JOHANNES KEPLER
UNIVERSITY LINZ

Author
Christoph Pichler, BSc

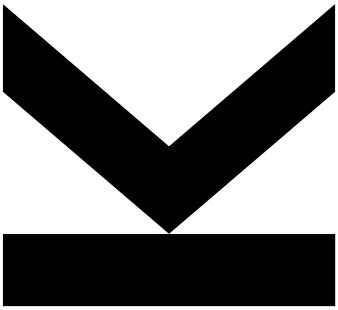
Submission
Institute for System Software (SSW)

Thesis Supervisor
a.Univ.-Prof. Dipl.-Ing. Dr. Herbert Prähofer

Assistant Thesis Supervisor
DI Dr. Roland Schatz

March 2022

RUNNING SWIFT-COMPILED LLVM IR CODE ON SULONG



Master's Thesis

to confer the academic degree of

Diplom-Ingenieur

in the Master's Program

Computer Science

JOHANNES KEPLER UNIVERSITY LINZ
Altenberger Straße 69
4040 Linz, Austria
jku.at

Abstract

Being currently on the 8th place of the most-used programming languages, Swift has been gaining its impact during the last few years and is used more and more often in software projects. As larger projects come with a higher effort of building and maintaining, an appropriate software architecture can reduce necessary resources by multiples thereof. Such an architecture where features including highly optimizing compilation, cross-language interoperability or monitoring and debugging tools are available is GraalVM. However, Swift code cannot be run on this architecture directly, as there exists no language implementation within GraalVM for Swift.

The approach described in this thesis is to run Swift on GraalVM by taking an intermediate step in between: First, the Swift compiler is used for LLVM IR generation from Swift code. Then, based on this LLVM code, Sulong (the LLVM runtime of GraalVM) can be extended and adjusted. By that, it is shown how Swift code can not only be run on GraalVM, but also interact with code of other GraalVM-supported languages. Furthermore, this thesis shows a proof of concept how to approach full Swift support, since it is still limited by parts of the Swift standard library.

Kurzfassung

Swift, derzeit auf Platz 8 der meistverwendeten Programmiersprachen, hat in den letzten Jahren an Einfluss gewonnen und wird immer häufiger in Softwareprojekten eingesetzt. Da bei großen Projekten sowohl die Erstellung als auch die Wartung viel Aufwand bedeutet, kann eine geeignete Software-Architektur die erforderlichen Ressourcen um ein Vielfaches reduzieren. Eine solche Architektur, in der Funktionen wie u.a. hochoptimiertes Kompilieren, sprachübergreifende Interoperabilität und Monitoring- und Debugging-Werkzeuge verfügbar sind, ist GraalVM. Nachdem es allerdings derzeit keine existierende Sprachimplementierung innerhalb von GraalVM für Swift gibt, kann Swift-Code nicht direkt auf dieser Architektur ausgeführt werden.

Der in dieser Arbeit beschriebene Ansatz ist es, über einen Zwischenschritt Swift auf GraalVM laufen zu lassen: Der Swift-Compiler wird zunächst für das Generieren von LLVM-Code aus Swift-Code verwendet. Anschließend können wir Sulong, die LLVM-Umgebung in GraalVM, basierend auf diesem LLVM-Code anpassen und erweitern. Dadurch wird gezeigt, wie Swift-Code auf GraalVM/Sulong ausgeführt werden und mit Code anderer von GraalVM unterstützten Sprachen interagieren kann. Außerdem beschreibt diese Arbeit, wie man volle Swift-Kompatibilität erreichen kann, da diese aktuell noch durch Teile der Swift-Standardbibliothek begrenzt ist.

Contents

Chapter 1

Introduction

The amount of software running in the world is increasing steadily. As software development also changes over time and is also dependent on the contexts of the tasks, different programming languages exist to write those projects. From the huge variety of available programming languages, only few achieve world-wide popularity and wide usage. The decision in which language to write a software project is a very important one and cannot be changed at a later point in time without enormous effort. Thus, languages that achieve high popularity are bound to have a very big impact in programming in the long run: Even if they are not considered for new software projects, the already existing amount of code will run and has to be maintained for a couple of years at least, reaching up to decades. Therefore, supporting an important and widely popular language is a huge benefit for a software system and comes with a big variety of interactions with existing and reusable code.

Recently, the programming language *Swift* [1] has become one of the fastest growing languages [2] and reached the top 10 of mostly-used programming languages worldwide [3, 4]. This leads to a very high propagation on the one hand of Swift code and the language itself. On the other hand, software systems that support Swift can make use of already existing code. Especially, recent products for Apple platforms (e.g. applications on iOS smart phones) are written in Swift. As a consequence, the importance of Swift is currently increasing, and the Swift programming language is going to have an impact on software development as long as the recently written applications and software products are going to stay in use. Therefore, having a system which can handle and execute Swift code will stay important within the next decade(s).

Looking at software systems and architectures, *GraalVM* [5] shows another important aspect of software development: Besides efficient compilation and interpretation by aggressively optimizing compilers and interpreters, it is also designed to easily provide interoperability across different programming languages. This cross-language interoperability (or polyglot programming, as it is also called) avoids rewriting/translation of code from one language into another, and thus becomes a powerful tool for combining and maintaining existing software systems.

The goal of this thesis is to integrate Swift into GraalVM, i.e. to enable that features and tools provided by GraalVM can also be used for Swift code.

1.1 Technological background and related work

To motivate the work of this thesis, the technological background of important components is shortly explained in the following.

1.1.1 Swift and LLVM

As mentioned above, Swift is a programming language, which means there exist tools to compile Swift source code [1]. Independent of the compilation output, Swift code is always compiled to LLVM as an intermediate representation. Thus, the Swift compiler can also explicitly produce LLVM from Swift code. In general (not only for Swift-compiled code), LLVM code is not human-written, but the (intermediate) result of compilation of certain languages [6].

To access existing functions from Swift code in languages other than Swift, cross-language interoperability is necessary. Currently, Swift only offers an interoperability mode with the language *Objective-C*, Swift's predecessor, and a C variant. Although this interoperability mode increases code re-usage, it is only possible to connect Swift code with C code or variants thereof. Code written in languages other than C variants (e.g. Java, JavaScript, or Python) do not have an existing interoperability mode with Swift yet.

1.1.2 GraalVM architecture

The GraalVM architecture consists of several components. Based on a Java Virtual Machine, it consists of a heavily optimizing compiler, and the ability to handle different programming languages via the Truffle framework [7].

The Truffle framework [8] is a language implementation framework which enables developers to implement AST[1] interpreters for programming languages [7]. Languages, for which a Truffle framework implementation exists, can also be easily used to perform cross-language interoperability between each other when they are executed within GraalVM.

