

CommonizedClassConD: ClassConDImpl ClassConDImpl with source and originalD stripped.

DefaultClassConD: ClassConDImpl Read as (default constructor) of given class, not default implementation of (any class constructor). So this is just the no-param constructor of a class. Its visibility depends on the classD's visibility.

DeserClassConD: ClassConDImpl & DeserCallableMemberD ProtoBuf based deserialized ClassConD. Holds a bunch of TypeTable, NameResolver, ContainerSource etc to help with further deserialization (so this is also in some sense lazy).

JClassConD: ClassConDImpl & JCallableMemberD A Java imported class's constructor. Since this is from JVM object code, it provides property impl of hasStableParamNames / hasSynthesizedParamNames.

enhance() implementation makes a copy with enhanced receiver param and value params.

SamAdapterClass: ClassConDImpl & SamAdapterD<JavaClassConD> A synthetic ClassConD that wraps another ClassConD.

4.2 ClassD

LazyJavaClassD Represents a JavaClass (Can be from Jar, or from source).

4.3 DFactory

In core.descriptors / resolve.

5 FIR

Front-end IR (not sure what this means), enabled by `-Xuse-fir` (This also implies using IR. But use-ir doesn't imply using FIR). The flag doc says it's in very early stage of development though.

compiler.fir.resolve: ResolutionStage sounds like something pipeline-related

5.1 Some observations

- See generated class FirVisitor for an overview of many of the Fir exprs
- compiler.frontend (not sure which step it is in, but it at least does symbol resolution, type checking, (Psi → CFG?)
  - See key classes: AnalysisResult, BindingContext, BindingTrace (records the collected binding / type substs?)
- compiler.resolution: tower/ReceiverValue etc
- compiler.fir: cones (types and symbols used in Fir?), fir2ir (lowering to Ir), psi2fir (lowering Psi to Fir), resolve, jvm, tree (Fir definitions and impl for psi2fir)
  - cones: StandardClassIds contains a bunch of core Kt (read: not JVM) type Ids. They have a JVM-like fqname.

    SyntheticCallableId contains when/try/nullcheck synthetic call exprs

– tree.gen contains all Fir expressions (see tree.tree-generator's readme for how they are generated), as well as extra info (FirTypeRef). And even on Fir level, the generic types are not yet erased (FirTypeProjectionWithVariance)

6 IR

IR (compiler.ir) seems to be an lower-level IR. This is an experimental IR that's intended to be used across all Kotlin backends. Also see this writeup.

A sample IR lowering pass can be found here. Looks that the compiler API provides a convenient IrBuilder, and a visitor style IrTransformer.

6.1 Playing with IR

This is enabled by passing `-Xuse-ir` as a CLI option to the compiler. IR backend currently doesn't support the script runner – KotlinToJVMBytecodeCompiler will ignore use-ir on kts files; Also ScriptCodegen calls KotlinTypeMapper.mapType which is not intended to be called with IR backend.

6.2 compiler.ir.tree

IrSymbol definitions. See IrSymbolVisitor for an overview. Looks that they have descriptors attached.

6.3 Phases

compiler.ir.backend.common defines CompilerPhase that's chainable. See PhaseBuilders.kt and JvmLower.kt.

6.4 IrSymbol vs IrElement

IrElement seems to be the implementation part while IrSymbol is more about the declaration part. E.g. IrClassImpl vs IrClassSymbolImpl. The former contain the concrete class members (init, methods) while the latter contains ClassD.

IrSymbol can be bound or not (what does this mean?), has a owner (an IrElement), an IdSignature (what's this?), and visibility (isPublicApi).

6.5 Psi $\rightarrow$ IR

compiler.ir.ir.psi2ir contains Psi2IrTranslator, which sounds like the entrypoint for this convertion.

The translator contains an IdSignatureComposer, which composes IdSignature from DeclarationDescriptor, and ClassD for enums.

package generator contains ThingGenerator that generates IrThing from KtThing. Generated IR will remember the KtElement's source position range (startOffset and endOffset).

