Scala Native Documentation

Release 0.3.8

Denys Shabalin

Dec 17, 2018

Contents

1 Community **3**

2 Documentation **5**

2.1 User's Guide 5
2.2 Libraries 16
2.3 Contributor's Guide 37
2.4 Blog 58
2.5 Changelog 61
2.6 FAQ 62

Scala Native is an optimizing ahead-of-time compiler and lightweight managed runtime designed specifically for Scala. It features:

- **Low-level primitives**.

  ```
  type Vec = CStruct3[Double, Double, Double]

  val vec = stackalloc[Vec] // allocate c struct on stack
  !vec._1 = 10.0            // initialize fields
  !vec._2 = 20.0
  !vec._3 = 30.0
  length(vec)               // pass by reference
  ```

  Pointers, structs, you name it. Low-level primitives let you hand-tune your application to make it work exactly as you want it to. You're in control.

- **Seamless interop with native code**.

  ```
  @extern object stdlib {
    def malloc(size: CSize): Ptr[Byte] = extern
  }

  val ptr = stdlib.malloc(32)
  ```

  Calling C code has never been easier. With the help of extern objects you can seamlessly call native code without any runtime overhead.

- **Instant startup time**.

  ```
  > time hello-native
  hello, native!

  real    0m0.005s
  user    0m0.002s
  sys     0m0.002s
  ```

  Scala Native is compiled ahead-of-time via LLVM. This means that there is no sluggish warm-up phase that's common for just-in-time compilers. Your code is immediately fast and ready for action.

CHAPTER 1

Community

- Want to follow project updates? Follow us on twitter.
- Want to chat? Join our Gitter chat channel.
- Have a question? Ask it on Stack Overflow with tag scala-native.
- Found a bug or want to propose a new feature? Open an issue on Github.

CHAPTER 2

Documentation

This documentation is divided into different parts. It's recommended to go through the *User's Guide* to get familiar with Scala Native. *Libraries* will walk you through all the known libraries that are currently available. *Contributor's Guide* contains valuable information for people who want to either contribute to the project or learn more about the internals and the development process behind the project.

2.1 User's Guide

2.1.1 Environment setup

Scala Native has the following build dependencies:

- Java 8 or newer
- sbt 0.13.13 or newer
- LLVM/Clang 3.7 or newer

And following runtime library dependencies:

- libunwind 0.99 or newer (built-in on macOS)
- Boehm GC 7.6.0 (optional)
- Re2 2017-01-01 (optional)
- zlib 1.2.8 or newer (optional)

Most of the runtime dependencies are completely optional and are only required if you use the corresponding feature.

Installing sbt

Please refer to this link for instructions for your operating system.

Installing clang and runtime dependencies

Scala Native requires Clang, which is part of the LLVM toolchain. The recommended LLVM version is 3.7 or newer, however, the Scala Native sbt plugin uses feature detection to discover the installed version of Clang so older versions may also work.

Scala Native uses Boehm garbage collector by default. Both the native library and header files must be provided at build time. One may use opt-in to use new experimental garbage collector called Immix to avoid this dependency.

To be able to use regular expressions, the RE2 library must be installed. You will also need to install zlib if you use classes from the *java.util.zip* package. If you don't use regular expressions or compression, you can skip these dependencies.

Note: Some package managers provide the library header files in separate *-dev* packages.

Here are install instructions for a number of operating systems Scala Native has been used with:

macOS

```
$ brew install llvm
$ brew install bdw-gc re2 # optional
```

Note: A version of zlib that is sufficiently recent comes with the installation of macOS.

Ubuntu

```
$ sudo apt install clang libunwind-dev
$ sudo apt install libgc-dev libre2-dev # optional
```

Note: libre2-dev is available since Ubuntu 16.04. Please refer to our travis environment setup script to install from source.

Arch Linux

```
$ sudo pacman -S llvm clang
$ sudo pacman -S gc re2 # optional
```

Note: A version of zlib that is sufficiently recent comes with the installation of Arch Linux.

Fedora 26

```
$ sudo dnf install llvm clang
$ sudo dnf install libunwind-devel gc-devel zlib-devel re2-devel # optional
```

FreeBSD

```
$ pkg install llvm38 libunwind
$ pkg install boehm-gc re2 # optional
```

Note: A version of zlib that is sufficiently recent comes with the installation of FreeBSD.

Nix/NixOS

```
$ wget https://raw.githubusercontent.com/scala-native/scala-native/master/scripts/
↪scala-native.nix
$ nix-shell scala-native.nix -A clangEnv
```

Continue to *Building projects with sbt*.

2.1.2 Building projects with sbt

If you have reached this section you probably have a system that is now able to compile and run Scala Native programs.

Minimal sbt project

The easiest way to make a fresh project is to use our official gitter8 template:

```
sbt new scala-native/scala-native.g8
```

This generates the following files:

- `project/plugins.sbt` to add a plugin dependency:

  ```
  addSbtPlugin("org.scala-native" % "sbt-scala-native" % "0.3.8")
  ```

- `project/build.properties` to specify the sbt version:

  ```
  sbt.version = 1.2.6
  ```

- `build.sbt` to enable the plugin and specify Scala version:

  ```
  enablePlugins(ScalaNativePlugin)

  scalaVersion := "2.11.12"
  ```

- `src/main/scala/Main.scala` with minimal application:

  ```
  object Main {
    def main(args: Array[String]): Unit =
      println("Hello, world!")
  }
  ```

Now, simply run `sbt run` to get everything compiled and have the expected output! Please refer to the *FAQ* if you encounter any problems.

Scala versions

Scala Native supports following Scala versions for corresponding releases:

Scala Native Version	Scala Versions
0.1.x	2.11.8
0.2.x	2.11.8, 2.11.11
0.3.0-0.3.3	2.11.8, 2.11.11
0.3.4+	2.11.8, 2.11.11, 2.11.12

Sbt settings and tasks

Since	Name	Type	Description
0.1	`compile`	`Analysis`	Compile Scala code to NIR
0.1	`run`	`Unit`	Compile, link and run the generated binary
0.1	`package`	`File`	Similar to standard package with addition of NIR
0.1	`publish`	`Unit`	Similar to standard publish with addition of NIR (1)
0.1	`nativeLink`	`File`	Link NIR and generate native binary
0.1	`nativeClang`	`File`	Path to `clang` command
0.1	`nativeClangPP`	`File`	Path to `clang++` command
0.1	`nativeCompileOptions`	`Seq[String]`	Extra options passed to clang verbatim during compilation
0.1	`nativeLinkingOptions`	`Seq[String]`	Extra options passed to clang verbatim during linking
0.1	`nativeMode`	`String`	Either `"debug"` or `"release"` (2)
0.2	`nativeGC`	`String`	Either `"none"`, `"boehm"` or `"immix"` (3)
0.3.3	`nativeLinkStubs`	`Boolean`	Whether to link `@stub` definitions, or to ignore them
0.3.9	`nativeLTO`	`String`	Either `"none"`, `"full"` or `"thin"` (4)

1. See *Publishing* and *Cross compilation* for details.
2. See *Compilation modes* for details.
3. See *Garbage collectors* for details.
4. See *Link-Time Optimization (LTO)* for details.

Compilation modes

Scala Native supports two distinct linking modes:

1. **debug.** (default)

   Default mode. Optimized for shortest compilation time. Runs fewer optimizations and is much more suited for iterative development workflow. Similar to clang's `-O0`.

2. **release.**

   Optimized for best runtime performance at expense of longer compilation time. Similar to clang's `-O2` with addition of link-time optimization over the whole application code.

Garbage collectors

1. **immix.** (default since 0.3.8, introduced in 0.3)

   Immix is a mostly-precise mark-region tracing garbage collector. More information about the collector is available as part of the original 0.3.0 announcement.

2. **boehm.** (default through 0.3.7)

   Conservative generational garbage collector. More information is available at the project's page.

3. **none.** (experimental, introduced in 0.2)

   Garbage collector that allocates things without ever freeing them. Useful for short-running command-line applications or applications where garbage collections pauses are not acceptable.

Link-Time Optimization (LTO)

Scala Native relies on link-time optimization to maximize runtime performance of release builds. There are three possible modes that are currently supported:

1. **none.** (default)

   Does not inline across Scala/C boundary. Scala to Scala calls are still optimized by emitting one fat LLVM IR module for the whole application.

2. **full.** (Clang 3.8 or older)

   Inlines across Scala/C boundary by merging all of the LLVM IR modules into a single module for the whole application. Unlike **none** this module also includes the runtime code thus allows for additional optimization opportunities.

3. **thin.** (recommended on Clang 3.9 or newer)

   Inlines across Scala/C boundary using LLVM's ThinLTO. Unlike **none** and **full** it's able to optimize the application in parallel. It also offers the best runtime performance according to our benchmarks.

Publishing

Scala Native supports sbt's standard workflow for the package distribution:

1. Compile your code.
2. Generate a jar with all of the class files and NIR files.
3. Publish the jar to sonatype, bintray or any other 3rd party hosting service.

Once the jar has been published, it can be resolved through sbt's standard package resolution system.

Cross compilation

sbt-crossproject is an sbt plugin that lets you cross-compile your projects against all three major platforms in Scala: JVM, JavaScript via Scala.js, and native via Scala Native. It is based on the original cross-project idea from Scala.js and supports the same syntax for existing JVM/JavaScript cross-projects. Please refer to the project's README for details.

Continue to *Language semantics*.

2.1.3 Language semantics

In general, the semantics of the Scala Native language are the same as Scala on the JVM. However, a few differences exist, which we mention here.

Interop extensions

Annotations and types defined `scala.scalanative.native` may modify semantics of the language for sake of interoperability with C libraries, read more about those in *Native code interoperability* section.

Multithreading

Scala Native doesn't yet provide libraries for parallel multi-threaded programming and assumes single-threaded execution by default.

It's possible to use C libraries to get access to multi-threading and synchronization primitives but this is not officially supported at the moment.

Finalization

Finalize method from `java.lang.Object` is never called in Scala Native.

Undefined behavior

Generally, Scala Native follows most of the special error conditions similarly to JVM:

1. Arrays throw `IndexOutOfBoundsException` on out-of-bounds access.
2. Casts throw `ClassCastException` on incorrect casts.
3. Accessing a field or method on `null`, throwing ``null` `` exception, throws `NullPointerException`.
4. Integer division by zero throws `ArithmeticException`.

There are a few exceptions:

1. Stack overflows are undefined behavior and would typically segfault on supported architectures instead of throwing `StackOverflowError`.
2. Exhausting a heap space results in crash with a stack trace instead of throwing `OutOfMemoryError`.

Continue to *Native code interoperability*.

2.1.4 Native code interoperability

Scala Native provides an interop layer that makes it easy to interact with foreign native code. This includes C and other languages that can expose APIs via C ABI (e.g. C++, D, Rust etc.)

All of the interop APIs discussed here are defined in `scala.scalanative.native` package. For brevity, we're going to refer to that namespace as just `native`.

Extern objects

Extern objects are simple wrapper objects that demarcate scopes where methods are treated as their native C ABI-friendly counterparts. They are roughly analogous to header files with top-level function declarations in C.

For example, to call C's `malloc` one might declare it as following:

```
@native.extern
object libc {
  def malloc(size: native.CSize): native.Ptr[Byte] = native.extern
}
```

`native.extern` on the right hand side of the method definition signifies that the body of the method is defined elsewhere in a native library that is available on the library path (see *Linking with native libraries*). The signature of the external function must match the signature of the original C function (see *Finding the right signature*).

