Scala 3 Internals

Internal Draft (rev cc47c561aa)

Martin Odersky et al.

17 July 2021

Latest HTML version available at https://dotty.epfl.ch/docs/internals/

Contents

1 Backend Internals

The code for the backend is split up by functionality and assembled in the object `GenBCode`.

```
object GenBCode    --- [defines] -->        PlainClassBuilder       GenBCode also defi
     |                                              |
 [extends]                                      [extends]
     |                                              |
BCodeSyncAndTry    ---------------->        SyncAndTryBuilder
     |                                              |
BCodeBodyBuilder   ---------------->        PlainBodyBuilder
     |                                              |
BCodeSkelBuilder   ---------------->        PlainSkelBuilder
     |                                       /      |       \
 BCodeHelpers      ---------------->  BCClassGen BCAnnotGen ...  (more components)
     |      \       \
     |       \       \------------->  helper methods
     |        \       \------------>  JMirrorBuilder, JBeanInfoBuilder (uses some comp
     |         \
     |   BytecodeWriters  --------->        methods and classes to write byte code fil
     |
 BCodeTypes        ---------------->        maps and fields for common BTypes, class T
     |
BCodeIdiomatic     ---------------->        utilities for code generation, e.g. genPri
     |
  BCodeGlue        ---------------->        BType class, predefined BTypes
```

1.0.1 Data Flow

Compiler creates a `BCodePhase`, calls `runOn(compilationUnits)`.

- initializes fields of `GenBCode` defined in `BCodeTypes` (BType maps, common BTypes like `StringReference`)
- initialize `primitives` map defined in `scalaPrimitives` (maps primitive members, like `int.+`, to bytecode instructions)
- creates `BytecodeWriter`, `JMirrorBuilder` and `JBeanInfoBuilder` instances (on each compiler run)
- `buildAndSendToDisk(units)`: uses work queues, see below.
  - `BCodePhase.addToQ1` adds class trees to `q1`
  - `Worker1.visit` creates ASM `ClassNodes`, adds to `q2`. It creates one `PlainClassBuilder` for each compilation unit.
  - `Worker2.addToQ3` adds byte arrays (one for each class) to `q3`
  - `BCodePhase.drainQ3` writes byte arrays to disk

1.0.2 Architecture

The architecture of `GenBCode` is the same as in Scalac. It can be partitioned into weakly coupled components (called “subsystems” below):

1.0.2.1 (a) The queue subsystem

Queues mediate between processors, queues don't know what each processor does.

The first queue contains AST trees for compilation units, the second queue contains ASM ClassNodes, and finally the third queue contains byte arrays, ready for serialization to disk.

Currently the queue subsystem is all sequential, but as can be seen in http://magarciaepfl.github.io/scala/ the above design enables overlapping (a.1) building of `ClassNodes`, (a.2) intra-method optimizations, and (a.3) serialization to disk.

This subsystem is described in detail in `GenBCode.scala`

1.0.2.2 (b) Bytecode-level types, BType

The previous bytecode emitter goes to great lengths to reason about bytecode-level types in terms of Symbols.

`GenBCode` uses `BType` as a more direct representation. A `BType` is immutable, and a value class (once the rest of GenBCode is merged from http://magarciaepfl.github.io/scala/).

Whether value class or not, its API is the same. That API doesn't reach into the type checker. Instead, each method on a `BType` answers a question that can be answered based on the `BType` itself. Sounds too simple to be good? It's a good building block, that's what it is.

The internal representation of a `BType` is based on what the JVM uses: internal names (e.g. `Ljava/lang/String;`) and method descriptors; as defined in the JVM spec (that's why they aren't documented in `GenBCode`, just read the JVM 8 spec).

All things `BType` can be found in `BCodeGlue.scala`

1.0.2.3 (c) Utilities offering a more "high-level" API to bytecode emission

Bytecode can be emitted one opcode at a time, but there are recurring patterns that call for a simpler API.

For example, when emitting a load-constant, a dedicated instruction exists for emitting load-zero. Similarly, emitting a switch can be done according to one of two strategies.

All these utilities are encapsulated in file `BCodeIdiomatic.scala`. They know nothing about the type checker (because, just between us, they don't need to).

1.0.2.4 (d) Mapping between type-checker types and BTypes

So that (c) can remain oblivious to what AST trees contain, some bookkeepers are needed:

- Tracked: for a bytecode class (BType), its superclass, directly declared interfaces, and inner classes.

To understand how it's built, see:

```
final def exemplar(csym0: Symbol): Tracked = { ... }
```

Details in `BCodeTypes.scala`

1.0.2.5 (e) More "high-level" utilities for bytecode emission

In the spirit of `BCodeIdiomatic`, utilities are added in `BCodeHelpers` for emitting:

- bean info class
- mirror class and their forwarders
- android-specific creator classes
- annotations

1.0.2.6 (f) Building an ASM ClassNode given an AST TypeDef It's done by `PlainClassBuilder`(see `GenBCode.scala`).

2 Classpaths

When ran from the `dotty` script, this is the classloader stack:

```
=====================================================
class sun.misc.Launcher$AppClassLoader  <= corresponds to java.class.path
sun.misc.Launcher$AppClassLoader@591ce4fe
file:/mnt/data-local/Work/Workspace/dev-2.11/dotty/target/scala-2.11.0-
M7/dotty_2.11.0-M7-0.1-SNAPSHOT.jar:file:/home/sun/.ivy2/cache/org.scala-
lang/scala-library/jars/scala-library-2.11.0-M7.jar:file:/home/sun/.ivy2/cache/org.sca
lang/scala-reflect/jars/scala-reflect-2.11.0-M7.jar
=====================================================
class sun.misc.Launcher$ExtClassLoader  <= corresponds to sun.boot.class.path
sun.misc.Launcher$ExtClassLoader@77fe0d66
file:/usr/lib/jvm/java-7-oracle/jre/lib/ext/sunpkcs11.jar:file:/usr/lib/jvm/java-
7-oracle/jre/lib/ext/localedata.jar:file:/usr/lib/jvm/java-7-oracle/jre/lib/ext/zipfs.
7-oracle/jre/lib/ext/sunec.jar:file:/usr/lib/jvm/java-7-oracle/jre/lib/ext/sunjce_prov
7-oracle/jre/lib/ext/dnsns.jar
=====================================================
```

When running from sbt or Eclipse, the classloader stack is:

```
=====================================================
class sbt.classpath.ClasspathUtilities$$anon$1
sbt.classpath.ClasspathUtilities$$anon$1@22a29f97
file:/mnt/data-local/Work/Workspace/dev-2.11/dotty/target/scala-2.11.0-
M7/classes/:file:/home/sun/.ivy2/cache/org.scala-lang/scala-library/jars/scala-
library-2.11.0-M7.jar:file:/home/sun/.ivy2/cache/org.scala-lang/scala-
reflect/jars/scala-reflect-2.11.0-M7.jar:file:/home/sun/.ivy2/cache/org.scala-
lang.modules/scala-xml_2.11.0-M7/bundles/scala-xml_2.11.0-M7-1.0.0-
RC7.jar
=====================================================
class java.net.URLClassLoader
java.net.URLClassLoader@2167c879
file:/home/sun/.ivy2/cache/org.scala-lang/scala-library/jars/scala-
library-2.11.0-M7.jar:file:/home/sun/.ivy2/cache/org.scala-lang/scala-
compiler/jars/scala-compiler-2.11.0-M7.jar:file:/home/sun/.ivy2/cache/org.scala-
lang/scala-reflect/jars/scala-reflect-2.11.0-M7.jar:file:/home/sun/.ivy2/cache/org.sca
lang.modules/scala-xml_2.11.0-M6/bundles/scala-xml_2.11.0-M6-1.0.0-
RC6.jar:file:/home/sun/.ivy2/cache/org.scala-lang.modules/scala-parser-
combinators_2.11.0-M6/bundles/scala-parser-combinators_2.11.0-M6-1.0.0-
RC4.jar:file:/home/sun/.ivy2/cache/jline/jline/jars/jline-2.11.jar
=====================================================
class xsbt.boot.BootFilteredLoader
xsbt.boot.BootFilteredLoader@73c74402
not a URL classloader
=====================================================
class sun.misc.Launcher$AppClassLoader  <= corresponds to java.class.path
```

```
sun.misc.Launcher$AppClassLoader@612dcb8c
file:/home/sun/.sbt/.lib/0.13.0/sbt-launch.jar
=====================================================
class sun.misc.Launcher$ExtClassLoader  <= corresponds to sun.boot.class.path
sun.misc.Launcher$ExtClassLoader@58e862c
file:/usr/lib/jvm/java-7-oracle/jre/lib/ext/sunpkcs11.jar:file:/usr/lib/jvm/java-
7-oracle/jre/lib/ext/localedata.jar:file:/usr/lib/jvm/java-7-oracle/jre/lib/ext/zipfs.
7-oracle/jre/lib/ext/sunec.jar:file:/usr/lib/jvm/java-7-oracle/jre/lib/ext/sunjce_prov
7-oracle/jre/lib/ext/dnsns.jar
=====================================================
```

Since scala/dotty only pick up `java.class.path` and `sun.boot.class.path`, it's clear why Dotty crashes in sbt and Eclipse unless we set the boot classpath explicitly.

3 Core Data Structures

(The following is work in progress)

3.1 Symbols and SymDenotations

- why symbols are not enough: their contents change all the time
- they change themselvesSo a `Symbol`
- reference: string + sig

Dotc is different from most other compilers in that it is centered around the idea of maintaining views of various artifacts associated with code. These views are indexed by tne

A symbol refers to a definition in a source program. Traditionally, compilers store context-dependent data in a symbol table. The symbol then is the central reference to address context-dependent data. But for `scalac`'s requirements it turns out that symbols are both too little and too much for this task.

