
Scala Native Documentation

Release 0.4.14

Denys Shabalin et al.

Jun 06, 2023

Contents

1	Community	3
2	Documentation	5
2.1	User's Guide	5
2.2	Libraries	25
2.3	Contributor's Guide	44
2.4	Blog	70
2.5	Changelog	73
2.6	FAQ	112

Version 0.4.14

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](#).

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 1.5.8 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

macOS, Linux, and Windows

Please refer to [this link](#) for instructions for your operating system.

FreeBSD

```
$ pkg install sbt
```

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 libstdc++-12-dev
$ sudo apt install libgc-dev # optional
```

Arch Linux

```
$ sudo pacman -S llvm clang
$ 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 12.2 and later

```
$ pkg install llvm10
$ 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/main/scripts/scala-
→native.nix
$ nix-shell scala-native.nix -A clangEnv
```

Windows

Corporate environments and Windows policies can affect the method used to setup your environment. The following procedure involves downloading installers and running the installers using Powershell (Administrative) to avoid some of these issues. If you have full access to your machine then you can install using your favorite method. [Chocolatey](#) or [Scoop](#) can be substituted as needed or desired and are also mentioned above in the installing *sbt* documentation.

1. Download and install Visual Studio Community 2019

<https://visualstudio.microsoft.com/>

You may install it via the command line if needed.

```
> .\vs_community__<version>.exe
```

Select the *Workloads* tab and then *Desktop development with C++* checkbox. The defaults are fine. The *C++ Clang tools for Windows* does not work so use the next step for details on installing LLVM.

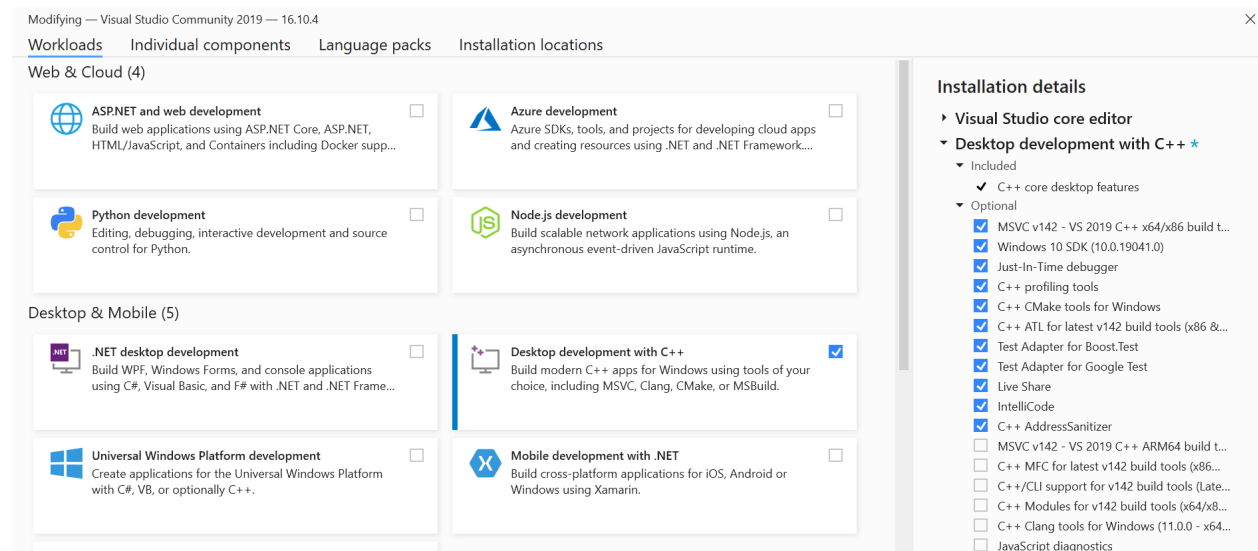


Fig. 1: Visual Studio install dialog showing options.

2. Download and install LLVM

<https://github.com/llvm/llvm-project/releases/tag/llvmorg-12.0.1>

Select *LLVM-12.0.1-win64.exe* or newer. Digital signatures are provided.

You may also install LLVM via the command line, and if needed, install it into your *C:\Users\<login>\AppData\Local* directory. The installer will add *LLVM* and the associated directories and files.

```
> .\LLVM-12.0.1-win64.exe
```

3. Add the binary location to your PATH

Using the install path above, you would add the following:

```
C:\Users\<login>\AppData\Local\LLVM\bin
```

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. In an empty working directory, execute:

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

Note: On Windows, new project should not be created in *mounted* directories, like external storage devices.

In the case of WSL2 (Windows Subsystem Linux), Windows file system drives like *C* or *D* are perceived as *mounted*. So creating new projects in these locations will not work.

In the WSL2 environment, it is recommended to create projects in the user files path, e.g `/home/<USER>/sn-projects`.

This will:

- start sbt.
- prompt for a project name
- use the `.g8 template`. to generate a basic project with that name.
- create a project sub-directory with the project name.
- copy the contents at these template links to the corresponding location in this new project sub-directory.
 - `project/plugins.sbt` adds the Scala Native plugin dependency and its version.
 - `project/build.properties` specifies the sbt version.
 - `build.sbt` enables the plugin and specifies the Scala version.
 - `src/main/scala/Main.scala` is a minimal application.

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

To use the new project:

- Change the current working directory to the new project directory.
 - For example, on linux with a project named `AnswerToProjectNamePrompt`, type `cd AnswerToProjectNamePrompt`.
- Type `sbt run`.

This will get everything compiled and should have the expected output!

Please refer to the [FAQ](#) if you encounter any problems.

The generated project is a starting point. After the first run, you should review the software versions in the generated files and, possibly, update or customize them. [Scaladex](#) is a useful resource for software versions.

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
0.4.1	2.11.12, 2.12.13, 2.13.4, 2.13.5
0.4.2	2.11.12, 2.12.{13..15}, 2.13.{4..8}
0.4.3-RC1, 0.4.3-RC2	2.11.12, 2.12.{13..15}, 2.13.{4..8}, 3.1.0
0.4.3	2.11.12, 2.12.{13..15}, 2.13.{4..8}, 3.1.{0..1}

Sbt settings and tasks

Use `nativeConfig` in `sbt` to provide settings. This is often done in a project's `build.sbt`.

Scala Native starts execution with a `NativeConfig` object, called `nativeConfig`, filled with default values:

```
show ThisBuild / nativeConfig
```

Each `withX()` method creates a new `NativeConfig` with the indicated `X` value set. All other settings are taken from the `Config` object being accessed.

To show `nativeConfig` values active in current scope at any point in time:

```
sbt> show nativeConfig
```

To set a new value and replace any previous setting:

```
import scala.scalanative.build._

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

To append a value to the right of any previous setting:

```
import scala.scalanative.build._

// Enable verbose reporting during compilation
nativeConfig ~= { c =>
  c.withCompileOptions(c.compileOptions ++ Seq("-v"))
}

// Use an alternate linker
nativeConfig ~= { c =>
  c.withLinkingOptions(c.linkingOptions ++ Seq("-fuse-ld=mold"))
}

/* The keen observer will note that "-fuse-ld=mold" could also have been
 * set using "withCompileOptions".
 */
```

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", "release-size" or "release-full" (2)
0.2	nativeGC	String	One of "none", " Boehm", "immix" or "commix" (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**.

3. **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.

4. **release-size.** (introduced in 0.4.10)

Optimize for reduced output size while still trying to keep quick compilation time and relatively fast runtime performance. Similar to clang's `-Oz` with addition of link-time optimization over the whole application code.

5. **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 aggressive optimizations such type-driven method duplication and more aggressive 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") }
```

you may create a few dedicated projects with different target triples. If you have multiple project definitions for different macOS architectures, eg:

```
lazy val sandbox64 = project.in(file("sandbox"))
  .settings(nativeConfig ~= { _.withTargetTriple("arm64-apple-darwin20.6.0") })

lazy val sandboxM1 = project.in(file("sandbox"))
  .settings(nativeConfig ~= { _.withTargetTriple("x86_64-apple-darwin20.6.0") })
```

These project definitions allow to produce different binaries - one dedicated for the *x86_64* platform and another one for *arm64*. You may easily combine them to one so called fat binary or universal binary via *lipo*:

```
lipo -create sandbox64/target/scala-2.12/sandbox64-out sandboxM1/target/scala-2.12/
↳ sandboxM1-out -output sandbox-out
```

which produces *sandbox-out* that can be used at any platform.

You may use *FatELF* <https://icculus.org/fatelf/> to build fat binaries for Linux.

Build target

Setting build target allows you to specify to what type of object your project should be linked to. As an example, to link it as dynamic library use the following command:

```
nativeConfig ~= { _.withBuildTarget (BuildTarget.libraryDynamic) }
```

1. **application** (default)

Results in creating ready to use executable program.

2. **libraryDynamic**

Results in dynamic library being built based on entry point methods annotated with *@exported*, for details see [Native code interoperability](#).

3. **libraryStatic**

Results in building static library using the same semantics as in the `libraryDynamic`. Exported methods should handle exceptions, as they might not be able to be caught in the program that is using a produced static library.

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
<code>void</code>	<code>Unit</code>
<code>bool</code>	<code>unsafe.CBool</code>
<code>char</code>	<code>unsafe.CChar</code>
<code>signed char</code>	<code>unsafe.CSignedChar</code>
<code>unsigned char</code>	<code>unsafe.CUnsignedChar</code> ¹
<code>short</code>	<code>unsafe.CShort</code>
<code>unsigned short</code>	<code>unsafe.CUnsignedShort</code> ¹
<code>int</code>	<code>unsafe.CInt</code>
<code>long int</code>	<code>unsafe.CLongInt</code>
<code>unsigned int</code>	<code>unsafe.CUnsignedInt</code> ¹
<code>unsigned long int</code>	<code>unsafe.CUnsignedLongInt</code> ¹
<code>long</code>	<code>unsafe.CLong</code>
<code>unsigned long</code>	<code>unsafe.CUnsignedLong</code> ¹
<code>long long</code>	<code>unsafe.CLongLong</code>
<code>unsigned long long</code>	<code>unsafe.CUnsignedLongLong</code> ¹
<code>size_t</code>	<code>unsafe.CSize</code>
<code>ssize_t</code>	<code>unsafe.CSSize</code>
<code>ptrdiff_t</code>	<code>unsafe.CPtrDiff</code> ²
<code>wchar_t</code>	<code>unsafe.CWideChar</code>
<code>char16_t</code>	<code>unsafe.CChar16</code>
<code>char32_t</code>	<code>unsafe.CChar32</code>
<code>float</code>	<code>unsafe.CFloat</code>
<code>double</code>	<code>unsafe.CDouble</code>
<code>void*</code>	<code>unsafe.Ptr[Byte]</code> ²
<code>int*</code>	<code>unsafe.Ptr[unsafe.CInt]</code> ²
<code>char*</code>	<code>unsafe.CString</code> ²³
<code>int (*)(int)</code>	<code>unsafe.CFuncPtr1[unsafe.CInt, unsafe.CInt]</code> ²⁴
<code>struct { int x, y; }*</code>	<code>unsafe.Ptr[unsafe.CStruct2[unsafe.CInt, unsafe.CInt]]</code> ²⁵
<code>struct { int x, y; }</code>	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.

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 =
```

(continues on next page)

¹ See *Unsigned integer types*.

² See *Pointer types*.

³ See *Byte strings*.

⁴ See *Function pointers*.

⁵ See *Memory layout types*.

(continued from previous page)

```
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)
```

Exported methods

When linking Scala Native as library, you can mark functions that should be visible in created library with `@exported(name: String)` annotation. In case if you omit or use null as the argument for name external function name match the name of the method. Currently, only static object methods can be exported. To export accessors of field or variable in static object use `@exportAccessors(getterName: String, setterName: String)`. If you omit the explicit names in the annotation constructor, Scala Native would create exported methods with `set_` and `get_` prefixes and name of field.

`int ScalaNativeInit(void)`; function is special exported function that needs to be called before invoking any code defined in Scala Native. It returns 0 on successful initialization and non-zero value in the otherwise. For dynamic libraries a constructor would be generated to invoke `ScalaNativeInit` function automatically upon loading library or startup of the program. If for some reason you need to disable automatic initialization of Scala Native upon loading dynamic library and invoke it manually in user code set `SCALANATIVE_NO_DYLIB_CTOR` environment variable. You can also disable generation of library constructors by defining `-DSCALANATIVE_NO_DYLIB_CTOR` in `NativeConfig::compileOptions` of your build.

```
import scala.scalanative.unsafe._

object myLib{
  @exportAccessors("mylib_current_count", "mylib_set_counter")
  var counter: Int = 0

  @exportAccessors("error_message")
  val ErrorMessage: CString = c"Something bad just happend!"

