Scala Native Documentation

Release 0.4.0

Denys Shabalin et al.

Jan 19, 2021

Contents

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 1.1.6 or newer
- LLVM/Clang 6.0 or newer

And following completely optional runtime library dependencies:

- Boehm GC 7.6.0 (optional)
- zlib 1.2.8 or newer (optional)

These 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 the most recent available for your system provided that it works with Scala Native. The Scala Native sbt plugin checks to ensure that *clang* is at least the minimum version shown above.

Scala Native uses the immix garbage collector by default. You can use the Boehm garbage collector instead. If you chose to use that alternate garbage collector both the native library and header files must be provided at build time.

If you use classes from the *java.util.zip* for compression zlib needs to be installed.

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 # optional
```

Note 1: Xcode should work as an alternative if preferred: https://apps.apple.com/us/app/xcode/id497799835

Note 2: A version of zlib that is sufficiently recent comes with the installation of macOS.

Ubuntu

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

Arch Linux

```
$ sudo pacman -S llvm clang build-essential
$ sudo pacman -S gc # optional
```

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

Fedora 33

```
$ sudo dnf install llvm clang
$ sudo dnf groupinstall "Development Tools"
$ sudo dnf install gc-devel zlib-devel # both optional
```

FreeBSD

```
$ pkg install llvm110
$ pkg install boehm-gc # 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" % "0.4.0-M2")
  ```

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

  ```
  sbt.version = 1.4.1
  ```

- `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+, 0.4.0-M1, 0.4.0-M2	2.11.8, 2.11.11, 2.11.12
0.4.0	2.11.12, 2.12.13, 2.13.4

Sbt settings and tasks

The settings now should be set via `nativeConfig` in *sbt*. Setting the options directly is now deprecated.

```
import scala.scalanative.build._

nativeConfig ~= {
  _.withLTO(LTO.thin)
    .withMode(Mode.releaseFast)
    .withGC(GC.commix)
}
```

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`	One of `"debug"`, `"release-fast"` or `"release-full"` (2)
0.2	`nativeGC`	`String`	One of `"none"`, `"boehm"` or `"immix"` (3)
0.3.3	`nativeLinkStubs`	`Boolean`	Whether to link `@stub` definitions, or to ignore them
0.4.0	`nativeConfig`	`NativeConfig`	Configuration of the Scala Native plugin
0.4.0	`nativeLTO`	`String`	One of `"none"`, `"full"` or `"thin"` (4)
0.4.0	`targetTriple`	`String`	The platform LLVM target triple
0.4.0	`nativeCheck`	`Boolean`	Shall the linker check intermediate results for correctness?
0.4.0	`nativeDump`	`Boolean`	Shall the linker dump intermediate results to disk?

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 three 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.** (deprecated since 0.4.0)

   Aliases to **release-full**.

2. **release-fast.** (introduced in 0.4.0)

   Optimize for runtime performance while still trying to keep quick compilation time and small emitted code size. Similar to clang's `-O2` with addition of link-time optimization over the whole application code.

3. **release-full.** (introduced in 0.4.0)

Optimized for best runtime performance, even if hurts compilation time and code size. This modes includes a number of more aggresive optimizations such type-driven method duplication and more aggresive inliner. Similar to clang's `-O3` 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. **commix.** (introduced in 0.4)

   Commix is parallel mark and concurrent sweep garbage collector based on Immix

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

   Conservative generational garbage collector. More information is available at the Github project "ivmai/bdgc" page.

4. **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.

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

   Inlines across Scala/C boundary using legacy FullLTO mode.

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

   Inlines across Scala/C boundary using LLVM's latest ThinLTO mode. Offers both better compilation speed and better runtime performance of the generated code than the legacy FullLTO mode.

Cross compilation using target triple

The target triple can be set to allow cross compilation (introduced in 0.4.0). Use the following approach in *sbt* to set the target triple:

```
nativeConfig ~= { _.withTargetTriple("x86_64-apple-macosx10.14.0") }
```

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.

Including Native Code in your Application or Library

Scala Native uses native C and C++ code to interact with the underlying platform and operating system. Since the tool chain compiles and links the Scala Native system, it can also compile and link C and C++ code included in an application project or a library that supports Scala Native that includes C and/or C++ source code.

Supported file extensions for native code are *.c*, *.cpp*, and *.S*.

Note that *.S* files or assembly code is not portable across different CPU architectures so conditional compilation would be needed to support more than one architecture. You can also include header files with the extensions *.h* and *.hpp*.

Applications with Native Code

In order to create standalone native projects with native code use the following procedure. You can start with the basic Scala Native template.

Add C/C++ code into *src/main/resources/scala-native*. The code can be put in subdirectories as desired inside the *scala-native* directory. As an example, create a file named *myapi.c* and put it into your *scala-native* directory as described above.

```
long long add3(long long in) { return in + 3; }
```

Next, create a main file as follows:

```
import scalanative.unsafe._

@extern
object myapi {
  def add3(in: CLongLong): CLongLong = extern
}

object Main {
  import myapi._
  def main(args: Array[String]): Unit = {
    val res = add3(-3L)
    assert(res == 0L)
    println(s"Add3 to -3 = $res")
  }
}
```

Finally, compile and run this like a normal Scala Native application.

Using libraries with Native Code

Libraries developed to target the Scala Native platform can have C, C++, or assembly files included in the dependency. The code is added to *src/main/resources/scala-native* and is published like a normal Scala library. The code can be put in subdirectories as desired inside the *scala-native* directory. These libraries can also be cross built to support Scala/JVM or Scala.js if the Native portions have replacement code on the respective platforms.

The primary purpose of this feature is to allow libraries to support Scala Native that need native "glue" code to operate. The current C interopt does not allow direct access to macro defined constants and functions or allow passing "struct"s from the stack to C functions. Future versions of Scala Native may relax these restrictions making this feature obsolete.

Note: This feature is not a replacement for developing or distributing native C/C++ libraries and should not be used for this purpose.

If the dependency contains native code, Scala Native will identify the library as a dependency that has native code and will unpack the library. Next, it will compile, link, and optimize any native code along with the Scala Native runtime and your application code. No additional information is needed in the build file other than the normal dependency so it is transparent to the library user.

This feature can be used in combination with the feature above that allows native code in your application.

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.unsafe` 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.unsafe` package. For brevity, we're going to refer to that namespace as just `unsafe`.

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:

```
import scala.scalanative.unsafe._

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

`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`	`unsafe.CBool`
`char`	`unsafe.CChar`
`signed char`	`unsafe.CSignedChar`
`unsigned char`	`unsafe.CUnsignedChar`[1]
`short`	`unsafe.CShort`
`unsigned short`	`unsafe.CUnsignedShort`[1]
`int`	`unsafe.CInt`
`long int`	`unsafe.CLongInt`
`unsigned int`	`unsafe.CUnsignedInt`[1]
`unsigned long int`	`unsafe.CUnsignedLongInt`[1]
`long`	`unsafe.CLong`
`unsigned long`	`unsafe.CUnsignedLong`[1]
`long long`	`unsafe.CLongLong`
`unsigned long long`	`unsafe.CUnsignedLongLong`[1]
`size_t`	`unsafe.CSize`
`ptrdiff_t`	`unsafe.CPtrDiff`[2]
`wchar_t`	`unsafe.CWideChar`
`char16_t`	`unsafe.CChar16`
`char32_t`	`unsafe.CChar32`
`float`	`unsafe.CFloat`
`double`	`unsafe.CDouble`
`void*`	`unsafe.Ptr[Byte]`[2]
`int*`	`unsafe.Ptr[unsafe.CInt]`[2]
`char*`	`unsafe.CString`[2,3]
`int (*)(int)`	`unsafe.CFuncPtr1[unsafe.CInt, unsafe.CInt]`[2,4]
`struct { int x, y; }*`	`unsafe.Ptr[unsafe.CStruct2[unsafe.CInt, unsafe.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 `@link` annotation.

```
import scala.scalanative.unsafe._

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

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

[1] See *Unsigned integer types*.
[2] See *Pointer types*.
[3] See *Byte strings*.
[4] See *Function pointers*.
[5] See *Memory layout types*.

As in C, library names are specified without the `lib` prefix. For example, the library libuv corresponds to `@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.unsafe._

@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

Scala Native supports native interoperability with C's variadic argument list type (i.e. `va_list`), but not `...` varargs. For example `vprintf` can be declared as:

```
import scala.scalanative.unsafe._

@extern
object mystdio {
  def vprintf(format: CString, args: CVarArgList): CInt = extern
}
```

One can wrap a function in a nicer API like:

```
import scala.scalanative.unsafe._

def myprintf(format: CString, args: CVarArg*): CInt =
  Zone { implicit z =>
    mystdio.vprintf(format, toCVarArgList(args.toSeq))
  }
```

And then call it just like a regular Scala function:

```
myprintf(c"2 + 3 = %d, 4 + 5 = %d", 2 + 3, 4 + 5)
```

Pointer types

Scala Native provides a built-in equivalent of C's pointers via `unsafe.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.atN`
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 follows in Scala Native:

```
def test(f: unsafe.CFuncPtr1[CString, Unit]): Unit = unsafe.extern
```

CFuncPtrN types are final classes containing pointer to underlying C function pointer. They automatically handle boxing call arguments and unboxing result. You can create them from C pointer using *CFuncPtr* helper methods:

```
def fnDef(str: CString): CInt = ???

val anyPtr: Ptr[Byte] = CFuncPtr.toPtr {
  CFuncPtr1.fromScalaFunction(fnDef)
}

type StringLengthFn = CFuncPtr1[CString, CInt]
val func: StringLengthFn = CFuncPtr.fromPtr[StringLengthFn](anyPtr)
func(c"hello")
```

It's also possible to create *CFuncPtrN* from Scala *FunctionN*. You can do this by using implicit method conversion method from the corresponding companion object.

```
import scalanative.unsafe.CFuncPtr0
def myFunc(): Unit = println("hi there!")

val myFuncPtr: CFuncPtr0[Unit] = CFuncPtr0.fromScalaFunction(myFunc)
val myImplFn: CFuncPtr0[Unit] = myFunc _
val myLambdaFuncPtr: CFuncPtr0[Unit] = () => println("hello!")
```

On Scala 2.12 or newer, the Scala language automatically converts from closures to SAM types:

```
val myfuncptr: unsafe.CFuncPtr0[Unit] = () => println("hi there!")
```

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:

   ```
   import scala.scalanative.unsafe._

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

   `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 `unsafe.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 `unsafe.stackalloc` function:

   ```
   val buffer = unsafe.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 `unsafe.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.

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

  Here `_N` is an accessor for the field number N.

- `unsafe.Ptr[unsafe.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.DigitN` constructors, where `N` refers to number of digits in the given number. So for example number `1024` is going to be encoded as following:

  ```
  import scalanative.unsafe._, Nat._

  type _1024 = Digit4[_1, _0, _2, _4]
  ```

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

  ```
  val arrptr = unsafe.stackalloc[CArray[Byte, _1024]]
  ```

  You can find an address of n-th array element via `arrptr.at(n)`.

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.unsafe._
import scalanative.libc._

// CString is an alias for Ptr[CChar]
val msg: CString = c"Hello, world!"
stdio.printf(msg)
```

It does not allow any octal values or escape characters not supported by Scala compiler, like `\a` or `\?`, but also unicode escapes. It is possible to use C-style hex values up to value 0xFF, eg. `c"Hello \x61\x62\x63"`

Additionally, we also expose two helper functions `unsafe.fromCString` and `unsafe.toCString` to convert between C-style *CString* (sequence of Bytes, usually interpreted as UTF-8 or ASCII) and Java-style *String* (sequence of 2-byte Chars usually interpreted as UTF-16).

It's worth to remember that `unsafe.toCString` and *c"..."* interpreter cannot be used interchangeably as they handle literals differently. Helper methods `unsafe.fromCString` and ``unsafe.toCString`` are charset aware. They will always assume *String* is UTF-16, and take a *Charset* parameter to know what encoding to assume for the byte string (*CString*) - if not present it is UTF-8.

If passed a null as an argument, they will return a null of the appropriate type instead of throwing a NullPointerException.

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

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

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

Aditionally you can also use `alignmentof` to find alignment of a given type:

```
println(unsafe.alignmentof[Int])                         // 4
println(unsafe.alignmentof[unsafe.CStruct2[Byte, Long]]) // 8
```

Unsigned integer types

Scala Native provides support for four unsigned integer types:

1. `unsigned.UByte`
2. `unsigned.UShort`
3. `unsigned.UInt`
4. `unsigned.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 `byteValue.toUByte`, `shortValue.toUShort`, `intValue.toUInt`, `longValue.toULong` and conversely `unsignedByteValue.toByte`, `unsignedShortValue.toShort`, `unsignedIntValue.toInt`, `unsignedLongValue.toLong`.

Continue to *Libraries*.

2.1.5 Testing

Scala Native comes with JUnit support out of the box. This means that you can write JUnit tests, in the same way you would do for a Java project.

To enable JUnit support, add the following lines to your *build.sbt* file:

```
libraryDependencies += "org.scala-native" %%% "junit-runtime" % "0.4.0"
addCompilerPlugin("org.scala-native" % "junit-plugin" % "0.4.0" cross CrossVersion.
↪full)
```

If you want to get more detailed output from the JUnit runtime, also include the following line:

```
testOptions += Tests.Argument(TestFrameworks.JUnit, "-a", "-s", "-v")
```

Then, add your tests, for example in the *src/test/scala/* directory:

```
import org.junit.Assert._
import org.junit.Test

class MyTest {
  @Test def superComplicatedTest(): Unit = {
    assertTrue("this assertion should pass", true)
  }
}
```

Finally, run the tests in *sbt* by running *test* to run all tests. You may also use *testOnly* to run a particular test, for example:

```
testOnly MyTest
testOnly MyTest.superComplicatedTest
```

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$Subset`
- `java.lang.Character$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.IntegerDecimalScale`
- `java.lang.InternalError`
- `java.lang.InterruptedException`
- `java.lang.Iterable`
- `java.lang.LinkageError`
- `java.lang.Long`
- `java.lang.LongCache`
- `java.lang.Math`
- `java.lang.MathRand`
- `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.Process`
- `java.lang.ProcessBuilder`
- `java.lang.ProcessBuilder$Redirect`
- `java.lang.ProcessBuilder$Redirect$Type`
- `java.lang.Readable`