Sulong is one language runtime in the GraalVM architecture and uses the Truffle framework for LLVM code execution [9]. Since LLVM is an intermediate representation, Sulong can be used to run code from languages that can be compiled to LLVM, e.g. C or C++. In contrast to a "direct" execution of a compiled, binary file of e.g. C/C++ code, execution via Sulong offers features of GraalVM, such as cross-language interoperability.

1.2 Approach

The approach to overcome the previously mentioned issue of Swift compatibility (only compatible with C/C++) is to combine the Swift language with the concept of GraalVM, i.e. to find a way how to execute Swift code within this GraalVM ecosystem. Running Swift code on top of GraalVM has the advantages of being able to use components and features including the highly-optimizing Graal compiler, or GraalVM's cross-language interoperability concept.

To avoid having to implement a completely new language runtime on GraalVM for Swift, LLVM can be used as an intermediate step: The Swift compiler produces LLVM bitcode as intermediate representation. This representation can be interpreted by Sulong on GraalVM. However, directly interpreting Swift-emitted LLVM code via Sulong is not trivial:

- Language runtime: As most modern languages, Swift also comes with a standard library where important functionality, which is not part of the language core, is specified and

[1] abstract syntax tree

written. While executing/processing Swift-compiled LLVM code by an LLVM interpreter or a further compiler, Swift functionality might be used from such a library (which is not part of the original source code in Swift). During execution in GraalVM, those Swift dependencies have to be provided for correct execution of code.

- Interoperability: From a user perspective, the goal is to achieve cross-language interoperability between Swift and a foreign language. However, from a GraalVM point-of-view, only interoperability between LLVM and a foreign language is possible. Therefore, some "translation" work between Swift and LLVM has to be implemented in order to support cross-language interoperability between a foreign language and Swift on GraalVM.

1.3 Thesis structure

This thesis is structured as follows:
In the next two sections (sections 2 and 3), the focus is laid on features and concepts of LLVM and the Swift language, respectively. After that, section 4 gives an introduction to GraalVM, Sulong and Truffle and their interaction. The following section (section 5) describes how to execute Swift-compiled LLVM IR code on Sulong, including current dependencies and limits thereof. Then, the next section (section 6) addresses necessary extensions to Sulong to make the Truffle interoperability concept work for Swift-compiled LLVM code. Finally, in section 7, an overall view is shown by providing a case study. The last section (section 8) concludes this thesis and discusses future work.

Chapter 2

LLVM

This section addresses LLVM with a focus on the compilation process(es), the different representation formats, and examples how LLVM supports higher-level language concepts by only using its lower-level representations.

In general, LLVM is a cross-platform compilation framework, i.e. a set of compilers and toolchains. It is built around its own intermediate representation (=IR) [6], which is described in more detail below.

2.1 LLVM representations

2.1.1 LLVM IR

LLVM's intermediate representation (IR) is the core of the LLVM framework and an assembly-like and target-independent language. It is defined by its instruction set. A short sample snippet of LLVM IR can be seen in Listing 2.2, which has been compiled from the C code in listing 2.1.

Basic features and principles of LLVM IR (cf. Listing 2.2) include:

- strong and static typing

- organized as a control flow graph (CFG), i.e. basic blocks (line ranges 9–12, 15–19, 22) with conditional and unconditional jumps only at the end of a block (lines 12, 19, 22)

- single static assignment (SSA – cf. lines 11, 15 or 16)

- (due to SSA) local variable names are dropped (and only kept in the debug information, cf. section 2.1.3)

- based on primitive types (i1, i32, ...), structured types (line 6) and pointers (lines 9, 10, ...)

```
1  struct Point {
2          int x;
3          int y;
4  };
5
6  void mirror_point(struct Point *p) {
7          if(!p) {return;}
8          p->y = - p->y;
9  }
```

Listing 2.1: C code (point mirror example)

```
6  %struct.Point = type { i32, i32 }
7
8  ; Function Attrs: nofree norecurse nounwind uwtable willreturn
9  define dso_local void @mirror_point(%struct.Point* %0) local_unnamed_addr #0 !dbg !9 {
10   call void @llvm.dbg.value(metadata %struct.Point* %0, metadata !20, metadata !DIExpression()), !dbg !21
11   %2 = icmp eq %struct.Point* %0, null, !dbg !22
12   br i1 %2, label %7, label %3, !dbg !24
13
14 3:                                                ; preds = %1
15   %4 = getelementptr inbounds %struct.Point, %struct.Point* %0, i64 0, i32 1, !dbg !25
16   %5 = load i32, i32* %4, align 4, !dbg !25, !tbaa !26
17   %6 = sub nsw i32 0, %5, !dbg !31
18   store i32 %6, i32* %4, align 4, !dbg !32, !tbaa !26
19   br label %7, !dbg !33
20
21 7:                                                ; preds = %3, %1
22   ret void, !dbg !33
23 }
```

Listing 2.2: LLVM IR (point mirror example), compiled from C code in listing 2.1

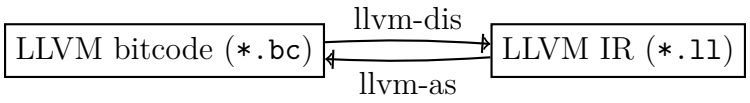


Figure 2.1: conversion between LLVM IR and LLVM bitcode

2.1.2 LLVM bitcode

LLVM bitcode is the "machine code" version of LLVM IR, thus it is more compact than the assembly-like IR and not human-readable. Since LLVM IR and LLVM bitcode only differ in the file format (human-readable vs. machine-readable) but not in the content, either of these two file formats can be converted into the other one by using the commands `llvm-as` and `llvm-dis` (cf. figure 2.1) [10].

Because of this "content equality", both formats will not always be clearly distinguished in this work – LLVM IR is shown (readability reasons) in examples in the written thesis, while for execution in Sulong, the bitcode format is used.

2.1.3 LLVM debug information

LLVM debug information is optional and additional information which can be included into LLVM IR.

As an intermediate representation, general LLVM IR does not contain variable names, type names or information about the position of corresponding code in the source file, since these data are not needed for code execution. However, for debugging the original source-level code (e.g. written in C, like in listing 2.1), exactly this type of information is needed. Thus, as an optional part of an LLVM IR file, these data can be attached to the LLVM IR as so-called *LLVM debug information*, such that the debugger can display e.g. debug values, types, and variables by name and location [11].

An incomplete excerpt of LLVM debug information (corresponding to the code example of listings 2.1 and 2.2) can be seen in listing 2.3.