  @exported
  def addLongs(l: Long, r: Long): Long = l + r

  @exported("mylib_addInts")
  def addInts(l: Int, r: Int): Int = l + r
}
```

```
# libmylib.h
int ScalaNativeInit(void);
long addLongs(long, long);
int addInts(int, int);
int mylib_current_count();
void mylib_set_counter(int);

# test.c
#include "libmylib.h"
#include <assert.h>

int main(int argc, char** argv){
  # This function needs to be called before invoking any methods defined in Scala_
  ↪Native.
```

(continues on next page)

(continued from previous page)

```

# Might be called automatically unless SCALANATIVE_NO_DYLIB_CTOR env variable is_
↪set.
assert(ScalaNativeInit() == 0);
addLongs(0L, 4L);
mylib_addInts(4, 0);
printf("Current count %d\n", mylib_current_count());
mylib_set_counter(42);
...
}

```

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	<code>*ptr</code>	<code>!ptr</code>
Store value	<code>*ptr = value</code>	<code>!ptr = value</code>
Pointer to index	<code>ptr + i, &ptr[i]</code>	<code>ptr + i</code>
Elements between	<code>ptr1 - ptr2</code>	<code>ptr1 - ptr2</code>
Load at index	<code>ptr[i]</code>	<code>ptr(i)</code>
Store at index	<code>ptr[i] = value</code>	<code>ptr(i) = value</code>
Pointer to field	<code>&ptr->name</code>	<code>ptr.atN</code>
Load a field	<code>ptr->name</code>	<code>ptr._N</code>
Store a field	<code>ptr->name = value</code>	<code>ptr._N = value</code>

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 **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
```

Additionally, you can also use `alignmentof` to find the 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:

```
enablePlugins(ScalaNativeJUnitPlugin)
```

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
```

2.1.6 Profiling

In this section you can find some tips on how to profile your Scala Native binary in Linux.

Measuring execution time and memory

- With the `time` command you can measure execution time:

```
$ time ./target/scala-2.13/scala-native-out
real  0m0,718s
user  0m0,419s
sys   0m0,299s
```

- With the `/usr/bin/time --verbose` command you can also see memory consumption:

Creating Flamegraphs

A [flamegraph](#) is a visualization of the most frequent code-paths of a program. You can use flamegraphs to see where your program spends most of its CPU time. Follow these steps:

- You need to install the `perf` command if you haven't got it already:

```
$ sudo apt update && sudo apt install linux-tools-generic
```

- Then clone the flamegraph repository into e.g. `~/git/hub/`

```
$ cd ~ && mkdir -p git/hub && cd git/hub/
$ git clone git@github.com:brendangregg/FlameGraph.git
```

- Then navigate to your Scala Native project and, after building your binary, you can create a flamegraph like so:

```
$ sudo perf record -F 1000 -a -g ./target/scala-2.13/scala-native-out
$ sudo perf script > out.perf
$ ~/git/hub/FlameGraph/stackcollapse-perf.pl out.perf > out.folded
$ ~/git/hub/FlameGraph/flamegraph.pl out.folded > kernel.svg
```

- Open the file `kernel.svg` in your browser and you can zoom in the interactive SVG-file by clicking on the colored boxes as explained [here](#). A box represents a stack frame. The broader a box is the more CPU cycles have been spent. The higher the box is, the deeper in the call-chain it is.
- The `perf` option `-F 1000` means that the sampling frequency is set to 1000 Hz. You can experiment with changing this option to get the right accuracy; start with e.g. `-F 99` and see what you get. You can then increase the sampling frequency to see if more details adds interesting information.

2.1.7 Runtime Settings

Scala Native comes with some ability to change the runtime characteristics.

Garbage Collector (GC) Settings

Scala Native supports the [Boehm-Demers-Weiser Garbage Collector](#). The environment variables defined in Boehm are planned to be shared by all the Garbage Collectors supported in Scala Native so they are consistent. The variables supported are listed below for each GC.

All Garbage Collectors

The following environment variables will be used for all GCs. They can go from 1 MB to the system memory maximum or up to about 512 GB. The size is in bytes, kilobytes(k or K), megabytes(m or M), or gigabytes(g or G). Examples: 1024k, 1M, or 1G etc.

- GC_INITIAL_HEAP_SIZE changes the minimum heap size.
- GC_MAXIMUM_MAX_HEAP_SIZE changes the maximum heap size.

The plan is to add more GC settings in the future using the Boehm setting names where applicable.

Boehm GC

The Boehm GC uses the two variables shown above. The following document shows all the variables available for Boehm: [README](#).

None GC

The None GC uses the two variables shown above.

Immix GC

The following variables have not been changed to match the standard variables in Immix yet.

- SCALANATIVE_MIN_HEAP_SIZE changes the minimum heap size.
- SCALANATIVE_MAX_HEAP_SIZE changes the maximum heap size.

Commix GC

In addition to the variables described above for Immix, Commix also adds a few more variables which do not match the Boehm settings yet.

- SCALANATIVE_GC_THREADS (default is processor count - 1)
- SCALANATIVE_TIME_RATIO (default is .05)
- SCALANATIVE_FREE_RATIO (default is .5)

Examples

If you are developing in the Scala Native sandbox, the following are examples showing some error conditions using Immix, the default GC. Adjust the path to your executable as needed:

```
$ export SCALANATIVE_MIN_SIZE=64k; export SCALANATIVE_MAX_SIZE=512k; sandbox/.2.13/  
↳target/scala-2.13/sandbox-out  
SCALANATIVE_MAX_HEAP_SIZE too small to initialize heap.  
Minimum required: 1m  
  
$ export SCALANATIVE_MIN_SIZE=2m; export SCALANATIVE_MAX_SIZE=1m; sandbox/.2.13/  
↳target/scala-2.13/sandbox-out  
SCALANATIVE_MAX_HEAP_SIZE should be at least SCALANATIVE_MIN_HEAP_SIZE
```

2.2 Libraries

2.2.1 Java Standard Library

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

Supported classes

The classes currently available are:

`java.io`

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

- `InputStream`
- `InputStreamReader`
- `InterruptedException`
- `LineNumberReader`
- `NotSerializableException`
- `ObjectStreamException`
- `OutputStream`
- `OutputStreamWriter`
- `PrintStream`
- `PrintWriter`
- `PushbackInputStream`
- `PushbackReader`
- `RandomAccessFile`
- `Reader`
- `Serializable`
- `StringReader`
- `StringWriter`
- `SyncFailedException`
- `UTFDataFormatException`
- `UncheckedIOException`
- `UnsupportedEncodingException`
- `Writer`

java.lang

- `AbstractMethodError`
- `AbstractStringBuilder`
- `Appendable`
- `ArithmeticException`
- `ArrayIndexOutOfBoundsException`
- `ArrayStoreException`
- `AssertionError`
- `AutoCloseable`
- `Boolean`
- `BootstrapMethodError`
- `Byte`
- `ByteCache`

- CharSequence
- Character
- Character.Subset
- Character.UnicodeBlock
- CharacterCache
- ClassCastException
- ClassCircularityError
- ClassFormatError
- ClassLoader
- ClassNotFoundException
- CloneNotSupportedException
- Cloneable
- Comparable
- Double
- Enum
- EnumConstantNotPresentException
- Error
- Exception
- ExceptionInInitializerError
- Float
- IllegalAccessException
- IllegalAccessException
- IllegalArgumentException
- IllegalMonitorStateException
- IllegalStateException
- IllegalThreadStateException
- IncompatibleClassChangeError
- IndexOutOfBoundsException
- InheritableThreadLocal
- InstantiationError
- InstantiationException
- Integer
- IntegerCache
- IntegerDecimalScale
- InternalError
- InterruptedException

- Iterable
- LinkageError
- Long
- LongCache
- Math
- MathRand
- NegativeArraySizeException
- NoClassDefFoundError
- NoSuchFieldError
- NoSuchFieldException
- NoSuchMethodError
- NoSuchMethodException
- NullPointerException
- Number
- NumberFormatException
- OutOfMemoryError
- Process
- ProcessBuilder
- ProcessBuilder.Redirect
- ProcessBuilder.Redirect.Type
- Readable
- ReflectiveOperationException
- RejectedExecutionException
- Runnable
- Runtime
- RuntimeException
- SecurityException
- Short
- StackOverflowError
- StackTrace
- StackTraceElement
- String
- StringBuffer
- StringBuilder
- StringIndexOutOfBoundsException
- System

- Thread
- Thread.UncaughtExceptionHandler
- ThreadDeath
- ThreadLocal
- Throwable
- TypeNotPresentException
- UnknownError
- UnsatisfiedLinkError
- UnsupportedClassVersionError
- UnsupportedOperationException
- VerifyError
- VirtualMachineError
- Void
- annotation.Annotation
- annotation.Retention
- annotation.RetentionPolicy
- constant.Constable
- constant.ConstantDesc
- ref.PhantomReference
- ref.Reference
- ref.ReferenceQueue
- ref.SoftReference
- ref.WeakReference
- reflect.AccessibleObject
- reflect.Array
- reflect.Constructor
- reflect.Executable
- reflect.Field
- reflect.InvocationTargetException
- reflect.Method
- reflect.UndeclaredThrowableException

java.math

- BigDecimal
- BigInteger
- BitLevel

- Conversion
- Division
- Elementary
- Logical
- MathContext
- Multiplication
- Primality
- RoundingMode

java.net

- BindException
- ConnectException
- Inet4Address
- Inet6Address
- InetAddress
- InetAddressBase
- InetSocketAddress
- MalformedURLException
- NoRouteToHostException
- PortUnreachableException
- ServerSocket
- Socket
- SocketAddress
- SocketException
- SocketImpl
- SocketInputStream
- SocketOption
- SocketOptions
- SocketOutputStream
- SocketTimeoutException
- URI
- URLEncoderDecoder
- URISyntaxException
- URL
- URLClassLoader
- URLConnection

- `URLDecoder`
- `URLEncoder`
- `UnknownHostException`
- `UnknownServiceException`

java.nio

- `Buffer`
- `BufferOverflowException`
- `BufferUnderflowException`
- `ByteBuffer`
- `ByteOrder`
- `CharBuffer`
- `DoubleBuffer`
- `FloatBuffer`
- `IntBuffer`
- `InvalidMarkException`
- `LongBuffer`
- `MappedByteBuffer`
- `ReadOnlyBufferException`
- `ShortBuffer`
- `channels.ByteChannel`
- `channels.Channel`
- `channels.Channels`
- `channels.ClosedChannelException`
- `channels.FileChannel`
- `channels.FileChannel.MapMode`
- `channels.FileLock`
- `channels.GatheringByteChannel`
- `channels.InterruptibleChannel`
- `channels.NonReadableChannelException`
- `channels.NonWritableChannelException`
- `channels.OverlappingFileLockException`
- `channels.ReadableByteChannel`
- `channels.ScatteringByteChannel`
- `channels.SeekableByteChannel`
- `channels.WritableByteChannel`

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

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

java.rmi

- Remote
- RemoteException

java.security

- AccessControlException
- CodeSigner
- DummyMessageDigest
- GeneralSecurityException
- MessageDigest
- MessageDigestSpi
- NoSuchAlgorithmException
- Principal
- Timestamp
- TimestampConstructorHelper
- cert.CertPath
- cert.Certificate
- cert.CertificateEncodingException
- cert.CertificateException
- cert.CertificateFactory
- cert.X509Certificate
- cert.X509Extension

java.util

- AbstractCollection
- AbstractList
- AbstractListView
- AbstractMap
- AbstractMap.SimpleEntry
- AbstractMap.SimpleImmutableEntry
- AbstractQueue
- AbstractRandomAccessListIterator
- AbstractSequentialList
- AbstractSet
- ArrayDeque

- `ArrayList`
- `Arrays`
- `BackedUpListIterator`
- `Base64`
- `Base64.Decoder`
- `Base64.Encoder`
- `BitSet`
- `Calendar`
- `Collection`
- `Collections`
- `Comparator`
- `ConcurrentModificationException`
- `Date`
- `Deque`
- `Dictionary`
- `DoubleSummaryStatistics`
- `DuplicateFormatFlagsException`
- `EmptyStackException`
- `EnumSet`
- `Enumeration`
- `FormatFlagsConversionMismatchException`
- `Formattable`
- `FormattableFlags`
- `Formatter`
- `Formatter.BigDecimalLayoutForm`
- `FormatterClosedException`
- `GregorianCalendar`
- `HashMap`
- `HashSet`
- `Hashtable`
- `IdentityHashMap`
- `IllegalFormatCodePointException`
- `IllegalFormatConversionException`
- `IllegalFormatException`
- `IllegalFormatFlagsException`
- `IllegalFormatPrecisionException`

- `IllegalFormatWidthException`
- `IllformedLocaleException`
- `InputMismatchException`
- `InvalidPropertiesFormatException`
- `IntSummaryStatistics`
- `Iterator`
- `LinkedHashMap`
- `LinkedHashSet`
- `LinkedList`
- `List`
- `ListIterator`
- `LongSummaryStatistics`
- `MissingFormatArgumentException`
- `MissingFormatWidthException`
- `MissingResourceException`
- `Map`
- `NavigableMap`
- `NavigableSet`
- `NoSuchElementException`
- `Objects`
- `Optional`
- `OptionalDouble`
- `OptionalInt`
- `OptionalLong`
- `PrimitiveIterator`
- `PrimitiveIterator.OfDouble`
- `PrimitiveIterator.OfInt`
- `PrimitiveIterator.OfLong`
- `PriorityQueue`
- `Properties`
- `Queue`
- `Random`
- `RandomAccess`
- `RandomAccessListIterator`
- `ServiceConfigurationError`
- `Set`

- `SizeChangeEvent`
- `SortedMap`
- `SortedSet`
- `Splititerator`
- `Splititerators`
- `StringTokenizer`
- `TooManyListenersException`
- `TreeSet`
- `UUID`
- `UnknownFormatConversionException`
- `UnknownFormatFlagsException`
- `WeakHashMap`

java.util.concurrent

- `Callable`
- `CancellationException`
- `ConcurrentHashMap`
- `ConcurrentHashMap.KeySetView`
- `ConcurrentLinkedQueue`
- `ConcurrentMap`
- `ConcurrentSkipListSet`
- `ExecutionException`
- `Executor`
- `RejectedExecutionException`
- `Semaphore`
- `ThreadFactory`
- `ThreadLocalRandom`
- `TimeUnit`
- `TimeoutException`
- `atomic.AtomicBoolean`
- `atomic.AtomicInteger`
- `atomic.AtomicLong`
- `atomic.AtomicLongArray`
- `atomic.AtomicReference`
- `atomic.AtomicReferenceArray`
- `atomic.LongAdder`

- `locks.AbstractOwnableSynchronizer`
- `locks.AbstractQueuedSynchronizer`
- `locks.Lock`
- `locks.ReentrantLock`

java.util.function

- `BiConsumer`
- `BiFunction`
- `BiPredicate`
- `BinaryOperator`
- `Consumer`
- `Function`
- `IntUnaryOperator`
- `Predicate`
- `Supplier`
- `UnaryOperator`

java.util.jar

- `Attributes`
- `Attributes.Name`
- `InitManifest`
- `JarEntry`
- `JarFile`
- `JarInputStream`
- `JarOutputStream`
- `Manifest`

java.util.regex

- `MatchResult`
- `Matcher`
- `Pattern`
- `PatternSyntaxException`

java.util.stream

- BaseStream
- Collector
- Collector.Characteristics
- Collectors
- DoubleStream
- DoubleStream.Builder
- DoubleStream.DoubleMapMultiConsumer
- Stream
- Stream.Builder
- StreamSupport

java.util.zip

- Adler32
- CRC32
- CheckedInputStream
- CheckedOutputStream
- Checksum
- DataFormatException
- Deflater
- DeflaterOutputStream
- GZIPInputStream
- GZIPOutputStream
- Inflater
- InflaterInputStream
- ZipConstants
- ZipEntry
- ZipException
- ZipFile
- ZipInputStream
- 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 ($\geq 7.0.0$). 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.

- The reference Java 8 regex package does not support certain commonly used Perl expressions supported by this implementation of RE2. For example, for named capture groups Java uses the expression “(?<foo>)” while Perl uses the expression “(?P<foo>)”.

Scala Native `java.util.regex` methods accept both forms. This extension is intended to be useful but is not strictly Java 8 compliant. Not all RE2 Perl expressions may be exposed in this way.

- 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.
- 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.

Embedding Resources

In Scala Native, resources are implemented via embedding a resource in a resulting binary file. Only `getClass().getResourceAsStream()` is implemented. For that to work, you have to specify an additional `NativeConfig` option:

```
nativeConfig ~= {
  _.withEmbedResources(true)
}
```

This will include the resource files found on the classpath in the resulting binary file. Please note that files with following extensions cannot be embedded and used as a resource:

```
".class", ".tasty", ".nir", ".scala", ".java", ".jar"
```

This is to avoid unnecessarily embedding source files. If necessary, please consider using a different file extension for embedding. Files found in the `resources/scala-native` directory will not be embedded as well. It is recommended to add the “.c” and “.h” files there.

Reasoning for the lack of `getResource()` and `getResources()`:

In Scala Native, the outputted file that can be run is a binary, unlike JVM’s classfiles and jars. For that reason, if `getResources()` URI methods would be implemented, a new URI format using a separate `FileSystem` would have to be added (e.g. instead of obtaining `jar:file:path.ext` you would obtain `embedded:path.ext`). As this still would provide a meaningful inconsistency between JVM’s `java.lang` API and Scala Native’s reimplementation, this remains not implemented for now. The added `getClass().getResourceAsStream()` however is able to be consistent between the platforms.

Continue to *ISO/IEC C Standard Library*.

2.2.2 ISO/IEC C Standard Library

Scala Native provides bindings for a core subset of the International Organization for Standardization/International Electrotechnical Commission (ISO/IEC) C standard library:

C Header	Scala Native Module
assert.h	N/A - <i>indicates binding not available</i>
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	scala.scalanative.libc.locale
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	scala.scalanative.libc.stddef
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. See indicated source module for limitations, if any, and usage:

C Header	Scala Native Module
aio.h	N/A - <i>indicates binding not available</i>
arpa/inet.h	scala.scalanative.posix.arpa.inet ¹
assert.h	N/A
complex.h	scala.scalanative.libc.complex
cpio.h	scala.scalanative.posix.cpio
ctype.h	scala.scalanative.libc.ctype

Continued on next page

Table 1 – continued from previous page

C Header	Scala Native Module
dirent.h	scala.scalanative.posix.dirent
dlfcn.h	scala.scalanative.posix.dlfcn
errno.h	scala.scalanative.posix.errno
fcntl.h	scala.scalanative.posix.fcntl
fenv.h	N/A
float.h	scala.scalanative.libc.float
fmtmsg.h	N/A
fnmatch.h	scala.scalanative.posix.fnmatch
ftw.h	N/A
getopt.h	scala.scalanative.posix.getopt
glob.h	scala.scalanative.posix.glob
grp.h	scala.scalanative.posix.grp
iconv.h	N/A
inttypes.h	scala.scalanative.posix.inttypes
iso646.h	N/A
langinfo.h	scala.scalanative.posix.langinfo
libgen.h	scala.scalanative.posix.libgen
limits.h	scala.scalanative.posix.limits
locale.h	N/A
math.h	scala.scalanative.libc.math
monetary.h	scala.scalanative.posix.monetary ³
mqueue.h	N/A
ndbm.h	N/A
net/if.h	scala.scalanative.posix.net.if
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	scala.scalanative.posix.nl_types
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	scala.scalanative.posix.signal
spawn.h	scala.scalanative.posix.spawn
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	scala.scalanative.posix.string
strings.h	scala.scalanative.posix.strings
stropts.h	N/A
sys/ipc.h	N/A
sys/mman.h	scala.scalanative.posix.sys.mman

Continued on next page

Table 1 – continued from previous page

C Header	Scala Native Module
sys/msg.h	N/A
sys/resource.h	scala.scalanative.posix.sys.resource
sys/select.h	scala.scalanative.posix.sys.select
sys/sem.h	N/A
sys/shm.h	N/A
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	scala.scalanative.posix.sys.times
sys/types.h	scala.scalanative.posix.sys.types
sys/uio.h	scala.scalanative.posix.sys.uio
sys/un.h	scala.scalanative.posix.sys.un
sys/utsname.h	scala.scalanative.posix.sys.utsname
sys/wait.h	scala.scalanative.posix.sys.wait
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	scala.scalanative.posix.wordexp

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

¹ The argument to `inet_ntoa()` differs from the POSIX specification because Scala Native supports only passing structures by reference. See code for details and usage.

³ See file for limit on number of variable arguments.

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 any version greater than `10` as most developers use a newer version of LLVM and `clang`. In order to make this easier we have a environment variable, `CLANG_FORMAT_PATH` which can be set to a compatible 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 of `clang` supported.

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

macOS

Normal macOS tools does not include clang-format so installing via `brew` is a good option.

```
% brew install clang-format
% export CLANG_FORMAT_PATH=/opt/homebrew/bin/clang-format
```

Note: `brew` for M1 installs at the above location which is in the `PATH` so the `export` is not needed and is for reference only. Other package managers may use different locations.

Ubuntu 20.04

```
$ sudo apt install clang-format-10
$ export CLANG_FORMAT_PATH=/usr/lib/llvm-10/bin/clang-format
```

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

C / POSIX Libraries

Both the `clib` and `posixlib` have coding styles that are unique compared to normal Scala coding style. Normal C code is written in lowercase snake case for function names and uppercase snake case for macro or pre-processor constants. Here is an example for Scala:

```
@extern
object cpio {
  @name("scalanative_c_issock")
  def C_ISSOCK: CUnsignedShort = extern

  @name("scalanative_c_islnk")
  def C_ISLNK: CUnsignedShort = extern
}
```

The following is the corresponding C file:

```
#include <cpio.h>

unsigned short scalanative_c_issock() { return C_ISSOCK; }
unsigned short scalanative_c_islnk() { return C_ISLNK; }
```

Since C has a flat namespace most libraries have prefixes and in general cannot use the same symbol names so there is no need to add additional prefixes. For Scala Native we use *scalanative_* as a prefix for functions.

This is the reason C++ added namespaces so that library designer could have a bit more freedom. The developer, however, still has to de-conflict duplicate symbols by using the defined namespaces.

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 main 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 main 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 main 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 main* (PRs coming from main 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 `javali` 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 platforms¹ and using different garbage collectors². 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 `release-size` mode optimizes for reduced size.

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`.

```
$ 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.

¹ <http://www.scala-native.org/en/latest/user/setup.html>

² <http://www.scala-native.org/en/latest/user/sbt.html>

Use the custom sbt command to publish all projects for a specific Scala version (x,y,z):

```
> publish-local-dev x.y.z
```

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 Scala 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`
 - `javali`
 - `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`
 - `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)`
 - `testInterfaceSbtDeps (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)
6. External tests and its dependencies. Sources of these tests are not stored in this project, but fetched from external sources, e.g.: Scala compiler repository. Sources in this project define interface used by Scala Native and tests filters.
 - `scalaPartest (scala-partest)` (JVM project, uses Scala Native artifacts)
 - `scalaPartestRuntime (scala-partest-runtime)` (Scala native project)
 - `scalaPartestTests (scala-partest-tests)` (JVM project)

- `scalaPartestJUnitTests` (`scala-partest-junit-tests`) (Scala Native project)

7. JUnit plugin, its tests and dependencies. Following sources define JUnit compiler for Scala Native and its runtime, as well as compliance tests and internal stubs.

- `junitPlugin` (`junit-plugin`)
- `junitRuntime` (`junit-runtime`)
- `junitTestOutputsJVM` (`junit-test/output-jvm`)
- `junitTestOutputsNative` (`junit-test/output-native`)
- `junitAsyncJVM` (`junit-async/jvm`)
- `junitAsyncNative` (`junit-async/native`)

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.

Working with scalalib overrides

Scalalib project does not introduce any new classes but provides overrides for the existing Scala standard library. Some of these overrides exist to improve the performance of Scala Native, eg. by explicit inlining of some methods. When running `scalalib/compile` it will automatically use existing `*.scala` files defined in `overrides` directories. To reduce the number of changes between overrides and original Scala sources, we have introduced a patching mechanism. Each file defined with the name `*.scala.patch` contains generated patch, which would be applied onto source defined for the current Scala version inside its standard library. In case `overrides*` directory contains both `*.scala` file and its corresponding patch file, only `*.scala` file would be added to the compilation sources.

To operate with patches it is recommended to use ScalaCLI script `scripts/scalalib-patch-tool.sc`. It takes 2 mandatory arguments: command to use and Scala version. There are currently 3 supported commands defined: `* recreate` - creates `*.scala` files based on original sources with applied patches corresponding to their name; `* create` - creates `*.scala.patch` files from defined `*.scala` files in `overrides` directory with corresponding name; `* prune` - deletes all `*.scala` files which does not have corresponding `*.scala.patch` file;

(e.g. `scala-cli scripts/scalalib-patch-tool.sc - recreate 2.13.10`)

Each of these commands is applied to all files defined in the `overrides` directory. By default override directory is selected based on the used scala version, if it's not the present script will try to use directory with corresponding Scala binary version, or it would try to use Scala epoch version or `overrides` directory. If none of these directories exists it will fail. It is also possible to define explicitly `overrides` directory to use by passing it as the third argument to the script.

Locally publish docs

Follow the steps after cloning the `scalnative` repo and changing to `scala-native` directory.

1. First time building the docs. This command will setup & build the docs.

```
$ bash scripts/makedocs setup
```

2. If setup is already done. This command will only build the docs assuming setup is already done.

```
$ bash scripts/makedocs
```

3. Navigate to `docs/_build/html` directory and open `index.html` file in your browser.

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.

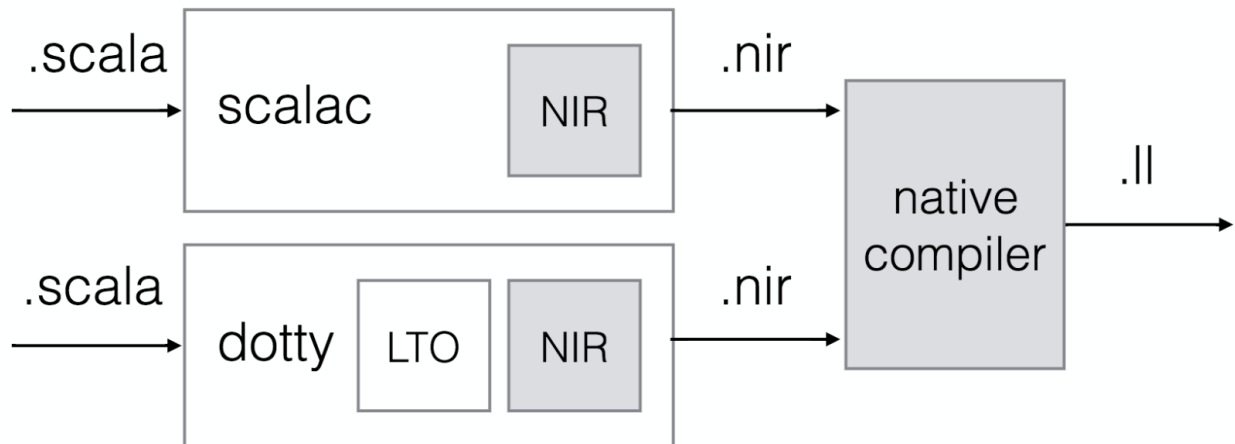


Fig. 2: High-level overview of the compilation process.

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.

The following directions are using the Scala 3 project. To use other Scala versions first find the project name and then use that instead of “sandbox3”, say “sandbox2_13”:

```
sbt> sandbox<TAB>
```

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

```
sbt> sandbox3/clean; sandbox3/nativeLink
```

Compiling will generate a number of files. The `.ll` LLVM assembly language files can be inspected to see what was passed to the LLVM step.

You can inspect the `.ll` files in the directories below `sandbox/.3/target/`. Because Scala Native is under active development, the directory layout, names of files and their specific content may change.