Too little: The attributes of a symbol depend on the phase. Examples: Types are gradually simplified by several phases. Owners are changed in phases `LambdaLift` (when methods are lifted out to an enclosing class) and Flatten (when all classes are moved to top level). Names are changed when private members need to be accessed from outside their class (for instance from a nested class or a class implementing a trait). So a functional compiler, a `Symbol` by itself met mean much. Instead we are more interested in the attributes of a symbol at a given phase.

`scalac` has a concept for "attributes of a symbol at

Too much: If a symbol is used to refer to a definition in another compilation unit, we get problems for incremental recompilation. The unit containing the symbol might be changed and recompiled, which might mean that the definition referred to by the symbol is deleted or changed. This leads to the problem of stale symbols that refer to definitions that no longer exist in this form. Scala 2 compiler tried to address this problem by rebinding symbols appearing in certain cross module references, but it turned out to be too difficult to do this reliably for all kinds of references. Scala 3 compiler attacks the problem at the root instead. The fundamental problem is that symbols are too specific to serve as a cross-module reference in a system with incremental compilation. They refer to a particular definition, but that definition may not persist unchanged after an edit.

`scalac` uses instead a different approach: A cross module reference is always type, either a `TermRef` or `TypeRef`. A reference type contains a prefix type and a name. The definition the type refers to is established dynamically based on these fields.

a system where sources can be recompiled at any instance,

the concept of a `Denotation`.

Since definitions are transformed by phases,

The Dotty project is a platform to develop new technology for Scala tooling and to try out concepts of future Scala language versions. Its compiler is a new design intended to reflect the lessons we learned from work with the Scala compiler. A clean redesign today will let us iterate faster with new ideas in the future.

Today we reached an important milestone: The Dotty compiler can compile itself, and the compiled compiler can act as a drop-in for the original one. This is what one calls a bootstrap.

3.2 Why is this important?

The main reason is that this gives us a some validation of the trustworthiness of the compiler itself. Compilers are complex beasts, and many things can go wrong. By far the worst things that can go wrong are bugs where incorrect code is produced. It's not fun debugging code that looks perfectly fine, yet gets translated to something subtly wrong by the compiler.

Having the compiler compile itself is a good test to demonstrate that the generated code has reached a certain level of quality. Not only is a compiler a large program (44k lines in the case of dotty), it is also one that exercises a large part of the language in quite intricate ways. Moreover, bugs in the code of a compiler don't tend to go unnoticed, precisely because every part of a compiler feeds into other parts and all together are necessary to produce a correct translation.

3.3 Are We Done Yet?

Far from it! The compiler is still very rough. A lot more work is needed to

- make it more robust, in particular when analyzing incorrect programs,
- improve error messages and warnings,
- improve the efficiency of some of the generated code,
- embed it in external tools such as sbt, REPL, IDEs,
- remove restrictions on what Scala code can be compiled,
- help in migrating Scala code that will have to be changed.

3.4 What Are the Next Steps?

Over the coming weeks and months, we plan to work on the following topics:

- Make snapshot releases.
- Get the Scala standard library to compile.
- Work on SBT integration of the compiler.
- Work on IDE support.
- Investigate the best way to obtaining a REPL.
- Work on the build infrastructure.

If you want to get your hands dirty with any of this, now is a good moment to get involved! To get started: https://github.com/lampepfl/dotty.

4 Dotty Overvall Structure

The compiler code is found in package dotty.tools. It spans the following three sub-packages:

```
backend     Compiler backends (currently for JVM and JS)
   dotc     The main compiler
     io     Helper modules for file access and classpath handling.
```

The dotc package contains some main classes that can be run as separate programs. The most important one is class Main. `Main` inherits from Driver which contains the highest level functions for starting a compiler and processing some sources. `Driver` in turn is based on two other high-level classes, Compiler and Run.

4.1 Package Structure

Most functionality of `scalac` is implemented in subpackages of `dotc`. Here's a list of sub-packages and their focus.

```
.
├── ast                    // Abstract syntax trees
├── config                 // Compiler configuration, settings, platform specific definit
├── core                   // Core data structures and operations, with specific subpacka
│   ├── classfile          // Reading of Java classfiles into core data structures
│   ├── tasty              // Reading and writing of TASTY files to/from core data struct
│   └── unpickleScala2     // Reading of Scala2 symbol information into core data structu
├── parsing                // Scanner and parser
├── printing               // Pretty-printing trees, types and other data
├── repl                   // The interactive REPL
├── reporting              // Reporting of error messages, warnings and other info.
├── rewrites               // Helpers for rewriting Scala 2's constructs into dotty's.
├── semanticdb             // Helpers for exporting semanticdb from trees.
├── transform              // Miniphases and helpers for tree transformations.
├── typer                  // Type-checking and other frontend phases
└── util                   // General purpose utility classes and modules.
```

4.2 Contexts

`scalac` has almost no global state (the only significant bit of global state is the name table, which is used to hash strings into unique names). Instead, all essential bits of information that can vary over a compiler run are collected in a Context. Most methods in `scalac` take a `Context` value as an implicit parameter.

Contexts give a convenient way to customize values in some part of the call-graph. To run, e.g. some compiler function `f` at a given phase `phase`, we invoke `f` with an explicit context parameter, like this

```
f(/*normal args*/)(using ctx.withPhase(phase))
```

This assumes that `f` is defined in the way most compiler functions are:

```
def f(/*normal parameters*/)(implicit ctx: Context) ...
```

Compiler code follows the convention that all implicit `Context` parameters are named `ctx`. This is important to avoid implicit ambiguities in the case where nested methods contain each a Context parameters. The common name ensures then that the implicit parameters properly shadow each other.

Sometimes we want to make sure that implicit contexts are not captured in closures or other long-lived objects, be it because we want to enforce that nested methods each get their own implicit context, or because we want to avoid a space leak in the case where a closure can survive several compiler runs. A typical case is a completer for a symbol representing an external class, which produces the attributes of the symbol on demand, and which might never be invoked. In that case we follow the convention that any context parameter is explicit, not implicit, so we can track where it is used, and that it has a name different from `ctx`. Commonly used is `ictx` for "initialization context".

With these two conventions in place, it has turned out that implicit contexts work amazingly well as a device for dependency injection and bulk parameterization. There is of course always the danger that an unexpected implicit will be passed, but in practice this has not turned out to be much of a problem.