6.5.1 DeclarationGenerator

Generates toplevel IrDecl from KtDecl. “Toplevel" is a very minimal set of KtDecl: KtNamedFunc, KtProperty, KtClassOrObject, KtTypeAlias, and KtScript. The resulting IrDecl contains both the declaration and the body implementation.

7 Type System

7.1 Types

Kotlin compiler uses Marker Interface pattern extensively in type definitions. See TypeSystemContext.kt (KotlinTypeMarker, TypeArgumentMarket etc).

KotlinType is the base class for all types in Kotlin. It has a tycon, list of tyargs, nullability (so nullability is built-in to any type – can this be problematic?). It also has a MemberScope (“what are the members in this type-based namespace?", see 8) and a refine(KotlinTypeRefiner) method.

7.2 core.type-system

type system core (equality, bounds checking etc). However this is more of an interface module – the actual impls are in core.descriptors, fir and ir modules.

7.3 TypeSystemTypeFactoryContext

Contains a bunch of common type factories:

- flexibleType has lower/upper bounds
- simpleType has tycon, tyargs, nullablep
- tyarg has ty and variance
- star has tyarg (why?)
- there’s also an error type used in diagnosis

7.4 TypeCheckerProviderContext

- modular axioms (errorType unifiable with all types etc)
- what is a stub type? (Probably PsiStub related?)

7.5 TypeSystemCommonSuperTypesContext

Used to check if two type has common super types, and lowest-common ancestor utils.

typeDepth is a safe overestimation of the depth (from ‘Any‘).

Seems to also be used in Fir.

7.6 TypeSystemInferenceExtensionContext

Inference related.

7.7 Type Checking During Analysis

7.7.1 ExpressionTypingServices

Seems to be the entrypoint, which creates an ExpressionTypingContext and calls ExpressionTypingVisitorDispatcher.getTypeInfo.

7.7.2 ExpressionTypingVisitorDispatcher

A delegating KtVisitor with context ExpressionTypingContext and returns KotlinTypeInfo.

Delegates its visiter methods to a bunch of sub-visitors: functions, control structures, patterns, basic expressions, and annotations/declarations etc.

Handles DeferredType by retrying type checking after the statement visitor returns.

7.7.3 ExpressionTypingVisitorForStatements

An ExpressionTypingVisitor for statements (XXX how can statements have types?).

.visitBinaryExpression Handles eq, add-eq, and other binary ops.

Take visitAssignment for example. So first it creates a new ResolutionContext with overridden expected type, scope, and context dependency.

Then it check lhs: Lhs can have annotation, which it resolves. Lhs can also be arrayAccess, which it handles in another way (BasicExpressionTypingVisitor-resolveArrayAccessSetMethod then checkLValue).

7.7.4 Questions

- What is a isCapturedTypeConstructor?
- What is a singleBestRepresentative?
- What is a noInferAnnotation?
- What is mayBeTypeVariable?
- What is a defaultType?
- Read impl of isUnit vs isUnitTypeConstructor
- Read impl of createCapturedType
- Read impl of createStubType
- Read impl createEmptySubstitutor, typeSubstitutorByTypeConstructor, safeSubstitute

7.8 TypeSystemContext

- fastCorrespondingSupertypes has no actual impl? (No, it's just that intellij's search functionality fail to find overridden extension methods)
- isCommonFinalClassConstructor is implemented in three (Psi, Fir, Ir) stage's TypeSystemContext:
  - ClassicTypeSystemContext: get ClassDescriptor from TypeConstructor's declarationDescriptor, then check it's final but not (enum or annotation). So the method really checks that the tycon is "final" but is not a uncommon (enum/annotation) class.
  - ConeTypeContext: Does almost the same thing, but also return true if is anonymous object (final by design). Works on FirBasedSymbol (some sort of class infotable?). Check that this is a FirRegularClassSymbol, whose FirRegularClass is final but not uncommon.
  - IrTypeSystemContext: Check this is a IrClassSymbol whose owner is final and not uncommon.