All definitions are generated into 1 or $N=(\text{number of CPUs}) * .ll$ files. A Linux example on system with 4 CPUs:

```
$ # on command line, with project root as current working directory.
$ find sandbox/.3/target -name "*.ll"
sandbox/.3/target/scala-3.1.3/native/3.ll
sandbox/.3/target/scala-3.1.3/native/2.ll
sandbox/.3/target/scala-3.1.3/native/1.ll
sandbox/.3/target/scala-3.1.3/native/0.ll
```

Any method, including the `main` method, might be defined in any of these files. [Appendix A: Finding main methods in .ll files](#) will help locating the code you are interested in.

Once you have located the code, you must be familiar with the [LLVM assembly language](#). NIR is a subset of the LLVM assembly language, so [Native Intermediate Representation](#) may be a gentler starting point.

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> sandbox3/clean; sandbox3/run
```

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

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

2.3.4 Native Intermediate Representation

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

Contents

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

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

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

Introduction

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

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

Would map to:

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

def @Test$::main_class.ssnr.ObjectArray_unit : (module @Test$, class @scala.
↳scalnative.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
```

Corresponds 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
```

(continues on next page)

(continued from previous page)

```

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
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.

2.3.8 Appendix A: Finding main methods in .ll files

Name mangling describes the precise, low level details Scala Native uses to transform names when generating code into `.ll` files.

This section shows how that information might be used to find a given method in those files. The `main` method is used as an example.

Scala 3 style main methods

The following code

```
package example
@main def run(): Unit = ???
```

creates a fully qualified class name `example.run`, with length 11.

```
C := example.run - fully qualified of the main class
N := 11          - length of fully qualified class name C
```

The entry point for this name has the form:

```
_SM<N><C>$D4mainLAL16java.lang.String_uEo
```

yielding:

```
_SM11example.run$D4mainLAL16java.lang.String_uEo
```

`LAL16java.lang.String_uEo` is the return type.

Scala 2 style main methods

The following code

```
package example

object Test {
  def main(args: Array[String]): Unit = ()
}
```

creates a fully qualified class name `example.Test`, with length 12.

```
C := example.Test - fully qualified of the main class
N := 12           - length of fully qualified class name C
```

A static main method forwarder is defined in a companion class to implement the companion and has the form:

```
_SM<N><C>D4mainLAL16java.lang.String_uEo
```

yielding:

```
_SM12example.TestD4mainLAL16java.lang.String_uEo.
```

The actual main method defined in the companion object has the form:

```
_SM<N+1><C>$D4mainLAL16java.lang.String_uEo
```

yielding:

```
_SM13example.Test$D4mainLAL16java.lang.String_uEo
```

LAL16java.lang.String_uEo is the return type.

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.

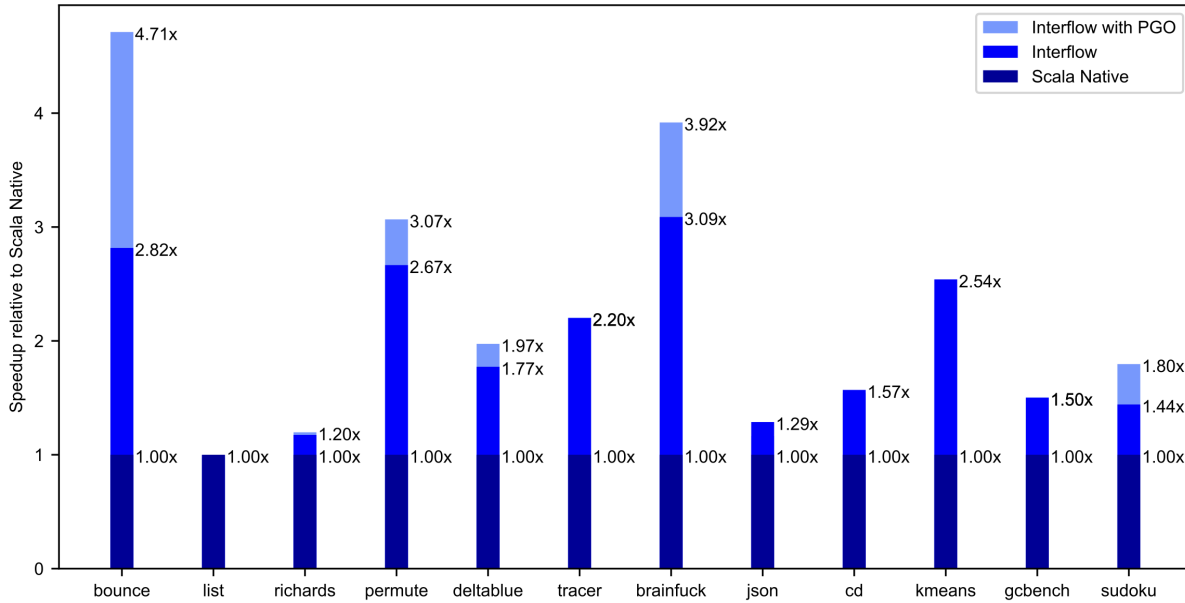
Additionally, we also add support for following *profile-guided optimizations*:

- *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.

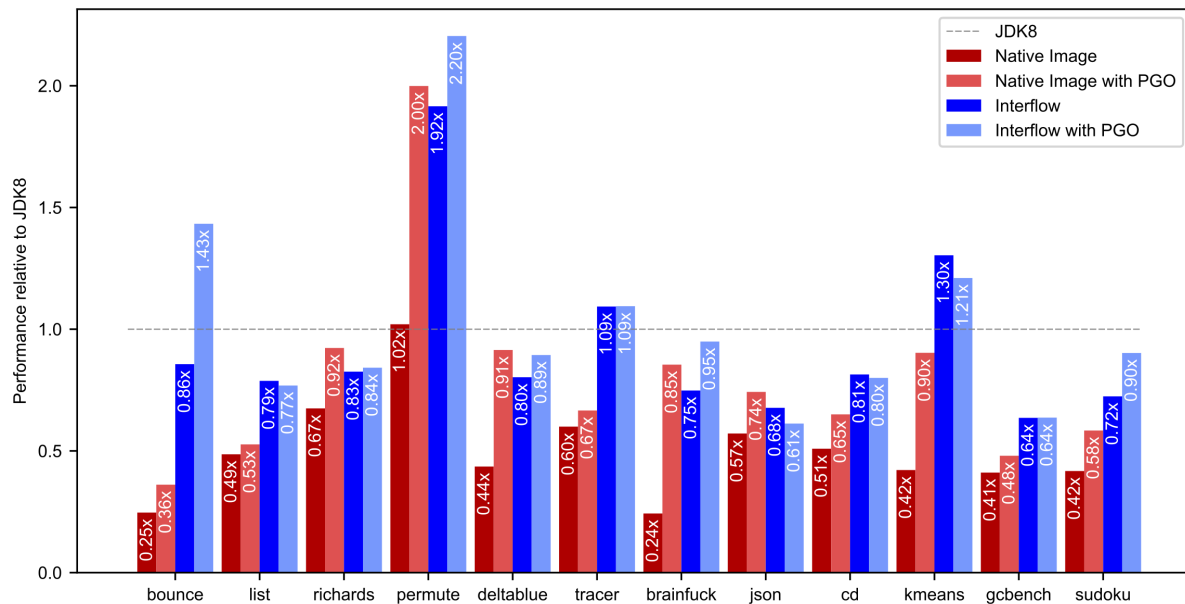
Evaluation

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 0.4.14 (2023-06-06)

We're happy to announce the release of Scala Native 0.4.14!

This patch version fixes backward-compatibility problems introduced in Scala Native 0.4.13. See [changelog of version 0.4.13](#) for more information.

Scala standard library used by this release is based on the following versions:

2.5.2 0.4.13 (2023-06-05)

We're happy to announce the release of Scala Native 0.4.13!

This release does not introduce any major improvements, but comes with variety of bugfixes. Scala Native 0.4.13 also updates the underlying Scala standard library, allowing to use its improvements in Scala 2.12, 2.13 and new definitions introduced in Scala 3.3.

Scala standard library used by this release is based on the following versions:

Contributors

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

```
$ git shortlog -sn --no-merges v0.4.12..
 20      Wojciech Mazur
   5      LeeTibbert
   3      Rikito Taniguchi
   2      Abdullah Sabaa Allil
   2      Arman Bilge
   1      Eric K Richardson
   1      Jarek Sacha
   1      Lorenzo Gabriele
   1      Vincent Lafeychine
   1      ankusharya
   1      kim / Motoyuki Kimura
```

Merged PRs

v0.4.13 (2023-06-03)

[Full Changelog](#)

Merged pull requests:

Compiler plugin

- Fix handling empty list of var args for extern method in Scala 2.13 #3240 (WojciechMazur)
- Report error when extern definition is marked as inlined #3241 (WojciechMazur)

Scala Native runtime

- Fix JUnit class cast when comparing Float/Double arrays #3249 (WojciechMazur)
- Fix referential equality of `scala.reflect.ClassTag` by caching underlying Manifest instances #3256 (WojciechMazur)
- Improved linktime resolved definitions #3266 (WojciechMazur)
- Expose parsed target triple in `LinktimeInfo` #3258 (armanbilge)
- Expose heap size information to the user #3275 (Abdullahsab3)
- Use `Array[Int]` instead of `Array[Array[Boolean]]` for `[class,trait]_has_trait` #3279 (lolgab)
- Backport `uioOps` to 0.4.x #3259 (armanbilge)

Java Standard Library

- Fix handling of `Path.relative` on Windows #3299 (jpsacha)
- Provide an evolution of Scala Native support for the Java Stream API #3268 (LeeTibbert)
- `Character.isWhitespace(-1)` returns false #3284 (tanishiking)

2.5.3 0.4.12 (2023-03-22)

We're happy to announce the release of Scala Native. It's the next maintenance release for Scala Native 0.4.x. This release fixes regressions introduced in the previous version and adds some requested features.

The Scala standard library used by this release is based on the following versions:

Notable changes

Composable extern definitions using `@extern trait`

Extern definitions can now be composed using traits annotated as `@extern`. Extern objects can now be composed using multiple extern traits allowing for better modeling of foreign APIs. A good candidate for modeling C bindings with this approach can be `errno.h` from C standard library and its POSIX extension. It can now be modeled as following

```
import scala.scalanative.unsafe.*

@extern trait errnoC {
  var errno: CInt = extern

  def EILSEQ: CInt = extern
}
```

(continues on next page)

(continued from previous page)

```
@extern trait errnoPosix extends errnoC {
  def EWOULDBLOCK: CInt = extern
  def EINPROGRESS: CInt = extern
  def EINTR: CInt = extern
}

@extern object errno extends errnoC with errnoPosix
```

The current bindings of POSIX and C standard library are not affected by this change, however, new model would be used in Scala Native 0.5.x

Contributors

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

```
$ git shortlog -sn --no-merges v0.4.11..v0.4.12
 8 Wojciech Mazur
 1 Eric K Richardson
 1 LeeTibbert
```

Merged PRs

(2023-03-22)

Full Changelog

Merged pull requests:

POSIX bindings

- Implement posixlib dlfcn #3234 (LeeTibbert)

Compiler plugin

- Improve resolving repeated parameters in Scala3 extern methods #3230 (WojciechMazur)
- Fix exports of extern methods using variadic arguments in Scala3 #3232 (WojciechMazur)
- Allow to compose extern definitions using @extern trait #2988 (WojciechMazur)

Toolchain

- Fix regression in handling opaque pointers on Windows #3226 (WojciechMazur)

2.5.4 0.4.11 (2023-03-15)

We're happy to announce the release of Scala Native. It's the next maintenance release for Scala Native 0.4.x. As always it brings bug fixes and minor improvements.

Notable changes

Extern methods with a variadic number of arguments

For a long time, Scala Native supported `C va_list` using `scalnative.unsafe.CVarArgList`. This allowed for interop with some of the C functions taking the variadic number of arguments. This release makes usage and definition of them easier, by restoring support for idiomatic ways of passing them using Scala variadic arguments lists.

```
void printf(char* format, ...);
```

```
@extern def printf(format: CString, args: Any*): Unit = extern

@main def test() =
  val msg = c"MyMessage"
  printf("String '%s' is allocated at %p and has %d characters\n", msg, msg,
  ↪strlen(msg))
```

Support for LLVM 15

The latest versions of LLVM added a new internal representation of pointers - their opaque variant replaces typed pointers. This change should not affect most of the users, but in some specific builds it could have lead to linking issues. Now Scala Native will try to detect version of LLVM toolchain. When using LLVM 15 or newer Scala Native toolchain would always generate opaque pointers in the compiled LLVM IR.