4.3 Compiler Phases

Seen from a temporal perspective, the `scalac` compiler consists of a list of phases. The current list of phases is specified in class Compiler as follows:

```
def phases: List[List[Phase]] =
frontendPhases ::: picklerPhases ::: transformPhases ::: backendPhases

/** Phases dealing with the frontend up to trees ready for TASTY pickling */
protected def frontendPhases: List[List[Phase]] =
  List(new FrontEnd) ::           // Compiler frontend: scanner, parser, namer, type
  List(new YCheckPositions) ::    // YCheck positions
  List(new sbt.ExtractDependencies) :: // Sends information on classes' dependencies
  List(new semanticdb.ExtractSemanticDB) :: // Extract info into .semanticdb files
  List(new PostTyper) ::          // Additional checks and cleanups after type check
  List(new sjs.PrepJSInterop) ::  // Additional checks and transformations for Scala
  List(new Staging) ::            // Check PCP, heal quoted types and expand macros
  List(new sbt.ExtractAPI) ::     // Sends a representation of the API of classes to
  List(new SetRootTree) ::        // Set the `rootTreeOrProvider` on class symbols
  Nil

/** Phases dealing with TASTY tree pickling and unpickling */
protected def picklerPhases: List[List[Phase]] =
  List(new Pickler) ::            // Generate TASTY info
  List(new PickleQuotes) ::       // Turn quoted trees into explicit run-time data s
  Nil
```

```
/** Phases dealing with the transformation from pickled trees to backend trees */
protected def transformPhases: List[List[Phase]] =
  List(new FirstTransform,         // Some transformations to put trees into a canon
       new CheckReentrant,         // Internal use only: Check that compiled program
       new ElimPackagePrefixes,    // Eliminate references to package prefixes in Se
       new CookComments,           // Cook the comments: expand variables, doc, etc.
       new CheckStatic,            // Check restrictions that apply to @static membe
       new BetaReduce,             // Reduce closure applications
       new init.Checker) ::        // Check initialization of objects
  List(new ElimRepeated,           // Rewrite vararg parameters and arguments
       new ExpandSAMs,             // Expand single abstract method closures to anon
       new ProtectedAccessors,     // Add accessors for protected members
       new ExtensionMethods,       // Expand methods of value classes with extension
       new UncacheGivenAliases,    // Avoid caching RHS of simple parameterless give
       new ByNameClosures,         // Expand arguments to by-name parameters to clos
       new HoistSuperArgs,         // Hoist complex arguments of supercalls to enclo
       new SpecializeApplyMethods, // Adds specialized methods to FunctionN
       new RefChecks) ::           // Various checks mostly related to abstract memb
  List(new ElimOpaque,             // Turn opaque into normal aliases
       new TryCatchPatterns,       // Compile cases in try/catch
       new PatternMatcher,         // Compile pattern matches
       new sjs.ExplicitJSClasses,  // Make all JS classes explicit (Scala.js only)
       new ExplicitOuter,          // Add accessors to outer classes from nested one
       new ExplicitSelf,           // Make references to non-trivial self types expl
       new ElimByName,             // Expand by-name parameter references
       new StringInterpolatorOpt) :: // Optimizes raw and s string interpolators by
  List(new PruneErasedDefs,        // Drop erased definitions from scopes and simpli
       new InlinePatterns,         // Remove placeholders of inlined patterns
       new VCInlineMethods,        // Inlines calls to value class methods
       new SeqLiterals,            // Express vararg arguments as arrays
       new InterceptedMethods,     // Special handling of `==`, `|=`, `getClass` met
       new Getters,                // Replace non-private vals and vars with getter
       new SpecializeFunctions,    // Specialized Function{0,1,2} by replacing super
       new LiftTry,                // Put try expressions that might execute on non-
       new CollectNullableFields,  // Collect fields that can be nulled out after us
       new ElimOuterSelect,        // Expand outer selections
       new ResolveSuper,           // Implement super accessors
       new FunctionXXLForwarders,  // Add forwarders for FunctionXXL apply method
       new ParamForwarding,        // Add forwarders for aliases of superclass param
       new TupleOptimizations,     // Optimize generic operations on tuples
       new LetOverApply,            // Lift blocks from receivers of applications
       new ArrayConstructors) ::   // Intercept creation of (non-generic) arrays and
  List(new Erasure) ::             // Rewrite types to JVM model, erasing all type p
  List(new ElimErasedValueType,    // Expand erased value types to their underlying
       new PureStats,              // Remove pure stats from blocks
       new VCElideAllocations,     // Peep-hole optimization to eliminate unnecessar
```

```
        new ArrayApply,             // Optimize `scala.Array.apply([....])` and `scal
        new sjs.AddLocalJSFakeNews, // Adds fake new invocations to local JS classes
        new ElimPolyFunction,       // Rewrite PolyFunction subclasses to FunctionN s
        new TailRec,                // Rewrite tail recursion to loops
        new CompleteJavaEnums,      // Fill in constructors for Java enums
        new Mixin,                  // Expand trait fields and trait initializers
        new LazyVals,               // Expand lazy vals
        new Memoize,                // Add private fields to getters and setters
        new NonLocalReturns,        // Expand non-local returns
        new CapturedVars) ::        // Represent vars captured by closures as heap ob
    List(new Constructors,          // Collect initialization code in primary constru
                                       // Note: constructors changes decls in transfo
        new Instrumentation) ::     // Count calls and allocations under -Yinstrument
    List(new LambdaLift,            // Lifts out nested functions to class scope, sto
                                    // Note: in this mini-phase block scopes are inco
        new ElimStaticThis,         // Replace `this` references to static objects by
        new CountOuterAccesses) ::  // Identify outer accessors that can be dropped
    List(new DropOuterAccessors,    // Drop unused outer accessors
        new Flatten,                // Lift all inner classes to package scope
        new RenameLifted,           // Renames lifted classes to local numbering sche
        new TransformWildcards,     // Replace wildcards with default values
        new MoveStatics,            // Move static methods from companion to the clas
        new ExpandPrivate,          // Widen private definitions accessed from nested
        new RestoreScopes,          // Repair scopes rendered invalid by moving defin
        new SelectStatic,           // get rid of selects that would be compiled into
        new sjs.JUnitBootstrappers, // Generate JUnit-specific bootstrapper classes f
        new CollectSuperCalls) ::   // Find classes that are called with super
    Nil

  /** Generate the output of the compilation */
  protected def backendPhases: List[List[Phase]] =
    List(new backend.sjs.GenSJSIR) :: // Generate .sjsir files for Scala.js (not enabl
    List(new GenBCode) ::             // Generate JVM bytecode
    Nil
```

Note that phases are grouped, so the `phases` method is of type `List[List[Phase]]`. The idea is that all phases in a group are fused into a single tree traversal. That way, phases can be kept small (most phases perform a single function) without requiring an excessive number of tree traversals (which are costly, because they have generally bad cache locality).

Phases fall into four categories:

- Frontend phases: `Frontend`, `PostTyper` and `Pickler`. `FrontEnd` parses the source programs and generates untyped abstract syntax trees, which are then typechecked and transformed into typed abstract syntax trees. `PostTyper` performs checks and cleanups that require a fully typed program. In particular, it
  - creates super accessors representing `super` calls in traits

  - creates implementations of synthetic (compiler-implemented) methods
  - avoids storing parameters passed unchanged from subclass to superclass in duplicate fields.

  Finally `Pickler` serializes the typed syntax trees produced by the frontend as TASTY data structures.

- High-level transformations: All phases from `FirstTransform` to `Erasure`. Most of these phases transform syntax trees, expanding high-level constructs to more primitive ones. The last phase in the group, `Erasure` translates all types into types supported directly by the JVM. To do this, it performs another type checking pass, but using the rules of the JVM's type system instead of Scala's.

- Low-level transformations: All phases from `ElimErasedValueType` to `CollectSuperCalls`. These further transform trees until they are essentially a structured version of Java bytecode.

- Code generators: These map the transformed trees to Java classfiles or .sjsir files.

5 Dotc's concept of time

Conceptually, the `scalac` compiler's job is to maintain views of various artifacts associated with source code at all points in time. But what is time for `scalac`? In fact, it is a combination of compiler runs and compiler phases.

The hours of the compiler's clocks are measured in compiler runs. Every run creates a new hour, which follows all the compiler runs (hours) that happened before. `scalac` is designed to be used as an incremental compiler that can support incremental builds, as well as interactions in an IDE and a REPL. This means that new runs can occur quite frequently. At the extreme, every keystroke in an editor or REPL can potentially launch a new compiler run, so potentially an "hour" of compiler time might take only a fraction of a second in real time.

The minutes of the compiler's clocks are measured in phases. At every compiler run, the compiler cycles through a number of phases. The list of phases is defined in the [Compiler]object There are currently about 60 phases per run, so the minutes/hours analogy works out roughly. After every phase the view the compiler has of the world changes: trees are transformed, types are gradually simplified from Scala types to JVM types, definitions are rearranged, and so on.

Many pieces in the information compiler are time-dependent. For instance, a Scala symbol representing a definition has a type, but that type will usually change as one goes from the higher-level Scala view of things to the lower-level JVM view. There are different ways to deal with this. Many compilers change the type of a symbol destructively according to the "current phase". Another, more functional approach might be to have different symbols representing the same definition at different phases, which each symbol carrying a different immutable type. `scalac` employs yet another scheme, which is inspired by functional reactive programming (FRP): Symbols carry not a single type, but a function from compiler phase to type. So the type of a symbol is a time-indexed function, where time ranges over compiler phases.

Typically, the definition of a symbol or other quantity remains stable for a number of phases. This leads us to the concept of a period. Conceptually, period is an interval of some given phases in a given compiler run. Periods are conceptually represented by three pieces of information

- the ID of the current run,
- the ID of the phase starting the period
- the number of phases in the period

All three pieces of information are encoded in a value class over a 32 bit integer. Here's the API for class `Period`:

```
class Period(val code: Int) extends AnyVal {
  def runId: RunId              // The run identifier of this period.
  def firstPhaseId: PhaseId     // The first phase of this period
  def lastPhaseId: PhaseId      // The last phase of this period
  def phaseId: PhaseId          // The phase identifier of this single-phase period

  def containsPhaseId(id: PhaseId): Boolean
```

```
  def contains(that: Period): Boolean
  def overlaps(that: Period): Boolean

  def & (that: Period): Period
  def | (that: Period): Period
}
```

We can access the parts of a period using `runId`, `firstPhaseId`, `lastPhaseId`, or using `phaseId` for periods consisting only of a single phase. They return `RunId` or `PhaseId` values, which are aliases of `Int`. `containsPhaseId`, `contains` and `overlaps` test whether a period contains a phase or a period as a sub-interval, or whether the interval overlaps with another period. Finally, `&` and `|` produce the intersection and the union of two period intervals (the union operation `|` takes as `runId` the `runId` of its left operand, as periods spanning different `runId`s cannot be constructed.

Periods are constructed using two `apply` methods:

```
object Period {
  /** The single-phase period consisting of given run id and phase id */
  def apply(rid: RunId, pid: PhaseId): Period

  /** The period consisting of given run id, and lo/hi phase ids */
  def apply(rid: RunId, loPid: PhaseId, hiPid: PhaseId): Period
}
```

As a sentinel value there's `Nowhere`, a period that is empty.

6 Scala 3 Syntax Summary

The following description of Scala tokens uses literal characters `‘c’` when referring to the ASCII fragment `\u0000` – `\u007F`.

Unicode escapes are used to represent the Unicode character with the given hexadecimal code:

```
UnicodeEscape ::= ‘\’ ‘u’ {‘u’} hexDigit hexDigit hexDigit hexDigit
hexDigit      ::= ‘0’ | … | ‘9’ | ‘A’ | … | ‘F’ | ‘a’ | … | ‘f’
```

Informal descriptions are typeset as `“some comment”`.