To include debug information into the emitted LLVM IR/bitcode, the clang (C $\rightarrow$ LLVM)

```
!0 = distinct !DICompileUnit(language: DW_LANG_C99, file: !1, producer: "clang␣version␣12.0.1␣(GraalVM.org
    ␣llvmorg-12.0.1-3-g6e0a5672bc-bgf11ed69a5a␣6e0a5672bc058d882dce3d56f90b72b64a6870d7)", isOptimized:
    true, runtimeVersion: 0, emissionKind: FullDebug, enums: !2, splitDebugInlining: false, nameTableKind
    : None)
!1 = !DIFile(filename: "/home/christoph/master-thesis/code/llvm/irExample.c", directory: "/home/christoph/
    sulong-dev/graal/sulong", checksumkind: CSK_MD5, checksum: "6bff4ed39be1564cbb7bfd1065ef083a")
!2 = !{}
!3 = !{i32 7, !"Dwarf␣Version", i32 5}
!13 = !DIDerivedType(tag: DW_TAG_pointer_type, baseType: !14, size: 64)
!14 = distinct !DICompositeType(tag: DW_TAG_structure_type, name: "Point", file: !10, line: 1, size: 64,
    elements: !15)
!15 = !{!16, !18}
!16 = !DIDerivedType(tag: DW_TAG_member, name: "x", scope: !14, file: !10, line: 2, baseType: !17, size:
    32)
!17 = !DIBasicType(name: "int", size: 32, encoding: DW_ATE_signed)
!22 = !DILocation(line: 7, column: 6, scope: !23)
!23 = distinct !DILexicalBlock(scope: !9, file: !10, line: 7, column: 5)
```

Listing 2.3: LLVM IR debug information (belonging to listings 2.1 and 2.2)

compiler provides the `-g` flag, which can be set for compilation.

2.2 Compilation process

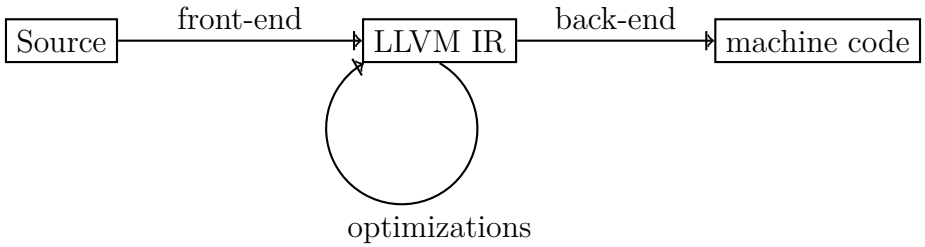


Figure 2.2: simple LLVM compilation process

A simple abstract LLVM compilation process [6] can be seen in Figure 2.2 and is described in the following:

- **front-end compilation**: Initial starting point is source language code of a supported language (e.g. C/C++). This source language code is then compiled down to LLVM IR with the corresponding compiler (clang/clang++ for C/C++ respectively).

- **optimizations**: As LLVM IR is a low-level representation, static optimizations can be performed, such as elimination of common sub expressions and dead code. These optimizations are not limited to LLVM IR, but are usually also applied during front-end and

back-end compilation.

- **back-end compilation**: At the end of the compilation process, the resulting LLVM IR code is given to a back-end compiler, which then generates (binary) machine code. The compiler and the exact binary format depend on the architecture of the target device. Instead of back-end compilation to a binary format, there also exist other output formats and execution modes, such as symbolic execution [12].

2.3 Breaking down source-language concepts to the LLVM instruction set

As mentioned above in section 2.2, LLVM front-end compilers map source code of higher-level languages (e.g. C, C++, Swift) to lower-level LLVM IR. Since there are operations and language concepts in the source languages which are not directly a part of the LLVM instruction set, those compilers have to break down specific operations. Most of these language concept changes, such as name mangling or receiver methods, will be discussed in the sections about interoperability (cf. sections 4.3.1 and 6) directly. However, since the basic principle of dynamic binding requires further knowledge, it will be explained in the following.

2.3.1 Static and dynamic binding

If a method, for which multiple implementations (with the same signature) exist, is invoked, then the execution semantics of the language define which of these implementations is executed. There are two common strategies, called *static* and *dynamic binding*.

In the following, it is assumed that a `Child` class inherits from a `Parent` class, and both classes implement the method `foo`. Then, given that `obj` is assignable to a `Parent` variable, `obj.foo()` requires a strategy to select the appropriate implementation. For cases like these, Table 2.1 shows which implementation is called, depending on the binding on the one hand, and based on the assignment of variable `obj` on the other hand.

variable assignment	**static binding** (C++)	**dynamic binding** (C++, Swift)
`Parent obj ← Parent()`	`Parent::foo`	`Parent::foo`
`Child obj ← Child()`	`Child::foo`	`Child::foo`
`Parent obj ← Child()`	`Parent::foo`	`Child::foo`

Table 2.1: Method resolving strategies: Static and dynamic binding (calling `obj.foo()`)

It can be seen that static binding follows the static type of the receiver, i.e. the container variable type where the object is currently stored. Since the method to be invoked is already known at compile time (by a simple lookup on the static receiver type), static binding is a cheap (easy and fast) way of resolving. Dynamic binding, however, always invokes the implementation of the current dynamic type of the receiver object (also cf. table 2.1). As a consequence, the implementation lookup can only happen at runtime, which makes dynamic binding more expensive in terms of time and complexity.

As a low-level IR, LLVM only supports static binding. This leads to a problem if higher-level languages with the semantics of dynamic binding are compiled down to LLVM. Thus, in order to keep execution semantics of these higher-level languages correct, their (implicit) dynamic binding lookups have to be converted to explicit LLVM instructions. In the following, this conversion is shown with a case study of C++/LLVM code.

2.3.2 Case study for resolution of dynamic binding: C++/LLVM

As an example, two types `A` and `B` are considered, where `B` inherits from `A`. Moreover, `A` implements three methods – `foo`, `bar` and `baz`. B also implements `foo` and `bar` (with its own implementations) and inherits `A`'s implementation of `baz`. In this example, calls of `foo` are statically bound, whereas `bar` and `baz` are resolved dynamically.

The left-hand side of Figure 2.3 shows the C++ view of the described example. By default, C++ methods are statically resolved – unless they are declared `virtual` in at least the base class, then dynamic binding is applied. As `bar` and `baz` are declared `virtual` in Figure 2.3, `foo` is statically resolved, while `bar` and `baz` are resolved dynamically.

As mentioned above, static binding can be applied at compile time and is supported in LLVM. Therefore, invocations of `foo` stay unchanged and can be hard-coded into LLVM already.