- `java.lang.ReflectiveOperationException`
- `java.lang.RejectedExecutionException`
- `java.lang.Runnable`
- `java.lang.Runtime`
- `java.lang.Runtime$ProcessBuilderOps`
- `java.lang.RuntimeException`
- `java.lang.SecurityException`
- `java.lang.Short`
- `java.lang.ShortCache`
- `java.lang.StackOverflowError`
- `java.lang.StackTrace`
- `java.lang.StackTraceElement`
- `java.lang.StackTraceElement$Fail`
- `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.annotation.Annotation`
- `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.BigInteger
- 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.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.ByteBuffer
- java.nio.ByteOrder
- java.nio.CharBuffer
- java.nio.DoubleBuffer
- java.nio.FloatBuffer
- java.nio.IntBuffer
- java.nio.InvalidMarkException
- java.nio.LongBuffer
- java.nio.MappedByteBuffer
- java.nio.ReadOnlyBufferException
- java.nio.ShortBuffer
- java.nio.channels.ByteChannel
- java.nio.channels.Channel
- java.nio.channels.Channels

- `java.nio.channels.ClosedChannelException`
- `java.nio.channels.FileChannel`
- `java.nio.channels.FileChannel$MapMode`
- `java.nio.channels.FileLock`
- `java.nio.channels.GatheringByteChannel`
- `java.nio.channels.InterruptibleChannel`
- `java.nio.channels.NonReadableChannelException`
- `java.nio.channels.NonWritableChannelException`
- `java.nio.channels.OverlappingFileLockException`
- `java.nio.channels.ReadableByteChannel`
- `java.nio.channels.ScatteringByteChannel`
- `java.nio.channels.SeekableByteChannel`
- `java.nio.channels.WritableByteChannel`
- `java.nio.channels.spi.AbstractInterruptibleChannel`
- `java.nio.charset.CharacterCodingException`
- `java.nio.charset.Charset`
- `java.nio.charset.CharsetDecoder`
- `java.nio.charset.CharsetEncoder`
- `java.nio.charset.CoderMalfunctionError`
- `java.nio.charset.CoderResult`
- `java.nio.charset.CodingErrorAction`
- `java.nio.charset.IllegalCharsetNameException`
- `java.nio.charset.MalformedInputException`
- `java.nio.charset.StandardCharsets`
- `java.nio.charset.UnmappableCharacterException`
- `java.nio.charset.UnsupportedCharsetException`
- `java.nio.file.AccessDeniedException`
- `java.nio.file.CopyOption`
- `java.nio.file.DirectoryIteratorException`
- `java.nio.file.DirectoryNotEmptyException`
- `java.nio.file.DirectoryStream`
- `java.nio.file.DirectoryStream$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$TerminateTraversalException
- java.nio.file.LinkOption
- java.nio.file.NoSuchFileException
- java.nio.file.NotDirectoryException
- java.nio.file.NotLinkException
- java.nio.file.OpenOption
- java.nio.file.Path
- java.nio.file.PathMatcher
- java.nio.file.Paths
- java.nio.file.RegexPathMatcher
- java.nio.file.SimpleFileVisitor
- java.nio.file.StandardCopyOption
- java.nio.file.StandardOpenOption
- java.nio.file.StandardWatchEventKinds
- java.nio.file.WatchEvent
- java.nio.file.WatchEvent$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.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.util.AbstractCollection
- java.util.AbstractList
- java.util.AbstractListView
- java.util.AbstractMap
- java.util.AbstractMap$SimpleEntry

- `java.util.AbstractMap$SimpleImmutableEntry`
- `java.util.AbstractQueue`
- `java.util.AbstractRandomAccessListIterator`
- `java.util.AbstractSequentialList`
- `java.util.AbstractSet`
- `java.util.ArrayDeque`
- `java.util.ArrayList`
- `java.util.Arrays`
- `java.util.Arrays$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$Wrapper`
- `java.util.Calendar`
- `java.util.Collection`
- `java.util.Collections`
- `java.util.Collections$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$WrappedEquals
- java.util.Collections$WrappedIterator
- java.util.Collections$WrappedList
- java.util.Collections$WrappedListIterator
- java.util.Collections$WrappedMap
- java.util.Collections$WrappedSet
- java.util.Collections$WrappedSortedMap
- java.util.Collections$WrappedSortedSet
- java.util.Comparator
- 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$BigDecimalLayoutForm
- java.util.FormatterClosedException
- java.util.GregorianCalendar
- java.util.HashMap
- java.util.HashSet
- java.util.Hashtable
- java.util.Hashtable$UnboxedEntry$1
- java.util.IdentityHashMap
- 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.List
- java.util.ListIterator
- java.util.Map
- java.util.Map$Entry
- java.util.MissingFormatArgumentException
- java.util.MissingFormatWidthException
- java.util.MissingResourceException
- java.util.NavigableMap
- java.util.NavigableSet
- java.util.NavigableView
- java.util.NoSuchElementException
- java.util.Objects
- java.util.PriorityQueue
- java.util.PriorityQueue$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.SortedMap
- java.util.SortedSet
- java.util.StringTokenizer
- java.util.TimeZone

- java.util.TooManyListenersException
- java.util.TreeSet
- java.util.TreeSet$BoxOrdering
- java.util.UUID
- java.util.UnknownFormatConversionException
- java.util.UnknownFormatFlagsException
- java.util.WeakHashMap
- java.util.concurrent.Callable
- java.util.concurrent.CancellationException
- java.util.concurrent.ConcurrentLinkedQueue
- java.util.concurrent.ExecutionException
- java.util.concurrent.Executor
- java.util.concurrent.RejectedExecutionException
- java.util.concurrent.TimeUnit
- java.util.concurrent.TimeoutException
- java.util.concurrent.atomic.AtomicBoolean
- java.util.concurrent.atomic.AtomicInteger
- java.util.concurrent.atomic.AtomicLong
- java.util.concurrent.atomic.AtomicLongArray
- java.util.concurrent.atomic.AtomicReference
- java.util.concurrent.atomic.AtomicReferenceArray
- java.util.concurrent.locks.AbstractOwnableSynchronizer
- java.util.concurrent.locks.AbstractQueuedSynchronizer
- java.util.function.BiConsumer
- java.util.function.BiFunction
- java.util.function.BiPredicate
- java.util.function.BinaryOperator
- java.util.function.Consumer
- java.util.function.Function
- java.util.function.Predicate
- java.util.function.Supplier
- java.util.function.UnaryOperator
- java.util.jar.Attributes
- java.util.jar.Attributes$Name
- java.util.jar.InitManifest
- java.util.jar.JarEntry

- `java.util.jar.JarFile`
- `java.util.jar.JarInputStream`
- `java.util.jar.JarOutputStream`
- `java.util.jar.Manifest`
- `java.util.regex.MatchResult`
- `java.util.regex.Matcher`
- `java.util.regex.Pattern`
- `java.util.regex.PatternSyntaxException`
- `java.util.stream.BaseStream`
- `java.util.stream.CompositeStream`
- `java.util.stream.EmptyIterator`
- `java.util.stream.Stream`
- `java.util.stream.Stream$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.ZipEntry`
- `java.util.zip.ZipException`
- `java.util.zip.ZipFile`
- `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. RE2 is not a drop-in replacement for *java.util.regex* but handles most common cases well.