6.0.1 Lexical Syntax

The lexical syntax of Scala is given by the following grammar in EBNF form.

```
whiteSpace       ::=  ‘\u0020’ | ‘\u0009’ | ‘\u000D’ | ‘\u000A’
upper            ::=  ‘A’ | … | ‘Z’ | ‘\$’ | ‘_’  “… and Unicode category Lu”
lower            ::=  ‘a’ | … | ‘z’ “… and Unicode category Ll”
letter           ::=  upper | lower “… and Unicode categories Lo, Lt, Lm, Nl”
digit            ::=  ‘0’ | … | ‘9’
paren            ::=  ‘(’ | ‘)’ | ‘[’ | ‘]’ | ‘{’ | ‘}’ | ‘'(’ | ‘'[’ | ‘'{’
delim            ::=  ‘`’ | ‘'’ | ‘"’ | ‘.’ | ‘;’ | ‘,’
opchar           ::=  ‘!’ | ‘#’ | ‘%’ | ‘&’ | ‘*’ | ‘+’ | ‘-’ | ‘/’ | ‘:’ |
                      ‘<’ | ‘=’ | ‘>’ | ‘?’ | ‘@’ | ‘\’ | ‘^’ | ‘|’ | ‘~’
                      “… and Unicode categories Sm, So”
printableChar    ::=  “all characters in [\u0020, \u007E] inclusive”
charEscapeSeq    ::=  ‘\’ (‘b’ | ‘t’ | ‘n’ | ‘f’ | ‘r’ | ‘"’ | ‘'’ | ‘\’)

op               ::=  opchar {opchar}
varid            ::=  lower idrest
alphaid          ::=  upper idrest
                   |  varid
plainid          ::=  alphaid
                   |  op
id               ::=  plainid
                   |  ‘`’ { charNoBackQuoteOrNewline | UnicodeEscape | charEscapeSeq }
idrest           ::=  {letter | digit} [‘_’ op]
quoteId          ::=  ‘'’ alphaid

integerLiteral   ::=  (decimalNumeral | hexNumeral) [‘L’ | ‘l’]
decimalNumeral   ::=  ‘0’ | nonZeroDigit [{digit | ‘_’} digit]
hexNumeral       ::=  ‘0’ (‘x’ | ‘X’) hexDigit [{hexDigit | ‘_’} hexDigit]
nonZeroDigit     ::=  ‘1’ | … | ‘9’

floatingPointLiteral
                 ::=  [decimalNumeral] ‘.’ digit [{digit | ‘_’} digit] [exponentPart]
```

```
                   |  decimalNumeral exponentPart [floatType]
                   |  decimalNumeral floatType
exponentPart     ::=  (‘E’ | ‘e’) [‘+’ | ‘-’] digit [{digit | ‘_’} digit]
floatType        ::=  ‘F’ | ‘f’ | ‘D’ | ‘d’

booleanLiteral   ::=  ‘true’ | ‘false’

characterLiteral ::=  ‘'’ (printableChar | charEscapeSeq) ‘'’

stringLiteral    ::=  ‘"’ {stringElement} ‘"’
                   |  ‘"""’ multiLineChars ‘"""’
stringElement    ::=  printableChar \ (‘"’ | ‘\’)
                   |  UnicodeEscape
                   |  charEscapeSeq
multiLineChars   ::=  {[‘"’] [‘"’] char \ ‘"’} {‘"’}
processedStringLiteral
                 ::=  alphaid ‘"’ {[‘\’] processedStringPart | ‘\\’ | ‘\"’} ‘"’
                   |  alphaid ‘"""’ {[‘"’] [‘"’] char \ (‘"’ | ‘$’) | escape} {‘"’} ‘"
processedStringPart
                 ::= printableChar \ (‘"’ | ‘$’ | ‘\’) | escape
escape           ::=  ‘$$’
                   |  ‘$’ letter { letter | digit }
                   |  ‘{’ Block  [‘;’ whiteSpace stringFormat whiteSpace] ‘}’
stringFormat     ::=  {printableChar \ (‘"’ | ‘}’ | ‘ ’ | ‘\t’ | ‘\n’)}

symbolLiteral    ::=  ‘'’ plainid // until 2.13

comment          ::=  ‘/*’ “any sequence of characters; nested comments are allowed” ‘
                   |  ‘//’ “any sequence of characters up to end of line”

nl               ::=  “new line character”
semi             ::=  ‘;’ |  nl {nl}
```

6.1 Optional Braces

The lexical analyzer also inserts `indent` and `outdent` tokens that represent regions of indented code at certain points

In the context-free productions below we use the notation <<< `ts` >>> to indicate a token sequence `ts` that is either enclosed in a pair of braces `{ ts }` or that constitutes an indented region `indent ts outdent`. Analogously, the notation :<<< `ts` >>> indicates a token sequence `ts` that is either enclosed in a pair of braces `{ ts }` or that constitutes an indented region `indent ts outdent` that follows a `:` at the end of a line.

```
 <<< ts >>>   ::=  ‘{’ ts ‘}’
                |  indent ts outdent
:<<< ts >>>   ::=  [nl] ‘{’ ts ‘}’
```

```
                 |  `:` indent ts outdent
```

6.2 Keywords

6.2.1 Regular keywords

```
abstract  case      catch     class     def       do        else
enum      export    extends   false     final     finally   for
given     if        implicit  import    lazy      match     new
null      object    override  package   private   protected return
sealed    super     then      throw     trait     true      try
type      val       var       while     with      yield
:         =         <-        =>        <:        >:        #
@         =>>       ?=>
```

6.2.2 Soft keywords

```
as  derives  end  extension  infix  inline  opaque  open  transparent  using  |  *  +
```

See the separate section on soft keywords for additional details on where a soft keyword is recognized.

6.3 Context-free Syntax

The context-free syntax of Scala is given by the following EBNF grammar:

6.3.1 Literals and Paths

```
SimpleLiteral     ::=  ['-'] integerLiteral
                    |  ['-'] floatingPointLiteral
                    |  booleanLiteral
                    |  characterLiteral
                    |  stringLiteral
Literal           ::=  SimpleLiteral
                    |  processedStringLiteral
                    |  symbolLiteral
                    |  'null'

QualId            ::=  id {'.' id}
ids               ::=  id {',' id}

SimpleRef         ::=  id
                    |  [id '.'] 'this'
                    |  [id '.'] 'super' [ClassQualifier] '.' id

ClassQualifier    ::=  '[' id ']'
```

6.3.2 Types

```
Type              ::=  FunType
                    |  HkTypeParamClause ‘=>>’ Type                         LambdaT
                    |  FunParamClause ‘=>>’ Type                            TermLa
                    |  MatchType
                    |  InfixType
FunType           ::=  FunTypeArgs (‘=>’ | ‘?=>’) Type                      Functi
                    |  HKTypeParamClause '=>' Type                          PolyFu
FunTypeArgs       ::=  InfixType
                    |  ‘(’ [ FunArgTypes ] ‘)’
                    |  FunParamClause
FunParamClause    ::=  ‘(’ TypedFunParam {‘,’ TypedFunParam } ‘)’
TypedFunParam     ::=  id ‘:’ Type
MatchType         ::=  InfixType `match` <<< TypeCaseClauses >>>
InfixType         ::=  RefinedType {id [nl] RefinedType}                    InfixO
RefinedType       ::=  AnnotType {[nl] Refinement}                          Refine
AnnotType         ::=  SimpleType {Annotation}                              Annota

SimpleType        ::=  SimpleLiteral                                        Single
                    |  ‘?’ TypeBounds
                    |  SimpleType1
SimpleType1       ::=  id                                                   Ident(
                    |  Singleton ‘.’ id                                     Select
                    |  Singleton ‘.’ ‘type’                                 Single
                    |  ‘(’ Types ‘)’                                        Tuple(
                    |  Refinement                                           Refine
                    |  ‘$’ ‘{’ Block ‘}’
                    |  SimpleType1 TypeArgs                                 Applie
                    |  SimpleType1 ‘#’ id                                   Select
Singleton         ::=  SimpleRef
                    |  SimpleLiteral
                    |  Singleton ‘.’ id
Singletons        ::=  Singleton { ‘,’ Singleton }
FunArgType        ::=  Type
                    |  ‘=>’ Type                                            Prefix
FunArgTypes       ::=  FunArgType { ‘,’ FunArgType }
ParamType         ::=  [‘=>’] ParamValueType
ParamValueType    ::=  Type [‘*’]                                           Postfi
TypeArgs          ::=  ‘[’ Types ‘]’                                        ts
Refinement        ::=  ‘{’ [RefineDcl] {semi [RefineDcl]} ‘}’               ds
TypeBounds        ::=  [‘>:’ Type] [‘<:’ Type]                              TypeBo
TypeParamBounds   ::=  TypeBounds {‘:’ Type}                                Contex
Types             ::=  Type {‘,’ Type}
```