    So basically ClassDescriptor, FirRegularClass and IrClassSymbol.owner are the same thing across three stages.

    Sounds that reading the common implemented methods of these three TySysCtx impl classes would be super helpful to understand the stages.

7.9 Playing with Types

DescriptorRenderer.SHORT_NAMES_IN_TYPES.renderType could be used to render KotlinType.

KotlinTypeChecker.DEFAULT could be used for simple type checking (equality, subtype relationship etc)

7.10 Type System for Fir

Read ConeTypeContext

7.11 Type System for Ir

Read IrTypeSystemContext

7.12 compiler.resolution/.inference

Type inference? constraint system, subst, fresh tycon, tyvar etc

7.13 compiler.frontend/.types

TypeIntersector (unify), DeferredType (I guess this is for when inference can't proceed at some first, and will retry when it has more information. Not really fully bidirectional (H-M style) type inference, but an approximation)

7.14 types.expressions

Contains a bunch of KtElement visitors that does type recon/checking:

- ExpressionTypingVisitorDispatcher
- ControlStructureTypingVisitor
- FunctionsTypingVisitor
- BasicExpressionTypingVisitor (constants etc)
  - This actually does a bit of parsing/validation work... e.g. understore on int literals.
  - Also uses ConstantExpressionEvaluator to check for possible compile time constants (this indeed sounds like something a parser would do).

    Folds boolean && and ||

    Look up simple unary and binary func in OperationsMapGenerated

8 Resolution

Package: compiler.frontend and compiler.resolution (specific types)

8.1 ResolutionContext

Kotlin compiler does data flow analysis in a top-down fashion. This class is used to pass data flow analysis results from AST parent to its children.

Known concrete subtypes: ExpressionTypingContext, CallCandidateResolutionContext, CallResolutionContext.

8.2 Scopes

core.descriptor / ResolutionScope: contains information about what identifier it contributes to a given lookup location. Identifiers have separate namespaces: variable, function, and classifier (type).

compiler.resolution / LexicalScope is a ResolutionScope that has a parent, a ownerD, a LexicalScopeKind (what kind of syntactical structure created this scope?)

8.3 Tower

i.e. ImplicitScopeTower. Some sort of multi-level scopes? Can't understand this part.

9 Analysis

Package: compiler.frontend. There are various analysis done in the frontend: Psi to DeclarationDescriptor, control flow graph (package cfg), type checking, method resolution (package resolve)...

This is a very complicated process – later stages depend on the result of the early stages. Information are passed either directly, or in the form of a shared mutable context (BindingTraceContext, TopDownAnalysisContext etc).

It would be extremely useful to understand the input and the output of each analysis stage! Unfortunately due to that the pipelines share results via mutable data, we won't know this by simply looking at their return types. Instead we have to read the implementation of each stage to see what they use.

9.1 compiler.cli.TopDownAnalyzerFacadeForJVM

Creates a DI container for all the components used throughout the analysis process, then call into LazyTopDownAnalyzer.

9.2 compiler.frontend.LazyTopDownAnalyzer

compiler.frontend.LazyTopDownAnalyzer contains the whole analysis pipeline: it ultimately converts Psi into DeclarationDescriptor. (Is there another analyzer that's not lazy?)

analyzeDeclarations analyzeDeclarations returns an AnalysisResult which contains a ModuleDescriptor and a BindingContext.

It goes through all stmts, calls:

- A bunch of resolvers: BodyResolver.resolveBodies, LazyDeclarationResolver etc.
- DeclarationsChecker.process, which goes through files, annotations, class's modifiers, idents, header (super+generic bounds); function, property, destructionDecl, typealias's modifiers and idents.

9.2.1 BodiesResolveContext

Stores the toplevel declarations (in typed maps) found during analysis. Concrete implementation: TopDownAnalysisContext, created in LazyTopDownAnalyzer.