Some notes on the implementation:

1. The included RE2 implements a Unicode version lower than the version used in the Scala Native Character class (>= 7.0.0). The RE2 Unicode version is in the 6.n range. For reference, Java 8 released with Unicode 6.2.0.

   The RE2 implemented may not match codepoints added or changed in later Unicode versions. Similarly, there may be slight differences for Unicode codepoints with high numeric value between values used by RE2 and those used by the Character class.

2. This implementation of RE2 does not support:

   - Character classes:
     - Unions: `[a-d[m-p]]`
     - Intersections: `[a-z&&[^aeiou]]`
   - Predefined character classes: `\h`, `\H`, `\v`, `\V`
   - Patterns:
     - Octal: `\0100` - use decimal or hexadecimal instead.
     - Two character Hexadecimal: `\xFF` - use `\x00FF` instead.
     - All alphabetic Unicode: `\uBEEF` - use hex `\xBEEF` instead.
     - Escape: `\e` - use `\u001B` instead.
   - Java character function classes:
     - `\p{javaLowerCase}`
     - `\p{javaUpperCase}`
     - `\p{javaWhitespace}`
     - `\p{javaMirrored}`
   - Boundary matchers: `\G`, `\R`, `\Z`
   - Possessive quantifiers: `X?+`, `X*+`, `X++`, `X{n}+`, `X{n,}+`, `X{n,m}+`
   - Lookaheads: `(?=X)`, `(?!X)`, `(?<=X)`, `(?<!X)`, `(?>X)`
   - Options
     - CANON_EQ
     - COMMENTS
     - LITERAL
     - UNICODE_CASE
     - UNICODE_CHARACTER_CLASS
     - UNIX_LINES
   - Patterns to match a Unicode binary property, such as `\p{isAlphabetic}` for a codepoint with the 'alphabetic' property, are not supported. Often another pattern `\p{isAlpha}` may be used instead, `\p{isAlpha}` in this case.

3. The following Matcher methods have a minimal implementation:

- Matcher.hasAnchoringBounds() - always return true.
- Matcher.hasTransparentBounds() - always throws UnsupportedOperationException because RE2 does not support lookaheads.
- Matcher.hitEnd() - always throws UnsupportedOperationException.
- Matcher.region(int, int)
- Matcher.regionEnd()
- Matcher.regionStart()
- Matcher.requireEnd() - always throws UnsupportedOperationException.
- **Matcher.useAnchoringBounds(boolean) - always throws** UnsupportedOperationException
- Matcher.useTransparentBounds(boolean) - always throws UnsupportedOperationException because RE2 does not support lookaheads.

4. Scala Native 0.3.8 required POSIX patterns to have the form `[[:alpha:]]`. Now the Java standard form `\p{Alpha}` is accepted and the former variant pattern is not. This improves compatibility with Java but, regrettably, may require code changes when upgrading from Scala Native 0.3.8.

Continue to *C Standard Library*.

2.2.2 C Standard Library

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

C Header	Scala Native Module
assert.h	N/A - *indicates binding not available*
complex.h	scala.scalanative.libc.complex
ctype.h	scala.scalanative.libc.ctype
errno.h	scala.scalanative.libc.errno
fenv.h	N/A
float.h	scala.scalanative.libc.float
inttypes.h	N/A
iso646.h	N/A
limits.h	N/A
locale.h	N/A
math.h	scala.scalanative.libc.math
setjmp.h	N/A
signal.h	scala.scalanative.libc.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.libc.stdio
stdlib.h	scala.scalanative.libc.stdlib
stdnoreturn.h	N/A
string.h	scala.scalanative.libc.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 a core subset of the POSIX library:

C Header	Scala Native Module
aio.h	N/A - *indicates binding not available*
arpa/inet.h	scala.scalanative.posix.arpa.inet
assert.h	N/A
complex.h	scala.scalanative.libc.complex
cpio.h	scala.scalanative.posix.cpio
ctype.h	scala.scalanative.libc.ctype
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	scala.scalanative.libc.float
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	scala.scalanative.libc.math
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	scala.scalanative.posix.netinet.tcp
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	scala.scalanative.posix.sys.statvfs
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	scala.scalanative.posix.sys.utsname
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 fulfill 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.

Formatting Scala via *scalafmt* downloads and runs the correct version and uses the *.scalafmt.conf* file at the root of the project. No configuration is needed.

Formatting C and C++ code uses *clang-format* which requires LLVM library dependencies. For *clang-format* we use the same version as the minimum version of LLVM and *clang*. This may not be the version of *clang* used for development as most developers will use a newer version. In order to make this easier we have a environment variable, *CLANG_FORMAT_PATH* which can be set to the older version. Another option is to make sure the correct version of *clang-format* is available in your path. Refer to *Environment setup* for the minimum version to install and use.

The following shows examples for two common operating systems. You may add the environment variable to your shell startup file for convenience:

macOS

```
$ export CLANG_FORMAT_PATH=/usr/local/opt/llvm@6/bin/clang-format
```

Note: Example for *brew*. Other package managers may use different locations.

Ubuntu

```
$ export CLANG_FORMAT_PATH=/usr/lib/llvm-6.0/bin/clang-format
```

The script *./scripts/clangfmt* will use the *.clang-format* file at the root of the project for settings used in formatting.

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.

Common sbt commands

Once you have cloned Scala Native from git, `cd` into the base directory and run `sbt` to launch the sbt build. Inside the sbt shell, the most common commands are the following:

- `sandbox/run` – run the main method of the *sandbox* project
- `tests/test` – run the unit tests
- `tools/test` – run the unit tests of the tools, aka the linker
- `sbtScalaNative/scripted` – run the integration tests of the sbt plugin (this takes a while)
- `clean` – delete all generated sources, compiled artifacts, intermediate products, and generally all build-produced files
- `reload` – reload the build, to take into account changes to the sbt plugin and its transitive dependencies

If you want to run all the tests and benchmarks, which takes a while, you can run the `test-all` command, ideally after `reload` and `clean`.

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:

```
> 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
```

Scripted tests are used when you need to interact with the file system, networking, or the build system that cannot be done with a unit test. They are located in the *scripted-tests* directory.

Run all the scripted tests or just one test using the following examples respectively. To run an individual test substitute the test to run for *native-code-include*:

```
> sbtScalaNative/scripted
> sbtScalaNative/scripted run/native-code-include
```

Some additional tips are as follows.

- If you modify the `nscplugin`, you will need to `clean` the project that you want to rebuild with its new version (typically `sandbox/clean` or `tests/clean`). For a full rebuild, use the global `clean` command.
- If you modify the sbt plugin or any of its transitive dependencies (`sbt-scala-native`, `nir`, `util`, `tools`, `test-runner`), you will need to `reload` for your changes to take effect with most test commands (except with the `scripted` tests).
- For a completely clean build, from scratch, run `reload` *and* `clean`.

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. The *release-fast* mode builds faster, performs less optimizations, but may perform better than *release*.

The *optimize* setting is controlled via the *SCALANATIVE_OPTIMIZE* environment variable. Valid values are *true* and *false*. The default value is *true*. This setting controls whether the Interflow optimizer is enabled or not.

The path to used include and library dirs is controlled via environment variables the *SCALANATIVE_INCLUDE_DIRS* and *SCALANATIVE_LIB_DIRS*.

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*.