6.3.3 Expressions

```
Expr              ::=  FunParams (‘=>’ | ‘?=>’) Expr                           Functi
                    |  HkTypeParamClause ‘=>’ Expr                             PolyFu
                    |  Expr1
BlockResult       ::=  FunParams (‘=>’ | ‘?=>’) Block
                    |  HkTypeParamClause ‘=>’ Block
                    |  Expr1
FunParams         ::=  Bindings
                    |  id
                    |  ‘_’
Expr1             ::=  [‘inline’] ‘if’ ‘(’ Expr ‘)’ {nl} Expr [[semi] ‘else’ Expr] If(
                    |  [‘inline’] ‘if’  Expr ‘then’ Expr [[semi] ‘else’ Expr]    If(co
                    |  ‘while’ ‘(’ Expr ‘)’ {nl} Expr                          WhileD
                    |  ‘while’ Expr ‘do’ Expr                                  WhileD
                    |  ‘try’ Expr Catches [‘finally’ Expr]                     Try(ex
                    |  ‘try’ Expr [‘finally’ Expr]                             Try(ex
                    |  ‘throw’ Expr                                            Throw(
                    |  ‘return’ [Expr]                                         Return
                    |  ForExpr
                    |  [SimpleExpr ‘.’] id ‘=’ Expr                            Assign
                    |  SimpleExpr1 ArgumentExprs ‘=’ Expr                      Assign
                    |  PostfixExpr [Ascription]
                    |  ‘inline’ InfixExpr MatchClause
Ascription        ::=  ‘:’ InfixType                                           Typed(
                    |  ‘:’ Annotation {Annotation}                             Typed(
Catches           ::=  ‘catch’ (Expr | ExprCaseClause)
PostfixExpr       ::=  InfixExpr [id]                                          Postfi
InfixExpr         ::=  PrefixExpr
                    |  InfixExpr id [nl] InfixExpr                             InfixO
                    |  InfixExpr id ‘:’ IndentedExpr
                    |  InfixExpr MatchClause
MatchClause       ::=  ‘match’ <<< CaseClauses >>>                             Match(
PrefixExpr        ::=  [‘-’ | ‘+’ | ‘~’ | ‘!’] SimpleExpr                      Prefix
SimpleExpr        ::=  SimpleRef
                    |  Literal
                    |  ‘_’
                    |  BlockExpr
                    |  ‘$’ ‘{’ Block ‘}’
                    |  Quoted
                    |  quoteId                                                 -- onl
                    |  ‘new’ ConstrApp {‘with’ ConstrApp} [TemplateBody]       New(co
                    |  ‘new’ TemplateBody
                    |  ‘(’ ExprsInParens ‘)’                                   Parens
                    |  SimpleExpr ‘.’ id                                       Select
                    |  SimpleExpr ‘.’ MatchClause
                    |  SimpleExpr TypeArgs                                     TypeAp
```

```
                    |  SimpleExpr ArgumentExprs                               Apply(
                    |  SimpleExpr1 ‘:’ IndentedExpr                           -- und
                    |  SimpleExpr1 FunParams (‘=>’ | ‘?=>’) IndentedExpr      -- und
                    |  SimpleExpr ‘_’                                         Postfi
                    |  XmlExpr                          -- to be dropped
IndentedExpr      ::=  indent CaseClauses | Block outdent
Quoted            ::=  ‘'’ ‘{’ Block ‘}’
                    |  ‘'’ ‘[’ Type ‘]’
ExprsInParens     ::=  ExprInParens {‘,’ ExprInParens}
ExprInParens      ::=  PostfixExpr ‘:’ Type                                   -- nor
                    |  Expr
ParArgumentExprs  ::=  ‘(’ [‘using’] ExprsInParens ‘)’                        exprs
                    |  ‘(’ [ExprsInParens ‘,’] PostfixExpr ‘*’ ‘)’            exprs
ArgumentExprs     ::=  ParArgumentExprs
                    |  BlockExpr
BlockExpr         ::=  <<< CaseClauses | Block >>>
Block             ::=  {BlockStat semi} [BlockResult]                         Block(
BlockStat         ::=  Import
                    |  {Annotation {nl}} {LocalModifier} Def
                    |  Extension
                    |  Expr1
                    |  EndMarker

ForExpr           ::=  ‘for’ ‘(’ Enumerators0 ‘)’ {nl} [‘do‘ | ‘yield’] Expr   ForYi
                    |  ‘for’ ‘{’ Enumerators0 ‘}’ {nl} [‘do‘ | ‘yield’] Expr
                    |  ‘for’     Enumerators0          (‘do‘ | ‘yield’) Expr
Enumerators0      ::=  {nl} Enumerators [semi]
Enumerators       ::=  Generator {semi Enumerator | Guard}
Enumerator        ::=  Generator
                    |  Guard
                    |  Pattern1 ‘=’ Expr                                      GenAli
Generator         ::=  [‘case’] Pattern1 ‘<-’ Expr
Guard             ::=  ‘if’ PostfixExpr

CaseClauses       ::=  CaseClause { CaseClause }                              Match(
CaseClause        ::=  ‘case’ Pattern [Guard] ‘=>’ Block                      CaseDe
ExprCaseClause    ::=  ‘case’ Pattern [Guard] ‘=>’ Expr
TypeCaseClauses   ::=  TypeCaseClause { TypeCaseClause }
TypeCaseClause    ::=  ‘case’ InfixType ‘=>’ Type [nl]

Pattern           ::=  Pattern1 { ‘|’ Pattern1 }                              Altern
Pattern1          ::=  Pattern2 [‘:’ RefinedType]                             Bind(n
Pattern2          ::=  [id ‘@’] InfixPattern                                  Bind(n
InfixPattern      ::=  SimplePattern { id [nl] SimplePattern }                InfixO
SimplePattern     ::=  PatVar                                                 Ident(
                    |  Literal                                                Bind(n
```

```
                     |  ‘(’ [Patterns] ‘)’                                   Parens
                     |  Quoted
                     |  XmlPattern (to be dropped)
                     |  SimplePattern1 [TypeArgs] [ArgumentPatterns]
                     |  ‘given’ RefinedType
SimplePattern1    ::=  SimpleRef
                     |  SimplePattern1 ‘.’ id
PatVar            ::=  varid
                     |  ‘_’
Patterns          ::=  Pattern {‘,’ Pattern}
ArgumentPatterns  ::=  ‘(’ [Patterns] ‘)’                                   Apply(
                     |  ‘(’ [Patterns ‘,’] PatVar ‘*’ ‘)’
```

6.3.4 Type and Value Parameters

```
ClsTypeParamClause::=  ‘[’ ClsTypeParam {‘,’ ClsTypeParam} ‘]’
ClsTypeParam      ::=  {Annotation} [‘+’ | ‘-’]                             TypeDe
                       id [HkTypeParamClause] TypeParamBounds               Bound(

DefTypeParamClause::=  ‘[’ DefTypeParam {‘,’ DefTypeParam} ‘]’
DefTypeParam      ::=  {Annotation} id [HkTypeParamClause] TypeParamBounds

TypTypeParamClause::=  ‘[’ TypTypeParam {‘,’ TypTypeParam} ‘]’
TypTypeParam      ::=  {Annotation} id [HkTypeParamClause] TypeBounds

HkTypeParamClause ::=  ‘[’ HkTypeParam {‘,’ HkTypeParam} ‘]’
HkTypeParam       ::=  {Annotation} [‘+’ | ‘-’] (id [HkTypeParamClause] | ‘_’)
                       TypeBounds

ClsParamClauses   ::=  {ClsParamClause} [[nl] ‘(’ [‘implicit’] ClsParams ‘)’]
ClsParamClause    ::=  [nl] ‘(’ ClsParams ‘)’
                     |  [nl] ‘(’ ‘using’ (ClsParams | FunArgTypes) ‘)’
ClsParams         ::=  ClsParam {‘,’ ClsParam}
ClsParam          ::=  {Annotation}                                         ValDef
                       [{Modifier} (‘val’ | ‘var’) | ‘inline’] Param
Param             ::=  id ‘:’ ParamType [‘=’ Expr]

DefParamClauses   ::=  {DefParamClause} [[nl] ‘(’ [‘implicit’] DefParams ‘)’]
DefParamClause    ::=  [nl] ‘(’ DefParams ‘)’ | UsingParamClause
UsingParamClause  ::=  [nl] ‘(’ ‘using’ (DefParams | FunArgTypes) ‘)’
DefParams         ::=  DefParam {‘,’ DefParam}
DefParam          ::=  {Annotation} [‘inline’] Param                        ValDef
```

6.3.5 Bindings and Imports

```
Bindings          ::=  ‘(’ [Binding {‘,’ Binding}] ‘)’
Binding           ::=  (id | ‘_’) [‘:’ Type]                                ValDef
```

```
Modifier          ::=  LocalModifier
                    |  AccessModifier
                    |  ‘override’
                    |  ‘opaque’
LocalModifier     ::=  ‘abstract’
                    |  ‘final’
                    |  ‘sealed’
                    |  ‘open’
                    |  ‘implicit’
                    |  ‘lazy’
                    |  ‘inline’
AccessModifier    ::=  (‘private’ | ‘protected’) [AccessQualifier]
AccessQualifier   ::=  ‘[’ id ‘]’

Annotation        ::=  ‘@’ SimpleType1 {ParArgumentExprs}                    Appl

Import            ::=  ‘import’ ImportExpr {‘,’ ImportExpr}
Export            ::=  ‘export’ ImportExpr {‘,’ ImportExpr}
ImportExpr        ::=  SimpleRef {‘.’ id} ‘.’ ImportSpec                     Impo
                    |  SimpleRef ‘as’ id                                     Impo
ImportSpec        ::=  NamedSelector
                    |  WildcardSelector
                    | ‘{’ ImportSelectors) ‘}’
NamedSelector     ::=  id [‘as’ (id | ‘_’)]
WildCardSelector  ::=  ‘*' | ‘given’ [InfixType]
ImportSelectors   ::=  NamedSelector [‘,’ ImportSelectors]
                    |  WildCardSelector {‘,’ WildCardSelector}

EndMarker         ::=  ‘end’ EndMarkerTag    -- when followed by EOL
EndMarkerTag      ::=  id | ‘if’ | ‘while’ | ‘for’ | ‘match’ | ‘try’
                    |  ‘new’ | ‘this’ | ‘given’ | ‘extension’ | ‘val’
```