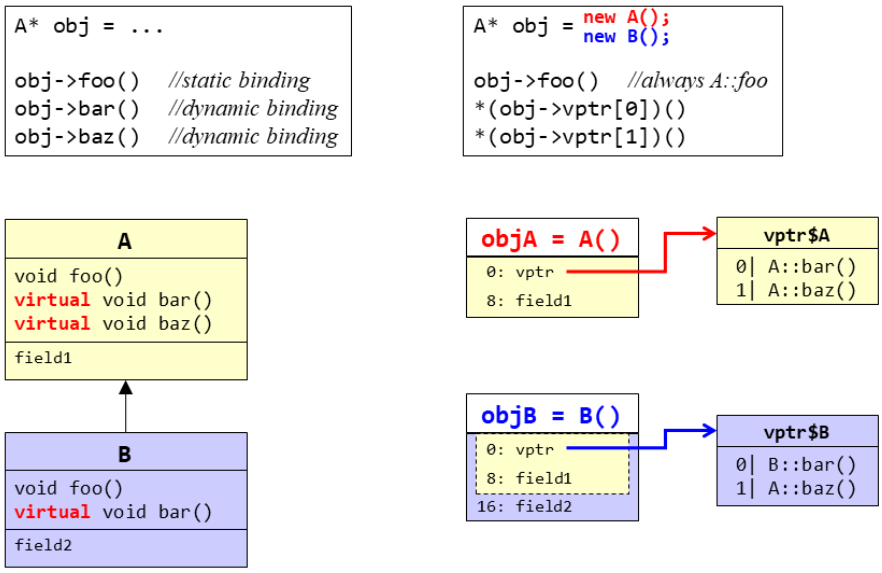


Figure 2.3: Method resolving: C++ view (left) and LLVM view (right)

For dynamic binding, the concrete implementation has to be selected depending on the dynamic type of the receiver. To apply this correctly, the following actions are taken by the LLVM front-end compiler:

- **vtable**: For every class, a so-called *vtable* (virtual method table) is created, which stores an index and a function pointer per entry. Every dynamically bound method where the current class type can be the receiver gets an index assigned, and is entered into this virtual method table. It is important that overridden methods (=methods with the same signature, but different implementations due to different classes) get the same index.
  As an example (cf. Figure 2.3), `bar` and `baz` are inserted into `A`'s vtable. As `B` inherits from `A`, these two methods are also added to `B`'s vtable with the same indices as their corresponding entries in `A`'s vtable. If implementations exist for the same type (e.g. `B::bar` for `B`), they are inserted directly. Otherwise (e.g. for `baz`), the implementation of an appropriate superclass method (i.e. `A::baz`) is taken.

- **vptr**: Instead of treating dynamically bound methods as methods of their type, the compiler adds a field for the `vptr` value. This `vptr` field is the first field of every type (=always located at index 0) and can store a pointer to a vtable. Usually, this pointer is implicitly set in the constructor. Thus, `vptr` always points to the vtable of the dynamic type of the current object.

- **call**: Every call of a dynamically bound method is replaced by a different structure using its virtual method index (cf. Figure 2.3). Instead of calling the function directly, the `vptr` is dereferenced to get the vtable of the current dynamic type. With the corresponding method offset (which is known at compile time), the function pointer of the method to be invoked can be found.
  Since this technique uses actual field data instead of static type information, selecting the method is independent of static type information, which is the necessity for dynamic binding.

By translating every dynamically bound method call to the procedure mentioned above and shown in Figure 2.3, dynamic binding can be also applied/simulated in a language with static binding only, as it is the case for LLVM.

Chapter 3

The Swift programming language

In this section, the language Swift is introduced together with its features and its compilation process.

Figure 3.1: Swift logo [13]

Swift is a multi-paradigm and general-purpose programming language whose first version was released in 2014. It is developed by Apple Inc.[1] and the open source community. Since the predecessor of Swift is Objective-C, a `C++`-like and thus unsafe language, the main aim of the Swift designers has been to create a fast and safe language which is compatible to Objective-C [1].

Although Swift has been originally designed for Apple platforms only, the current major version (Swift 5) is also available for Linux (Ubuntu, CentOS, Amazon Linux) and Windows systems [1].

3.1 Features and concepts

As the focus of this thesis is not to list all concepts and features of the Swift programming language itself, only a small selection will be provided below. A short part of a Swift program (to take a glimpse how the language looks like) can be seen in Listing 3.1, which shows a sample implementation for a struct data type for complex numbers:

[1] https://www.apple.com/

```
1  struct Complex {
2          var re: Double //struct members/fields
3          var im: Double
4
5          func abs() -> Double {
6                  let a = self.re*self.re + self.im*self.im
7                  return a.squareRoot()
8          }
9
10         func conjugate() -> Complex {
11                 return Complex(re: self.re, im: -self.im) //default constructor
12         }
13
14         func getAsTuple() -> (Double, Double) { //tuple types as return types
15                 return (self.re, self.im)
16         }
17
18         func addComplex(c: Complex) -> Complex { //named parameters
19                 return Complex(re: self.re+c.re, im: self.im+c.im)
20         }
21 }
```

Listing 3.1: Example of a Swift type representing complex numbers

- Fields/struct members: denoted by the `var` keyword, fields are listed with their corresponding (static) type.

- Methods: In Swift, methods are also supported for structs as the receiver type (not class types only) and are denoted by the keyword `func`. Listing 3.1 contains the methods (`abs`, `conjugate`, `getAsTuple`, `addComplex`).

- Tuple types/multiple return values: Tuple types (supported in Swift) can be used to return multiple values from a function (e.g. `getAsTuple` in Listing 3.1).

- Named parameters: Swift has named parameters, i.e. the names of the formal parameters also have to be provided by the programmer for the actual parameters (can be seen in Listing 3.1 when the default constructor of `Complex` is called).

The following features are also supported by Swift (only a selection), but will not be further shown in detail [1]:

- Object orientation principles: classes, (single) inheritance, dynamic binding (also for member fields), information hiding

- Error handling (including exception control flow)

- Generics
- Fast and concise iteration over collections and ranges
- Functional programming patterns, e.g. map and filter
- Closures unified with function pointers (also known as *lambdas* in other languages)

As mentioned before, Swift has been designed to be a safe Objective-C successor. Thus, the following safety mechanisms are present in Swift [1]:

- Automatic memory management: Memory for e.g. objects on the heap does not have to be requested manually. However, in contrast to many other languages with automatic memory management, Swift does not make use of a garbage collector. Instead, automated reference counting is used to find objects which are not referenced any more.
- Variable initialization checks: Every read access to a variable x requires that x has to be initialized beforehand.
- Over-/Underflow checks: Values are checked at runtime if they are within a valid value range. In Swift, these checks are performed for array indices (like in Java), but also for arithmetic operations.
- Static and strict typing: Type checks are performed at compile time already by type inference. Except for a special kind of data types (`UnsafePointer` and its sub-types, which are necessary for Objective-C interoperability, cf. section 3.4), pointers are not allowed.
- Non-nullable types as standard: variables of any type cannot be set to `nil` (=Swift's null reference/empty value). Instead, every type `T` has its corresponding optional type `T?`, which accepts every value of T, but also `nil`.

3.2 Swift compiler

3.2.1 Basic compilation process

The compilation process from a Swift file to machine code is sketched in Figure 3.2 and is described in more detail below [14].