Finding the right signature

To find a correct signature for a given C function one must provide an equivalent Scala type for each of the arguments:

C Type	Scala Type
`void`	`Unit`
`bool`	`native.CBool`
`char`	`native.CChar`
`signed char`	`native.CSignedChar`
`unsigned char`	`native.CUnsignedChar` [*1*]
`short`	`native.CShort`
`unsigned short`	`native.CUnsignedShort` [*1*]
`int`	`native.CInt`
`long int`	`native.CLongInt`
`unsigned int`	`native.CUnsignedInt` [*1*]
`unsigned long int`	`native.CUnsignedLongInt` [*1*]
`long`	`native.CLong`
`unsigned long`	`native.CUnsignedLong` [*1*]
`long long`	`native.CLongLong`
`unsigned long long`	`native.CUnsignedLongLong` [*1*]
`size_t`	`native.CSize`
`ptrdiff_t`	`native.CPtrDiff` [2]
`wchar_t`	`native.CWideChar`
`char16_t`	`native.CChar16`
`char32_t`	`native.CChar32`
`float`	`native.CFloat`
`double`	`native.CDouble`
`void*`	`native.Ptr[Byte]` [2]
`int*`	`native.Ptr[native.CInt]` [2]
`char*`	`native.CString` [2] [*3*]
`int (*)(int)`	`native.CFunctionPtr1[native.CInt, native.CInt]` [2] [*4*]
`struct { int x, y; }*`	`native.Ptr[native.CStruct2[native.CInt, native.CInt]]` [2] [*5*]
`struct { int x, y; }`	Not supported

Linking with native libraries

C compilers typically require to pass an additional `-l mylib` flag to dynamically link with a library. In Scala Native, one can annotate libraries to link with using the `@native.link` annotation.

```
@native.link("mylib")
@native.extern
object mylib {
  def f(): Unit = native.extern
}
```

Whenever any of the members of `mylib` object are reachable, the Scala Native linker will automatically link with the corresponding native library.

As in C, library names are specified without the `lib` prefix. For example, the library libuv corresponds to `@native.link("uv")` in Scala Native.

It is possible to rename functions using the `@name` annotation. Its use is recommended to enforce the Scala naming conventions in bindings:

```
import scala.scalanative.native._
@link("uv")
@extern
object uv {
  @name("uv_uptime")
  def uptime(result: Ptr[CDouble]): Int = extern
}
```

If a library has multiple components, you could split the bindings into separate objects as it is permitted to use the same `@link` annotation more than once.

Variadic functions

One can declare variadic functions like `printf` using `native.CVararg` auxiliary type:

```
@native.extern
object stdio {
  def printf(format: native.CString,
             args: native.CVararg*): native.CInt = native.extern
}
```

Pointer types

Scala Native provides a built-in equivalent of C's pointers via `native.Ptr[T]` data type. Under the hood pointers are implemented using unmanaged machine pointers.

Operations on pointers are closely related to their C counterparts and are compiled into equivalent machine code:

Operation	C syntax	Scala Syntax
Load value	`*ptr`	`!ptr`
Store value	`*ptr = value`	`!ptr = value`
Pointer to index	`ptr + i`, `&ptr[i]`	`ptr + i`
Elements between	`ptr1 - ptr2`	`ptr1 - ptr2`
Load at index	`ptr[i]`	`ptr(i)`
Store at index	`ptr[i] = value`	`ptr(i) = value`
Pointer to field	`&ptr->name`	`ptr._N`
Load a field	`ptr->name`	`!ptr._N`
Store a field	`ptr->name = value`	`!ptr._N = value`

Where `N` is the index of the field `name` in the struct. See *Memory layout types* for details.

Function pointers

It is possible to use external functions that take function pointers. For example given the following signature in C:

```
void test(void (* f)(char *));
```

One can declare it as following in Scala Native:

```
def test(f: CFunctionPtr1[CString, Unit]): Unit = native.extern
```

To pass a Scala function to `CFunctionPtrN`, you need to use the conversion function `CFunctionPtr.fromFunctionN()`:

```
def f(s: CString): Unit = ???
def g(): Unit = test(CFunctionPtr.fromFunction1(f))
```

Memory management

Unlike standard Scala objects that are managed automatically by the underlying runtime system, one has to be extra careful when working with unmanaged memory.

1. **Zone allocation.** (since 0.3)

   Zones (also known as memory regions/contexts) are a technique for semi-automatic memory management. Using them one can bind allocations to a temporary scope in the program and the zone allocator will automatically clean them up for you as soon as execution goes out of it:

   ```
   native.Zone { implicit z =>
     val buffer = native.alloc[Byte](n)
   }
   ```

   native.alloc requests memory sufficient to contain *n* values of a given type. If number of elements is not specified, it defaults to a single element. Memory is zeroed out by default.

   Zone allocation is the preferred way to allocate temporary unmanaged memory. It's idiomatic to use implicit zone parameters to abstract over code that has to zone allocate.

   One typical example of this are C strings that are created from Scala strings using `native.toCString`. The conversion takes implicit zone parameter and allocates the result in that zone.

   When using zone allocated memory one has to be careful not to capture this memory beyond the lifetime of the zone. Dereferencing zone-allocated memory after the end of the zone is undefined behavior.

2. **Stack allocation.**

   Scala Native provides a built-in way to perform stack allocations of using `native.stackalloc` function:

   ```
   val buffer = native.stackalloc[Byte](256)
   ```

   This code will allocate 256 bytes that are going to be available until the enclosing method returns. Number of elements to be allocated is optional and defaults to 1 otherwise. Memory is not zeroed out by default.

   When using stack allocated memory one has to be careful not to capture this memory beyond the lifetime of the method. Dereferencing stack allocated memory after the method's execution has completed is undefined behavior.

3. **Manual heap allocation.**

   Scala Native's library contains a bindings for a subset of the standard libc functionality. This includes the trio of `malloc`, `realloc` and `free` functions that are defined in `native.stdlib` extern object.

   Calling those will let you allocate memory using system's standard dynamic memory allocator. Every single manual allocation must also be freed manually as soon as it's not needed any longer.

   Apart from the standard system allocator one might also bind to plethora of 3-rd party allocators such as jemalloc to serve the same purpose.

Undefined behavior

Similarly to their C counter-parts, behavior of operations that access memory is subject to undefined behaviour for following conditions:

1. Dereferencing null.
2. Out-of-bounds memory access.
3. Use-after-free.
4. Use-after-return.
5. Double-free, invalid free.

Memory layout types

Memory layout types are auxiliary types that let one specify memory layout of unmanaged memory. They are meant to be used purely in combination with native pointers and do not have a corresponding first-class values backing them.

- `native.Ptr[native.CStructN[T1, ..., TN]]`

  Pointer to a C struct with up to 22 fields. Type parameters are the types of corresponding fields. One may access fields of the struct using `_N` helper methods on a pointer value:

  ```
  val ptr = native.stackalloc[native.CStruct2[Int, Int]]
  !ptr._1 = 10
  !ptr._2 = 20
  println(s"first ${!ptr._1}, second ${!ptr._2}")
  ```

  Here `_N` computes a derived pointer that corresponds to memory occupied by field number N.

- `native.Ptr[native.CArray[T, N]]`

  Pointer to a C array with statically-known length `N`. Length is encoded as a type-level natural number. Natural numbers are types that are composed of base naturals `Nat._0, ... Nat._9` and an additional `Nat.Digit` constructor. So for example number `1024` is going to be encoded as following:

  ```
  import scalanative.native._, Nat._

  type _1024 = Digit[_1, Digit[_0, Digit[_2, _4]]]
  ```

  Once you have a natural for the length, it can be used as an array length:

  ```
  val ptr = native.stackalloc[CArray[Byte, _1024]]
  ```

  Addresses of the first twenty two elements are accessible via `_N` accessors. The rest are accessible via `ptr._1 + index`.

Byte strings

Scala Native supports byte strings via `c"..."` string interpolator that gets compiled down to pointers to statically-allocated zero-terminated strings (similarly to C):

```
import scalanative.native._

// CString is an alias for Ptr[CChar]
```

(continues on next page)

(continued from previous page)

```
val msg: CString = c"Hello, world!"
stdio.printf(msg)
```

Additionally, we also expose two helper functions `native.toCString` and `native.fromCString` to convert between C-style and Java-style strings.

Unchecked casts

Quite often, C interfaces expect the user to perform unchecked casts to convert between different pointer types, or between pointers and integer values. For this particular use case, we provide `obj.cast[T]` that is defined in the implicit class `native.CCast`. Unlike Scala's `asInstanceOf`, `cast` does not provide any safety guarantees.

Platform-specific types

Scala Native defines the type `Word` and its unsigned counterpart, `UWord`. A word corresponds to `Int` on 32-bit architectures and to `Long` on 64-bit ones.

Size of types

In order to statically determine the size of a type, you can use the `sizeof` function which is Scala Native's counterpart of the eponymous C operator. It returns the size in bytes:

```
println(sizeof[Byte])    // 1
println(sizeof[CBool])   // 1
println(sizeof[CShort])  // 2
println(sizeof[CInt])    // 4
println(sizeof[CLong])   // 8
```

It can also be used to obtain the size of a structure:

```
type TwoBytes = CStruct2[Byte, Byte]
println(sizeof[TwoBytes])  // 2
```

Unsigned integer types

Scala Native provides support for four unsigned integer types:

1. `native.UByte`
2. `native.UShort`
3. `native.UInt`
4. `native.ULong`

They share the same primitive operations as signed integer types. Primitive operation between two integer values are supported only if they have the same signedness (they must both signed or both unsigned.)

Conversions between signed and unsigned integers must be done explicitly using `signed.toUByte`, `signed.toUShort`, `signed.toUInt`, `signed.toULong` and conversely `unsigned.toByte`, `unsigned.toShort`, `unsigned.toInt`, `unsigned.toLong`.

Continue to *Libraries*.

2.2 Libraries

2.2.1 Java Standard Library

Scala Native supports a subset of the JDK core libraries reimplemented in Scala.

Supported classes

Here is the list of currently available classes:

- `java.io.BufferedInputStream`
- `java.io.BufferedOutputStream`
- `java.io.BufferedReader`
- `java.io.BufferedWriter`
- `java.io.ByteArrayInputStream`
- `java.io.ByteArrayOutputStream`
- `java.io.Closeable`
- `java.io.DataInput`
- `java.io.DataInputStream`
- `java.io.DataOutput`
- `java.io.DataOutputStream`
- `java.io.EOFException`
- `java.io.File`
- `java.io.FileDescriptor`
- `java.io.FileFilter`
- `java.io.FileInputStream`
- `java.io.FileNotFoundException`
- `java.io.FileOutputStream`
- `java.io.FileReader`
- `java.io.FileWriter`
- `java.io.FilenameFilter`
- `java.io.FilterInputStream`
- `java.io.FilterOutputStream`
- `java.io.FilterReader`
- `java.io.Flushable`
- `java.io.IOException`
- `java.io.InputStream`
- `java.io.InputStreamReader`
- `java.io.InterruptedIOException`

- `java.io.LineNumberReader`
- `java.io.NotSerializableException`
- `java.io.ObjectStreamException`
- `java.io.OutputStream`
- `java.io.OutputStreamWriter`
- `java.io.PrintStream`
- `java.io.PrintWriter`
- `java.io.PushbackInputStream`
- `java.io.PushbackReader`
- `java.io.RandomAccessFile`
- `java.io.Reader`
- `java.io.Serializable`
- `java.io.StringReader`
- `java.io.StringWriter`
- `java.io.SyncFailedException`
- `java.io.UTFDataFormatException`
- `java.io.UnsupportedEncodingException`
- `java.io.Writer`
- `java.lang.AbstractMethodError`
- `java.lang.AbstractStringBuilder`
- `java.lang.Appendable`
- `java.lang.ArithmeticException`
- `java.lang.ArrayIndexOutOfBoundsException`
- `java.lang.ArrayStoreException`
- `java.lang.AssertionError`
- `java.lang.AutoCloseable`
- `java.lang.Boolean`
- `java.lang.BootstrapMethodError`
- `java.lang.Byte`
- `java.lang.ByteCache`
- `java.lang.CharSequence`
- `java.lang.Character`
- `java.lang.Character$CaseFolding`
- `java.lang.Character$Subset`
- `java.lang.Character$UnicodeBlock`
- `java.lang.CharacterCache`