6.3.6 Declarations and Definitions

```
RefineDcl         ::=  ‘val’ ValDcl
                    |  ‘def’ DefDcl
                    |  ‘type’ {nl} TypeDcl
Dcl               ::=  RefineDcl
                    |  ‘var’ VarDcl
ValDcl            ::=  ids ‘:’ Type                                         PatDef
VarDcl            ::=  ids ‘:’ Type                                         PatDef
DefDcl            ::=  DefSig ‘:’ Type                                      DefDef
DefSig            ::=  id [DefTypeParamClause] DefParamClauses
TypeDcl           ::=  id [TypeParamClause] {FunParamClause} TypeBounds     TypeDe
                       [‘=’ Type]
```

```
Def               ::=  'val' PatDef
                    |  'var' PatDef
                    |  'def' DefDef
                    |  'type' {nl} TypeDcl
                    |  TmplDef
PatDef            ::=  ids [':' Type] '=' Expr
                    |  Pattern2 [':' Type] '=' Expr                 PatDef
DefDef            ::=  DefSig [':' Type] '=' Expr                   DefDef
                    |  'this' DefParamClause DefParamClauses '=' ConstrExpr     DefDef

TmplDef           ::=  (['case'] 'class' | 'trait') ClassDef
                    |  ['case'] 'object' ObjectDef
                    |  'enum' EnumDef
                    |  'given' GivenDef
ClassDef          ::=  id ClassConstr [Template]                    ClassD
ClassConstr       ::=  [ClsTypeParamClause] [ConstrMods] ClsParamClauses     with D
ConstrMods        ::=  {Annotation} [AccessModifier]
ObjectDef         ::=  id [Template]                                ModuleD
EnumDef           ::=  id ClassConstr InheritClauses EnumBody
GivenDef          ::=  [GivenSig] (AnnotType ['=' Expr] | StructuralInstance)
GivenSig          ::=  [id] [DefTypeParamClause] {UsingParamClause} ':'     -- one
StructuralInstance ::=  ConstrApp {'with' ConstrApp} ['with' TemplateBody]
Extension         ::=  'extension' [DefTypeParamClause] {UsingParamClause}
                       '(' DefParam ')' {UsingParamClause} ExtMethods
ExtMethods        ::=  ExtMethod | [nl] <<< ExtMethod {semi ExtMethod} >>>
ExtMethod         ::=  {Annotation [nl]} {Modifier} 'def' DefDef
Template          ::=  InheritClauses [TemplateBody]
InheritClauses    ::=  ['extends' ConstrApps] ['derives' QualId {',' QualId}]
ConstrApps        ::=  ConstrApp ({',' ConstrApp} | {'with' ConstrApp})
ConstrApp         ::=  SimpleType1 {Annotation} {ParArgumentExprs}
ConstrExpr        ::=  SelfInvocation
                    |  <<< SelfInvocation {semi BlockStat} >>>
SelfInvocation    ::=  'this' ArgumentExprs {ArgumentExprs}

TemplateBody      ::=  :<<< [SelfType] TemplateStat {semi TemplateStat} >>>
TemplateStat      ::=  Import
                    |  Export
                    |  {Annotation [nl]} {Modifier} Def
                    |  {Annotation [nl]} {Modifier} Dcl
                    |  Extension
                    |  Expr1
                    |  EndMarker
                    |
SelfType          ::=  id [':' InfixType] '=>'                      ValDef
                    |  'this' ':' InfixType '=>'
```

```
EnumBody          ::=  :<<< [SelfType] EnumStat {semi EnumStat} >>>
EnumStat          ::=  TemplateStat
                    |  {Annotation [nl]} {Modifier} EnumCase
EnumCase          ::=  ‘case’ (id ClassConstr [‘extends’ ConstrApps]] | ids)

TopStats          ::=  TopStat {semi TopStat}
TopStat           ::=  Import
                    |  Export
                    |  {Annotation [nl]} {Modifier} Def
                    |  Extension
                    |  Packaging
                    |  PackageObject
                    |  EndMarker
                    |
Packaging         ::=  ‘package’ QualId :<<< TopStats >>>
PackageObject     ::=  ‘package’ ‘object’ ObjectDef

CompilationUnit   ::=  {‘package’ QualId semi} TopStats
```

7 Type System

The types are defined in dotty/tools/dotc/core/Types.scala

7.1 Class diagram

- PDF, generated with a fork of scaladiagrams

7.2 Proxy types and ground types

A type which inherits `TypeProxy` is a proxy for another type accessible using the `underlying` method, other types are called ground types and inherit `CachedGroundType` or `UncachedGroundType`.

Here's a diagram, copied from dotty/tools/dotc/core/Types.scala:

```
Type -+- ProxyType --+- NamedType ----+--- TypeRef
      |              |                 \
      |              +- SingletonType-+-+- TermRef
      |              |                |
      |              |                +--- ThisType
      |              |                +--- SuperType
      |              |                +--- ConstantType
      |              |                +--- TermParamRef
      |              |                +----RecThis
      |              |                +--- SkolemType
      |              +- TypeParamRef
      |              +- RefinedOrRecType -+-- RefinedType
      |              |                    -+-- RecType
      |              +- AppliedType
      |              +- TypeBounds
      |              +- ExprType
      |              +- AnnotatedType
      |              +- TypeVar
      |              +- HKTypeLambda
      |              +- MatchType
      |
      +- GroundType -+- AndType
                     +- OrType
                     +- MethodOrPoly ---+-- PolyType
                     |                  +-- MethodType
                     +- ClassInfo
                     |
                     +- NoType
                     +- NoPrefix
                     +- ErrorType
                     +- WildcardType
```

7.3 Representations of types

Type	Representation
`p.x.type`	`TermRef(p, x)`
`p#T`	`TypeRef(p, T)`
`p.x.T` == `p.x.type#T`	`TypeRef(TermRef(p, x), T)`
`this.type`	`ThisType`
`A & B`	`AndType(A, B)`
A | B	`OrType(A, B)`
`=> T`	`ExprType(T)`
`p { refinedName }`	`RefinedType(p, refinedName)`
type of the value `super`	`SuperType`
`type T >: A <: B`	`TypeRef` with underlying type `RealTypeBounds(A, B)`
`type T = A`	`TypeRef` with underlying type `TypeAlias(A)`
`class p.C ...`	`ClassInfo(p, C, ...)`

7.3.1 Representation of methods

```
def f[A, B <: Ord[A]](x: A, y: B): Unit
```

is represented as:

```
val p = PolyType(List("A", "B"))(
  List(TypeBounds(Nothing, Any),
       TypeBounds(Nothing,
         RefinedType(Ordering,
           scala$math$Ordering$$T, TypeAlias(PolyParam(p, 0))))),
  m)

val m = MethodType(List("x", "y"),
  List(PolyParam(p, 0), PolyParam(p, 1)))(Unit)
```

(This is a slightly simplified version, e.g. we write `Unit` instead of `TypeRef(TermRef(ThisType(TypeRef(NoPr`

Note that a PolyParam refers to a type parameter using its index (here A is 0 and B is 1).

7.4 Subtyping checks

`topLevelSubType(tp1, tp2)` in dotty/tools/dotc/core/TypeComparer.scala checks if `tp1` is a subtype of `tp2`.

7.4.1 Type rebasing

FIXME: This section is no longer accurate because https://github.com/lampepfl/dotty/pull/331 changed the handling of refined types.

Consider tests/pos/refinedSubtyping.scala

```
class Test {

  class C { type T; type Coll }

  type T1 = C { type T = Int }

  type T11 = T1 { type Coll = Set[Int] }

  type T2 = C { type Coll = Set[T] }

  type T22 = T2 { type T = Int }

  var x: T11 = _
  var y: T22 = _

  x = y
  y = x

}
```

We want to do the subtyping checks recursively, since it would be nice if we could check if `T22 <: T11` by first checking if `T2 <: T1`. To achieve this recursive subtyping check, we remember that `T2#T` is really `T22#T`. This procedure is called rebasing and is done by storing refined names in `pendingRefinedBases` and looking them up using `rebase`.

7.5 Type caching

TODO

7.6 Type inference via constraint solving

TODO

8 Dotty Internals 1

layout: doc-page title: Dotty Internals 1: Trees & Symbols (Meeting Notes)

These are meeting notes for the Dotty Internals 1: Trees & Symbols talk by Dmitry Petrashko on Mar 21, 2017.

9 Entry point

`dotc/Compiler.scala`

The entry point to the compiler contains the list of phases and their order.

10 Phases

Some phases executed independently, but others (miniphases) are grouped for efficiency. See the paper "Miniphases: Compilation using Modular and Efficient Tree Transformation" for details.

11 Trees

`dotc/ast/Trees.scala`

Trees represent code written by the user (e.g. methods, classes, expressions). There are two kinds of trees: untyped and typed.

Unlike other compilers (but like `scalac`), dotty doesn't use multiple intermediate representations (IRs) during the compilation pipeline. Instead, it uses trees for all phases.

Dotty trees are immutable, so they can be shared.

11.1 Untyped trees

`dotc/ast/untpd.scala`

These are the trees as output by the parser.

Some trees only exist as untyped: e.g. `WhileDo` and `ForDo`. These are desugared by the typechecker.

11.2 Typed trees

`dotc/ast/tpd.scala`

Typed trees contain not only the user-written code, but also semantic information (the types) about the code.