Locally publish to test in other builds

If you need to test your copy of Scala Native in the larger context of a separate build, you will need to locally publish all the artifacts of Scala Native.

You can do this with:

```
> publishLocal
```

Afterwards, set the version of *sbt-scala-native* in the target project's *project/plugins.sbt* to the current SNAPSHOT version of Scala Native, and use normally.

Organization of the build

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

1. The compiler plugin, which generates NIR files. It is used in all the Scana Native artifacts in the build, with `.dependsOn(nscplugin % "plugin")`. This is a JVM project.
    - `nscplugin`
2. The Scala Native core libraries. Those are core artifacts which the sbt plugin adds to the `Compile` configuration of all Scala Native projects. The libraries in this group are themselves Scala Native projects. Projects further in the list depend on projects before them.
    - `nativelib`
    - `clib`
    - `posixlib`
    - `javalib`
    - `auxlib`
    - `scalalib`
3. The Scala Native sbt plugin and its dependencies (directory names are in parentheses). These are JVM projects.
    - `sbtScalaNative (sbt-scala-native)`
    - `tools`
    - `nir`, `util`
    - `nirparser`
    - `testRunner (test-runner)`

4. The Scala Native test interface and its dependencies. The sbt plugin adds them to the `Test` configuration of all Scala Native projects. These are Scala Native projects.
   - `testInterface (test-interface)`
   - `testInterfaceSbtDefs (test-interface-sbt-defs)`
5. Tests and benchmarks (no dependencies on each other).
   - `tests (unit-tests)` (Scala Native project)
   - `tools` This has tests within the project (JVM project)
   - `(scripted-tests)` (JVM project)

Apart from those mentioned sub-projects it is possible to notice project-like directory `testInterfaceCommon (test-interface-common)`. Its content is shared as unmanaged source dependency between JVM and Native sides of test interface.

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.

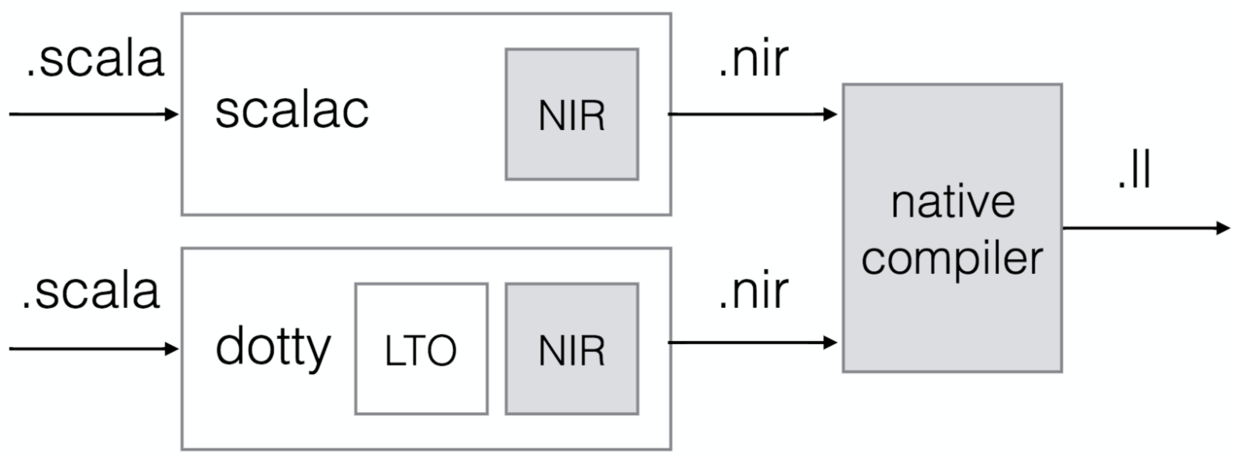


Fig. 1: High-level overview of the compilation process.

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
```

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 compiler plugin you'll need to clean the sandbox (or other Scala Native projects) if you want it to be recompiled with the newer version of the compiler plugin. This can be achieved with:

```
sbt> 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

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 =
```

(continues on next page)

(continued from previous page)

```
    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> <scope>                    // field name
    R <type-name>+ E                    // constructor name
    D <name> <type-name>+ E <scope>     // 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

<scope> ::=
    P <defn-name>                  // private to defn-name
    O                              // public