- `java.lang.ClassCastException`
- `java.lang.ClassCircularityError`
- `java.lang.ClassFormatError`
- `java.lang.ClassLoader`
- `java.lang.ClassNotFoundException`
- `java.lang.CloneNotSupportedException`
- `java.lang.Cloneable`
- `java.lang.Comparable`
- `java.lang.Double`
- `java.lang.Enum`
- `java.lang.EnumConstantNotPresentException`
- `java.lang.Error`
- `java.lang.Exception`
- `java.lang.ExceptionInInitializerError`
- `java.lang.Float`
- `java.lang.IllegalAccessError`
- `java.lang.IllegalAccessException`
- `java.lang.IllegalArgumentException`
- `java.lang.IllegalMonitorStateException`
- `java.lang.IllegalStateException`
- `java.lang.IllegalThreadStateException`
- `java.lang.IncompatibleClassChangeError`
- `java.lang.IndexOutOfBoundsException`
- `java.lang.InheritableThreadLocal`
- `java.lang.InstantiationError`
- `java.lang.InstantiationException`
- `java.lang.Integer`
- `java.lang.IntegerCache`
- `java.lang.InternalError`
- `java.lang.InterruptedException`
- `java.lang.Iterable`
- `java.lang.LinkageError`
- `java.lang.Long`
- `java.lang.LongCache`
- `java.lang.Math`
- `java.lang.NegativeArraySizeException`

- `java.lang.NoClassDefFoundError`
- `java.lang.NoSuchFieldError`
- `java.lang.NoSuchFieldException`
- `java.lang.NoSuchMethodError`
- `java.lang.NoSuchMethodException`
- `java.lang.NullPointerException`
- `java.lang.Number`
- `java.lang.NumberFormatException`
- `java.lang.OutOfMemoryError`
- `java.lang.PipeIO`
- `java.lang.PipeIO$NullInput`
- `java.lang.PipeIO$NullOutput`
- `java.lang.PipeIO$Stream`
- `java.lang.PipeIO$Stream$class`
- `java.lang.PipeIO$StreamImpl`
- `java.lang.Process`
- `java.lang.ProcessBuilder`
- `java.lang.ProcessBuilder$Redirect`
- `java.lang.ProcessBuilder$Redirect$RedirectImpl`
- `java.lang.ProcessBuilder$Redirect$Type`
- `java.lang.Readable`
- `java.lang.ReflectiveOperationException`
- `java.lang.RejectedExecutionException`
- `java.lang.Runnable`
- `java.lang.Runtime`
- `java.lang.Runtime$ProcessBuilderOps`
- `java.lang.RuntimeException`
- `java.lang.SecurityException`
- `java.lang.Short`
- `java.lang.ShortCache`
- `java.lang.StackOverflowError`
- `java.lang.StackTrace`
- `java.lang.StackTraceElement`
- `java.lang.String`
- `java.lang.StringBuffer`
- `java.lang.StringBuilder`

- `java.lang.StringIndexOutOfBoundsException`
- `java.lang.System`
- `java.lang.Thread`
- `java.lang.Thread$UncaughtExceptionHandler`
- `java.lang.ThreadDeath`
- `java.lang.ThreadLocal`
- `java.lang.Throwable`
- `java.lang.TypeNotPresentException`
- `java.lang.UnixProcess`
- `java.lang.UnixProcess$ProcessMonitor`
- `java.lang.UnknownError`
- `java.lang.UnsatisfiedLinkError`
- `java.lang.UnsupportedClassVersionError`
- `java.lang.UnsupportedOperationException`
- `java.lang.VerifyError`
- `java.lang.VirtualMachineError`
- `java.lang.Void`
- `java.lang._String$CaseInsensitiveComparator`
- `java.lang.annotation.Retention`
- `java.lang.annotation.RetentionPolicy`
- `java.lang.ref.PhantomReference`
- `java.lang.ref.Reference`
- `java.lang.ref.ReferenceQueue`
- `java.lang.ref.SoftReference`
- `java.lang.ref.WeakReference`
- `java.lang.reflect.AccessibleObject`
- `java.lang.reflect.Array`
- `java.lang.reflect.Constructor`
- `java.lang.reflect.Executable`
- `java.lang.reflect.Field`
- `java.lang.reflect.InvocationTargetException`
- `java.lang.reflect.Method`
- `java.lang.reflect.UndeclaredThrowableException`
- `java.math.BigDecimal`
- `java.math.BigDecimal$QuotAndRem`
- `java.math.BigDecimal$StringOps`

- `java.math.BigInteger`
- `java.math.BigInteger$QuotAndRem`
- `java.math.BitLevel`
- `java.math.Conversion`
- `java.math.Division`
- `java.math.Elementary`
- `java.math.Logical`
- `java.math.MathContext`
- `java.math.Multiplication`
- `java.math.Primality`
- `java.math.RoundingMode`
- `java.net.BindException`
- `java.net.ConnectException`
- `java.net.Inet4Address`
- `java.net.Inet6Address`
- `java.net.InetAddress`
- `java.net.InetAddressBase`
- `java.net.InetAddressBase$class`
- `java.net.InetSocketAddress`
- `java.net.MalformedURLException`
- `java.net.NoRouteToHostException`
- `java.net.PlainSocketImpl`
- `java.net.PortUnreachableException`
- `java.net.ServerSocket`
- `java.net.Socket`
- `java.net.SocketAddress`
- `java.net.SocketException`
- `java.net.SocketImpl`
- `java.net.SocketInputStream`
- `java.net.SocketOption`
- `java.net.SocketOptions`
- `java.net.SocketOutputStream`
- `java.net.SocketTimeoutException`
- `java.net.URI`
- `java.net.URI$Helper`
- `java.net.URIEncoderDecoder`

- `java.net.URISyntaxException`
- `java.net.URL`
- `java.net.URLClassLoader`
- `java.net.URLConnection`
- `java.net.URLEncoder`
- `java.net.UnknownHostException`
- `java.net.UnknownServiceException`
- `java.nio.Buffer`
- `java.nio.BufferOverflowException`
- `java.nio.BufferUnderflowException`
- `java.nio.ByteArrayBits`
- `java.nio.ByteBuffer`
- `java.nio.ByteOrder`
- `java.nio.CharBuffer`
- `java.nio.DoubleBuffer`
- `java.nio.FloatBuffer`
- `java.nio.GenBuffer`
- `java.nio.GenHeapBuffer`
- `java.nio.GenHeapBuffer$NewHeapBuffer`
- `java.nio.GenHeapBufferView`
- `java.nio.GenHeapBufferView$NewHeapBufferView`
- `java.nio.HeapByteBuffer`
- `java.nio.HeapByteBuffer$NewHeapByteBuffer`
- `java.nio.HeapByteBufferCharView`
- `java.nio.HeapByteBufferCharView$NewHeapByteBufferCharView`
- `java.nio.HeapByteBufferDoubleView`
- `java.nio.HeapByteBufferDoubleView$NewHeapByteBufferDoubleView`
- `java.nio.HeapByteBufferFloatView`
- `java.nio.HeapByteBufferFloatView$NewHeapByteBufferFloatView`
- `java.nio.HeapByteBufferIntView`
- `java.nio.HeapByteBufferIntView$NewHeapByteBufferIntView`
- `java.nio.HeapByteBufferLongView`
- `java.nio.HeapByteBufferLongView$NewHeapByteBufferLongView`
- `java.nio.HeapByteBufferShortView`
- `java.nio.HeapByteBufferShortView$NewHeapByteBufferShortView`
- `java.nio.HeapCharBuffer`

- `java.nio.HeapCharBuffer$NewHeapCharBuffer`
- `java.nio.HeapDoubleBuffer`
- `java.nio.HeapDoubleBuffer$NewHeapDoubleBuffer`
- `java.nio.HeapFloatBuffer`
- `java.nio.HeapFloatBuffer$NewHeapFloatBuffer`
- `java.nio.HeapIntBuffer`
- `java.nio.HeapIntBuffer$NewHeapIntBuffer`
- `java.nio.HeapLongBuffer`
- `java.nio.HeapLongBuffer$NewHeapLongBuffer`
- `java.nio.HeapShortBuffer`
- `java.nio.HeapShortBuffer$NewHeapShortBuffer`
- `java.nio.IntBuffer`
- `java.nio.InvalidMarkException`
- `java.nio.LongBuffer`
- `java.nio.MappedByteBuffer`
- `java.nio.ReadOnlyBufferException`
- `java.nio.ShortBuffer`
- `java.nio.StringCharBuffer`
- `java.nio.channels.ByteChannel`
- `java.nio.channels.Channel`
- `java.nio.channels.Channels`
- `java.nio.channels.ClosedChannelException`
- `java.nio.channels.FileChannel`
- `java.nio.channels.FileChannel$MapMode`
- `java.nio.channels.FileChannelImpl`
- `java.nio.channels.FileLock`
- `java.nio.channels.GatheringByteChannel`
- `java.nio.channels.InterruptibleChannel`
- `java.nio.channels.NonReadableChannelException`
- `java.nio.channels.NonWritableChannelException`
- `java.nio.channels.OverlappingFileLockException`
- `java.nio.channels.ReadableByteChannel`
- `java.nio.channels.ScatteringByteChannel`
- `java.nio.channels.SeekableByteChannel`
- `java.nio.channels.WritableByteChannel`
- `java.nio.channels.spi.AbstractInterruptibleChannel`

- `java.nio.charset.CharacterCodingException`
- `java.nio.charset.Charset`
- `java.nio.charset.CharsetDecoder`
- `java.nio.charset.CharsetEncoder`
- `java.nio.charset.CoderMalfunctionError`
- `java.nio.charset.CoderResult`
- `java.nio.charset.CodingErrorAction`
- `java.nio.charset.IllegalCharsetNameException`
- `java.nio.charset.MalformedInputException`
- `java.nio.charset.StandardCharsets`
- `java.nio.charset.UnmappableCharacterException`
- `java.nio.charset.UnsupportedCharsetException`
- `java.nio.file.AccessDeniedException`
- `java.nio.file.CopyOption`
- `java.nio.file.DirectoryIteratorException`
- `java.nio.file.DirectoryNotEmptyException`
- `java.nio.file.DirectoryStream`
- `java.nio.file.DirectoryStream$Filter`
- `java.nio.file.DirectoryStreamImpl`
- `java.nio.file.FileAlreadyExistsException`
- `java.nio.file.FileSystem`
- `java.nio.file.FileSystemException`
- `java.nio.file.FileSystemLoopException`
- `java.nio.file.FileSystemNotFoundException`
- `java.nio.file.FileSystems`
- `java.nio.file.FileVisitOption`
- `java.nio.file.FileVisitResult`
- `java.nio.file.FileVisitor`
- `java.nio.file.Files`
- `java.nio.file.Files$TerminateTraversalException`
- `java.nio.file.LinkOption`
- `java.nio.file.NoSuchFileException`
- `java.nio.file.NotDirectoryException`
- `java.nio.file.NotLinkException`
- `java.nio.file.OpenOption`
- `java.nio.file.Path`