11.3 Notes on some tree types

- `RefTree`: trees that refer to something. There are multiple subtypes
  - `Ident`: by-name reference
  - `Select`: select (e.g. a field) from another tree (e.g. `a.foo` is represented as `Select(Ident(a), foo))`
- `This`: the this pointer
- `Apply`: function application: e.g. `a.foo(1, 2)(3, 4)` becomes `Apply(Apply(Select(Ident(a), foo), List(1, 2)), List(3, 4))`
- `TypeApply`: type application: `def foo[T](a: T) = ??? foo[Int](1)` becomes `Apply(TypeApply(Ident(foo), List(Int)), List(1))`
- `Literal`: constants (e.g. integer constant 1)

- `Typed`: type ascription (e.g. for widening, as in `(1: Any)`)
- `NamedArg`: named arguments (can appear out-of-order in untyped trees, but will appear in-order in typed ones)
- `Assign`: assignment. The node has a `lhs` and a `rhs`, but the `lhs` can be arbitrarily complicated (e.g. `(new C).f = 0`).
- `If`: the condition in an if-expression can be arbitrarily complex (e.g. it can contain class definitions)
- `Closure`: the free variables are stored in the `env` field, but are only accessible "around" the `LambdaLift` phase.
- `Match` and `CaseDef`: pattern-matching trees. The `pat` field in `CaseDef` (the pattern) is, in turn, populated with a subset of trees like `Bind` and `Unapply`.
- `Return`: return from a method. If the `from` field is empty, then we return from the closest enclosing method. The `expr` field should have a types that matches the return type of the method, but the `Return` node itself has type bottom.
- `TypeTree`: tree representing a type (e.g. for `TypeApply`).
- `AndType`, `OrType`, etc.: these are other trees that represent types that can be written by the user. These are a strict subset of all types, since some types cannot be written by the user.
- `ValDef`: defines fields or local variables. To differentiate between the two cases, we can look at the denotation. The `preRhs` field is lazy because sometimes we want to "load" a definition without know what's on the rhs (for example, to look up its type).
- `DefDef`: method definition.
- `TypeDef`: type definition. Both `type A = ???` and `class A {}` are represented with a `TypeDef`. To differentiate between the two, look at the type of the node (better), or in the case of classes there should be a `Template` node in the rhs.
- `Template`: describes the "body" of a class, including inheritance information and constructor. The `constr` field will be populated only after the `Constructors` phase; before that the constructor lives in the `preBody` field.
- `Thicket`: allows us to return multiple trees when a single one is expected. This kind of tree is not user-visible. For example, `transformDefDef` in `LabelDefs` takes in a `DefDef` and needs to be able to sometimes break up the method into multiple methods, which are then returned as a single tree (via a `Thicket`). If we return a thicket in a location where multiple trees are expected, the compiler will flatten them, but if only one tree is expected (for example, in the constructor field of a class), then the compiler will throw.

11.3.1 ThisTree

Tree classes have a `ThisTree` type field which is used to implement functionality that's common for all trees while returning a specific tree type. See `withType` in the `Tree` base class, for an example.

Additionally, both `Tree` and `ThisTree` are polymorphic so they can represent both untyped and typed trees.

For example, `withType` has signature `def withType(tpe: Type)(implicit ctx: Context): ThisTree[Type]`. This means that `withType` can return the most-specific tree type for the current tree, while at the same time guaranteeing that the returned tree will be typed.

11.4 Creating trees

You should use the creation methods in `untpd.scala` and `tpd.scala` to instantiate tree objects (as opposed to creating them directly using the case classes in `Trees.scala`).

11.5 Meaning of trees

In general, the best way to know what a tree represents is to look at its type or denotation; pattern matching on the structure of a tree is error-prone.

11.6 Errors

`dotc/typer/ErrorReporting.scala`

Sometimes there's an error during compilation, but we want to continue compilling (as opposed to failing outright), to uncover additional errors.

In cases where a tree is expected but there's an error, we can use the `errorTree` methods in `ErrorReporting` to create placeholder trees that explicitly mark the presence of errors.

Similarly, there exist `ErrorType` and `ErrorSymbol` classes.

11.7 Assignment

The closest in Dotty to what a programming language like C calls an "l-value" is a `RefTree` (so an `Ident` or a `Select`). However, keep in mind that arbitrarily complex expressions can appear in the lhs of an assignment: e.g.

```
trait T {
  var s = 0
}
{
  class T2 extends T
  while (true) 1
  new Bla
}.s = 10
```

Another caveat, before typechecking there can be some trees where the lhs isn't a `RefTree`: e.g. `(a, b) = (3, 4)`.

12 Symbols

`dotc/core/Symbols.scala`

Symbols are references to definitions (e.g. of variables, fields, classes). Symbols can be used to refer to definitions for which we don't have ASTs (for example, from the Java standard library).

`NoSymbol` is used to indicate the lack of a symbol.

Symbols uniquely identify definitions, but they don't say what the definitions mean. To understand the meaning of a symbol we need to look at its denotation (spefically for symbols, a `SymDenotation`).

Symbols can not only represent terms, but also types (hence the `isTerm`/`isType` methods in the `Symbol` class).

12.1 ClassSymbol

`ClassSymbol` represents either a `class`, or an `trait`, or an `object`. For example, an object

```
object O {
  val s = 1
}
```

is represented (after `Typer`) as

```
class O$ { this: O.type =>
  val s = 1
}
val O = new O$
```

where we have a type symbol for `class O$` and a term symbol for `val O`. Notice the use of the selftype `O.type` to indicate that `this` has a singleton type.

12.2 SymDenotation

`dotc/core/SymDenotations.scala`

Symbols contain `SymDenotations`. The denotation, in turn, refers to:

- the source symbol (so the linkage is cyclic)
- the "owner" of the symbol:
  - if the symbol is a variable, the owner is the enclosing method
  - if it's a field, the owner is the enclosing class
  - if it's a class, then the owner is the enclosing class
- a set of flags that contain semantic information about the definition (e.g. whether it's a trait or mutable). Flags are defined in `Flags.scala`.
- the type of the definition (through the `info` method)

13 Debug Macros

Complex macros may break invariants of the compiler, which leads to compiler crashes. Here we list common compiler crashes and how to deal with them.

13.1 Enable checks

- Always enable `-Xcheck-macros`
- May also enable `-Ycheck:all`

13.2 position not set

For this problem, here is the log that is usually shown:

```
[error] assertion failed: position not set for org.scalactic.anyvals.PosZInt.+$extensi
[error]   org.scalactic.anyvals.PosZInt.widenToInt(SizeParam.this.sizeRange)
[error] ) # 2326942 of class dotty.tools.dotc.ast.Trees$Apply in library/src-
bootstrapped/scala/tasty/reflect/utils/TreeUtils.scala
```

To debug why the position is not set, note the tree id `2326942`, and enable the following compiler option:

```
-Ydebug-tree-with-id 2326942
```

With the option above, the compiler will crash when the tree is created. From the stack trace, we will be able to figure out where the tree is created.

If the position is in the compiler, then either report a compiler bug or fix the problem with `.withSpan(tree.span)`. The following fix is an example:

- https://github.com/lampepfl/dotty/pull/6581

13.3 unresolved symbols in pickling

Here is the usually stacktrace for unresolved symbols in pickling:

```
[error] java.lang.AssertionError: assertion failed: unresolved symbols: value pos (lin
test.dotty/target/scala-0.17/src_managed/test/org/scalatest/AssertionsSpec.scala
[error]  at dotty.tools.dotc.core.tasty.TreePickler.pickle(TreePickler.scala:699)
[error]  at dotty.tools.dotc.transform.Pickler.run$$anonfun$10$$anonfun$8(Pickler.scal
[error]  at dotty.runtime.function.JProcedure1.apply(JProcedure1.java:15)
[error]  at dotty.runtime.function.JProcedure1.apply(JProcedure1.java:10)
[error]  at scala.collection.immutable.List.foreach(List.scala:392)
[error]  at dotty.tools.dotc.transform.Pickler.run$$anonfun$2(Pickler.scala:83)
[error]  at dotty.runtime.function.JProcedure1.apply(JProcedure1.java:15)
[error]  at dotty.runtime.function.JProcedure1.apply(JProcedure1.java:10)
[error]  at scala.collection.immutable.List.foreach(List.scala:392)
[error]  at dotty.tools.dotc.transform.Pickler.run(Pickler.scala:83)
[error]  at dotty.tools.dotc.core.Phases$Phase.runOn$$anonfun$1(Phases.scala:316)
[error]  at scala.collection.immutable.List.map(List.scala:286)
```

```
[error]  at dotty.tools.dotc.core.Phases$Phase.runOn(Phases.scala:318)
[error]  at dotty.tools.dotc.transform.Pickler.runOn(Pickler.scala:87)
```

From the stack trace, we know `pos` at line 5565 cannot be resolved. For the compiler, it means that the name `pos` (usually a local name, but could also be a class member) is used in the code but its definition cannot be found.

A possible cause of the problem is that the macro implementation accidentally dropped the definition of the referenced name.

If you are confident that the macro implementation is correct, then it might be a bug of the compiler. Try to minimize the code and report a compiler bug.

14 Differences between Scalac and Dotty

Overview explanation how symbols, named types and denotations hang together: Denotations1

14.0.1 Denotation

Comment with a few details: Denotations2

A `Denotation` is the result of a name lookup during a given period

- Most properties of symbols are now in the denotation (name, type, owner, etc.)
- Denotations usually have a reference to the selected symbol
- Denotations may be overloaded (`MultiDenotation`). In this case the symbol may be `NoSymbol` (the two variants have symbols).
- Non-overloaded denotations have an `info`

Denotations of methods have a signature (Signature1), which uniquely identifies overloaded methods.

14.0.1.1 Denotation vs. SymDenotation

A `SymDenotation` is an extended denotation that has symbol-specific properties (that may change over phases) * `flags` * `annotations` * `info`

`SymDenotation` implements lazy types (similar to scalac). The type completer assigns the denotation's `info`.