The Scala standard library used by this release is based on the following versions:

Contributors

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

```
$ git shortlog -sn --no-merges v0.4.10..v0.4.11
 24      Wojciech Mazur
 14      LeeTibbert
  2      Arman Bilge
  1      João Costa
  1      Ondra Pelech
  1      philwalk
```

Merged PRs

(2023-03-16)

Full Changelog

Merged pull requests:

Java Standard Library

- Partial Fix #3090: `j.nio.MappedByteBuffer` no longer causes segmentation fault on FreeBSD64 #3113 (LeeTibbert)
- Port all missing `java.util.function` types #3127 (WojciechMazur)

- Fix #3131: javalib ServerSocket should now be more accepting #3140 (LeeTibbert)
- Fix #3153: j.nio.fs.FileHelpers uses only java.io.tmp property for temporary files/dirs #3155 (LeeTibbert)
- Fix #3071, #3135: Implement Java 11 writeString & readString methods and Java 10 transferTo #3159 (LeeTibbert)
- Fix #2937, 3163: improved j.nio.f.Files default directory idiom handling #3166 (LeeTibbert)
- Partial fix #3165: Port two JSR-166 concurrent interfaces/traits: BlockingDeque, TransferQueue #3188 (LeeTibbert)
- Fix #3192, #3194: Implement limited java spliterator support #3202 (LeeTibbert)
- javalib Spliterators trySplit() methods now split #3218 (LeeTibbert)

POSIX bindings

- posixlib socket.c now compiles on FreeBSD arm64 #3112 (LeeTibbert)
- Fix #1642: posixlib stdio.scala is now mostly Open Group 2018 compliant #3160 (LeeTibbert)
- Fix #3206: posixlib unistd and monetary use new CVarArgs support #3209 (LeeTibbert)

Compiler plugin

- Fix generation of CFuncPtr extern forwarders using opaque types #3182 (WojciechMazur)
- Don't emit Inst.Jump/Inst.Label in NIR taking single Unit argument #3201 (WojciechMazur)
- Restore support for C VarArgs alongside current CVarArgLists #3204 (WojciechMazur)
- Allow for materialization of Tag[Ptr[_]] or taking abstract type #3207 (WojciechMazur)

Toolchain

- Define null guards for methods using this #3123 (WojciechMazur)
- Fix #3173: Linux executable file .comment section shows build info #3183 (LeeTibbert)
- Fix cygwin and msys build problems #3180 (philwalk)
- Use opaque pointers in generated LLVM IR when possible #3190 (WojciechMazur)
- Don't emit debug logs when skipping embedding source files #3191 (WojciechMazur)
- Emit Val.Unit/Type.Unit as void in LLVM IR instead of ref to BoxedUnit #3200 (WojciechMazur)
- Restore stack state after executing inlined function to prevent stack overflows #3199 (WojciechMazur)
- Fix poisonous new lines escapes in nir.Show leading to linker failures #3208 (WojciechMazur)

Runtime

- Update LLVM libunwind to 15.0.7 (was 12.0.1) #3184 (WojciechMazur)
- Commix GC - fix deadlocks due to misaligned pointers when marking range #3185 (WojciechMazur)

sbt plugin

- Port `definedTestNames` override from Scala.js #3203 (armanbilge)

2.5.5 0.4.10 (2023-01-27)

We're happy to announce the release of Scala Native.

Scala Native 0.4.10 adds support for Scala 3.2.2 with its new lazy vals implementation, and drops the support for Scala 2.11 which has been EOL for over 5 years. The latest release also contains multiple bug fixes and improvements, including a new release mode `scala.scalanative.build.Mode.releaseSize` oriented for the size of the produced binaries.

Scala standard library used by this release is based on the following versions:

Contributors

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

```
$ git shortlog -sn --no-merges v0.4.9..v0.4.10
 23 LeeTibbert
  9 Wojciech Mazur
  7      110416
  6 Arman Bilge
  3 Eric K Richardson
  1      Hossein Naderi
  1      Dong Nguyen
```

Merged PRs

v0.4.10 (2023-01-27)

[Full Changelog](#)

Merged pull requests:

Supported Scala versions

- Drop Scala 2.11 #3028 (ekrich)
- Support Scala 3.2.2 #3094 (WojciechMazur)

Java Standard Library

- Better error handling for `Files#createLink` #3012 (armanbilge) #3015
- Fix #2755: `j.nio.Files#readAllBytes` reports failed Unix file open call #3026 (LeeTibbert)
- Fix #2973: Implement some requested `j.io.InputStream` Java 9 & 11 methods #3031 (LeeTibbert)
- Make initialization of `System.properties` a lazy operation #3061 (WojciechMazur)
- Add `java.util.concurrent.Flow` #3099 (armanbilge)
- Port `java.util.TreeMap` and friends #3102 (armanbilge)

- Add `java.nio.file.Path.of` methods #3083 (i10416)
- Generate scaladoc for `java.lib` in Scala 2 #3035 (ekrich)

POSIX bindings

- Add POSIX `sys/un` bindings #3025 (LeeTibbert)
- Add POSIX `sys/times` bindings #3032 (LeeTibbert)
- Add POSIX bindings for `glob`, `fnmatch` and `libgen` #3041 (LeeTibbert)
- Add POSIX `wordexp` bindings
- #3042 (LeeTibbert)
- Add subset of C and POSIX locale related method #3034 (LeeTibbert)
- Add POSIX `langinfo.h` and `nl_types.h` bindings #3044 (LeeTibbert)
- Reduce memory usage in `posixlib` `spawn` #3040 (LeeTibbert)

Compiler plugin

- Improve: report error on `extern` in `val def` #3033 (i10416)
- Report error on default arguments in `extern` method #3045 (i10416)
- Fix issue with using opaque types in `CFuncPtr` #3096 (WojciechMazur)
- Fix `NullPointerException` when writing NIR to virtual files #3108 (WojciechMazur)
- Relativize paths in NIR using `mapSourceURI` setting in compiler plugin #3109 (WojciechMazur)

JUnit runtime

- Allow to use inherited `org.junit.{After, Before}Class` methods #3055 (WojciechMazur)

Build toolchain

- Handle whitespaces in the files passed to the clang when linking #3062 (WojciechMazur)
- Fix #3078, allow methods to have more than 10k instructions #3095 (WojciechMazur)
- Introduce `scalnative.build.Mode.ReleaseSize` #3091 (dongnguyenvt)

sbt plugin

- Add `NativeTags.Link` to limit concurrency of `nativeLink` #3064 (armanbilge)

Other bugfixes

- Fix #3065: FreeBSD once again compiles #3077 (LeeTibbert)
- Fix #3069: `Commix gc` runs once again on FreeBSD #3079 (LeeTibbert)
- Fix #3073: `ProcessMonitor` semaphore now works on FreeBSD #3080 (LeeTibbert)

2.5.6 0.4.9 (2022-11-23)

We're happy to announce the release of Scala Native 0.4.9.

It's a patch release fixing linkage errors when building Scala 2.13 libraries using Scala 3 dependencies.

It does also reverse version policy changes leading to problems in sbt-crossproject. Improved version policy would be restored in Scala Native 0.5.x.

Scala Native 0.4.9 introduces a new feature - an experimental support for incremental compilation.

Scala standard library used by this release is based on the following versions:

New features

Incremental compilation

A new experimental compilation mode was being developed during the latest edition of Google Summer of Code by [Liangyong Yu](#). This feature splits generated code based on the package names and allows to skip re-compilation of generated LLVM IR if changes to definitions are detected. Incremental compilation can allow reducing compilation times by up to 20%. You can read more about this change in Liangyongs [GSoC report](#) To enable this experimental feature modify your nativeConfig:

```
nativeConfig ~= {  
  _.withIncrementalCompilation(true)  
}
```

Contributors

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

```
$ git shortlog -sn --no-merges v0.4.8..v0.4.9  
 5 Wojciech Mazur  
 3 Arman Bilge  
 3 LeeTibbert  
 1 yuly16
```

Merged PRs

v0.4.9 (2022-11-23)

[Full Changelog](#)

Merged pull requests:

- [Remove version scheme #2985](#) (armanbilge)
- [Fix UnknownHostException message #2984](#) (armanbilge)
- [Fix codegen when accessing Scala 3 enum in Scala 2.13 codebase #2989](#) (WojciechMazur)
- [Add more JDK9+ Math methods #2990](#) (armanbilge)
- [Experimental incremental compilation #2777](#) (yuly16)

- Cleanup incremental compilation and fix build issues #2998 (WojciechMazur)
- Implement posixlib net/if.{scala,c} #3000 (LeeTibbert)
- Partial fix #3005: j.n.InetAddress now uses more UnknownHostException #3007 (LeeTibbert)
- Fix #2987: j.n.InetAddress#toString now handles interface names #3002 (LeeTibbert)

2.5.7 0.4.8 (2022-11-09)

We're happy to announce the release of Scala Native 0.4.8. The latest release brings multiple bugfixes and implementation optimizations, as well as new exciting features including building dynamic libraries, configurable optimizer, and easier access to arrays underlying memory.

Scala standard library used by this release is based on the following versions:

Contributors

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

```
$ git shortlog -sn --no-merges v0.4.7..v0.4.8
 18      LeeTibbert
 18      Wojciech Mazur
 11      Arman Bilge
  6      Eric K Richardson
  5      David Bouyssié
  2      Mark Hammons
  1      Daniel Esik
  1      Jamie Willis
  1      João Costa
  1      Liangyong Yu
```

New features

Producing native libraries using Scala Native

Scala Native can now produce both dynamic and static libraries. This feature allows to use Scala Native code in foreign runtimes, eg. in C, Rust or on the JVM. To enable this feature switch the build target of your project and annotate entry points for library using `@exported` annotation.

```
import scala.scalanative.build.BuildTarget
nativeConfig ~= {
  _.withBuildTarget(BuildTarget.libraryDynamic)
}
```

Only statically reachable functions can be exported in your library. Exporting module fields is not allowed, however, you can export their accessor using `@exportAccessors` annotation.

```
object Foo {
  @exportAccessors
  var counter: Int = 0

  @exportAccessors("error_message")
```

(continues on next page)

(continued from previous page)

```

val ErrorMessage: CString = c"Something bad just happened!"

@exported
def addLongs(l: Long, r: Long): Long = l + r
}

```

This feature is treated as experimental and might change in the future. For more information see [dedicated interop section](#) and [available build targets list](#).

Configurable Scala Native optimizer

In the past, we have observed very long build times for some projects using Scala Native. In most of them, most of the time was spent in the optimizer, especially when using `release-full` build mode. Now Scala Native can be configured to limit the amount of inlines, which in some cases might have been too eager.

```

// build.sbt
nativeConfig ~= { config =>
  val optimizerConfig = config.optimizerConfig
  config.withOptimizerConfig{
    optimizerConfig
      .withMaxInlineDepth(10) // Maximal amount of nested inlines - default=None
      .withMaxCallerSize(8192) // Maximal number of instructions in caller function -
↳ default=8192
      .withMaxInlineSize(8) // Maximal number of instructions in inlined function,
↳- default=8
  }
}

```

Easier access to arrays underlying memory

When interacting with native code it is sometimes expected to allocate memory Garbage Collected memory and access it directly as a pointer type. So far it was allowed to access the underlying memory of Scala Arrays, but it needed knowledge about internals of the Scala Native runtime. Now you can use the dedicated extension method instead to make this easier.

```

@main def sandbox = {
  import scala.scalanative.unsafe.*
  val arr: scala.Array[Byte] = new Array[Byte](1024)
  val ptr: Ptr[Byte] = arr.at(0)
}

```

Merged PRs

v0.4.8 (2022-11-09)

[Full Changelog](#)

Merged pull requests:

Scala Native Compiler Plugin

- Don't unapply unnecessary unboxing in lambdas #2938 (WojciechMazur)
- Fix encoding of `scala.Nothing` and `scala.Null` in type signatures #2949 (WojciechMazur)
- Handle passing null for unboxed arguments of extern methods #2950 (WojciechMazur)
- Report error when referencing local state in `CFuncPtr` #2957 (WojciechMazur)
- Fix lost information about value class instance #2959 (WojciechMazur)

Scala Native toolchain

- Encode main class name to match outputs of the compiler #2955 (WojciechMazur)
- Configurable Interflow optimizer #2819 (yuly16)
- Allow to link as dynamic library #2145 (WojciechMazur)

Scala Native Standard Library

- Add `UnsafeRichArray#at` syntax extension #2888 (armanbilge)
- Add `LinktimeInfo.{debugMode, releaseMode}` #2886 (armanbilge)
- Fix #2921: Commix Heap.c now compiles with Clang 15.0.3 #2922 (LeeTibbert)

Posix library

- Fix #2841: complete POSIX `string.h`, `strings.h` #2855 (LeeTibbert)
- Fix #2892: Implement `posixlib sys/select pselect()` #2895 (LeeTibbert)
- Fix #2891: `posixlib spawn` is now implemented. #2894 (LeeTibbert)
- Partial fix #2963: Add missing `SIGTERM` & `kin` to `posixlib signal.scala` #2964 (LeeTibbert)
- Fix #2893: Implement `posixlib wait.scala` #2969 (LeeTibbert)

Java Standard Library

- Fix `FileChannel#write` for read-only buffers #2884 (armanbilge)
- Port `j.u.SplittableRandom` from `Scala.js` #2879 (armanbilge)
- Adding missing `java.lang.Character` functionality #2871 (j-mie6)
- Port `j.u.ArrayDeque` from `JSR 166` #2898 (armanbilge)
- Fix #2903: avoid systematic checking of `String` integrity in `IEEE754Helpers` #2907 (david-bouyssie)
- Fix #2927: Expunge non-JVM `j.l.String#getValue()` #2928 (LeeTibbert)
- Fix #12925: A `j.l.String` constructor now yields immutable strings #2929 (LeeTibbert)
- Fix #2935: Ensure `StringBuilder` does not alter existing child `Strings` #2936 (LeeTibbert)
- Optimize method `AbstractStringBuilder.append0(CharSequence, Int, Int)` #2909 (david-bouyssie)

- A few `java.net.Inet*Address` fixes #2877 (LeeTibbert)
- Add JDK9 constructors to `j.m.BigInteger` #2974 (armanbilge)

Other

- Fix #1826: Add documentation for GC settings #2910 (ekrich)
- Fix #2678: Provide examples of using `NativeConfig` #2926 (LeeTibbert)
- Fix `ScalaRunTime` patch for 2.12 #2876 (armanbilge)

2.5.8 0.4.7 (2022-09-01)

We're happy to announce the release of Scala Native 0.4.7. This patch version fixes Scala Native 0.4.6 problems using JDK 8.