Figure 3.2: Swift compilation process

- Assuming the **Swift** code has been written into one or more `.swift` files, the Swift frontend (`swiftc`) performs recursive decent parsing and creates an
- abstract syntax tree (**AST**). This graph is then used for type and semantic checks. Having a checked AST, the Swift frontend starts to generate code in the form of the so-called
- Swift intermediate language (**SIL**). This Swift-specific and high-level representation is used for dataflow diagnostics and further, high-level optimizations. After that,
- **LLVM bitcode** is generated, which is a more low-level intermediate representation. Here, the task of the Swift compiler, which serves here as an LLVM front-end, has ended, since further (low-level) optimizations and the transformation to **machine code** is then the task of an LLVM back-end.

3.2.2 Compiling multiple Swift files

Compiled single Swift files can be used as both, an executable directly and as a library. For multiple Swift files, the output (executable or library) depends on the content [15]:

- **Executable file**: If an executable file shall be created, then there has to be exactly one file called `main.swift` containing a "main method", i.e. code at the top/global declaration level. Specifying a module name is optional.

- **Library**: For creating a library, a main method must not be present in any of the files. Instead, during compilation, a module name has to be specified (which is the name where the containing functions can be found by the linker afterwards).

3.2.3 Swift compiler output: binary and LLVM

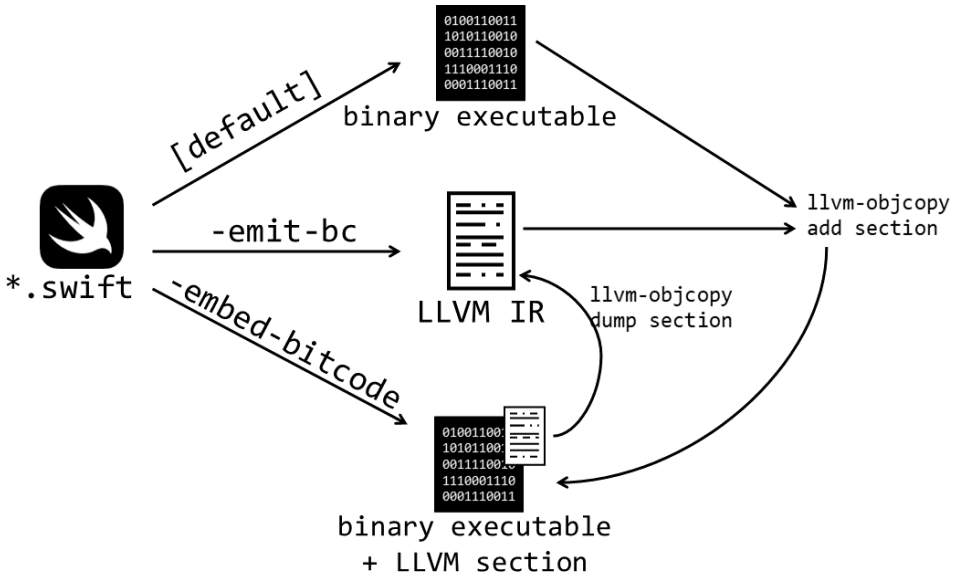


Figure 3.3: Swift compiler options for LLVM bitcode

As the standard Swift compiler is also a fork of the LLVM project, it usually creates a binary executable object file from the produced LLVM IR. By setting a flag as sketched in Figure 3.3, however, the standard compiler can also include the LLVM bitcode as a section into the executable object file (`-embed-bitcode`), or emit the LLVM bitcode into an extra output file (`-emit-bc`) [15].

Although the output format of the compilation process can be specified in advance, LLVM bitcode can be included and extracted also after compiling via the `llvm-objcopy` tool. The necessary commands for including and extracting (also shown in Figure 3.3) are `add-section` and `dump-section`, respectively.

Dealing with LLVM sections will be further discussed in sections 5 and 7 below.

3.3 Swift standard library

A standard library is a collection of code which is shipped with the runtime environment of a certain language. In contrast to the "computational core" of a language, a standard library usually contains code of architecture- or system-dependent behavior (e.g. standard input/output). Most of the popular programming languages make use of such a standard library.

Swift has its own standard library, which is open source [15], and included in official compiler and integrated development environment distributions [16, 17]. The Swift standard library consists of at least two major parts: the core standard library and the runtime implementation.

- As the name suggests, the **Swift core library** contains the core of the standard library, including type definitions of functions, data types and protocols. It is written in Swift (and Swift-GYB, cf. below) itself.

- The **runtime** implementation is written in C++ and Objective-C and deals with dynamic operations, e.g. memory management (reference counting) or casting operations.

- Further optional parts are platform-dependent components (e.g. an SDK overlay for Apple platforms).

As mentioned above, not all of the core library is written in Swift itself. Some parts are written in Swift GYB, which is a Swift generation language [15]. The GYB compiler takes GYB code and then automatically generates Swift code (GYB = generate your boilerplate). GYB code is e.g. used for defining the different Swift integer types: Written once, Swift code can be generated for each integer type width automatically.

3.4 Interoperability between Swift and C/Objective-C/C++

As Swift is a successor of the C variant *Objective-C*, Swift supports interoperability with Objective-C code [18]. The key principle of Swift's existing Objective-C interoperability is

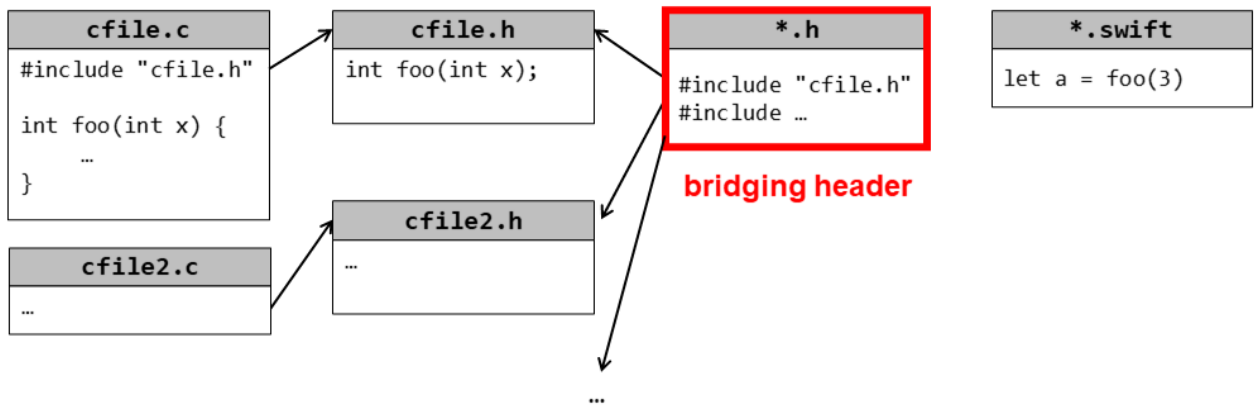


Figure 3.4: Principle and concept of Swift-C/C++ interoperability

the so-called bridging header – a file containing all Objective-C functions that should be reachable from Swift. As every C header is also an Objective-C header, interoperability between Swift and C code can be achieved by applying the same principle (i.e. treating C code as Objective-C code).