9.2.2 ThingResolver

E.g. LazyDeclarationResolver. They have a .trace that is a LockProtectedTrace (under LockBasedLazyResolverStorageManager). The real trace is NoScopeRecordCliBindingTrace (i.e. doesn't records scope information).

9.3 BindingTrace

Has a BindingContext. Is writable. Can record/inquiry KotlinType for a KtElement. Impls: BindingTraceContext and ObservingBindingTrace

9.3.1 BindingContext

Sounds like a read-only counterpart to the BindingTrace.

9.3.2 BindingTraceContext

Concrete implementation of BindingTrace. Has a map: SlicedMapImpl.

SlicedMapImpl This is a two-level map: .map maps a key (e.g. a KtElement) to a holder: KeyFMap. The holder takes a slice.key (usually the key is the slice itself) and return the value (e.g. a DeclarationDescriptor).

KeyFMap An abstract immutable map. But it's not backed by any purely functional data structures... (See impl: OneElementFMap, ArrayBackedFMap etc)

Slice A Slice<K, V> is an identifier for a mapping from K to V. Such mapping often represents the analysis results (e.g. what's the DeclarationDescriptor of this KtElement?). See BindingContext.java for a list of all the common slices.

9.4 KotlinCodeAnalyzer

See concrete impl ResolveSession.

9.5 DeclarationChecker

Check a KtDecl against a DeclarationDescriptor within a DeclarationCheckerContext. Has lots of subtypes. Reports errors to context's BindingTrace. Some checkers are just linters that enforce certain code style. Others are necessarily to ensure soundness.

PlatformConfiguratiorBase contains a bunch of linter-like checkers.

9.6 Control Flow Analysis

package: cfg. Looks like traditional control flow analysis (not PDG). Nodes are represented by class Instruction – it has def-use chain (class PseudoValue), prev/succ edges, and is implemented by lots of concrete instructions. See InstructionVisitor for a list of concrete instrs.

PseudoCode Looks like basic block but maybe not. Groups the instructions. Can be nested.

ControlFlowAnalyzer Entrypoint of control flow analysis. Its .process method checks BodiesResolveContext's files, classes, function, and properties.

ControlFlowProcessor Converts a KtElement (with the its BindingTrace) to a PseudoCode.

ControlFlowBuilder Concrete impl: ControlFlowInstructionsGenerator. Used by ControlFlowProcessor, knows how to generate individual instructions (i.e. How to map a KtParameter to a VariableDeclInstruction?). Also holds the context of the current graph.

ControlFlowInformationProvider Does real work (e.g. checkDeclaration) to analyze the control flow. E.g. checkFunction will make sure a function with a non-unit return type always returns something. Records a few flow related things: MUST_BE_LATEINIT etc.

9.6.1 Survey of CFG Instructions

Subroutine{Enter,Exit,Sink}Instruction Well known nodes for function start and end. Sounds that Exit is for returns / exceptions, and Sink is the unique end of the graph?

9.7 Smartcasting

compiler.frontend smartcasts.DataFlowInfo: bunch of maps to stores the data flow analysis result useful for smart casts.

DataFlowValue: one instance of a value in a dataflow

DataFlowValue.Kind: classify exprs into smart cast enabled, possible, or disabled ones. Quite intuitive.

IdentifierInfo: represents both qualifier and ident name. what is this for?

10 Codegen

There are currently two codegen systems and both target JS, JVM, etc. They differ in the choice of IR.

- The existing production codegen uses KtElement (Psi node) as the IR. JVM codegen lives in compiler.backend / codegen (e.g. ClassBodyCodeGen); JS codegen lives in js.js.translator (e.g. PropertyTranslator).
- The other “experimental" codegen uses compiler.ir as the IR. All targets specific code lives in compiler.ir.backend.TARGET.

10.1 Psi-based Codegen

10.1.1 JVM

ExpressionCodeGen Generates JVM bytecode directly from KtElement. This is a KtVisitor<StackValue[2]>. Has an InstructionAdapter (JVM bytecode emitter).