- `java.nio.file.PathMatcher`
- `java.nio.file.PathMatcherImpl`
- `java.nio.file.Paths`
- `java.nio.file.RegexPathMatcher`
- `java.nio.file.SimpleFileVisitor`
- `java.nio.file.StandardCopyOption`
- `java.nio.file.StandardOpenOption`
- `java.nio.file.StandardWatchEventKinds`
- `java.nio.file.WatchEvent`
- `java.nio.file.WatchEvent$Kind`
- `java.nio.file.WatchEvent$Modifier`
- `java.nio.file.WatchKey`
- `java.nio.file.WatchService`
- `java.nio.file.Watchable`
- `java.nio.file.attribute.AclEntry`
- `java.nio.file.attribute.AclFileAttributeView`
- `java.nio.file.attribute.AttributeView`
- `java.nio.file.attribute.BasicFileAttributeView`
- `java.nio.file.attribute.BasicFileAttributes`
- `java.nio.file.attribute.DosFileAttributeView`
- `java.nio.file.attribute.DosFileAttributes`
- `java.nio.file.attribute.FileAttribute`
- `java.nio.file.attribute.FileAttributeView`
- `java.nio.file.attribute.FileAttributeView$class`
- `java.nio.file.attribute.FileOwnerAttributeView`
- `java.nio.file.attribute.FileStoreAttributeView`
- `java.nio.file.attribute.FileTime`
- `java.nio.file.attribute.GroupPrincipal`
- `java.nio.file.attribute.PosixFileAttributeView`
- `java.nio.file.attribute.PosixFileAttributes`
- `java.nio.file.attribute.PosixFilePermission`
- `java.nio.file.attribute.PosixFilePermissions`
- `java.nio.file.attribute.UserDefinedFileAttributeView`
- `java.nio.file.attribute.UserPrincipal`
- `java.nio.file.attribute.UserPrincipalLookupService`
- `java.nio.file.spi.FileSystemProvider`

- `java.rmi.Remote`
- `java.rmi.RemoteException`
- `java.security.AccessControlException`
- `java.security.CodeSigner`
- `java.security.DummyMessageDigest`
- `java.security.GeneralSecurityException`
- `java.security.MessageDigest`
- `java.security.MessageDigestSpi`
- `java.security.NoSuchAlgorithmException`
- `java.security.Principal`
- `java.security.Timestamp`
- `java.security.TimestampConstructorHelper`
- `java.security.cert.CertPath`
- `java.security.cert.Certificate`
- `java.security.cert.CertificateEncodingException`
- `java.security.cert.CertificateException`
- `java.security.cert.CertificateFactory`
- `java.security.cert.X509Certificate`
- `java.security.cert.X509Extension`
- `java.text.DateFormatSymbols`
- `java.text.DecimalFormat`
- `java.text.DecimalFormat$BigDecimalFormatting`
- `java.text.DecimalFormat$BigIntegerFormatting`
- `java.text.DecimalFormat$DoubleFormatting`
- `java.text.DecimalFormat$Formatting`
- `java.text.DecimalFormat$Formatting$Digits`
- `java.text.DecimalFormat$Formatting$class`
- `java.text.DecimalFormat$LongFormatting`
- `java.text.DecimalFormat$PatternSyntax`
- `java.text.DecimalFormat$PatternSyntax$Affix`
- `java.text.DecimalFormat$PatternSyntax$Exponent`
- `java.text.DecimalFormat$PatternSyntax$Fraction`
- `java.text.DecimalFormat$PatternSyntax$Fraction$$plus$plus`
- `java.text.DecimalFormat$PatternSyntax$Integer`
- `java.text.DecimalFormat$PatternSyntax$MinimumExponent`
- `java.text.DecimalFormat$PatternSyntax$MinimumFraction`

- java.text.DecimalFormat$PatternSyntax$MinimumInteger
- java.text.DecimalFormat$PatternSyntax$Number
- java.text.DecimalFormat$PatternSyntax$Number$Fraction_$plus$plus
- java.text.DecimalFormat$PatternSyntax$Number$Integer_$plus$plus
- java.text.DecimalFormat$PatternSyntax$OptionalFraction
- java.text.DecimalFormat$PatternSyntax$Pattern
- java.text.DecimalFormat$PatternSyntax$Pattern$$plus$plus
- java.text.DecimalFormat$PatternSyntax$SignedPattern
- java.text.DecimalFormat$PatternSyntax$SignedPattern$Number_$plus$plus
- java.text.DecimalFormat$PatternSyntax$SignedPattern$Prefix_$plus$plus
- java.text.DecimalFormatSymbols
- java.text.FieldPosition
- java.text.Format
- java.text.Format$Field
- java.text.NumberFormat
- java.text.ParseException
- java.time.Instant
- java.util.AbstractCollection
- java.util.AbstractList
- java.util.AbstractListView
- java.util.AbstractMap
- java.util.AbstractMap$SimpleEntry
- java.util.AbstractMap$SimpleImmutableEntry
- java.util.AbstractQueue
- java.util.AbstractRandomAccessListIterator
- java.util.AbstractSequentialList
- java.util.AbstractSet
- java.util.ArrayList
- java.util.Arrays
- java.util.Arrays$AsRef
- java.util.BackedUpListIterator
- java.util.Base64
- java.util.Base64$Decoder
- java.util.Base64$DecodingInputStream
- java.util.Base64$Encoder
- java.util.Base64$EncodingOutputStream

- `java.util.Base64$Wrapper`
- `java.util.Calendar`
- `java.util.Collection`
- `java.util.Collections`
- `java.util.Collections$BasicSynchronizedList$1`
- `java.util.Collections$CheckedCollection`
- `java.util.Collections$CheckedList`
- `java.util.Collections$CheckedListIterator`
- `java.util.Collections$CheckedMap`
- `java.util.Collections$CheckedSet`
- `java.util.Collections$CheckedSortedMap`
- `java.util.Collections$CheckedSortedSet`
- `java.util.Collections$EmptyIterator`
- `java.util.Collections$EmptyListIterator`
- `java.util.Collections$ImmutableList`
- `java.util.Collections$ImmutableMap`
- `java.util.Collections$ImmutableSet`
- `java.util.Collections$UnmodifiableCollection`
- `java.util.Collections$UnmodifiableIterator`
- `java.util.Collections$UnmodifiableList`
- `java.util.Collections$UnmodifiableListIterator`
- `java.util.Collections$UnmodifiableMap`
- `java.util.Collections$UnmodifiableSet`
- `java.util.Collections$UnmodifiableSortedMap`
- `java.util.Collections$UnmodifiableSortedSet`
- `java.util.Collections$WrappedCollection`
- `java.util.Collections$WrappedCollection$class`
- `java.util.Collections$WrappedEquals`
- `java.util.Collections$WrappedEquals$class`
- `java.util.Collections$WrappedIterator`
- `java.util.Collections$WrappedIterator$class`
- `java.util.Collections$WrappedList`
- `java.util.Collections$WrappedList$class`
- `java.util.Collections$WrappedListIterator`
- `java.util.Collections$WrappedListIterator$class`
- `java.util.Collections$WrappedMap`

- `java.util.Collections$WrappedMap$class`
- `java.util.Collections$WrappedSet`
- `java.util.Collections$WrappedSortedMap`
- `java.util.Collections$WrappedSortedMap$class`
- `java.util.Collections$WrappedSortedSet`
- `java.util.Collections$WrappedSortedSet$class`
- `java.util.Comparator`
- `java.util.Comparator$class`
- `java.util.ConcurrentModificationException`
- `java.util.Date`
- `java.util.Deque`
- `java.util.Dictionary`
- `java.util.DuplicateFormatFlagsException`
- `java.util.EmptyStackException`
- `java.util.EnumSet`
- `java.util.Enumeration`
- `java.util.FormatFlagsConversionMismatchException`
- `java.util.Formattable`
- `java.util.FormattableFlags`
- `java.util.Formatter`
- `java.util.Formatter$BigDecimalLayoutForm`
- `java.util.Formatter$DateTimeUtil`
- `java.util.Formatter$FloatUtil`
- `java.util.Formatter$FormatToken`
- `java.util.Formatter$ParserStateMachine`
- `java.util.Formatter$Transformer`
- `java.util.FormatterClosedException`
- `java.util.GregorianCalendar`
- `java.util.HashMap`
- `java.util.HashMap$AbstractMapView`
- `java.util.HashMap$AbstractMapView$class`
- `java.util.HashMap$AbstractMapViewIterator`
- `java.util.HashMap$EntrySet`
- `java.util.HashMap$KeySet`
- `java.util.HashMap$ValuesView`
- `java.util.HashSet`

- `java.util.Hashtable`
- `java.util.Hashtable$UnboxedEntry$1`
- `java.util.IllegalFormatCodePointException`
- `java.util.IllegalFormatConversionException`
- `java.util.IllegalFormatException`
- `java.util.IllegalFormatFlagsException`
- `java.util.IllegalFormatPrecisionException`
- `java.util.IllegalFormatWidthException`
- `java.util.IllformedLocaleException`
- `java.util.InputMismatchException`
- `java.util.InvalidPropertiesFormatException`
- `java.util.Iterator`
- `java.util.LinkedHashMap`
- `java.util.LinkedHashSet`
- `java.util.LinkedList`
- `java.util.LinkedList$Node`
- `java.util.List`
- `java.util.ListIterator`
- `java.util.Locale`
- `java.util.Map`
- `java.util.Map$Entry`
- `java.util.MissingFormatArgumentException`
- `java.util.MissingFormatWidthException`
- `java.util.MissingResourceException`
- `java.util.NavigableSet`
- `java.util.NavigableView`
- `java.util.NoSuchElementException`
- `java.util.Objects`
- `java.util.PriorityQueue`
- `java.util.PriorityQueue$BoxOrdering`
- `java.util.Properties`
- `java.util.Queue`
- `java.util.Random`
- `java.util.RandomAccess`
- `java.util.RandomAccessListIterator`
- `java.util.ServiceConfigurationError`

- java.util.Set
- java.util.SizeChangeEvent
- java.util.SizeChangeEvent$class
- java.util.SortedMap
- java.util.SortedSet
- java.util.StringTokenizer
- java.util.TimeZone
- java.util.TooManyListenersException
- java.util.TreeSet
- java.util.TreeSet$BoxOrdering
- java.util.UUID
- java.util.UnknownFormatConversionException
- java.util.UnknownFormatFlagsException
- java.util.WeakHashMap
- java.util.WeakHashMap$AbstractMapView
- java.util.WeakHashMap$AbstractMapView$class
- java.util.WeakHashMap$AbstractMapViewIterator
- java.util.WeakHashMap$EntrySet
- java.util.WeakHashMap$KeySet
- java.util.WeakHashMap$ValuesView
- java.util.concurrent.Callable
- java.util.concurrent.CancellationException
- java.util.concurrent.ExecutionException
- java.util.concurrent.Executor
- java.util.concurrent.RejectedExecutionException
- java.util.concurrent.TimeUnit
- java.util.concurrent.TimeoutException
- java.util.concurrent.atomic.AtomicBoolean
- java.util.concurrent.atomic.AtomicInteger
- java.util.concurrent.atomic.AtomicLong
- java.util.concurrent.atomic.AtomicLongArray
- java.util.concurrent.atomic.AtomicReference
- java.util.concurrent.atomic.AtomicReferenceArray
- java.util.concurrent.locks.AbstractOwnableSynchronizer
- java.util.concurrent.locks.AbstractQueuedSynchronizer
- java.util.function.BiFunction

- `java.util.function.BiFunction$class`
- `java.util.function.BiPredicate`
- `java.util.function.BiPredicate$class`
- `java.util.function.Function`
- `java.util.function.Function$class`
- `java.util.function.Predicate`
- `java.util.function.Predicate$class`
- `java.util.function.Supplier`
- `java.util.function.UnaryOperator`
- `java.util.jar.Attributes`
- `java.util.jar.Attributes$Name`
- `java.util.jar.InitManifest`
- `java.util.jar.JarEntry`
- `java.util.jar.JarFile`
- `java.util.jar.JarFile$JarFileEnumerator$1`
- `java.util.jar.JarFile$JarFileInputStream`
- `java.util.jar.JarInputStream`
- `java.util.jar.JarOutputStream`
- `java.util.jar.JarVerifier`
- `java.util.jar.JarVerifier$VerifierEntry`
- `java.util.jar.Manifest`
- `java.util.jar.Manifest$Chunk`
- `java.util.package`
- `java.util.package$Box`
- `java.util.package$CompareNullablesOps`
- `java.util.regex.MatchResult`
- `java.util.regex.Matcher`
- `java.util.regex.Pattern`
- `java.util.regex.Pattern$CompiledPatternStore`
- `java.util.regex.Pattern$CompiledPatternStore$Key`
- `java.util.regex.Pattern$CompiledPatternStore$Node`
- `java.util.regex.PatternSyntaxException`
- `java.util.regex.cre2`
- `java.util.regex.cre2h`
- `java.util.regex.cre2h$RE2RegExpOps`
- `java.util.regex.cre2h$RE2StringOps`