Figure 3.4 shows an example:

- (Objective-)C code and its signatures is written normally into code (`*.c`) and header (`*.h`) files (left-hand side of Figure 3.4).
- For the bridging header (red part in Figure 3.4), an extra file is created. By including every (Objective-)C header file which contains a function that should be reachable from Swift, it is ensured that the bridging header contains all necessary function signatures.
- In the Swift code (right-hand side of Figure 3.4), the corresponding (Objective-)C functions are visible as-is. For every data type in (Objective-)C, there exists its corresponding Swift data type [19].
- To compile the Swift files, the bridging header argument of the Swift compiler (`-import-objc-header`) has to be set to the path of the bridging header file.

By following that principle, (Objective-)C and Swift code can interact easily.

Since C++ is also a variant of C, interoperability with C++ code is a further concept to investigate. Recently (March 2022), Swift announced the formation of an official workgroup for advancing interoperability between Swift and C++ [20, 21]. The current development seems that C++ code cannot be imported as it can be done for C – instead, all C++ code has to be exported as an extra Swift module [22].

Chapter 4

Sulong and the GraalVM architecture

In this section, the basic GraalVM architecture and its major components are described. They are shown in figure 4.1 and are then described in the following.

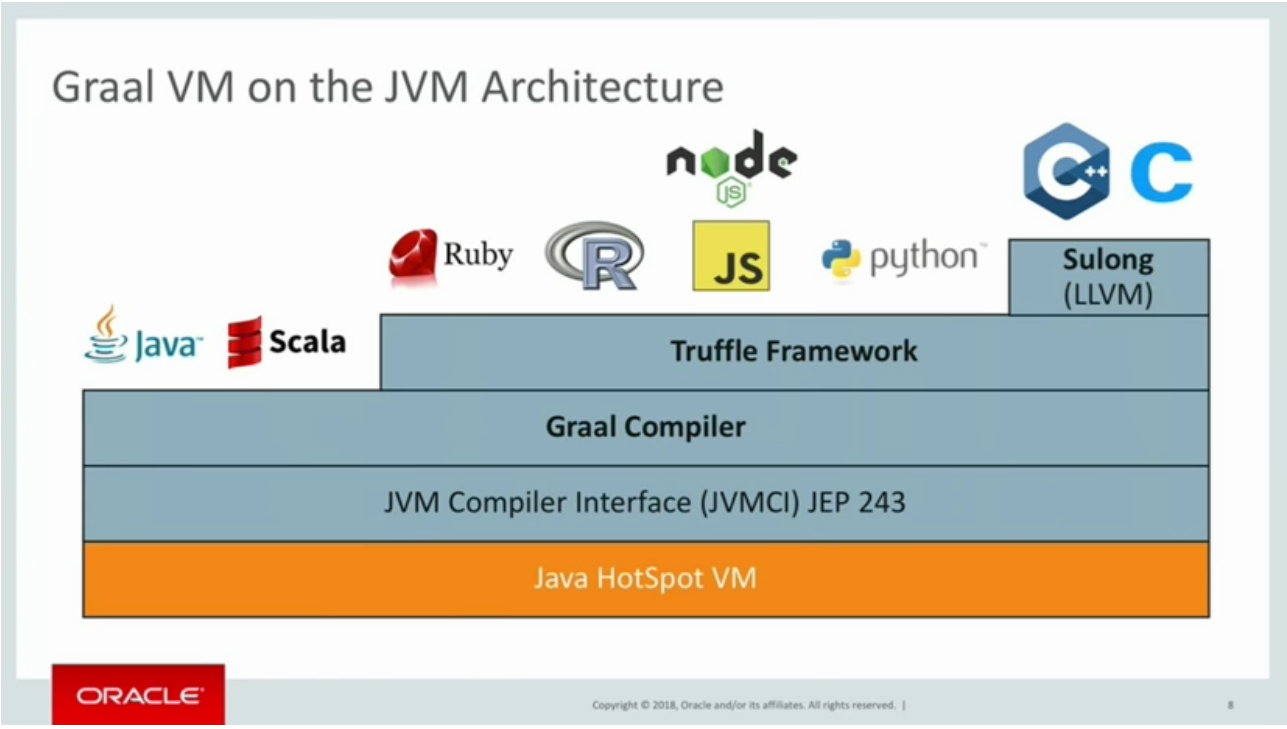


Figure 4.1: GraalVM Architecture [23]

4.1 GraalVM

GraalVM [7, 5], proposed in 2013 and first released in 2019, is a Java virtual machine with the following key features:

- Based on Java Hotspot VM/OpenJDK, developed by Oracle Inc.
- High-performance VM: via the JVM compiler interface (JVMCI), the *Graal compiler* can connect to the Hotspot VM. This compiler uses several dynamic compilation principles (e.g. speculation, partial escape analysis, partial evaluation, etc.) to heavily optimize the code. Besides just-in-time (JIT), also ahead-of-time (AOT) compilation can be applied to get native binaries with a fast start-up.
- Polyglot programming, which is described in more detail in section 4.2. The concrete operations still have to be covered by the corresponding language runtime implementations, but Truffle provides a message API to connect end-user code of different (Truffle) languages.
- Advanced tools: Written code and resource consumption can be easily analyzed. Code cannot only be optimized across different languages, there is also a debugger available. Another tool is the profiler, which allows the monitoring of code and execution behavior in different ways.

As GraalVM is based on a JVM, programming languages that can be compiled down to Java Bytecode (like Java or Scala) can be directly used with the Graal compiler. For implementing other programming languages, the Truffle framework serves as a basis (cf. figure 4.1) [23].

4.2 Truffle

Truffle [24, 8] is a language implementation framework which enables developers to implement interpreters for programming languages [7], where usually an AST (abstract syntax tree) or bytecode is used as a source. It is based on the Graal compiler, thus the data structures produced by Truffle (and the corresponding language implementation) can be compiled directly by

the Graal compiler (see Figure 4.1). Languages, for which Truffle implementations exist(*Truffle languages*) are for example Ruby, Javascript or Python.

The most important properties and tools of the Truffle framework are:

- **Optimization**: In the Graal compiler, dynamic optimization techniques are used for efficient compilation: The Truffle interpreter for a certain language is converted to a compiler using the principle of so-called first Futamura projection [25]. By that, source code data structures such as an AST or bytecode can be directly compiled to machine code in a very efficient way.