StackValue Sounds like a helper class to represent operands on the JVM stack. Has a couple subclasses.

ClassBodyCodeGen Generate class body from a KtPureClassOrObject with a ClassDescriptor. Also does bridge generation.

FunctionCodeGen Generate class body from a KtPureClassOrObject with a ClassDescriptor. Also does bridge generation.

10.1.2 JS

e.S. js.js.translator / PropertyTranslator. Looks that it directly translates descriptor + Psi (KtElement) into JS statement / expressions.

Also see FunctionBodyTranslator – takes a FunD and a KtElement (KtDeclWithBody).

10.2 IR-based Codegen

10.2.1 JVM

ClassCodegen Takes an IrClass and generates into a JvmBackendContext.

11 Compiler Plugin

11.1 Case study: All Open

The all-open plugin marks all classes annotated with some annotations to have the open modality. The implementation of the plugin lives in the Kotlin compiler repo and is easy to understand.

The plugin has a couple components: core (Kt compiler plugin), gradle plugin, and IDE plugin.

11.1.1 plugins.allopen.allopen-cli

Core implementation: the actual plugin (called extension), plugin registrar, and cli option processor.

AbstractAllOpenDeclarationAttributeAltererExtension Implements DeclarationAttributeAltererExtension which has a refineDeclarationModality method that overrides the modality of some declarations.

AllOpenCommandLineProcessor Read CLI options and store them as typed configuration keys. The options will be consumed by the registrar.

AllOpenComponentRegistrar Registers the extension as a Kotlin compiler plugin. Also see resource/META-INF/services for service discovery.

11.2 Plugin API

The plugin API is currently under changes – Some of the extensions that deal with type checking and call resolution are marked as unsable (e.g. InternalNonStableExtensionPoints) but jetpack-compose uses these.

Plugins are integrated to the compiler all over around the pipeline: some are used in frontend, some are used in backend.

ProjectExtensionDescriptor Knows how exactly to register an extension, and how to find one. Each extension has an instance of this to register itself.

DeclarationAttributeAltererExtension A frontend extension. Is used in ModifiersChecker to override modalities.

ExpressionCodegenExtension A backend extension. The noarg plugin uses this.

AnalysisHandlerExtension A Java frontend extension. Maybe it can help with overriding types?

12 Scratch

12.1 Questions

- What is a LazyTypeParameterDescriptor?
- What is a core.descriptors.MemberScope?
- What's the diff between TypeCheckerContext and TypeSystemContext?

12.2 Stack traces

12.2.1 Pipeline for `kotlin` Script Runner

Note that the runner uses coroutine in its eval methods and therefore can be hard to trace.

Runner: AbstractScriptEvaluationExtension → ScriptJvmCompilerImpls Parse: jvmCompilationUtil.getScriptKtFile: text-based Kt source → KtFile (Psi-based KtElement) Analysis: TopDownAnalyzerFacadeForJVM → LazyTopDownAnalyzer Codegen (Psi-based): KotlinCodegenFacade → DefaultCodegenFactory → THINGCodegen (e.g. Package / Member / Script / ImplementationBody / ClassBody) Eval: AbstractScriptEvaluationExtension → BasicJvmScriptEvaluator

12.2.2 Pipeline for `kotlinc` Compiler on Kt files

Runner: compiler.preloader / Preloader → compiler.cli / K2JVMCompiler → KotlinToJVMBytecodeCompiler (checks IR flag) Parse: Not sure Analysis: KotlinToJVMBytecodeCompiler.analyze → TopDownAnalyzerFacadeForJVM Codegen (Psi-based): KotlinToJVMBytecodeCompiler.generate → KotlinCodegenFacade Lowering (Ir-based): analysis is the same; but ktjvmbcc.generate is different? Uses JvmIrCodeGenFactory with a PhaseConfig. I.e., .generate → JvmIrCodeGenFactory → JvmBackendFacade → JvmLower → CompilerPhase.invokeToplevel(PhaseConfig, JvmBackendContext, IrModuleFragment) → a bunch of abstraction layers around lowering phases → SomeLoweringPass.lower Codegen (Ir-based): JvmBackendFacade → ClassCodeGen (IrFile.declarations should only contain IrClass after lowering) Write: KotlinToJVMBytecodeCompiler.writeOutputs

12.2.3 Pipeline for `kotlinc` Compiler on Kt file calling compiled Java code

Java code resolution happens at various places.