- `java.util.stream.BaseStream`
- `java.util.stream.CompositeStream`
- `java.util.stream.EmptyIterator`
- `java.util.stream.Stream`
- `java.util.stream.Stream$Builder`
- `java.util.stream.Stream$Builder$class`
- `java.util.stream.WrappedScalaStream`
- `java.util.stream.WrappedScalaStream$Builder`
- `java.util.zip.Adler32`
- `java.util.zip.CRC32`
- `java.util.zip.CheckedInputStream`
- `java.util.zip.CheckedOutputStream`
- `java.util.zip.Checksum`
- `java.util.zip.DataFormatException`
- `java.util.zip.Deflater`
- `java.util.zip.DeflaterOutputStream`
- `java.util.zip.GZIPInputStream`
- `java.util.zip.GZIPOutputStream`
- `java.util.zip.Inflater`
- `java.util.zip.InflaterInputStream`
- `java.util.zip.ZipConstants`
- `java.util.zip.ZipConstants$class`
- `java.util.zip.ZipEntry`
- `java.util.zip.ZipEntry$LittleEndianReader`
- `java.util.zip.ZipException`
- `java.util.zip.ZipFile`
- `java.util.zip.ZipFile$RAFStream`
- `java.util.zip.ZipFile$ZipInflaterInputStream`
- `java.util.zip.ZipInputStream`
- `java.util.zip.ZipOutputStream`

Note: This is an ongoing effort, some of the classes listed here might be partially implemented. Please consult javalib sources for details.

Regular expressions (java.util.regex)

Scala Native implements *java.util.regex*-compatible API using Google's RE2 library. There is some differences in terms of the support of the regular expression language.

Some expressions are not supported:

- **Character classes:**
  - Unions: `[a-d[m-p]]`
  - Intersections: `[a-z&&[^aeiou]]`
- Predefined character classes: `\h`, `\H`, `\v`, `\V`
- **Java character function classes:**
  - `\p{javaLowerCase}`
  - `\p{javaUpperCase}`
  - `\p{javaWhitespace}`
  - `\p{javaMirrored}`
- Boundary matchers: `\G`, `\Z`, `\R`
- Possessive quantifiers: `X?+`, `X*+`, `X++`, `X{n}+`, `X{n,}+`, `X{n,m}+`
- Lookaheads: `(?=X)`, `(?!X)`, `(?<=X)`, `(?<!X)`, `(?>X)`

Some expressions have an alternative syntax:

Java	RE2
`(?<foo>a)`	`(?P<foo>a)`
`p{Alnum}`	`[[:alnum:]]`
`p{Alpha}`	`[[:alpha:]]`
`p{ASCII}`	`[[:ascii:]]`
`p{Blank}`	`[[:blank:]]`
`p{Cntrl}`	`[[:cntrl:]]`
`p{Digit}`	`[[:digit:]]`
`p{Graph}`	`[[:graph:]]`
`p{Lower}`	`[[:lower:]]`
`p{Print}`	`[[:print:]]`
`p{Punct}`	`[[:punct:]]`
`p{Space}`	`[[:space:]]`
`p{Upper}`	`[[:upper:]]`
`p{XDigit}`	`[[:xdigit:]]`
`p{InGreek}`	`p{Greek}`
`p{IsLatin}`	`p{Latin}`

Continue to *C Standard Library*.

2.2.2 C Standard Library

Scala Native provides bindings for the core subset of the C standard library:

C Header	Scala Native Module
assert.h	N/A
complex.h	scala.scalanative.native.complex
ctype.h	scala.scalanative.native.ctype
errno.h	scala.scalanative.native.errno
fenv.h	N/A
float.h	N/A
inttypes.h	N/A
iso646.h	N/A
limits.h	N/A
locale.h	N/A
math.h	scala.scalanative.native.math
setjmp.h	N/A
signal.h	scala.scalanative.native.signal
stdalign.h	N/A
stdarg.h	N/A
stdatomic.h	N/A
stdbool.h	N/A
stddef.h	N/A
stdint.h	N/A
stdio.h	scala.scalanative.native.stdio
stdlib.h	scala.scalanative.native.stdlib
stdnoreturn.h	N/A
string.h	scala.scalanative.native.string
tgmath.h	N/A
threads.h	N/A
time.h	N/A
uchar.h	N/A
wchar.h	N/A
wctype.h	N/A

Continue to *C POSIX Library*.

2.2.3 C POSIX Library

Scala Native provides bindings for the core subset of the POSIX library:

C Header	Scala Native Module
aio.h	N/A
arpa/inet.h	scala.scalanative.posix.arpa.inet
assert.h	N/A
complex.h	scala.scalanative.native.complex
cpio.h	scala.scalanative.posix.cpio
ctype.h	N/A
dirent.h	scala.scalanative.posix.dirent
dlfcn.h	N/A
errno.h	scala.scalanative.posix.errno
fcntl.h	scala.scalanative.posix.fcntl
fenv.h	N/A

Continued on next page

Table 1 – continued from previous page

C Header	Scala Native Module
float.h	N/A
fmtmsg.h	N/A
fnmatch.h	N/A
ftw.h	N/A
getopt.h	scala.scalanative.posix.getopt
glob.h	N/A
grp.h	scala.scalanative.posix.grp
iconv.h	N/A
inttypes.h	scala.scalanative.posix.inttypes
iso646.h	N/A
langinfo.h	N/A
libgen.h	N/A
limits.h	scala.scalanative.posix.limits
locale.h	N/A
math.h	N/A
monetary.h	N/A
mqueue.h	N/A
ndbm.h	N/A
net/if.h	N/A
netdb.h	scala.scalanative.posix.netdb
netinet/in.h	scala.scalanative.posix.netinet.in
netinet/tcp.h	N/A
nl_types.h	N/A
poll.h	scala.scalanative.posix.poll
pthread.h	scala.scalanative.posix.pthread
pwd.h	scala.scalanative.posix.pwd
regex.h	scala.scalanative.posix.regex
sched.h	scala.scalanative.posix.sched
search.h	N/A
semaphore.h	N/A
setjmp.h	N/A
signal.h	N/A
spawn.h	N/A
stdarg.h	N/A
stdbool.h	N/A
stddef.h	N/A
stdint.h	N/A
stdio.h	N/A
stdlib.h	scala.scalanative.posix.stdlib
string.h	N/A
strings.h	N/A
stropts.h	N/A
sys/ipc.h	N/A
sys/mman.h	N/A
sys/msg.h	N/A
sys/resource.h	N/A
sys/select.h	scala.scalanative.posix.sys.select
sys/sem.h	N/A
sys/shm.h	N/A

Continued on next page

Table 1 – continued from previous page

C Header	Scala Native Module
sys/socket.h	scala.scalanative.posix.sys.socket
sys/stat.h	scala.scalanative.posix.sys.stat
sys/statvfs.h	N/A
sys/time.h	scala.scalanative.posix.sys.time
sys/times.h	N/A
sys/types.h	scala.scalanative.posix.sys.types
sys/uio.h	scala.scalanative.posix.sys.uio
sys/un.h	N/A
sys/utsname.h	N/A
sys/wait.h	N/A
syslog.h	scala.scalanative.posix.syslog
tar.h	N/A
termios.h	scala.scalanative.posix.termios
tgmath.h	N/A
time.h	scala.scalanative.posix.time
trace.h	N/A
ulimit.h	N/A
unistd.h	scala.scalanative.posix.unistd
utime.h	scala.scalanative.posix.utime
utmpx.h	N/A
wchar.h	N/A
wctype.h	N/A
wordexp.h	N/A

Continue to *Community Libraries*.

2.2.4 Community Libraries

Third-party libraries for Scala Native can be found using:

- Scala Native libraries indexed by MVN Repository.
- Awesome Scala Native, a curated list of Scala Native libraries and projects.

Continue to *FAQ*.

2.3 Contributor's Guide

2.3.1 Contributing guidelines

Very important notice about Javalib

Scala Native contains a re-implementation of part of the JDK.

Although the GPL and Scala License are compatible and the GPL and Scala CLA are compatible, EPFL wish to distribute scala native under a permissive license.

When you sign the Scala CLA you are confirming that your contributions are your own creation. This is especially important, as it denies you the ability to copy any source code, e.g. Android, OpenJDK, Apache Harmony, GNU Classpath or Scala.js. To be clear, you are personally liable if you provide false information regarding the authorship of your contribution.

However, we are prepared to accept contributions that include code copied from Scala.js or Apache Harmony project on a case-by-case basis. In such cases, you must fulfil your obligations and include the relevant copyright / license information.

Coding style

Scala Native is formatted via *./scripts/scalafmt* and *./scripts/clangfmt*. Make sure that all of your contributions are properly formatted before suggesting any changes.

General workflow

This the general workflow for contributing to Scala Native.

1. Make sure you have signed the Scala CLA. If not, sign it.
2. You should always perform your work in its own Git branch. The branch should be given a descriptive name that explains its intent.
3. When the feature or fix is completed you should open a Pull Request on GitHub.
4. The Pull Request should be reviewed by other maintainers (as many as feasible/practical), among which at least one core developer. Independent contributors can also participate in the review process, and are encouraged to do so.
5. After the review, you should resolve issues brought up by the reviewers as needed (amending or adding commits to address reviewers' comments), iterating until the reviewers give their thumbs up, the "LGTM" (acronym for "Looks Good To Me").
6. Once the code has passed review the Pull Request can be merged into the distribution.

Git workflow

Scala Native repositories maintain a linear merge-free history on the master branch. All of the incoming pull requests are merged using squash and merge policy (i.e. one merged pull request corresponds to one squashed commit to the master branch.)

You do not need to squash commits manually. It's typical to add new commits to the PR branch to accommodate changes that were suggested by the reviewers. Squashing things manually and/or rewriting history on the PR branch is all-right as long as it's clear that concerns raised by reviewers have been addressed.

Maintaining a long-standing work-in-progress (WIP) branch requires one to rebase on top of latest master using `git rebase --onto` from time to time. It's strongly recommended not to perform any merges on your branches that you are planning to use as a PR branch.

Pull Request Requirements

In order for a Pull Request to be considered, it has to meet these requirements:

1. Live up to the current code standard:
   - Be formatted with *./scripts/scalafmt* and *./scripts/clangfmt*.
   - Not violate DRY.
   - Boy Scout Rule should be applied.
2. Be accompanied by appropriate tests.

3. Be issued from a branch *other than master* (PRs coming from master will not be accepted.)

If not *all* of these requirements are met then the code should **not** be merged into the distribution, and need not even be reviewed.

Documentation

All code contributed to the user-facing standard library (the *nativelib/* directory) should come accompanied with documentation. Pull requests containing undocumented code will not be accepted.

Code contributed to the internals (nscplugin, tools, etc.) should come accompanied by internal documentation if the code is not self-explanatory, e.g., important design decisions that other maintainers should know about.

Creating Commits And Writing Commit Messages

Follow these guidelines when creating public commits and writing commit messages.