- **Tools**: Being an AST interpreter framework, Truffle also contains a framework for implementing widely-used code developing tools, such as debugging or monitoring software. As mentioned in the next paragraph, this is especially interesting for new languages.

- **Rapid prototyping of new languages**: Before new languages can be adopted, many things have to be manually implemented, including an optimizing compiler, debuggers, etc. When implementing a new language with the Truffle framework, preexisting tools and the optimizing Graal compiler can be leveraged with little additional effort.

- **Tool extensions**: If new developer tools (debug support, monitoring, ...) arise, they only have to be implemented once in Truffle and can then be used by all Truffle languages easily.

- **Polyglot programming**: Different Truffle languages can interact with each other without performance disadvantages. This cross-language concept is explained in more detail in the next section, 4.2.1.

4.2.1 Cross-language interoperability

As mentioned before, Truffle also supports cross-language interoperability between different languages. This includes accessing array elements of foreign arrays, calling methods of foreign objects, passing foreign objects as parameters, writing member fields of foreign objects etc. [7, 8].

An example of such an interoperability call is shown in the following. The first listing (4.1) shows a simple JavaScript function in a file "InteropExample.js". The second listing (4.2) shows how this JavaScript code can be called from Python.

```
1 function square(x) {
2         return x*x;
3 }
```

Listing 4.1: JavaScript code file "InteropExample.js"

```
1 import polyglot
2 # get file reference
3 llvmFile = polyglot.eval(language = "js", path = "InteropExample.js");
4 # call foreign function
5 res = llvmFile.square(3); # res==9
```

Listing 4.2: Python code, calling foreign function of Listing 4.1

From the Truffle perspective, this concept is implemented via interop message passing: Corresponding classes in the Truffle framework and its language implementations can send and accept pre-defined messages, such as `readMember(name)`, `writeMember(name, value)`, `invokeMember(name, args[])`, `readArrayElement(index)`, etc. Results of cross-language function/method calls can then again be inspected via these messages.

In the Graal compiler, the interoperability access node of the created multilingual Truffle AST can then be optimized away, such that the cross-language operation does not cause any additional performance drawbacks.

4.3 Sulong

Sulong[1] is the implementation of the LLVM bitcode "language" on top of the Truffle framework, and thus serves as the LLVM runtime in the GraalVM architecture [26, 27]. By being based on Truffle, Sulong also supports features like high-performance compilation, debugging support, monitoring, and cross-language interoperability (cf. section 4.3.1). Running on a JVM, Sulong also offers safe execution for unsafe languages such as C or C++ [27, 9]. Thus, errors can be observed more easily and automatically (as additional safety checks do not have to be inserted by certain tools or even manually).

[1] Chinese term for "fast dragon", cf. the LLVM logo

As mentioned in section 2.2, LLVM is not a classical programming language, but more an intermediate representation. Since Sulong is an LLVM runtime, it is used to (indirectly) execute languages that can be compiled to LLVM bitcode (e.g. C, C++, Rust, cf. figure 4.1). Language runtimes other than Sulong, like GraalJS for JavaScript or GraalPython for Python, act on the source code (written by the user) directly. Sulong, however, does not act on user-written code, but on a compiled version (LLVM) of it, as it can be seen in Figure 4.2.

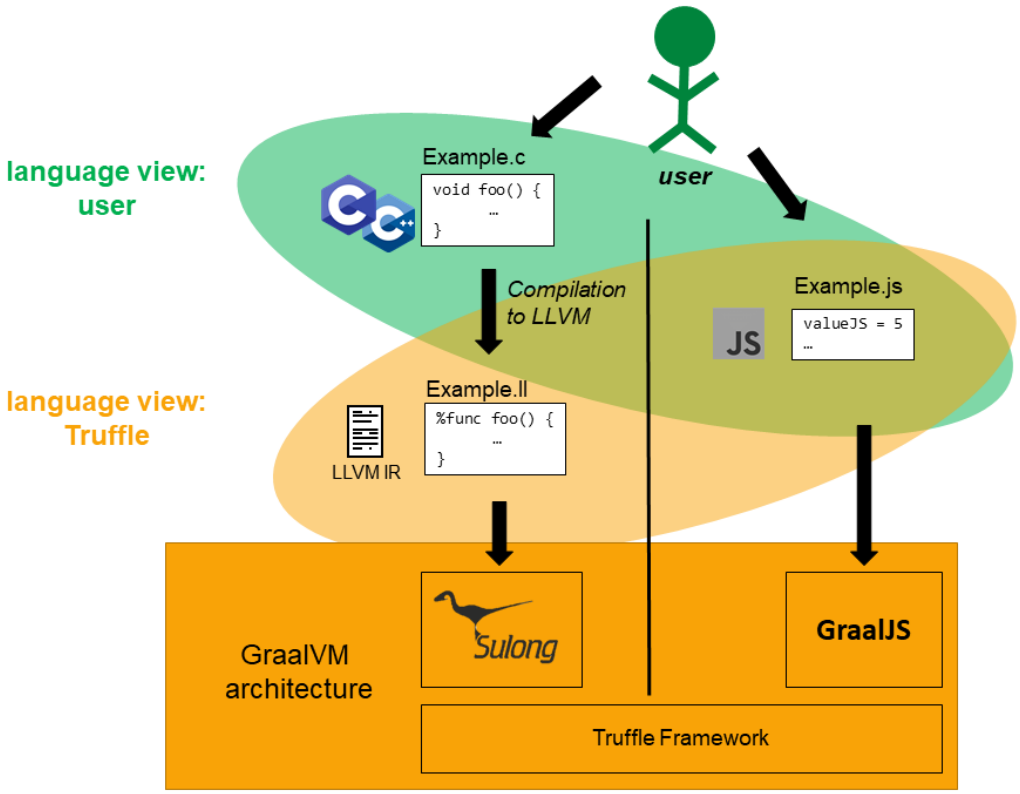


Figure 4.2: For JavaScript, user-written code is compiled/interpreted directly. For languages that can be compiled to LLVM, the user-written source language differs from the corresponding Truffle language.

This extra language step becomes relevant for Truffle features (e.g. (source-level) debugging [28] or cross-language interoperability) which usually work on both, the user perspective and the corresponding Truffle language code. In these cases, the language runtime implementation (e.g. Sulong for LLVM) has to internally support access to the source language code for a given Truffle language code input.

4.3.1 Truffle's existing interoperability concept in Sulong

As mentioned above in section 4.2.1, the interoperability concept is based on message passing and its API – Truffle forwards corresponding messages between different Truffle language implementations. Since Sulong is the LLVM implementation, other Truffle languages can interact with LLVM code.