14.0.1.2 Implicit Conversion

There is an implicit conversion:

```
core.Symbols.toDenot(sym: Symbol)(implicit ctx: Context): SymDenotation
```

Because the class `Symbol` is defined in the object `core.Symbols`, the implicit conversion does not need to be imported, it is part of the implicit scope of the type `Symbol` (check the Scala spec). However, it can only be applied if an implicit `Context` is in scope.

14.0.2 Symbol

- `Symbol` instances have a `SymDenotation`
- Most symbol properties in the Scala 2 compiler are now in the denotation (in the Scala 3 compiler).

Most of the `isFooBar` properties in scalac don't exist anymore in dotc. Use flag tests instead, for example:

```
if (sym.isPackageClass)         // Scala 2
if (sym is Flags.PackageClass)  // Scala 3 (*)
```

`(*)` Symbols are implicitly converted to their denotation, see above. Each `SymDenotation` has flags that can be queried using the `is` method.

14.0.3 Flags

- Flags are instances of the value class `FlagSet`, which encapsulates a `Long`
- Each flag is either valid for types, terms, or both

```
000..0001000..01
        ^     ^^
        flag  | \
              |  valid for term
              valid for type
```

- Example: `Module` is valid for both module values and module classes, `ModuleVal` / `ModuleClass` for either of the two.
- `flags.is(Method | Param)`: true if `flags` has either of the two

14.0.4 Tree

- Trees don't have symbols
  - `tree.symbol` is `tree.denot.symbol`
  - `tree.denot` is `tree.tpe.denot` where the `tpe` is a `NamdedType` (see next point)
- Subclasses of `DenotingTree` (`Template`, `ValDef`, `DefDef`, `Select`, `Ident`, etc.) have a `NamedType`, which has a `denot` field. The denotation has a symbol.
  - The `denot` of a `NamedType` (prefix + name) for the current period is obtained from the symbol that the type refers to. This symbol is searched using `prefix.member(name)`.

14.0.5 Type

- `MethodType(paramSyms, resultType)` from scalac => `mt @ MethodType(paramNames, paramTypes)`. Result type is `mt.resultType`

```
@todo
```

15 Contexts

The `Context` contains the state of the compiler, for example

- `settings`
- `freshNames` (`FreshNameCreator`)
- `period` (run and phase id)
- `compilationUnit`
- `phase`
- `tree` (current tree)
- `typer` (current typer)
- `mode` (type checking mode)
- `typerState` (for example undetermined type variables)
- …

15.0.1 Contexts in the typer

The type checker passes contexts through all methods and adapts fields where necessary, e.g.

```
case tree: untpd.Block => typedBlock(desugar.block(tree), pt)(ctx.fresh.withNewScope)
```

A number of fields in the context are typer-specific (`mode`, `typerState`).

15.0.2 In other phases

Other phases need a context for many things, for example to access the denotation of a symbols (depends on the period). However they typically don't need to modify / extend the context while traversing the AST. For these phases the context can be simply an implicit class parameter that is then available in all members.

Careful: beware of memory leaks. Don't hold on to contexts in long lived objects.

15.0.3 Using contexts

Nested contexts should be named `ctx` to enable implicit shadowing:

```
scala> class A

scala> def foo(implicit a: A) { def bar(implicit b: A) { println(implicitly[A]) } }
<console>:8: error: ambiguous implicit values:
 both value a of type A
 and value b of type A
 match expected type A
       def foo(implicit a: A) { def bar(implicit b: A) { println(implicitly[A]) } }

scala> def foo(implicit a: A) { def bar(implicit a: A) { println(implicitly[A]) } }
foo: (implicit a: A)Unit
```

16 Explicit Nulls

The explicit nulls feature (enabled via a flag) changes the Scala type hierarchy so that reference types (e.g. `String`) are non-nullable. We can still express nullability with union types: e.g. `val x: String | Null = null`.

The implementation of the feature in Scala 3 can be conceptually divided in several parts:

1. changes to the type hierarchy so that `Null` is only a subtype of `Any`
2. a "translation layer" for Java interoperability that exposes the nullability in Java APIs
3. a `unsafeNulls` language feature which enables implicit unsafe conversion between `T` and `T | Null`

16.1 Explicit-Nulls Flag

The explicit-nulls flag is currently disabled by default. It can be enabled via `-Yexplicit-nulls` defined in `ScalaSettings.scala`. All of the explicit-nulls-related changes should be gated behind the flag.

16.2 Type Hierarchy

We change the type hierarchy so that `Null` is only a subtype of `Any` by:

- modifying the notion of what is a nullable class (`isNullableClass`) in `SymDenotations` to include only `Null` and `Any`, which is used by `TypeComparer`
- changing the parent of `Null` in `Definitions` to point to `Any` and not `AnyRef`
- changing `isBottomType` and `isBottomClass` in `Definitions`

16.3 Working with Nullable Unions

There are some utility functions for nullable types in `NullOpsDecorator.scala`. They are extension methods for `Type`; hence we can use them in this way: `tp.f(...)`.

- `stripNull` syntactically strips all `Null` types in the union: e.g. `T | Null => T`. This should only be used if we can guarantee `T` is a reference type.
- `isNullableUnion` determines whether `this` is a nullable union.
- `isNullableAfterErasure` determines whether `this` type can have `null` value after erasure.

Within `Types.scala`, we also defined an extractor `OrNull` to extract the non-nullable part of a nullable unions .

```scala
(tp: Type) match
  case OrNull(tp1) => // if tp is a nullable union: tp1 | Null
  case _ => // otherwise
```

16.4 Java Interoperability

The problem we're trying to solve here is: if we see a Java method `String foo(String)`, what should that method look like to Scala?

- since we should be able to pass `null` into Java methods, the argument type should be `String | Null`
- since Java methods might return `null`, the return type should be `String | Null`

At a high-level:

- we track the loading of Java fields and methods as they're loaded by the compiler
- we do this in two places: `Namer` (for Java sources) and `ClassFileParser` (for bytecode)
- whenever we load a Java member, we "nullify" its argument and return types

The nullification logic lives in `compiler/src/dotty/tools/dotc/core/JavaNullInterop.scala`.

The entry point is the function `def nullifyMember(sym: Symbol, tp: Type, isEnumValueDef: Boolean)(implicit ctx: Context): Type` which, given a symbol, its "regular" type, and a boolean whether it is a Enum value definition, produces what the type of the symbol should be in the explicit nulls world.

1. If the symbol is a Enum value definition or a `TYPE_` field, we don't nullify the type
2. If it is `toString()` method or the constructor, or it has a `@NotNull` annotation, we nullify the type, without a `Null` at the outmost level.
3. Otherwise, we nullify the type in regular way.

The `@NotNull` annotations are defined in `Definitions.scala`.

See `JavaNullMap` in `JavaNullInterop.scala` for more details about how we nullify different types.

16.5 Relaxed Overriding Check

If the explicit nulls flag is enabled, the overriding check between Scala classes and Java classes is relaxed.

The `matches` function in `Types.scala` is used to select condidated for overriding check.

The `compatibleTypes` in `RefCheck.scala` determines whether the overriding types are compatible.

16.6 Nullified Upper Bound

Suppose we have a type bound `class C[T >: Null <: String]`, it becomes unapplicable in explicit nulls, since we don't have a type that is a supertype of `Null` and a subtype of `String`.

Hence, when we read a type bound from Scala 2 Tasty or Scala 3 Tasty, the upper bound is nullified if the lower bound is exactly `Null`. The example above would become `class C[T >: Null <: String | Null]`.

16.7 Unsafe Nulls Feature and SafeNulls Mode

The `unsafeNulls` language feature is currently disabled by default. It can be enabled by importing `scala.language.unsafeNulls` or using `-language:unsafeNulls`. The feature object is defined in `library/src/scalaShadowing/language.scala`. We can use `config.Feature.enabled(nme.unsafeNulls)` to check if this feature is enabled.

We use the `SafeNulls` mode to track `unsafeNulls`. If explicit nulls is enabled without `unsafeNulls`, there is a `SafeNulls` mode in the context; when `unsafeNulls` is enabled, `SafeNulls` mode will be removed from the context.

Since we want to allow selecting member on nullable values, when searching a member of a type, the `| Null` part should be ignored. See `goOr` in `Types.scala`.

16.8 Flow Typing

As typing happens, we accumulate a set of `NotNullInfos` in the `Context` (see `Contexts.scala`). A `NotNullInfo` contains the set of `TermRefs` that are known to be non-null at the current program point. See `Nullables.scala` for how `NotNullInfos` are computed.

During type-checking, when we type an identity or a select tree (in `typedIdent` and `typedSelect`), we will call `toNotNullTermRef` on the tree before return the typed tree. If the tree `x` has nullable type `T|Null` and it is known to be not null according to the `NotNullInfo` and it is not on the lhs of assignment, then we cast it to `x.type & T` using `defn.Any_typeCast`.

The reason for casting to `x.type & T`, as opposed to just `T`, is that it allows us to support flow typing for paths of length greater than one.

```
abstract class Node:
  val x: String
  val next: Node | Null

def f =
  val l: Node | Null = ???
  if l != null && l.next != null then
    val third: l.next.next.type = l.next.next
```

After typing, `f` becomes:

```
def f =
  val l: Node | Null = ???
  if l != null && l.$asInstanceOf$[l.type & Node].next != null then
    val third:
      l.$asInstanceOf$[l.type & Node].next.$asInstanceOf$[(l.type & Node).next.type
      l.$asInstanceOf$[l.type & Node].next.$asInstanceOf$[(l.type & Node).next.type
```

Notice that in the example above `(l.type & Node).next.type & Node` is still a stable path, so we can use it in the type and track it for flow typing.