Prepare meaningful commits

If your work spans multiple local commits (for example; if you do safe point commits while working in a feature branch or work in a branch for long time doing merges/rebases etc.) then please do not commit it all but rewrite the history by squashing the commits into **one commit per useful unit of change**, each accompanied by a detailed commit message. For more info, see the article: Git Workflow. Additionally, every commit should be able to be used in isolation–that is, each commit must build and pass all tests.

First line of the commit message

The first line should be a descriptive sentence about what the commit is doing, written using the imperative style, e.g., "Change this.", and should not exceed 70 characters. It should be possible to fully understand what the commit does by just reading this single line. It is **not ok** to only list the ticket number, type "minor fix" or similar. If the commit has a corresponding ticket, include a reference to the ticket number, with the format "Fix #xxx: Change that.", as the first line. Sometimes, there is no better message than "Fix #xxx: Fix that issue.", which is redundant. In that case, and assuming that it aptly and concisely summarizes the commit in a single line, the commit message should be "Fix #xxx: Title of the ticket.".

Body of the commit message

If the commit is a small fix, the first line can be enough. Otherwise, following the single line description should be a blank line followed by details of the commit, in the form of free text, or bulleted list.

2.3.2 Guide to the sbt build

This section gives some basic information and tips about the build system. The `sbt` build system is quite complex and effectively brings together all the components of Scala Native. The `build.sbt` file is at the root of the project along with the sub-projects that make up the system.

Overview

In order to effectively work with Scala Native, a knowledge of the build system is very helpful. In general the code is built and published to your local Ivy repository using the *sbt publishLocal* command so that other components in the system can depend on each other via normal *sbt* dependencies. Although the `build.sbt` file and other code in the system is the way to learn the system thoroughly, the following sections will give information that should be helpful to get started.

The build has roughly four groups of sub-projects as follows:

1. The Native Scala Compiler plugin and libraries. Each of these projects depend on the next project in the list.
   - *nscplugin*
   - *nativelib*
   - *javalib*
   - *auxlib*
   - *scalalib*
2. The Scala Native plugin and dependencies (directory names are in parentheses).
   - *sbtScalaNative (sbt-scala-native)*
   - *tools*
   - *nir*
   - *util*
3. The Scala Native test interface and dependencies.
   - *testInterface (test-interface)*
   - *testInterfaceSerialization (test-interface-serialization)*
   - *testInterfaceSbtDefs (test-interface-sbt-defs)*
4. Tests and benchmarks (no dependencies on each other).
   - *tests (unit-tests)*
   - *tools* This has tests within the project
   - *(scripted-tests)*
   - *benchmarks*

Each of the groups above also depend on the previous group being compiled and published locally. The sbt plugin `sbtScalaNative` is used inside Scala Native exactly as it is used in a project using Scala Native. The plugin is needed by the *testInterface* and also the *tests* that use the *testInterface* to compile native code.

Building Scala Native

Once you have cloned Scala Native from git, *cd* into the base directory. Inside this directory is the *build.sbt* file which is used to build Scala Native. This file has *sbt* command aliases which are used to help build the system. In order to build Scala Native for the first time you should run the following commands:

```
$ sbt
> rebuild
```

If you want to run all the tests and benchmarks, which takes awhile, you can run the *test-all* command after the systems builds.

Normal development workflow

Let us suppose that you wish to work on the *javalib* project to add some code or fix a bug. Once you make a change to the code, run the following command at the sbt prompt to compile the code and run the tests:

```
> javalib/publishLocal
> tests/test
```

You can run only the test of interest by using one of the following commands:

```
> tests/testOnly java.lang.StringSuite
> tests/testOnly *StringSuite
```

Some additional tips are as follows.

- If you change anything in *tools* (linker, optimizer, codegen), you need to *reload*, not *rebuild*. It's possible because we textually include code of the *sbt* plugin and the toolchain.
- If you change *nscplugin*, *rebuild* is the only option. This is because the Scala compiler uses this plugin to generate the code that Scala Native uses.

Build settings via environment variables

Two build settings, `nativeGC` and `nativeMode` can be changed via environment variables. They have default settings that are used unless changed. The setting that controls the garbage collector is *nativeGC*. Scala Native has a high performance Garbage Collector (GC) `immix` that comes with the system or the *boehm* GC which can be used when the supporting library is installed. The setting *none* also exists for a short running script or where memory is not an issue.

Scala Native uses Continuous integration (CI) to compile and test the code on different platforms[1] and using different garbage collectors[2]. The Scala Native *sbt* plugin includes the ability to set an environment variable *SCALANATIVE_GC* to set the garbage collector value used by *sbt*. Setting this as follows will set the value in the plugin when *sbt* is run.

```
$ export SCALANATIVE_GC=immix
$ sbt
> show nativeGC
```

This setting remains unless changed at the *sbt* prompt. If changed, the value will be restored to the environment variable value if *sbt* is restarted or *reload* is called at the *sbt* prompt. You can also revert to the default setting value by running *unset SCALANATIVE_GC* at the command line and then restarting *sbt*.

The *nativeMode* setting is controlled via the *SCALANATIVE_MODE* environment variable. The default mode, *debug* is designed to optimize but compile fast whereas the *release* mode performs additional optimizations and takes longer to compile.

Setting the GC setting via *sbt*

The GC setting is only used during the link phase of the Scala Native compiler so it can be applied to one or all the Scala Native projects that use the *sbtScalaNative* plugin. This is an example to only change the setting for the *sandbox*.

[1] http://www.scala-native.org/en/latest/user/setup.html
[2] http://www.scala-native.org/en/latest/user/sbt.html

```
$ sbt
> show nativeGC
> set nativeGC in sandbox := "none"
> show nativeGC
> sandbox/run
```

The following shows how to set `nativeGC` on all the projects.

```
> set every nativeGC := "immix"
> show nativeGC
```

The same process above will work for setting *nativeMode*.

The next section has more build and development information for those wanting to work on *The compiler plugin and code generator*.

2.3.3 The compiler plugin and code generator

Compilation to native code happens in two steps. First, Scala code is compiled into *Native Intermediate Representation* by nscplugin, the Scala compiler plugin. It runs as one of the later phases of the Scala compiler and inspects the AST and generates `.nir` files. Finally, the `.nir` files are compiled into `.ll` files and passed to LLVM by the native compiler.

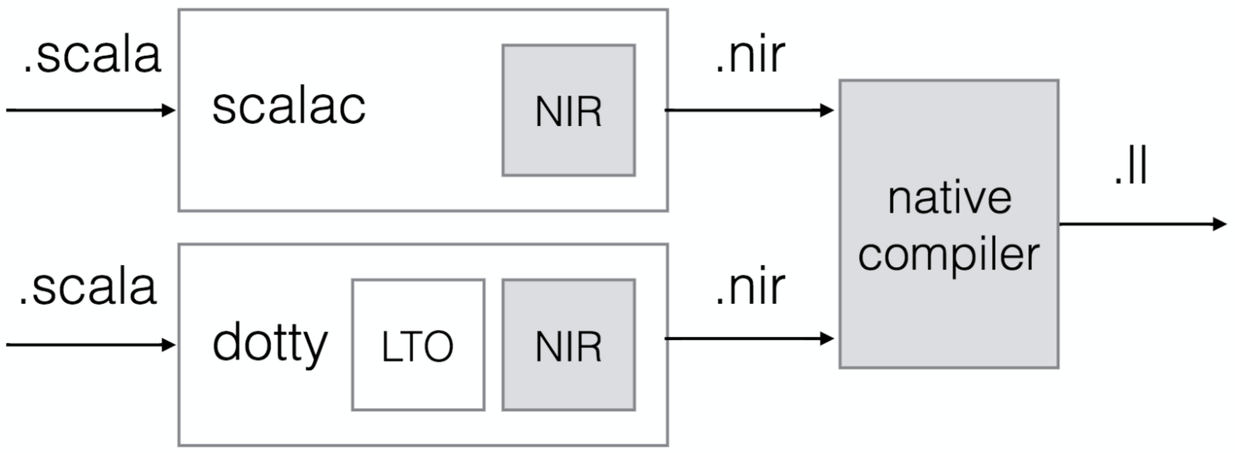


Fig. 1: High-level overview of the compilation process.

Tips for working on the compiler

When adding a new intrinsic, the first thing to check is how clang would compile it in C. Write a small program with the behavior you are trying to add and compile it to `.ll` using:

```
clang -S -emit-llvm foo.c
```

Now write the equivalent Scala code for the new intrinsic in the sandbox project. This project contains a minimal amount of code and has all the toolchain set up which makes it fast to iterate and inspect the output of the compilation.

To compile the sandbox project run the following in the sbt shell:

```
sbt> ;sandbox/clean;sandbox/nativeLink
```

If the example code for the new intrinsic requires you to change APIs in `nativelib`, then remember to also publish the changes with `nativelib/publishLocal`.

After compiling the sandbox project you can inspect the `.ll` files inside `sandbox/target/scala-<version>/ll`. The files are grouped by the package name. By default the `Test.scala` file doesn't define a package, so the resulting file will be `__empty.ll`. Locating the code you are interested in might require that you get more familiar with the LLVM assembly language.

When working on the compile plugin you'll need to publish it and reload each time you want to recompile the sandbox project. This can be achieved with:

```
sbt> ;nscplugin/publishLocal;reload;sandbox/clean;sandbox/run
```

Certain intrinsics might require adding new primitives to the compiler plugin. This can be done in `NirPrimitives` with an accompanying definition in `NirDefinitions`. Ensure that new primitives are correctly registered.

The NIR code generation uses a builder to maintain the generated instructions. This allows to inspect the instructions before and after the part of the compilation you are working on has generated code.

2.3.4 Native Intermediate Representation

NIR is high-level object-oriented SSA-based representation. The core of the representation is a subset of LLVM instructions, types and values, augmented with a number of high-level primitives that are necessary to efficiently compile modern languages like Scala.

Contents

- *Native Intermediate Representation*
  - *Introduction*
  - *Definitions*
    - *Var*
    - *Const*
    - *Declare*
    - *Define*
    - *Struct*
    - *Trait*
    - *Class*
    - *Module*
  - *Types*
    - *Void*
    - *Vararg*
    - *Pointer*
    - *Boolean*

    * *Integer*
    * *Float*
    * *Array*
    * *Function*
    * *Struct*
    * *Unit*
    * *Nothing*
    * *Class*
    * *Trait*
    * *Module*
  - *Control-Flow*
    * *unreachable*
    * *ret*
    * *jump*
    * *if*
    * *switch*
    * *invoke*
    * *throw*
    * *try*
  - *Operands*
    * *call*
    * *load*
    * *store*
    * *elem*
    * *extract*
    * *insert*
    * *stackalloc*
    * *bin*
    * *comp*
    * *conv*
    * *sizeof*
    * *classalloc*
    * *field*
    * *method*
    * *dynmethod*

    * *as*
    * *is*
  - *Values*
    * *Boolean*
    * *Zero and null*
    * *Integer*
    * *Float*
    * *Struct*
    * *Array*
    * *Local*
    * *Global*
    * *Unit*
    * *Null*
    * *String*
  - *Attributes*
    * *Inlining*
      · *mayinline*
      · *inlinehint*
      · *noinline*
      · *alwaysinline*
    * *Linking*
      · *link*
      · *pin*
      · *pin-if*
      · *pin-weak*
      · *stub*
    * *Misc*
      · *dyn*
      · *pure*
      · *extern*
      · *override*

Introduction

Lets have a look at the textual form of NIR generated for a simple Scala module:

```
object Test {
  def main(args: Array[String]): Unit =
    println("Hello, world!")
}
```

Would map to:

```
pin(@Test$::init) module @Test$ : @java.lang.Object

def @Test$::main_class.ssnr.ObjectArray_unit : (module @Test$, class @scala.
↪scalanative.runtime.ObjectArray) => unit {
  %src.2(%src.0 : module @Test$, %src.1 : class @scala.scalanative.runtime.
↪ObjectArray):
    %src.3 = module @scala.Predef$
    %src.4 = method %src.3 : module @scala.Predef$, @scala.Predef$::println_class.
↪java.lang.Object_unit
    %src.5 = call[(module @scala.Predef$, class @java.lang.Object) => unit] %src.4 :␣
↪ptr(%src.3 : module @scala.Predef$, "Hello, world!")
    ret %src.5 : unit
}

def @Test$::init : (module @Test$) => unit {
  %src.1(%src.0 : module @Test$):
    %src.2 = call[(class @java.lang.Object) => unit] @java.lang.Object::init : ptr(
↪%src.0 : module @Test$)
    ret unit
}
```

Here we can see a few distinctive features of the representation:

1. At its core NIR is very much a classical SSA-based representation. The code consists of basic blocks of instructions. Instructions take value and type parameters. Control flow instructions can only appear as the last instruction of the basic block.
2. Basic blocks have parameters. Parameters directly correspond to phi instructions in the classical SSA.
3. The representation is strongly typed. All parameters have explicit type annotations. Instructions may be overloaded for different types via type parameters.
4. Unlike LLVM, it has support for high-level object-oriented features such as garbage-collected classes, traits and modules. They may contain methods and fields. There is no overloading or access control modifiers so names must be mangled appropriately.
5. All definitions live in a single top-level scope indexed by globally unique names. During compilation they are lazily loaded until all reachable definitions have been discovered. *pin* and *pin-if* attributes are used to express additional dependencies.