<name> ::=
    <length number> [-] <chars>    // raw identifier of given length; `-` separator␣
↪is only used when <chars> starts with digit or `-` itself
```

Mangling identifiers containing special characters follows Scala JVM conventions. Each double-quote " character is always converted to *$u0022*

2.3.6 IntelliJ IDEA

- Select "Create project from existing sources" and choose the `build.sbt` file. When prompted, select "Open as project". Make sure you select the "Use sbt shell" for both import and build.
- When the import is complete, we need to fix some module dependencies:
  - `scalalib`: Right-click on the module, "Mark directory as" -> "Excluded". This is needed because `scalalib` is only meant to be used at runtime (it is the Scala library that the executables link against). Not excluding it makes IDEA think that the Scala library comes from it, which results into highlighting errors.
  - `nscplugin`: We need to add what SBT calls `unmanagedSourceDirectories` as dependencies. Go go Project Structure -> Modules -> `nscplugin` -> Dependencies and click the + icon. Select "JARs or Directories" and navigate to the `nir` directory at the root of the Scala Native project. Repeat for the `util` directory.
  - `native-build`: We need to add the `sbt-scala-native` module as a dependency. Go go Project Structure -> Modules -> `native-build` -> Dependencies and click the + icon. Select "Module Dependency" and select the `sbt-scala-native` module.

The above is not an exhaustive list, but it is the bare minimum to have the build working. Please keep in mind that you will have to repeat the above steps, in case you reload (re-import) the SBT build. This will need to happen if you change some SBT-related file (e.g. `build.sbt`).

2.3.7 Metals

Metals import should work out of the box for most of the modules.

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.

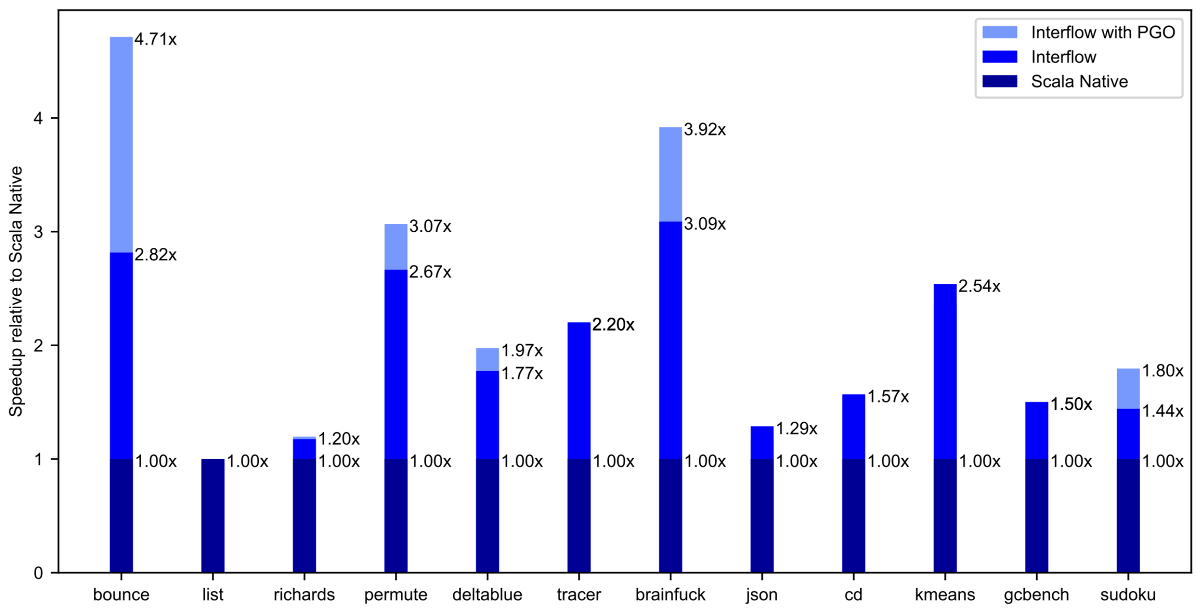
Additionally, we also add support for following *profile-guided optimizations*:

- *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.

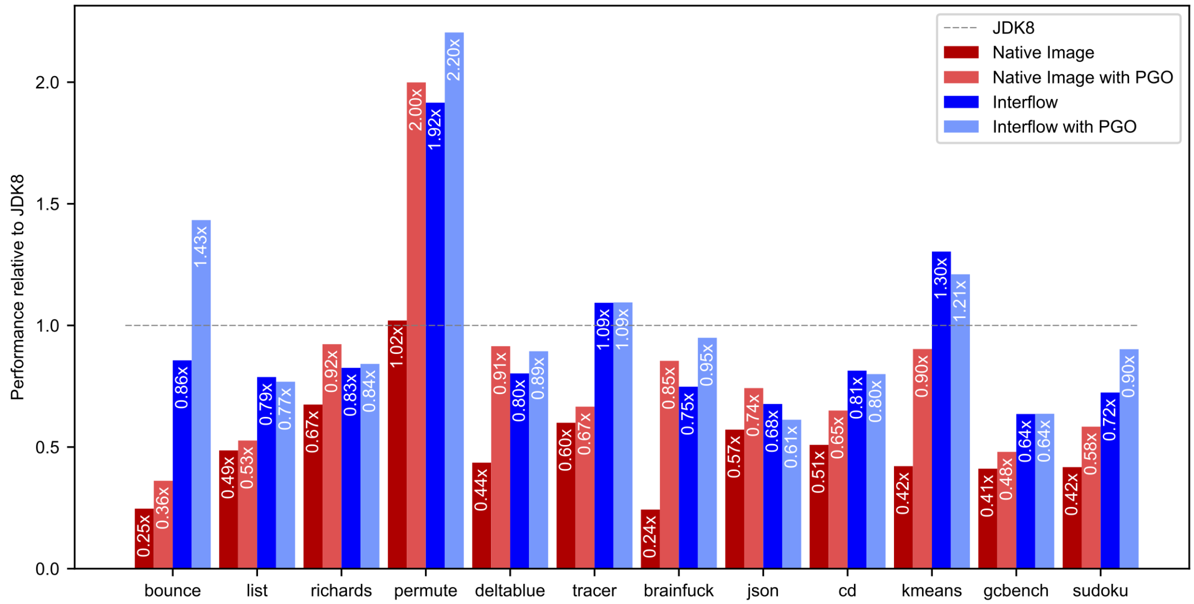
Evaluation

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 0.4.0 (Jan 19, 2021)

We are happy to announce the release of Scala Native 0.4.0!

Scala Native is an optimizing ahead-of-time compiler and lightweight managed runtime designed specifically for Scala. It is developed at the Scala Center and with the help from VirtusLab along with contributors from the community.

Check out the documentation at https://scala-native.readthedocs.io/

TL;DR

- **Not backward compatible with previous releases**,
- A unique `nativeConfig` setting replaces the set of `nativeX` settings,

- The partial implementation of the JDK packages `java.time` and `java.text` were removed from core repo. Third-party libraries such as scala-java-time and scala-java-locales should be used instead,
- `CFuncPtr` is now created by implicit conversion from ordinary `scala.Function`
- Added Scala 2.12 and 2.13 support,
- Added support for JUnit,
- Additional C/C++ can be added to compilation pipeline,
- Allowed for cross compilation using custom target triple
- Allowed reflective instantiation by using `@EnableReflectiveInstantiation` annotation,
- Added new Garbage Collector - Concurrent Mark and Parallel Sweep Garbage Collector, called Commix,
- Various bug fixes

Breaking changes

Broken backward compatibility

Scala Native 0.4.0 breaks backward binary compatibility with previous releases of Scala Native. Libraries published using version 0.4.0-M2 or older must be republished for Scala Native 0.4.x.

Removal of `java.time` / `java.text`

This release removes the partial implementations of the `java.time` and `java.text` packages from Scala Native core. This will allow third-party libraries, like `scala-java-time` and `scala-java-locales`, to provide more complete versions thereof.

Using methods that directly or transitively need the removed classes will require an additional dependency on the appropriate third-party library. For example:

```
val str: String = "Hello Native"
str.toLowerCase()              // works as before
str.toLowerCase(Locale.French) // requires scala-java-locales to link
```

`NativeConfig` replaces setting keys

The `nativeXyz` setting keys are now deprecated in favor of a single `nativeConfig` setting, which can be used as follows:

```
// build.sbt
nativeConfig ~= {
  _.withMode(build.Mode.releaseFast)
   .withGC(build.GC.immix)
   .withLTO(build.LTO.full)
   .withOptimize(true)
   .withCompileOptions(Nil)
   .withLinkingOptions(Nil)
}
```

Old style settings keys are still supported, but they have lower priority than the new config and will be removed at some point in the future. In the following example resulting `LTO` setting would be set to `thin`

```
nativeConfig := nativeConfig.value.withLTO(build.LTO.thin)

nativeLTO := "none"
```

CFuncPtr changes

You no longer need to implement the `CFuncPtrN` trait which is now private for Scala Native implementation. Instead, you can use an implicit conversion method taking arbitrary `scala.FunctionN` and returning `CFuncPtrN`.

```
type Callback = CFuncPtr1[CInt,Unit]
def registerCallback(cFn: Callback): Unit = extern
def fn(n: CInt): Unit = ???

registerCallback(CFuncPtr1.fromScalaFunction(fn))
registerCallback(fn)

registerCallback { (n: CInt) => println("hello native") }
```

It's now also possible to work with an arbitrary pointer and convert it to `CFuncPtrN` that can be called in your Scala code or to convert your function to any pointer if your native library needs this.

```
import scala.scalanative.unsafe.Ptr
val cFnPtr: CFuncPtr0[CInt]     = ???
val fnPtr: Ptr[Byte]            = CFuncPtr.toPtr(cFnPtr)
val fnFromPtr: CFuncPtr0[CInt] = CFuncPtr.fromPtr[CFuncPtr0[CInt]](fnPtr)
```

Other breaking changes:

- Sbt 0.13.x is no longer supported - upgrade to 1.1.6 or newer.
- The minimal version of Clang working with Scala Native is now 6.0
- `CSize` is now unsigned numeral type
- Usage of signed numeral types for methods expecting `CSize` was deprecated.

New features

Supported Scala versions

We added support for Scala 2.12.13 and 2.13.4, in addition to the existing support for 2.11.12.

JUnit Support

Scala Native now comes with JUnit support out of the box, this means that you can write tests in the same way you would do for a Scala/JVM or Scala.js project. To enable JUnit tests all you will need to do is to add the two following lines to your `build.sbt`.