See [changelog of version 0.4.6](#) for more information.

2.5.9 0.4.6 (2022-09-01)

We're happy to announce the release of Scala Native 0.4.6. The new release brought support for Scala 3.2.0 and multiple bug fixes.

Scala 3.2.0 support

The new version of Scala Native is now supporting Scala 3.2.0. The latest release of Scala introduced a change to the internal implementation of lazy vals. These changes were not compatible with previous releases of the Scala Native compiler plugin.

The latest Scala Native standard library *does not* enforce an upgrade to Scala 3.2.0. All artifacts are still published using Scala 3.1.x. It also includes the Scala 3 standard library, which is now based on the sources of Scala 3.2.0. The publishing model of native artifacts of Scala 3 standard library is currently flawed. Artifacts are not cross-compiled for each Scala 3 version. Instead, we (maintainers), are forced on choosing only 1 version of Scala 3 sources. Fortunately, Scala 3.2.0 standard library fully sources compatible with 3.1.x releases. This oversight would be fixed in the next major version of Scala Native.

Scala standard library used by this release is based on the following versions:

Contributors

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

```
$ git shortlog -sn --no-merges v0.4.5..v0.4.6
 16      Arman Bilge
 13      Wojciech Mazur
 11      LeeTibbert
  3      Eric K Richardson
  1      James You
  1      João Costa
  1      Yilin Wei
  1      yuly16
```

Merged PRs

v0.4.6 (2022-09-01)

Full Changelog

Merged pull requests:

Java standard library

- Drop empty elements in `Paths.get` #2745 (armanbilge)
- Path extends `Watchable` #2749 (armanbilge)
- Port `CopyOnWriteArrayList` from `Scala.js` #2725 (armanbilge)
- Extract `getCanonicalHostName` from `getHostName` #2752 (armanbilge)
- Port `InflaterOutputStream` from `Android Luni` #2748 (armanbilge)
- Add `java.nio.ch.CompletionHandler` #2796 (armanbilge)
- Add `java.net.ProtocolException` #2791 (armanbilge)
- Add `j.u.s.Stream#forEach` #2747 (armanbilge)
- Implement `UUID.randomUUID()` using `java.security.SecureRandom` #2759 (armanbilge)
- Fix #2814: Port and extend `StandardSocketOptions` #2816 (LeeTibbert)

Posix bindings

- Add `SO_REUSEPORT` #2806 (armanbilge)
- Handle null args to `accept` #2807 (armanbilge)
- Add `getpeername` #2812 (armanbilge)
- Fix #2717: Provide missing POSIX symbol `EADDRNOTAVAIL` #2718 (LeeTibbert)
- Fix #2706: Implement clock methods and test cases, complete `time.h` #2713 (LeeTibbert)

Scala Native standard library

- Add `LinktimeInfo.is{FreeBSD,Mac,Linux}` #2809 (armanbilge)

Scala Native compiler plugin

- Fixes to usage of structural types in `Scala 3` #2737 (WojciechMazur)
- Use correct parameters lists when generating extern methods in `Scala 3` #2736 (WojciechMazur)

Bug fixes

- Fix #2751: Files#delete now throws informative DirectoryNotEmptyException #2754 (LeeTibbert)
- Align error messages for UnknownHost with JVM #2805 (armanbilge)
- Fix #2793: Two regex patterns now tell the truth #2795 (LeeTibbert)
- Add CharsetTest, fix Charset#aliases #2792 (armanbilge)
- Fix #2769, #2650: Remove several defects discovered by ProcessTest #2776 (LeeTibbert)
- Fix #2750: Two javalib entities now throw expected Exceptions #2756 (LeeTibbert)
- Fix #2604: Print signal name when test interface exits #2722 (jmesyou)
- Fixes: #1915: Consider clock_gettime for macos and linux #2704 (ekrich)
- Fix #2730: Package private at the top level of build breaks easy import #2744 (ekrich)

2.5.10 0.4.5 (2022-06-27)

We're happy to announce the release of Scala Native 0.4.5. This patch release targets bugfixes and implements some of the missing Java standard library classes, but does not include any new features. This release also adds initial support for compilation using Java 17.

Scala standard library used by this release is based on the following versions:

Contributors

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

```
$ git shortlog -sn --no-merges v0.4.4..v0.4.5
 23      LeeTibbert
 10      Wojciech Mazur
  6      Arman Bilge
  4      110416
  4      Eric K Richardson
  1      Daniel Esik
  1      Felix Yan
  1      Guillaume Raffin
  1      Lorenzo Gabriele
```

Merged PRs

v0.4.5 (2022-06-27)

[Full Changelog](#)

Merged pull requests:

Java standard library

- Add a timeout to InetAddress host caching, fix #2310 #2580 (TheElectronWill)
- Port java.util.Optional from Scala.js #2596 (ekrich)

- Fix CRC32 update with empty array #2618 (armanbilge)
- Fix #2608, #1543: Correct/complete regex Matcher region support #2619 (LeeTibbert)
- Port ThreadLocalRandom #2674 (armanbilge)
- Port ConcurrentHashMap #2676 (armanbilge)
- Port new implementation of TreeSet and PriorityQueue from Scala.js #2677 (WojciechMazur)
- Port stubs for java.util.concurrent data structures from Scala.js #2682 (WojciechMazur)
- Port ConcurrentSkipListSet #2675 (armanbilge)
- Implement `java.lang.reflect::Array.newInstance` for multi-dimensional overload #2683 (WojciechMazur)
- Port IntUnaryOperator #2613 (armanbilge)
- Add stubs for `java.lang.constant.{Constantable, ConstantDesc}` #2684 (WojciechMazur)
- Fix #2586: Port three Scala.js fixes #2694 (LeeTibbert)

C/POSIX standard library bindings

- add tgmath for real #104 #2587 (i10416)
- add fenv.h #2591 (i10416)
- add some int-related bindings #2594 (i10416)
- Fix #2627: posix socket.c#getaddrinfo no longer frees unallocated memory #2628 (LeeTibbert)
- Fix I2633: posix netdb.c now has FreeBSD required include file #2637 (LeeTibbert)
- Fix #2660, #2661: Provide Posix netdb gai_strerror method and associated EAI definitions #2662 (LeeTibbert)
- Fix #2314: Posix netdb getaddrinfo now handles null hints arguments correctly #2663 (LeeTibbert)
- Fix #2395: posixlib usleep method is now deprecated #2669 (LeeTibbert)
- Fix #2657: Synthesize posix timezone and daylight variables on FreeBSD #2695 (LeeTibbert)

Scala Native runtime library

- Fix #2635: implement natively Platform.scala#isFreeBSD #2636 (LeeTibbert)
- Fix Immix GC compilation errors #2685 (WojciechMazur)

2.5.11 0.4.4 (2022-03-02)

We're happy to announce the new patch release of Scala Native v0.4.4, which brings improved support for Arm64, introduces experimental embedding of resources and adds dedicated ScalaNativeJUnitPlugin for easier setup of tests. Also in this release we've fixed all the issues found in the recent support for Scala 3, including Selectables, cross version with Scala 2.13 dependencies and many more.

Top changes

Experimental support for embedding resources

Thanks to the effort of @jchyb you can embed resources of your project and use them at runtime using `Class[_].getResourceAsStream()` in the same way as on the JVM. If you use some native sources in your project, e.g. written in C, you don't need to worry about increased size of binary - all files in `resources/scala-native` directory would be ignored. To enable experimental support of this feature enable it in Native Config:

```
nativeConfig ~= { _.withEmbedResources(true) }
```

More information about this new feature can be found [here](#)

New sbt plugin - ScalaNativeJUnitPlugin

In this release we have added a new sbt plugin dedicated for easier setup of JUnit tests framework. `ScalaNativeJUnitPlugin` is a port of similar feature from `Scala.js` - it allows you to forget about specifying dedicated library dependencies for tests runtime. It does also ensure that JUnit plugin would no longer be included in dependencies of downstream libraries.

When migrating to newer version of Scala Native you enable both Scala Native support and setup JUnit runtime with the current version using only one setting in your project `enablePlugins(ScalaNativeJUnitPlugin)`. If building with a cross project you can use the following syntax to add it to the cross project: `.nativeConfigure(_.enablePlugins(ScalaNativeJUnitPlugin))`. Remember to remove the old settings needed for JUnit support:

```
addCompilerPlugin("org.scala-native" % "junit-plugin" % nativeVersion cross_  
↳CrossVersion.full)  
libraryDependencies += "org.scala-native" %%% "junit-runtime" % nativeVersion % Test
```

Improved Arm64 support

Last but not least we have improved our support for Arm64 architecture. This means that you can now use Scala Native on Apple M1 chips and Linux based arm64 devices.

Contributors

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

```
$ git shortlog -sn --no-merges v0.4.3..v0.4.4  
19      Wojciech Mazur  
4       Jan Chyb  
3       Eric K Richardson  
1       Arman Bilge  
1       Shadaj Laddad  
1       algobardo
```

Merged PRs

v0.4.4 (2022-03-02)

Full Changelog

Merged pull requests:

New features

- Add `getResourceAsStream()` via embedding resources #2475 (jchyb)
- Add `ScalaNativeJUnitPlugin` for easy setup of JUnit tests #2556 (WojciechMazur)
- Port `java.util.BitSet` #2565 (armanbilge)
- Support `PathMatching` via `Globs` #2559 (jchyb)

Bug fixes

- Fix `UnixPath.relativeize` when containing `..` #2542 (algotardo)
- Fix #2547: `java.util.TimeZone` stub is used instead of `scala-java-time` #2548 (ekrich)
- Fix failing to link `CFuncPtr` created from adapted function #2551 (WojciechMazur)
- Add support for `scala.reflect.Selectable` #2549 (WojciechMazur)
- Fix incorrect self type of closures #2554 (WojciechMazur)
- Fix conflicting cross version suffixes when using `Cross.for3Use_2.13` #2553 (WojciechMazur)
- Fix failing to link usages of `.asInstanceOf[Nothing]` #2555 (WojciechMazur)
- Fix issues with “Is a directory” error in `ResourceEmbedder` #2568 (jchyb)
- Fix runtime errors when cross-compiling to `Arm64` #2573 (WojciechMazur)
- Fix deprecated constructor `String(Array[Byte], Int, Int, Int)` #2576 (WojciechMazur)
- Fix `NullPointerException` when assigning null to non-Ptr types #2575 (WojciechMazur)
- Fix negation of floating point numbers #2577 (WojciechMazur)
- Add missing constructors for throwables and improve JVM compliance #2578 (WojciechMazur)
- Fix runtime problems on MacOS using M1 chips #2579 (WojciechMazur)

Misc

- Update `libunwind` to LLVM 12.0.1 #2502 (shadaj)

2.5.12 0.4.3 (2022-01-21)

We’re happy to announce the release of Scala Native v0.4.3, which brings support for Scala 3!

This release does also contain performance-critical bug fixes to performing operations on Strings, leading to significant improvement of regexes throughput. Furthermore it comes with improvements to the sbt plugin in the area of incremental compilation and implementation of some missing methods from Java standard library.

Breaking changes in release 0.4.3

The minimal version of Scala 3 supported by Scala Native is 3.1.0, due to fatal blockers in Scala 3.0.x. In the future, when improved backward compatibility support in Scala 3.2 would be implemented, Scala Native might be backported to the older version of the compiler.

Changes to the main method resolving

In the previous versions of Scala Native the main entry point was always resolved based on the companion object defining the main method. This behaviour however led to multiple issues in situations when an actual main method was introduced via inheritance. Because of that issue, build tools needed to always add `$` suffix to actual main class to make sure that entry method would be found. With the 0.4.3 release, Scala Native does generate static method forwarders in the same way as it is done in JVM and Scala.js. Build tools should make sure to no longer add `$` to the discovered main class name.