Type checking / call resolution By instantiating a Java class `val foo = JClass()`, Kotlin type checker will try to infer the result type of such call. To do that, it will consult CallResolver (resolveFunctionCall) to resolve the call first. This calls PSICallResolver (runResolutionAndInference), which goes through a chain of ResolutionScope (getContributedClassifer) calls (ScopeTower / TowerResolver / ChainedMemberScope / JvmPackageScope / LazyJavaPackageScope), which finally instantiates a LazyJavaClassD from the Jar.

Java class loading compiler.frontend.java contains some compiled Java class parsing functionalities.

BinaryJavaClass loads Jar files. It reads class members (fields etc) and then read their annotations.

BinaryClassSignatureParser parses types (type parameters, use-site variances etc).

12.2.4 Pipeline for Analysis

LazyTopDownAnalyzer.analyzeDeclarations → lots of resolvers and checkers. BodyResolver.resolveBodies: resolve behavior decl bodies (what is a behavior decl?), build CFG, check declarations, run extension checkers.

Looks that at least BodyResolver.resolveBehaviorDeclarationBodies does type checking. E.g. resolveFunctionBodies checks expression types in function bodies, using ExpressionTypingServices.

CFG depends on DeclarationDescriptors built from a previous step (See ControlFlowInformationProvider). Also seems to depend on method calls being resolved (See CFP (generateArrayAssignment) – at least for INDEXED_LVALUE_SET).

- INDEXED_LVALUE_{GET,SET} (e.g. `arr[ix] = val`): recorded in BasicExpressionTypingVisitor (resolveArrayAccessSpecialMethod). This is interesting – It

suggests that expression type checking and resolution are done in the same place.

There's a CFG to Dot graph printer: CFGraphToDotFilePrinter. It sounds useful to visualize the CFG, but it's in tests-common.

Nomenclature

Bridge Kotlin's way (i.e. doesn't require Java 8) of representing default implementation for interfaces in generate code. See compiler.backend-common / impl.kt's code doc.

Commonization Seems to be a klib process to strip source and original information from descriptors. For example see CommonizedClassConD. Maybe this is to reduce storage size for these Ds in klibs?

Con Constructor. Can be either a type constructor, or a value / class constructor like `<init>`

D DeclarationDescriptor or just Descriptor. See the DeclarationDescriptor section

Deser Deserialization. See core.deserialization – The compiler is able to serialize / deserialize descriptors. And it seems to be done lazily – Many deserialized descriptors still hold TypeTable / NameResolver / VersionRequirementTable to help with further deserialization.

FIR "Front-end IR", an intermediate level IR, below KtElement and above IR. See 5.

Fun Function

Impl Implementation

IR Can either be a general term (Intermediate Representation), or specifically Kotlin compiler's new experimental lowlevel IR (see 6).

J Java

Param Parameter

PSI JetBrain's universal parse tree API. See its doc here.

Stub In the context of PSI, this is the interface part of a PSI tree. It's initially calculated from PSI trees and then cached for efficient retrival. See JetBrain's doc.

Subst Substitution, usually in the context of type checking / reconstruction / unification algorithms. See Wikipedia: Unification and nLab: Substitution.

TyCon Type constructor. Basically a type-level function that takes zero or more types, and returns another type. E.g. List is a tycon: ∀ `a, List<a>`

Var Variable

Vis Visibility, as in public / private / internal etc.