```
addCompilerPlugin("org.scala-native" % "junit-plugin" % nativeVersion cross␣
↪CrossVersion.full)
libraryDependencies += "org.scala-native" %%% "junit-runtime" % nativeVersion % "test"
```

Reflective instantiation

Since this release you are able to reflectively instantiate definitions marked with the `@EnableReflectiveInstantation` annotation, as well as its descendants. Annotated classes and modules, having a concrete implementation, can be accessed via the provided `scalanative.reflect.Reflect` API. If you have used Scala.js before, it may seem similar to you, as the new implementation uses exactly the same API.

Scala Native does not support full reflection support, although this feature might fix most of the issues that could occur in users code.

```scala
package x.y.z

@EnableReflectiveInstantation
trait ReflectiveFoo {
  val value: String = "foo"
}

object SingleFoo extends ReflectiveFoo

case class MultipleFoo(times: Int) extends ReflectiveFoo {
  override val value: String = super.value * times
}

for {
  cls  <- lookupInstantiatableClass("x.y.z.MultipleFoo")
  ctor <- cls.getConstructor(classOf[Int])
  obj  <- ctor.newInstance(5)
} yield obj // results in Some(new MultipleFoo(5))

for {
  cls <- lookupLoadableModule("x.y.z.SingleFoo")
  obj <- cls.loadModule()
} yield obj // results Some(SingleFoo)
```

Cross compilation

It is now possible to define a custom target for the compiler by providing an LLVM-style TargetTriple in your config. The default behavior is still to target the host architecture and operating system.

For example, if you're working on Linux and would like to create an executable suitable for MacOS without changing your whole build, you can use the following sbt setting::

```
sbt 'set nativeConfig ~= {_.withTargetTriple("x86_64-apple-darwin<version>")}' myApp/
↪nativeLink
```

We consider changing target triple as a feature for advanced users, and cannot promise it would currently work with any possible configuration yet. However, the number of supported architectures and operating systems would definitely grow in the future.

> When using Linux / MacOS, you can check the target triple used in your environment with the command `llvm-config --host-target`.

Native sources in the build

With the 0.4.0 release you're able to put your C/C++ sources in the `resources/scala-native` directory inside your project, so they will be linked and compiled inside the SN pipeline.

As an example you can use it to access macro-defined constants and functions or to pass `structs` from the stack to C functions.

```
// src/resources/scala-native/example.c
typedef int (*Callback0) (void);

const int EXAMPLE_CONSTANT = 42;

int exec(Callback0 f) {
 return f();
};
```

```
// src/main/example.scala
@extern
object example {
 def exec(cb: CFuncPtr0[CInt]): ExecResult = extern

 @name("EXAMPLE_CONSTANT")
 final val someConstant: Int = extern
}
```

Commix GC

This release also adds a new Garbage Collector - Commix, a parallel mark, and concurrent sweep GC, based on the well known Immix GC. It reduces GC pause times by utilizing additional processor cores during mark and sweep phases.

While the GC itself will use multiple threads, Scala Native still does not support multi-threading in the application code. Commix GC was written in C and uses `pthread` to work. In case your application needs concurrency support, you may try the experimental library scala-native-loop

Bugfixes

- Failures during the build of multiple parallel projects using common jar were fixed,
- Lowered overall memory usage when compiling and linking,
- Value classes are now correctly handled in lambda functions,
- The `synchronized` flag in now taken into account when generating methods,
- Constructors are no longer treated are virtual methods, they're always resolved statically,
- Generic `CFuncPtr` can be passed as method arguments,
- Binary operations with `Nothing` arguments will no longer break compilation,
- Resolving of public method no longer can result in private method with the same name,
- Instances of `java.lang.Class` are now cached and can be correctly tested using reference equality,
- Triple-quoted `CString`'s are now correctly escaped,
- Identifiers starting with digits are now correctly handled,

- Fixed errors with too many open files after consecutive runs,
- Fixed crashes when HOME env variable was not set,
- Boehm GC installed using MacPorts is now supported,
- Fixed segmentation fault when trying to access current, unlinked directory,
- `malloc` will now throw `OutOfMemoryError` when it cannot allocate memory,
- `toCString` & `fromCString` now correctly return null,
- Fixed errors with not cleared `errno` when using POSIX `readdir`
- Array operation now throw JVM-compilant `ArrayIndexOutOfBoundsException`,
- Fix bug with `BufferedInputStream.read()` for values bigger then 0x7f,
- `Files.walk` accepts non-directory files,
- Improved IEEE754 specification compliance when parsing strings,
- Fixed infinite loop in `java.io.RandomAccesFile.readLine`,
- Added multiple missing `javalib` classes and methods

Contributors

Big thanks to everybody who contributed to this release or reported an issue!