Source compatibility of `stackalloc[T]` and `alloc[T]` methods

No-param methods `scalanative.unsafe.stackalloc[T]` and `scalanative.unsafe.alloc[T]` were removed from Scala 3 `unsafe` package. In Scala 2 these methods have been deprecated and it is recommended to use empty-param variants of these methods instead, eg. `scalanative.unsafe.stackalloc[T]()`. This change was introduced due to changes of how does Scala 3 interpret source code as it could lead to dangerous, hard to debug, runtime errors. As an example call to `stackalloc[Byte](10)` in Scala 2 would result in allocating memory of size `10 * sizeof(Byte)`. In Scala 3 however, the given snippet would be interpreted as `stackalloc[Byte].apply(10)` which is still valid, but leads to allocating `1 * sizeof(Byte)` chunk of memory and accessing its 10th element, which would lead to undefined behaviour. More information can be found in [#2480](#)

Known issues

Cannot create documentation using scaladoc in Scala Native sbt project

When trying to generate documentation using Scaladoc for projects using Scala Native compilation would fail with an exception. This issue is common for most of the compiler plugins and was already fixed in Scala 3.1.1 compiler. As a workaround in Scala 3.1.0 we would recommend adjusting sbt build, by filtering out all `-Xplugin` settings from `Compile / doc / scalacOptions`. Workaround can be found in [this discussion](#)

Contributors

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

```
$ git shortlog -sn --no-merges v0.4.2..v0.4.3
 54      Wojciech Mazur
   5      Eric K Richardson
   2      Jan Chyb
   1      Anton Sviridov
   1      Kamil Podsiadło
   1      Lorenzo Gabriele
   1      Ruslan Shevchenko
```

Merged PRs

v0.4.3 (2022-01-21)

Full Changelog

Merged pull requests:

Bug fixes

- Initialize `_String` with `_Array[Char]` #2449 (lolgab)
- Fix ARM64 MacOS varargs #2448 (keynmol)
- Fix certain absolute file paths being shown as relative on Windows #2471 (jchyb)
- Make `java.nio.MappedByteBuffer` JDK9+ compliant #2453 (WojciechMazur)
- Remove redundant conversion of `CharSequence` into `Array` in `Character.codePoint{at, before}` methods #2495 (WojciechMazur)

Native library

- Adapt `unsafe` package for upcoming Scala 3 support #2454 (WojciechMazur)
- Enforce usage of empty parens variant of `stackalloc[T]` and `alloc[T]` in Scala 3 #2480 (WojciechMazur)

Standard Java library

- Port `URLDecoder` from `Scala.js` #2506 (kpodsiad)
- Align implementation of UTF-8 and UTF-16 charsets with `Scala.js` #2515 (WojciechMazur)
- `ProcessMonitor` should not busy wait if there are no active processes #2518 (WojciechMazur)
- `System.properties` define `os.arch` #2452 (WojciechMazur)
- Start adding JDK 11 support #2450 (ekrich)
- Touch up NIO for Java 9+ #2455 (ekrich)
- Add missing `java.io.channels.Channels` methods #2532 (jchyb)
- Fix runtime errors in `BigDecimal` arithmetic operations #2534 (WojciechMazur)

Compiler plugin / NIR

- Use encoded names for methods and fields in NIR #2458 (WojciechMazur) (WojciechMazur)
- Define `Field` operation in NIR #2460 (WojciechMazur)
- Port NIR generation compiler plugin to Scala 3 #2462 (WojciechMazur)
- Adapt `LazyVals` to work in Scala Native #2465 (WojciechMazur)
- Port JUnit Plugin to Scala 3 #2467 (WojciechMazur)
- Introduce static scope and generate static method forwarders in Scala 3 #2472 (WojciechMazur)

- Generate static method forwarders in Scala 2 #2476 (WojciechMazur)
- Use public static main method forwarder as entry point #2481 (WojciechMazur)
- Improvements to generation and invocation of static methods #2489 (WojciechMazur)
- Fix #2504 - Use correct self value when generating Apply trees #2509 (WojciechMazur)
- Fix off by 1 `nir.Positions` produced in Scala 3 #2510 (WojciechMazur)
- Fix compilation errors when using lazy vals in quoted macros #2511 (WojciechMazur)
- Better error messages for `CFuncPtr.fromScalaFunction` #2512 (WojciechMazur)
- When enabling reflective instantiation check base classess instead of parents #2521 (WojciechMazur)

sbt plugin

- Store config cached fingerprints seperetly for each version of Scala #2516 (WojciechMazur)
- Add `-test` suffix to artifact path of test executables #2524 (WojciechMazur)

Linker

- Make reachablity phase more user friendly #2469 (WojciechMazur)

Misc

- Allow to define top-level extern definitons in Scala 3 #2496 (WojciechMazur)
- Add missing `Show` implementation for `Inst.LinktimeIf` #2513 (WojciechMazur)
- Implement time zone offset #2508 (ekrich)
- Add Scala 2.13.8 to the build #2525 (WojciechMazur)
- Allow to compile Scala 2.11 scalalib with JDK 17 #2530 (rssh)
- Add Scala 3.1.1 to the build #2536 (WojciechMazur)

2.5.13 0.4.3-RC2 (2022-01-11)

We're happy to announce the release of Scala Native v0.4.3-RC2, which brings support for Scala 3, and multiple bugfixes found since previous release candidate.

This release does also contain performance-critical bug fixes to performing operations on Strings, leading to significant improvement of regexes throughput.

Due to multiple internal changes and to ensure compatibility with libraries published using Scala 2.13, we've decided that the 0.4.3 will be preceded by release candidates. If no new bugs will be discovered until 21st January 2022 it would be promoted to a full release.

Breaking changes in release 0.4.3

The minimal version of Scala 3 supported by Scala Native is 3.1.0, due to fatal blockers in Scala 3.0.x.

Changes to the main method resolving

In the previous versions of Scala Native main entry point was always resolved based on the companion object defining the main method. This behaviour however lead to multiple issues in situations when an actual main method was introduced via inheritance. Because of that issue, build tools needed to always add `$` suffix to actual main class to make sure that entry method would be found. With the 0.4.3 release, Scala Native does generate static method forwarders in the same way as it is done in JVM and Scala.js. Build tools should make sure to no longer add `$` to the discovered main class name.

Source compatibility of `stackalloc[T]` and `alloc[T]` methods

No-param methods `scalnative.unsafe.stackalloc[T]` and `scalnative.unsafe.alloc[T]` were removed from Scala 3 `unsafe` package. In Scala 2 these methods have been deprecated and it is recommended to use empty-param variants of these methods instead, eg. `scalnative.unsafe.stackalloc[T]()`. This change was introduced due to changes of how does Scala 3 interpret source code as it could lead to dangerous, hard to debug, runtime errors. As an example call to `stackalloc[Byte](10)` in Scala 2 would result in allocating memory of size `10 * sizeof(Byte)`. In Scala 3 however, the given snippet would be interpreted as `stackalloc[Byte].apply(10)` which is still valid, but leads to allocating `1 * sizeof(Byte)` chunk of memory and accessing its 10th element, which would lead to undefined behaviour. More information can be found in [#2480](#)

Known issues

Cannot create documentation using scaladoc in Scala Native sbt project

When trying to generate documentation using Scaladoc for projects using Scala Native compilation would fail with an exception. This issue is common for most of the compiler plugins and was already fixed in Scala 3.1.1-RC2 compiler. As a workaround in Scala 3.1.0 we would recomend adjusting sbt build, by filtering out all `-Xplugin` settings from `Compile / doc / scalacOptions`. Workaround can be find in [this discussion](#)

Contributors

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

```
$ git shortlog -sn --no-merges v0.4.3-RC1..v0.4.3-RC2
 14      Wojciech Mazur
   2      Eric K Richardson
   1      Kamil Podsiadło
```

Merged PRs

v0.4.3-RC2 (2022-01-11)

[Full Changelog](#)

Merged pull requests:

Compiler plugin

- Fix #2504 - Use correct self value when generating Apply trees #2509 (WojciechMazur)
- Fix off by 1 `nir.Positions` produced in Scala 3 #2510 (WojciechMazur)
- Fix compilation errors when using lazy vals in quoted macros #2511 (WojciechMazur)
- Better error messages for `CFuncPtr.fromScalaFunction` #2512 (WojciechMazur)
- When enabling reflective instantiation check base classes instead of parents #2521 (WojciechMazur)

Standard Java library

- Port URLDecoder from Scala.js #2506 (kpodsiad)
- Align implementation of UTF-8 and UTF-16 charsets with Scala.js #2515 (WojciechMazur)
- ProcessMonitor should not busy wait if there are no active processes #2518 (WojciechMazur)

sbt plugin

- Store config cached fingerprints seperetly for each version of Scala #2516 (WojciechMazur)
- Add `-test` suffix to artifact path of test executables #2524 (WojciechMazur)

Misc

- Add missing `Show` implementation for `Inst.LinktimeIf` #2513 (WojciechMazur)
- Implement time zone offset #2508 (ekrich)
- Add Scala 2.13.8 to the build #2525 (WojciechMazur)

2.5.14 0.4.3-RC1 (2022-01-04)

We're happy to announce the release of Scala Native v0.4.3-RC1, which brings support for Scala 3. This release does also contain performance-critical bug fixes to performing operations on Strings, leading to significant improvement of regexes throughput.

Due to multiple internal changes and to ensure compatibility with libraries published using Scala 2.13, we've decided that the 0.4.3 will be preceded by release candidates. If no new bugs will be discovered until 17th January 2022 it would be promoted to a full release.

Breaking changes

The minimal version of Scala 3 supported by Scala Native is 3.1.0, due to fatal blockers in Scala 3.0.x.

Changes to the main method resolving

In the previous versions of Scala Native main entry point was always resolved based on the companion object defining the main method. This behaviour however lead to multiple issues in situations when an actual main method was introduced via inheritance. Because of that issue, build tools needed to always add `$` suffix to actual main class to make sure that entry method would be found. With the 0.4.3 release, Scala Native does generate static method forwarders in the same way as it is done in JVM and Scala.js. Build tools should make sure to no longer add `$` to the discovered main class name.

Source compatibility of `stackalloc[T]` and `alloc[T]` methods

No-param methods `scalanative.unsafe.stackalloc[T]` and `scalanative.unsafe.alloc[T]` were removed from Scala 3 `unsafe` package. In Scala 2 these methods have been deprecated and it is recommended to use empty-param variants of these methods instead, eg. `scalanative.unsafe.stackalloc[T]()`. This change was introduced due to changes of how does Scala 3 interpret source code as it could lead to dangerous, hard to debug, runtime errors. As an example call to `stackalloc[Byte](10)` in Scala 2 would result in allocating memory of size `10 * sizeof(Byte)`. In Scala 3 however, the given snippet would be interpreted as `stackalloc[Byte].apply(10)` which is still valid, but leads to allocating `1 * sizeof(Byte)` chunk of memory and accessing its 10th element, which would lead to undefined behaviour. More information can be found in [#2480](#)

Known issues

Cannot create documentation using scaladoc in Scala Native sbt project

When trying to generate documentation using Scaladoc for projects using Scala Native compilation would fail with an exception. This issue is common for most of the compiler plugins and was already fixed in Scala 3.1.1-RC2 compiler. As a workaround in Scala 3.1.0 we would recomend adjusting sbt build, by filtering out all `-Xplugin` settings from `Compile / doc / scalacOptions`.

Contributors

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

```
$ git shortlog -sn --no-merges v0.4.2..v0.4.3-RC1
 34      Wojciech Mazur
   3      Eric K Richardson
   1      Anton Sviridov
   1      Jan Chyb
   1      Lorenzo Gabriele
```

Merged PRs

v0.4.3-RC1 (2022-01-04)

[Full Changelog](#)

Merged pull requests:

Bug fixes

- Initialize `_String` with `_Array[Char]` #2449 (lolgab)
- Fix ARM64 MacOS varargs #2448 (keynmol)
- Fix certain absolute file paths being shown as relative on Windows #2471 (jchyb)
- Make `java.nio.MappedByteBuffer` JDK9+ compliant #2453 (WojciechMazur)
- Remove redundant conversion of `CharSequence` into `Array` in `Character.codePoint{at, before}` methods #2495 (WojciechMazur)

Features

- `System.properties` define `os.arch` #2452 (WojciechMazur)
- Start adding JDK 11 support #2450 (ekrich)
- Touch up NIO for Java 9+ #2455 (ekrich)
- Allow to define top-level extern definitions in Scala 3 #2496 (WojciechMazur)

Native library

- Adapt `unsafe` package for upcoming Scala 3 support #2454 (WojciechMazur)
- Enforce usage of empty parens variant of `stackalloc[T]` and `alloc[T]` in Scala 3 #2480 (WojciechMazur)

Compiler plugin / NIR

- Use encoded names for methods and fields in NIR #2458 (WojciechMazur) (WojciechMazur)
- Define `Field` operation in NIR #2460 (WojciechMazur)
- Port NIR generation compiler plugin to Scala 3 #2462 (WojciechMazur)
- Adapt `LazyVals` to work in Scala Native #2465 (WojciechMazur)
- Port JUnit Plugin to Scala 3 #2467 (WojciechMazur)
- Introduce static scope and generate static method forwarders in Scala 3 #2472 (WojciechMazur)
- Generate static method forwarders in Scala 2 #2476 (WojciechMazur)
- Use public static main method forwarder as entry point #2481 (WojciechMazur)
- Improvements to generation and invocation of static methods #2489 (WojciechMazur)

Linker

- Make reachability phase more user friendly #2469 (WojciechMazur)

2.5.15 0.4.2 (2021-12-06)

We're happy to announce the patch release of Scala Native 0.4.2, which brings improvements to Java HashMaps implementation and faster Zone allocations.

Contributors

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

```

$ git shortlog -sn --no-merges v0.4.1..v0.4.2
 4          Eric K Richardson
 3          Wojciech Mazur
 1          Denys Shabalin
 1          Jan Chyb
 1          Waris Radji
 1          ankusharya

```

Merged PRs

v0.4.2 (2021-12-06)

Full Changelog

Merged pull requests:

- Fix #2422: Add additional include and lib dir for brew, macOS m1 (arm) #2423 (ekrich)
- Support Scala 2.13.7 #2427 (WojciechMazur)
- Fix #2413: Add note to docs for creating sn project in mounted directories #2426 (ankusharya)
- Port and update collections and tests to support IdentityHashMap #2425 (ekrich)
- Fix #2433: spaces not being allowed in Paths on Windows #2434 (jchyb)
- Add missing unistd function getppid() #2440 (riiswa)
- Fix incorrect definition of Class.getConstructor stub method #2439 (WojciechMazur)
- Clean up reading unix env #2442 (ekrich)
- Fix #735: improve throughput of zone allocation #2443 (densh)

2.5.16 0.4.1 (2021-10-20)

We're happy to announce the release of Scala Native v0.4.1!