Definitions

Var

```
..$attrs var @$name: $ty = $value
```

Corresponds to LLVM's global variables when used in the top-level scope and to fields, when used as a member of classes and modules.

Const

```
..$attrs const @$name: $type = $value
```

Corresponds to LLVM's global constant. Constants may only reside on the top-level and can not be members of classes and modules.

Declare

```
..$attrs def @$name: $type
```

Correspond to LLVM's declare when used on the top-level of the compilation unit and to abstract methods when used inside classes and traits.

Define

```
..$attrs def @$name: $type { ..$blocks }
```

Corresponds to LLVM's define when used on the top-level of the compilation unit and to normal methods when used inside classes, traits and modules.

Struct

```
..$attrs struct @$name { ..$types }
```

Corresponds to LLVM's named struct.

Trait

```
..$attrs trait @$name : ..$traits
```

Scala-like traits. May contain abstract and concrete methods as members.

Class

```
..$attrs class @$name : $parent, ..$traits
```

Scala-like classes. May contain vars, abstract and concrete methods as members.

Module

```
..$attrs module @$name : $parent, ..$traits
```

Scala-like modules (i.e. `object $name`) May only contain vars and concrete methods as members.

Types

Void

```
void
```

Corresponds to LLVM's void.

Vararg

```
...
```

Corresponds to LLVM's varargs. May only be nested inside function types.

Pointer

```
ptr
```

Corresponds to LLVM's pointer type with a major distinction of not preserving the type of memory that's being pointed at. Pointers are going to become untyped in LLVM in near future too.

Boolean

```
bool
```

Corresponds to LLVM's i1.

Integer

```
i8
i16
i32
i64
```

Corresponds to LLVM integer types. Unlike LLVM we do not support arbitrary width integer types at the moment.

Float

```
f32
f64
```

Corresponds to LLVM's floating point types.

Array

```
[$type x N]
```

Corresponds to LLVM's aggregate array type.

Function

```
(..$args) => $ret
```

Corresponds to LLVM's function type.

Struct

```
struct @$name
struct { ..$types }
```

Has two forms: named and anonymous. Corresponds to LLVM's aggregate structure type.

Unit

```
unit
```

A reference type that corresponds to `scala.Unit`.

Nothing

```
nothing
```

Corresponds to `scala.Nothing`. May only be used a function return type.

Class

```
class @$name
```

A reference to a class instance.

Trait

```
trait @$name
```

A reference to a trait instance.

Module

```
module @$name
```

A reference to a module.

Control-Flow

unreachable

```
unreachable
```

If execution reaches undefined instruction the behaviour of execution is undefined starting from that point. Corresponds to LLVM's unreachable.

ret

```
ret $value
```

Returns a value. Corresponds to LLVM's ret.

jump

```
jump $next(..$values)
```

Jumps to the next basic block with provided values for the parameters. Corresponds to LLVM's unconditional version of br.

if

```
if $cond then $next1(..$values1) else $next2(..$values2)
```

Conditionally jumps to one of the basic blocks. Corresponds to LLVM's conditional form of br.

switch

```
switch $value {
   case $value1 => $next1(..$values1)
   ...
   default      => $nextN(..$valuesN)
}
```

Jumps to one of the basic blocks if `$value` is equal to corresponding `$valueN`. Corresponds to LLVM's switch.

invoke

```
invoke[$type] $ptr(..$values) to $success unwind $failure
```

Invoke function pointer, jump to success in case value is returned, unwind to failure if exception was thrown. Corresponds to LLVM's invoke.

throw

```
throw $value
```

Throws the values and starts unwinding.

try

```
try $succ catch $failure
```

Operands

All non-control-flow instructions follow a general pattern of `%$name = $opname[..$types] ..$values`. Purely side-effecting operands like `store` produce `unit` value.

call

```
call[$type] $ptr(..$values)
```

Calls given function of given function type and argument values. Corresponds to LLVM's call.

load

```
load[$type] $ptr
```

Load value of given type from memory. Corresponds to LLVM's load.

store

```
store[$type] $ptr, $value
```

Store value of given type to memory. Corresponds to LLVM's store.

elem

```
elem[$type] $ptr, ..$indexes
```

Compute derived pointer starting from given pointer. Corresponds to LLVM's getelementptr.

extract

```
extract[$type] $aggrvalue, $index
```

Extract element from aggregate value. Corresponds to LLVM's extractvalue.

insert

```
insert[$type] $aggrvalue, $value, $index
```

Create a new aggregate value based on existing one with element at index replaced with new value. Corresponds to LLVM's insertvalue.

stackalloc

```
stackalloc[$type]
```

Stack allocate a slot of memory big enough to store given type. Corresponds to LLVM's alloca.

bin

```
$bin[$type] $value1, $value2`
```

Where `$bin` is one of the following: `iadd`, `fadd`, `isub`, `fsub`, `imul`, `fmul`, `sdiv`, `udiv`, `fdiv`, `srem`, `urem`, `frem`, `shl`, `lshr`, `ashr` , `and`, `or`, `xor`. Depending on the type and signedness, maps to either integer or floating point binary operations in LLVM.

comp

```
$comp[$type] $value1, $value2
```

Where `$comp` is one of the following: `eq`, `neq`, `lt`, `lte`, `gt`, `gte`. Depending on the type, maps to either icmp or fcmp with corresponding comparison flags in LLVM.

conv

```
$conv[$type] $value
```

Where `$conv` is one of the following: `trunc`, `zext`, `sext`, `fptrunc`, `fpext`, `fptoui`, `fptosi`, `uitofp`, `sitofp`, `ptrtoint`, `inttoptr`, `bitcast`. Corresponds to LLVM conversion instructions with the same name.

sizeof

```
sizeof[$type]
```

Returns a size of given type.

classalloc

```
classalloc @$name
```

Roughly corresponds to `new $name` in Scala. Performs allocation without calling the constructor.

field

```
field[$type] $value, @$name
```

Returns a pointer to the given field of given object.

method

```
method[$type] $value, @$name
```

Returns a pointer to the given method of given object.

dynmethod

```
dynmethod $obj, $signature
```

Returns a pointer to the given method of given object and signature.

as

```
as[$type] $value
```

Corresponds to `$value.asInstanceOf[$type]` in Scala.

is

```
is[$type] $value
```

Corresponds to `$value.isInstanceOf[$type]` in Scala.

Values

Boolean

```
true
false
```

Corresponds to LLVM's `true` and `false`.

Zero and null

```
null
zero $type
```

Corresponds to LLVM's `null` and `zeroinitializer`.

Integer

```
Ni8
Ni16
Ni32
Ni64
```

Correponds to LLVM's integer values.

Float

```
N.Nf32
N.Nf64
```

Corresponds to LLVM's floating point values.

Struct

```
struct @$name {..$values}`
```

Corresponds to LLVM's struct values.

Array

```
array $ty {..$values}
```

Corresponds to LLVM's array value.

Local

```
%$name
```

Named reference to result of previously executed instructions or basic block parameters.

Global

```
@$name
```

Reference to the value of top-level definition.

Unit

```
unit
```

Corresponds to `()` in Scala.

Null

```
null
```

Corresponds to null literal in Scala.

String

```
"..."
```

Corresponds to string literal in Scala.

Attributes

Attributes allow one to attach additional metadata to definitions and instructions.

Inlining

mayinline

```
mayinline
```

Default state: optimiser is allowed to inline given method.

inlinehint

```
inlinehint
```

Optimiser is incentivized to inline given methods but it is allowed not to.

noinline

```
noinline
```

Optimiser must never inline given method.

alwaysinline

```
alwaysinline
```

Optimiser must always inline given method.

Linking

link

```
link($name)
```

Automatically put `$name` on a list of native libraries to link with if the given definition is reachable.

pin

```
pin(@$name)
```

Require `$name` to be reachable, whenever current definition is reachable. Used to introduce indirect linking dependencies. For example, module definitions depend on its constructors using this attribute.

pin-if

```
pin-if(@$name, @$cond)
```

Require `$name` to be reachable if current and `$cond` definitions are both reachable. Used to introduce conditional indirect linking dependencies. For example, class constructors conditionally depend on methods overridden in given class if the method that are being overridden are reachable.

pin-weak

```
pin-weak(@$name)
```

Require `$name` to be reachable if there is a reachable dynmethod with matching signature.

stub

```
stub
```

Indicates that the annotated method, class or module is only a stub without implementation. If the linker is configured with `linkStubs = false`, then these definitions will be ignored and a linking error will be reported. If `linkStubs = true`, these definitions will be linked.

Misc

dyn

```
dyn
```

Indication that a method can be called using a structural type dispatch.

pure

```
pure
```

Let optimiser assume that calls to given method are effectively pure. Meaning that if the same method is called twice with exactly the same argument values, it can re-use the result of first invocation without calling the method twice.

extern

```
extern
```

Use C-friendly calling convention and don't name-mangle given method.

override

```
override(@$name)
```

Attributed method overrides `@$name` method if `@$name` is reachable. `$name` must be defined in one of the super classes or traits of the parent class.

2.3.5 Name mangling

Scala Native toolchain mangles names for all definitions except the ones which have been explicitly exported to C using `extern`. Mangling scheme is defined through a simple grammar that uses a notation inspired by Itanium ABI:

```
<mangled-name> ::=
    _S <defn-name>

<defn-name> ::=
    T <name>                        // top-level name
    M <name> <sig-name>             // member name

<sig-name> ::=
    F <name>                        // field name
    R <type-name>+ E                // constructor name
    D <name> <type-name>+ E         // method name
    P <name> <type-name>+ E         // proxy name
    C <name>                        // c extern name
    G <name>                        // generated name
    K <sig-name> <type-name>+ E     // duplicate name

<type-name> ::=
    v                               // c vararg
    R _                             // c pointer type-name
    R <type-name>+ E                // c function type-name
    S <type-name>+ E                // c anonymous struct type-name
    A <type-name> <number> _        // c array type-name
    <integer-type-name>             // signed integer type-name
    z                               // scala.Boolean
    c                               // scala.Char
    f                               // scala.Float
    d                               // scala.Double
```

(continues on next page)

(continued from previous page)

```
    u                               // scala.Unit
    l                               // scala.Null
    n                               // scala.Nothing
    L <nullable-type-name>          // nullable type-name
    A <type-name> _                 // nonnull array type-name
    X <name>                        // nonnull exact class type-name
    <name>                          // nonnull class type-name

<nullable-type-name> ::=
    A <type-name> _                 // nullable array type-name
    X <name>                        // nullable exact class type-name
    <name>                          // nullable class type-name

<integer-type-name> ::=
    b                               // scala.Byte
    s                               // scala.Short
    i                               // scala.Int
    j                               // scala.Long

<name> ::=
    <length number> <chars>         // raw identifier of given length
```

2.4 Blog

2.4.1 Interflow: Scala Native's upcoming flow-sensitive, profile-guided optimizer

June 16, 2018.