```
$ git shortlog -sn --no-merges v0.4.0-M2..v0.4.0
    64        LeeTibbert
    58        Wojciech Mazur
    37        Eric K Richardson
    13        Kirill A. Korinsky
    10        Ergys Dona
     8        Lorenzo Gabriele
     4        Sébastien Doeraene
     3        Valdis Adamsons
     2        Denys Shabalin
     2        Ondra Pelech
     2        kerr
     1        Danny Lee
     1        Nadav Samet
     1        Richard Whaling
     1        jokade
```

Full Changelog

The most impacting merged pull requests:

Compiler

- Fix #1928: show file name for NIR version mismatch during linking #1929 (jokade)
- Fix #2084 Allow identifiers containing double-quote characters #2085 (WojciechMazur)
- Fix #2035: Guard virtual lookup of non virtual methods #2051 (WojciechMazur)
- Fix #415: Report usage positions of missing definitions when linking #2069 (WojciechMazur)

- Fix #899: Allow binary operations with Nothing arguments #2065 (WojciechMazur)
- Fix #1435: Cache instances of j.l.Class #1894 (WojciechMazur)
- Fix #1950 Enable handling value classes when generating lambda #1952 (WojciechMazur)
- Fix #2012: Fix not reachable definitions of default methods #2040 (WojciechMazur)
- Fix #1972: Implement JavaDefaultMethods on Scala 2.11 #1997 (LeeTibbert)
- Fix Build crashes in releaseFull mode #1980 (WojciechMazur)
- Store source code positions in NIR #1878 (WojciechMazur)
- Add Scala 2.13.x support #1916 (WojciechMazur)
- Add Scala 2.12 support #1877 (errikos)
- Fix #1669: Put private methods in a separate scope through mangling. #1898 (WojciechMazur)
- Update NIR version to 5.8 #1912 (WojciechMazur)
- Fix #1627: Allow passing generic functions ptr as method args #1901 (WojciechMazur)
- Fix #1091 Take the `synchronized` flag of methods into account #1988 (WojciechMazur)
- Fix #1909, #1843: Statically resolve constructors, not as virtual methods #1957 (WojciechMazur)
- Lower memory usage in CodeGen #1979 (WojciechMazur)
- Fix #1943: Support JUnit's @Ignore on test class #1961 (WojciechMazur)
- Fix #1944: Compile error for non-public methods with JUnit annotations #1958 (WojciechMazur)
- Fix #1652: Allow declaration of external functions with varying signatures in separate objects #1746 (lolgab)
- Fix #1496: Encode Strings in NIR as char code units instead of UTF-8 #1883 (WojciechMazur)
- Fix #1801: Store the already processed byte string in Val.Chars #1855 (WojciechMazur)
- Fix #1256: Add JUnit support #1841 (errikos)
- Fix #1279: Enable reflective instantiation via static initializers #1728 (errikos)
- Fix #1770: Fix _*scala*== with ScalaNumbers. #1805 (LeeTibbert)

Sbt plugin

- Fix #1849: Streamline clang version detection #2099 (ekrich)
- Remove target triple discovery code #2033 (ekrich)
- Fix #2024: Use a shared Scope in the sbt plugin for all parallel tasks #2039 (WojciechMazur)
- Fix #1999: Clear errno before readdir in posixlib dirent.c #2000 (LeeTibbert)
- Fix #1711: Ignore non-jar non-directory elements on the classpath #1987 (ekrich)
- Fix #1970: Restrict native code to a specified subdirectory #1876 (ekrich)
- Fix #1597: Introduce nativeConfig instead of multiple nativeX keys #1864 (WojciechMazur)
- Import the testing infrastructure of Scala.js. #1869 (WojciechMazur)
- Discover and use clang 11 if present, and drop clang < 5.0. #1874 (LeeTibbert)
- Fix #657: Give libraries a way to include native C code to be compiled. #1637 (ekrich)
- Fix "too many open files" after consecutive runs #1839 (errikos)

- Drop support for sbt 0.13.x. #1712 (ekrich)
- Support boehm installed from macports #2071 (catap)
- Never use default path that doesn't exists #2091 (catap)
- Fix crash when HOME env variable is not set #1738 (lolgab)

Native library

- Fix 2059: Remove non-standard fcntl.close() & use proper unistd.close(). #1633 (LeeTibbert)
- Fix #519: Make CSize an unsigned type #1949 (WojciechMazur)
- Never use a memory after it's freed #2072 (catap)
- Throw an OutOfMemoryError if malloc cannot allocate #2073 (catap)
- Fix #1631: CFuncPtr <-> Ptr[Byte] conversion #1845 (WojciechMazur)
- Move native code into posixlib and clib respectively #1885 (ekrich)
- Fix #1796: Nativelib toCString() & fromCString() now return nulls. #1945 (LeeTibbert)
- Fix #1613: Restore the two argument fcntl.open() method. #1614 (LeeTibbert)
- Fix #1768: Handle segfault when current directory was unlinked #1842 (WojciechMazur)
- Commix: named semaphores #1658 (valdisxp1)
- Commix: parallel mark and concurrent sweep GC #1423 (valdisxp1)
- Avoid defining NDEBUG if it's already defined #1791 (lolgab)
- Fix warning with musl libc due to wrong include #1745 (lolgab)
- Fix #1606: Add printf vararg helpers #1636 (rwhaling)

Java standard library

- Partial fix #1023: Port j.u.NavigableMap #1893 (LeeTibbert)
- Support `java.util.Date` methods using `java.time.Instant`. #2088 (WojciechMazur)
- Remove dummy java.time implementation from core repo. #2087 (WojciechMazur)
- String to{Lower, Upper}Case handles Unicode special, non-locale dependent, casing rules #2098 (WojciechMazur)
- Port localized String.to{Lower, Upper}Case from Scala.js #2095 (WojciechMazur)
- Port optional Locale `j.u.Formatter` from Scala.js #2079 (WojciechMazur)
- Implement j.u.Map default methods. #2061 (LeeTibbert)
- Fix #2049: Use natural ordering for Arrays#sort with null comparator #2050 (LeeTibbert)
- Fix #1993: Port ju.ConcurrentLinkedQueue from Scala.js #1994 (lolgab)
- Fix #2044: Throw JVM-compliant ArrayIndexOutOfBoundsException for array ops #2047 (WojciechMazur)
- Work around limitation for JDK12+ about j.l.constant.Constable #1941 (catap)
- Port j.u.Objects#requireNonNull with Supplier argument #1975 (LeeTibbert)
- Port Scala.js j.u.Objects parameter widening & later changes. #1953 (LeeTibbert)

- Add j.l.Iterable.forEach #1934 (LeeTibbert)
- Add the default methods of j.u.Iterator default methods #1937 (LeeTibbert)
- Fix BufferedInputStream.read() for values bigger than 0x7f #1922 (catap)
- Update java.util.Properties to match Scala.js changes #1892 (ekrich)
- Provide more useful j.l.Thread#getStackTrace stub #1899 (LeeTibbert)
- Fix #1780: Fix an ambiguous overload about java.nio.FileSystems.newFileSystem on JDK 13+. #1873 (WojciechMazur)
- Fix #1871: Fix a corner case of defaults in ju.Properties.{stringP,p}ropertyNames. #1872 (ekrich)
- Fix #1064: Implement java.util.Properties.{load,store}. #1653 (ekrich)
- Fix #1758: Accept a non-directory file in Files.walk() #1838 (WojciechMazur)
- Fix #1559: Improve spec compliance when parsing IEEE754 strings. #1703 (LeeTibbert)
- Port/implement j.u.ArrayDeque #1696 (LeeTibbert)
- Fix #1693: j.u.AbstractCollection#toString output now matches Scala JVM #1697 (LeeTibbert)
- Fix #1683: Implement suppression and non-writable trace for Throwable #1688 (LeeTibbert)
- Fix #1623: Fix an infinite loop in j.i.RandomAccessFile#readLine #2100 (LeeTibbert)
- Fix scala-js#4088: Avoid an Int overflow in BigDecimal.toString(). #1837 (LeeTibbert)
- Update uppercase lowercase to use UnicodeData.txt vs CaseFolding.txt #1611 (ekrich)

2.5.2 0.4.0-M2 (May 23, 2019)

Read release notes for 0.4.0-M2 on GitHub.

2.5.3 0.4.0-M1 (May 23, 2019)

Read release notes for 0.4.0-M1 on GitHub.

2.5.4 0.3.9 (Apr 23, 2019)

Read release notes for 0.3.9 on GitHub.

2.5.5 0.3.8 (Jul 16, 2018)

Read release notes for 0.3.8 on GitHub.

2.5.6 0.3.7 (Mar 29, 2018)

Read release notes for 0.3.7 on GitHub.

2.5.7 0.3.6 (Dec 12, 2017)

Read release notes for 0.3.6 on GitHub.

2.5.8 0.3.5 (Dec 12, 2017)

Read release notes for 0.3.5 on GitHub.

2.5.9 0.3.4 (Dec 12, 2017)

Read release notes for 0.3.4 on GitHub.

2.5.10 0.3.3 (Sep 7, 2017)

Read release notes for 0.3.3 on GitHub.

2.5.11 0.3.2 (Aug 8, 2017)

Read release notes for 0.3.2 on GitHub.

2.5.12 0.3.1 (June 29, 2017)

Read release notes for 0.3.1 on GitHub.

2.5.13 0.3.0 (June 15, 2017)

Read release notes for 0.3.0 on GitHub.

2.5.14 0.2.1 (April 27, 2017)

Read release notes for 0.2.1 on GitHub.

2.5.15 0.2.0 (April 26, 2017)

Read release notes for 0.2.0 on GitHub.

2.5.16 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.