TL;DR

- **Backward compatible with previous 0.4.x releases,**
- Windows support
- Immix and Commix GC supports WeakReferences
- FileChannels improvements and support for MappedByteBuffers
- Updated JUnit runtime

- Faster builds in release modes
- New mechanism - linktime resolved values and conditional branching
- Improvements to Java stdlib implementation
- Multiple bugfixes

Most impacting changes

Windows support

This release of Scala Native brings native support for compilation and running Scala Native applications on Windows. For instructions how to setup your environment check out our [documentation](#)

Linktime resolved expressions

To make Windows support possible we introduced a new way of interacting with native code, especially with OS native functions that might not be existing on different platforms. Our solution to this problem is the introduction of linktime resolved expressions - special values using `@scala.scalanative.unsafe.resolvedAtLinktime` annotation. Each of such a value provides a stub in NIR (Native Intermediate Representation), which would be replaced when performing linking, based on values defined in `scala.scalanative.build.NativeConfig`. Currently, the list of linktime resolved expressions is limited and users cannot create new ones, but it might change in the future.

```
// Example of linktime resolved expressions definition
package scala.scalanative.meta
import scala.scalanative.unsafe._

object LinktimeInfo {
  @resolvedAtLinktime("scala.scalanative.meta.linktimeinfo.isWindows")
  def isWindows: Boolean = resolved
}
```

In case of usage of `if` condition containing linktime resolved value, at the time of performing linking of NIR files, whole condition expression would be replaced with constant boolean value equal to `true` or `false`. At this point we can dead-code eliminate never taken branch. This feature is crucial for cross compilation on different platforms - if we eliminate branch containing call to OS specific function it would be never used, and though we would not need its definition when building.

In the snippet below you can see an example of linktime conditional branching. Depending on Native Config settings only one of the `if` condition branch would be taken, the second one would be always discarded.

```
import scala.scalanative.meta.LinktimeInfo.isWindows
import scala.scalanative.unsafe._

def isDirectory(path: CString): Boolean = {
  if (isWindows) {
    val flag = FILE_ATTRIBUTE_DIRECTORY
    GetFileAttributesA(path) & FILE_ATTRIBUTE_DIRECTORY != 0
  } else {
    val buf = alloc[stat.stat]()
    stat.stat(path, buf)
    stat.S_ISDIR(buf._13) != 0
  }
}
```

Weak References support

The next new feature is partial support for Weak References, however usage of this feature is only possible when using Immix or Commix GC - they have been adopted to handle proper marking of WeakReference objects. Other Garbage Collector implementations, like external Boehm GC do not contain mechanisms allowing to perform WeakReference semantics. To check at runtime if your current GC supports weak references you can use the dedicated linktime resolved value `LinktimeInfo.isWeakReferenceSupported`

Contributors

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

```
$ git shortlog -sn --no-merges v0.4.0..v0.4.1
 56      Wojciech Mazur
 40      LeeTibbert
 19      Jan Chyb
 18      Eric K Richardson
 6       Nacho Cordón
 4       Kirill A. Korinsky
 4       Sébastien Doeraene
 2       João Costa
 2       Lorenzo Gabriele
 2       Teodor Dimov
 1       Alex Dupre
 1       Bjorn Regnell
 1       Denys Shabalin
 1       Ergys Dona
 1       philwalk
```

Merged PRs

v0.4.1 (2021-10-20)

[Full Changelog](#)

Merged pull requests:

Scala compiler plugin

- Fix: #2187 Match with guards does not compile #2188 (WojciechMazur)
- Fix #2305 compile error on macro using Scala 2.11.12 #2336 (jchyb)
- Fix: #2144, Lambda function generation should check if boxing is needed. #2147 (WojciechMazur)

Scala Native build

- Speedup Interflow #2134 (JD557)
- Remove legacy textual NIR parser #2158 (WojciechMazur)
- Allow for cross building tools #2156 (WojciechMazur)
- Improve speed of release-full compilation #2183 (catap)

- Fix reporting undefined symbols not marked with stub #2162 (WojciechMazur)
- Refactor build to compile each project and ll separately #2234 (ekrich)
- Allow for linktime conditional branching #2206 (WojciechMazur)
- Ignore no longer existing objects when inheriting Interflow state #2344 (WojciechMazur)
- Source compatibility with Scala 3 for tools #2347 (WojciechMazur)
- Always pass the LTO config without considering the release mode #2111 (lolgab)
- Fix #2211: NIR warnings from incorrect subtyping resolution #2357 (jchyb)
- Use LLD linker when setting LTO on Windows #2381 (WojciechMazur)
- Fix: Type.isPtrBox always returns false #2133 (WojciechMazur)
- Fix invalid nullability resolution for boxed primitives #2390 (WojciechMazur)
- Fix #805, extending class containing main does not include it in scope #1984 (WojciechMazur)

Sbt plugin

- Fix sbt plugin incompatibilities on Windows #2189 (WojciechMazur)
- Fix #2321: Fix Clang discovery on Windows #2334 (ekrich)
- if Windows, append .exe to binaryName in discover(), to support mingw64 and cygwin compilers #2349 (philwalk)
- Fix resolving binary version in ScalaNativeCrossVersion for snapshots #2207 (WojciechMazur)
- Fix 2378; Allow to read from stdin when using run command #2384 (WojciechMazur)
- Skip linking on subsequent nativeLink calls in SN sbt plugin #2389 (jchyb)
- Fix #2146: Publish sbt-scala-native to Sonatype instead of Bintray. #2386 (sjrd)
- Sbt nativeLink command should fail in case of clang failure #2394 (WojciechMazur)

Garbage Collector

- Corrects bug in named semaphores length for commix garbage collector #2216 (ncordon)
- Performs better checks in commix semaphore operations #2229 (ncordon)
- Removes magic constants from the commix semaphore length code #2232 (ncordon)
- Support GC shared code and add experimental GC setup #2231 (ekrich)
- Allow compilation of GCs on Windows #2264 (WojciechMazur)
- Add a configurable env flag that controls memory preallocation when G... #2205 (teodimoff)
- Respect doPrealloc if we are not on linux. #2304 (teodimoff)
- Incrementally commit memory in GC on Windows #2293 (WojciechMazur)
- Allow to use CommixGC on Windows #2294 (WojciechMazur)
- Allow to use Boehm GC on Windows #2369 (WojciechMazur)

Java Standard Library

- Fix #1590: Correct `j.i.DataInputStream` EOF handling & `skipBytes` #2104 (LeeTibbert)
- Fix #2137: Two defects in `j.n.PlainSocketImpl#read` #2140 (LeeTibbert)
- Simplify match case in `j.n.PlainSocketImpl` #2149 (LeeTibbert)
- Fix #2138: `j.i.DataInputStream#rebuffer` handles short reads correctly. #2142 (LeeTibbert)
- Fix #2143: `j.i.DataOutputStream` now does bulk writes where possible. #2152 (LeeTibbert)
- Fix #2164: Return `-1` at first EOF at `Base64.DecodingInputStream` #2165 (catap)
- Port `DataInputStream` `readUTF` code & all of `DataInputStreamTest` #2153 (LeeTibbert)
- Fix #2163: Do not use `regexp` in `j.u.Formatter` #2169 (WojciechMazur)
- Fix #2178: `j.l.{Double,Float}#isFinite` now handles NaN as specified. #2180 (LeeTibbert)
- Correct `j.i.DataInputStream` static `readUTF` implementation. #2172 (LeeTibbert)
- Reduce indirection for the implementation of `getClass` #2139 (densh)
- Fix #2063: Modify `j.u.Date#toString` to be Java 8 compliant. #2110 (LeeTibbert)
- Update `j.u.Formatter` to the latest version from `Scala.js`. #2196 (sjrd)
- Implement `j.nio.file.PosixException` object #2198 (LeeTibbert)
- Make it explicit that `Unix FileAttributeView` is provided by Posix calls. #2227 (LeeTibbert)
- Update & supersede PR #1609: Implement `java.nio` `UserPrincipal` infrastructure. #2244 (LeeTibbert)
- Homogenises numeric exception messages #2270 (ncordon)
- Fix #12283 - `j.nio.file.Files.copy()` now sets `lastModified` time correctly #2284 (LeeTibbert)
- Use `Unicode 13` for `Character` `toUpperCase` and `toLowerCase` #2103 (ekrich)
- Fix #2313: Defect in `j.n.InetAddress#createIPStringFromByteArray` #2348 (jchyb)
- A few improvements inside FS related API #2081 (catap)
- Correct `BufferedInputStream` `mark()` and `close()` behaviour #2354 (WojciechMazur)
- Implement `java.io` for Windows #2355 (WojciechMazur)
- Implement `java.nio` on Windows #2358 (WojciechMazur)
- Implement `java.lang.ProcessBuilder` for Windows #2360 (WojciechMazur)
- Implement `java.net` for Windows #2364 (WojciechMazur)
- Allow to use `java.util.zip` in Windows #2361 (WojciechMazur)
- Port `CVarArgList` implementation for Windows #2376 (WojciechMazur)
- Fix: #2135, segmentation-faults when working with file channels #2141 (WojciechMazur)
- Add `WeakReference` functionality and finalization-like operation via `WeakReferenceRegistry` #2368 (jchyb)
- Fix bug with `File.getCanonicalFile` leading to infinite recursion loop on Windows #2374 (WojciechMazur)
- Remove memory leak when reading user locale country and language data on Windows #2383 (WojciechMazur)
- Reimplement `Thread.sleep` to be OS-specific and multithreading safe #2373 (WojciechMazur)
- Fix #2396: `URI` `normalize` doesn't seem to work #2397 (jchyb)
- Update RE2 regex implementation #2402 (jchyb)

- Fix java.nio.file.Files inconsistencies between JVM and Native #2408 (jchyb)
- Match regex implementation of re2j version 1.3 #2407 (jchyb)
- Fix sn.regex parsing of OR expressions with a common prefix and \Q\E quoted expressions #2410 (jchyb)
- Add missing FileChannel functionality in javalib #2393 (jchyb)

Native library

- Fix 1664: Correct two memory access flaws in POSIX time #2160 (LeeTibbert)
- Fix #1665: posixlib localtime() now ensures tzset() has been called. #2269 (LeeTibbert)
- Support for POSIX signal #1362 (ekrich)
- posix.time strptime improvements & TimeTest macOS correction #2203 (LeeTibbert)
- Fix #1646: Implement posixlib sys/resource.scala #2193 (LeeTibbert)
- Makes dirent.c more readable and returns -1 only on empty dir #2221 (ncordon)
- Fix #2251: size of scalanative_sockaddr_in == struct sockaddr #2252 (LeeTibbert)
- Fix #1921, #2250: Evolve hacktoberfest Posix socket sendto() & recvfrom() #2258 (LeeTibbert)
- Improve POSIX pthread code quality #2307 (ekrich)
- Implement shutdown using atexit instead of c++ #1906 (lolgab)

Documentation

- Make corresponding Scala Native version evident in documentation. #2116 (LeeTibbert)
- Remove two instances of a hard-coded Scala Native version in documentation. #2121 (LeeTibbert)
- Generate latest documentation copyright year #2115 (LeeTibbert)
- Add missing commix reference in the documentation #2136 (JD557)
- add docs on profiling including how to make flamegraphs #2226 (bjornregnell)
- Fix #2267: Correct a broken link in docs/user/profiling.rst. #2268 (LeeTibbert)
- Explain how to create universal binary on macOS #2405 (catap)

Misc

- Update junit-runtime to match JUnit 4.13.2 #2331 (jchyb)
- Add a default signal handler to TestMain #2338 (jchyb)
- Restore scala.collection optimization overrides as patch files #2168 (WojciechMazur)
- Allow testing against Scala partest and JUnit tests #2154 (WojciechMazur)
- Change testing setup to allow javalib testing against the JVM #2335 (jchyb)
- Setup MiMa checks #2356 (WojciechMazur)
- Allow scala-native to be used with FreeBSD #2148 (alexdupre)

2.5.17 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-locale`s 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-locale`s, 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-locale 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 `@EnableReflectiveInstantiation` annotation, as well as its descendants. Annotated classes and modules, having a concrete implementation, can be accessed via the provided `scalnative.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.

```
package x.y.z

@EnableReflectiveInstantiation
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 is now taken into account when generating methods,
- Constructors are no longer treated as 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-compiled `ArrayIndexOutOfBoundsException`,
- Fix bug with `BufferedInputStream.read()` for values bigger than `0x7f`,
- `Files.walk` accepts non-directory files,
- Improved IEEE754 specification compliance when parsing strings,
- Fixed infinite loop in `java.io.RandomAccessFile.readLine`,
- Added multiple missing `java.lib` 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      Lee Tibbert
 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
```

(continues on next page)

(continued from previous page)

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 exist #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.{string,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)

2.5.18 0.4.0-M2 (May 23, 2019)

Read release notes for 0.4.0-M2 on [GitHub](#).

2.5.19 0.4.0-M1 (May 23, 2019)

Read release notes for 0.4.0-M1 on [GitHub](#).

2.5.20 0.3.9 (Apr 23, 2019)

Read release notes for 0.3.9 on [GitHub](#).

2.5.21 0.3.8 (Jul 16, 2018)

Read release notes for 0.3.8 on [GitHub](#).

2.5.22 0.3.7 (Mar 29, 2018)

Read release notes for 0.3.7 on [GitHub](#).

2.5.23 0.3.6 (Dec 12, 2017)

Read release notes for 0.3.6 on [GitHub](#).

2.5.24 0.3.5 (Dec 12, 2017)

Read release notes for 0.3.5 on [GitHub](#).

2.5.25 0.3.4 (Dec 12, 2017)

Read release notes for 0.3.4 on [GitHub](#).

2.5.26 0.3.3 (Sep 7, 2017)

Read release notes for 0.3.3 on [GitHub](#).

2.5.27 0.3.2 (Aug 8, 2017)

Read release notes for 0.3.2 on [GitHub](#).

2.5.28 0.3.1 (June 29, 2017)

Read release notes for 0.3.1 on [GitHub](#).

2.5.29 0.3.0 (June 15, 2017)

Read release notes for 0.3.0 on [GitHub](#).

2.5.30 0.2.1 (April 27, 2017)

Read release notes for 0.2.1 on GitHub.

2.5.31 0.2.0 (April 26, 2017)

Read release notes for 0.2.0 on GitHub.

2.5.32 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.