This post provides a sneak peak at Interflow, an upcoming optimizer for Scala Native. For more details, see our publication preprint.

The Interflow Optimizer

Scala Native relies on LLVM as its primary optimizer as of the latest 0.3.7 release. Overall, we've found that LLVM fits this role quite well, after all, it is an industry-standard toolchain for AOT compilation of statically typed programming languages. LLVM produces high-quality native code, and the results are getting better with each release.

However, we have also found that LLVM intermediate representation is sometimes too low-level for the Scala programming language. For example, it does not have direct support for object-oriented features such as classes, allocations, virtual calls on them, instance checks, casts, etc. We encode all of those features by lowering them into equivalent code using C-like abstractions LLVM provides us. As a side effect of this lossy conversion, some of the optimization opportunities are irreversibly lost.

To address the abstraction gap between Scala's high-level features and LLVM's low-level representation, we developed our own interprocedural, flow-sensitive optimizer called Interflow. It operates on the Scala Native's intermediate representation called NIR. Unlike LLVM IR, it preserves full information about object-oriented features.

Interflow fuses following *static* optimizations in a single optimization pass:

- *Flow-sensitive type inference.* Interflow discards most of the original type information ascribed to the methods. Instead, we recompute it using flow-sensitive type inference starting from the entry point of the program. Type inference infers additional `exact` and `nonnull` type qualifiers which are not present in the original program. Those qualifiers aid partial evaluation in the elimination of instance checks and virtual calls.

- *Method duplication.* To propagate inferred type information across method boundaries, Interflow relies on duplication. Methods are duplicated once per unique signature, i.e., a list of inferred parameter types. Method duplication is analogous (although not strictly equivalent) to monomorphization in other languages such as C++ or Rust.
- *Partial evaluation.* As part of its traversal, Interflow partially evaluates instance checks, casts, and virtual calls away and replace them with statically predicted results. Partial evaluation removes computations that can be done at compile time and improves the precision of inferred types due to elimination of impossible control flow paths.
- *Partial escape analysis.* Interflow elides allocations which do not escape. It relies on a variation of a technique called partial escape analysis and scalar replacement. This optimization enables elimination of unnecessary closures, boxes, decorators, builders and other intermediate allocations.
- *Inlining.* Interflow performs inlining in the same pass as the rest of the optimizations. This opens the door for caller sensitive information based on partial evaluation and partial escape analysis to be taken into account to decide if method call should be inlined.

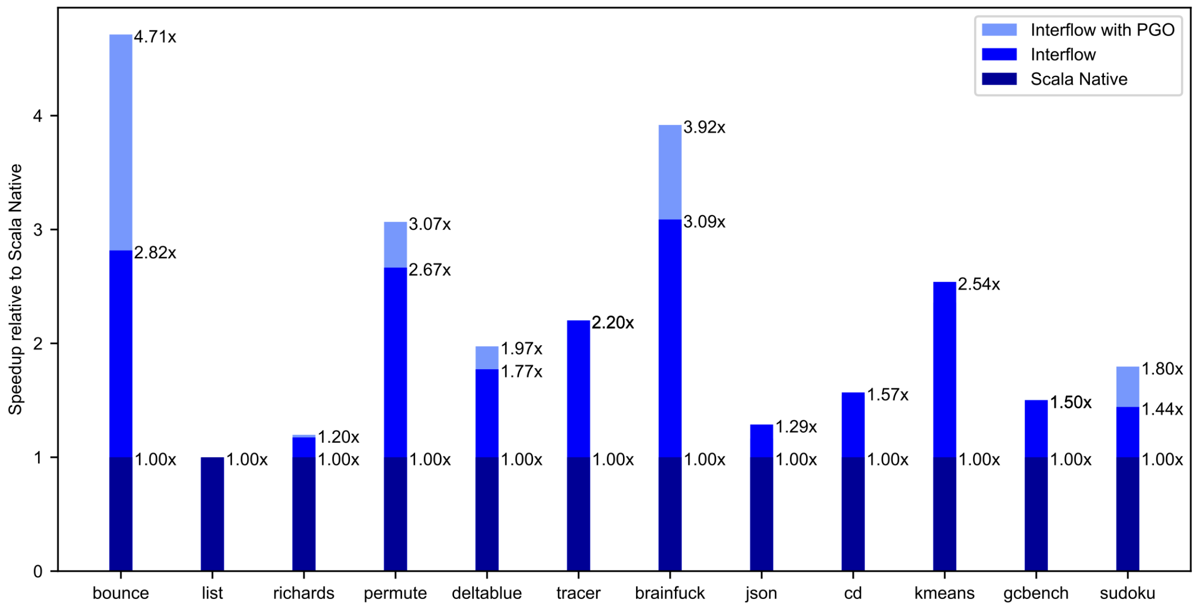
Additionally, we also add support for following *profile-guided optimizations*:

- *Polymorphic inline caching.* Interflow devirtualizes based on flow-sensitive type inference, but it can not predict all of the virtual calls. To aid static devirtualization, we also add support for dynamic devirtualization based on collected type profiles.
- *Untaken branch pruning.* Some of the application code paths (such as error handling) are rarely taken on typical workloads. Untaken branch pruning detects them based on profile data and hoists them out of a method. This optimization reduces code bloat and helps the inliner due to smaller code size left in the method.
- *Profile-directed code placement.* Using the basic block frequency LLVM optimizer can improve native code layout to have the likely branches closer together. It improves generated code quality and can have a significant performance impact on some of the workloads.

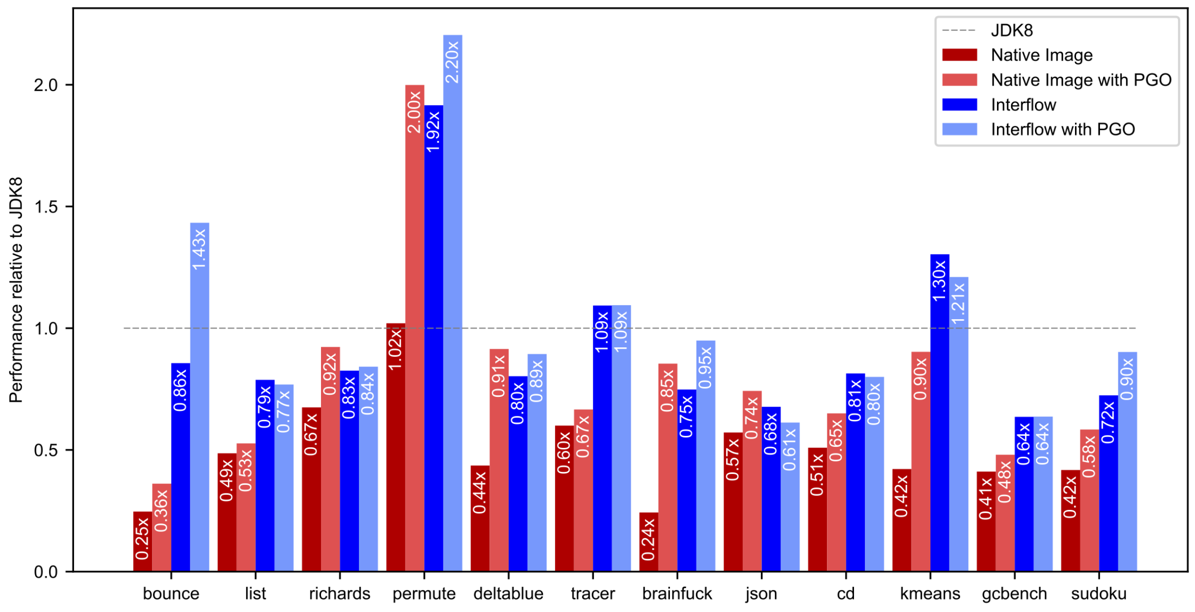
Evaluation

Note: the performance numbers shown here are based on the current development snapshot of the Interflow, they may change *substantially* in the final released version.

We run our current prototype of Interflow on Scala Native benchmarks on a machine equipped with Intel i9 7900X CPU. Interflow achieves up to 3.09x higher throughput (with a geometric mean speedup of 1.8x) than Scala Native 0.3.7. Moreover, with the addition of PGO, Interflow gets up to 4.71x faster (with a geometric mean speedup 1.96x) faster than the Scala Native:



Additionally, we also compare our performance results with Graal Native Image (1.0-RC1 Enterprise Edition) and warmed up HotSpot JDK (1.8.0-1711-b11).



Both Scala Native 0.3.7 (geomean 0.49x) and Native Image 1.0-RC1 (geomean 0.47x) without PGO fail to achieve performance comparable to the a warmed-up JIT compiler. Native Image's implementation of PGO obtains impressive speedups, but it is still behind JDK8 (geomean 0.73x).

On the other hand, Interflow (geomean 0.89x) outperforms Graal Native Image statically. With the addition of PGO, Interflow gets quite close to the throughput of a fully warmed JIT compiler (geomean 0.96x).

Interestingly enough, with Interflow, profile-guided optimizations are not strictly required to get the best performance on 7 out of 12 benchmarks. PGO is just an added extra that can push last 5-10% of the performance envelope.

Conclusion

This post provides a sneak peak at Interflow, an upcoming optimizer for Scala Native. Additionally, we're also going to provide support for profile-guided optimization as an opt-in feature for users who want to obtain absolute best peak performance for Scala Native compiled code. Interflow and PGO are currently in development. Stay tuned for updates on general availability on twitter.com/scala_native.

2.5 Changelog

2.5.1 0.3.8 (Jul 16, 2018)

Read release notes for 0.3.8 on GitHub.

2.5.2 0.3.7 (Mar 29, 2018)

Read release notes for 0.3.7 on GitHub.

2.5.3 0.3.6 (Dec 12, 2017)

Read release notes for 0.3.6 on GitHub.

2.5.4 0.3.5 (Dec 12, 2017)

Read release notes for 0.3.5 on GitHub.

2.5.5 0.3.4 (Dec 12, 2017)

Read release notes for 0.3.4 on GitHub.

2.5.6 0.3.3 (Sep 7, 2017)

Read release notes for 0.3.3 on GitHub.

2.5.7 0.3.2 (Aug 8, 2017)

Read release notes for 0.3.2 on GitHub.

2.5.8 0.3.1 (June 29, 2017)

Read release notes for 0.3.1 on GitHub.

2.5.9 0.3.0 (June 15, 2017)

Read release notes for 0.3.0 on GitHub.

2.5.10 0.2.1 (April 27, 2017)

Read release notes for 0.2.1 on GitHub.

2.5.11 0.2.0 (April 26, 2017)

Read release notes for 0.2.0 on GitHub.

2.5.12 0.1.0 (March 14, 2017)

Read original announcement on scala-lang.org

2.6 FAQ

—

Q: How do I make the resulting executable smaller?

A: Compress the binary with https://upx.github.io/

—

Q: Does Scala Native support WebAssembly?

A: Support for WebAssembly is out of scope for the project. If you need to run Scala code in the browser, consider using Scala.js instead.

2.6.1 Troubleshooting

When compiling your Scala Native project, the linker `ld` may fail with the following message:

```
relocation R_X86_64_32 against `.rodata.str1.1' can not be used when making a shared␣
↪object; recompile with -fPIC
```

It is likely that the `LDFLAGS` environment variable enables hardening. For example, this occurs when the `hardening-wrapper` package is installed on Arch Linux. It can be safely removed.