However, the intention of polyglot programming via the interoperability concept is to write and call code in/from different source languages. In the case of Sulong, polyglot code written by programmers refers to the C language rather than to the "automatically" supported language LLVM [29]. Thus, polyglot API messages in Sulong refer to concepts from C code (e.g. struct members), while Sulong itself has to work with those C messages on the corresponding concepts in compiled LLVM code (e.g. byte offsets). So, while polyglot programming with other Truffle language implementations like JavaScript or Python works relatively straight-forward, Sulong has to provide additional mappings between the "C front-end" and the "LLVM back-end". These mappings will be shown in the following for different language concepts in C.

Function names and parameter order

foreign language, e.g. Python (source)

```
sq = llvmFile.square(3)
```

C (source)

```
int square(int x) {     return x*x; }
```

LLVM (from C)

```
define internal i32 @square(i32 %0) #0 !dbg !10 {
  call void @llvm.dbg.value(metadata i32 %0, metadata !15, metadata !DIExpression()), !dbg !16
  %2 = mul nsw i32 %0, %0, !dbg !17
  ret i32 %2, !dbg !18
}
```

Figure 4.3: C/LLVM function is called from python via the polyglot message `invokeMember("square", 3)`

Referring to the example case in Figure 4.3: Calling the `square` function from python works by writing python names and arguments the same way as they have been declared in the corresponding C source file. Function names, parameter order and basic types are preserved during compilation from C to LLVM, thus no additional effort is needed in this case.

foreign language, e.g. Python (source)

```
point = llvmFile.allocNativePoint();
point.x = 5;
```

C (source)

```
#include<graalvm/llvm/polyglot.h>

struct Point {
    double x;
    double y;
};

POLYGLOT_DECLARE_STRUCT(Point)

double getYofPoint(struct Point *p) {    return p->y;}

void *allocNativePoint() {
    struct Point *ret = malloc(sizeof(*ret));
    return polyglot_from_Point(ret);
}

//further methods for point deallocation, ...
```

LLVM (from C)

```
%struct.Point = type { double, double }
; Function Attrs: norecurse nounwind readonly uwtable willreturn
define dso_local double @getYofPoint(%struct.Point* nocapture readonly %0) #0 !dbg !29 {
  call void @llvm.dbg.value(metadata %struct.Point* %0, metadata !34, metadata !DIExpression()), !dbg !35
  %2 = getelementptr inbounds %struct.Point, %struct.Point* %0, i64 0, i32 1, !dbg !36
  %3 = load double, double* %2, align 8, !dbg !36, !tbaa !37
  ret double %3, !dbg !42
}
```

Figure 4.4: Cross-language example with python, C and the C-compiled LLVM code

Struct access

While function names and parameter order do not change during compilation from C to LLVM, accessing structs works differently, as it can be seen in Figure 4.4 in the corresponding functions `getYofPoint`: In C, struct members are resolved by their name (`p->y`), whereas LLVM uses offsets (`%2=getelementptr... %struct.Point* %0... i32 1, load double... %2`). Sulong therefore maps member names to offsets and vice versa by accessing the corresponding part of the debug information (cf. section 2.1.3) [29].

Chapter 5

Executing Swift-compiled LLVM IR code on Sulong

In this section, the necessary prerequisites and adaptions are described to execute Swift-compiled LLVM IR on Sulong. First, a look is taken at the LLVM IR emitted by the Swift compiler, and its necessary changes to make it compatible with Sulong. Then, the focus is put on the basic compilation and execution steps. Finally, it is investigated how to make the standard library of Swift accessible to Sulong.

5.1 Analyzing LLVM bitcode produced by Swift compiler

In the following, the implemented changes on Sulong are described, which were necessary to run specific LLVM operations emitted by the Swift compiler.

5.1.1 LLVM section names

As mentioned in section 3.2.3 and Figure 3.3, the Swift compiler can emit binary executables only, LLVM bitcode only, or also binary executables which contain the LLVM bitcode as a

section of the file. Also, LLVM sections can be extracted and included after compilation by the `llvm-objcopy` tool.

The embedded LLVM section in an executable binary file is usually denoted by `__LLVM,__Bitcode` for MachO files, and by `.llvmbc` for other formats [10], such as ELF (executable linking format). If Sulong looks for LLVM bitcode, it looks for the corresponding section name, depending on the file type. For ELF files, the name to look for has been `.llvmbc`.

However, the Swift compiler always puts LLVM bitcode into the `__LLVM,__Bitcode` section, even if the emitted file is an ELF file. Therefore, LLVM sections in Swift-compiled ELF files could not be found by Sulong first. The working and straight-forward solution for Sulong is to also look for `__LLVM,__Bitcode` in ELF files if LLVM sections need to be found.

5.1.2 Swift's getElementPointer aliases

Struct member accesses are often encoded by a fixed (member-dependent) offset, encoded as a `getelementptr` instruction. For example, a field of a C struct in Figure can be accessed by a fixed offset, e.g. a`[getelementptr base, offset]` instruction, given `base` holds the base address of the structure. As long as the offset is known at compile time (e.g. field accesses in C or C++), C/C++-compiled LLVM IR usually contains hard-coded `getelementptr` instructions in the compiled LLVM IR.

For Swift, however, the offsets are not always known at compile time, but rather at linking time. Therefore, read/write access of a struct member cannot be performed via direct encoding, but has to happen via a dynamic lookup (similar to dynamic binding). To overcome this problem, Swift-compiled LLVM IR makes use of `getelementptr` alias symbols.

A `getelementptr` alias is a symbol for a named `getelementptr` expression. As a simplified example, there is a Point struct as in Figure 5.1. Since not all offsets are used at compile time, the compiler can simply put the name of the corresponding `getelementptr` expression `yInP` into the code. As soon as offsets are known, the `getelementptr` instructions are then filled with concrete offsets (`yInP = hidden alias ...getElementPointer (Point, 4)`). Later, i.e. during linking time, all offsets can be obtained by resolving the appropriate `getelementptr` symbol. Note that the example in 5.1 is simplified, as Swift type members are dynamically

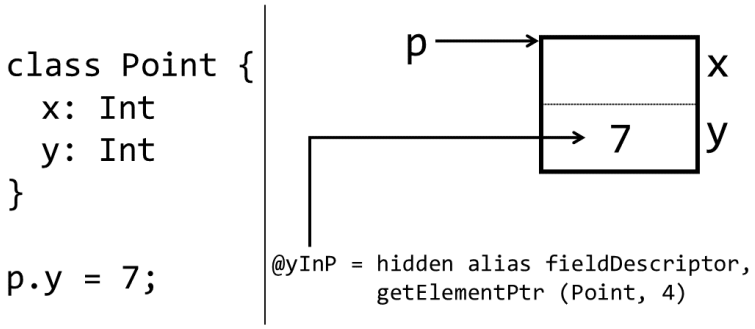


Figure 5.1: (Simplified) `getelementptr` alias for a member field of Swift struct

Swift

```
1 class Point {
2         var x: Int = 0
3         var y: Int = 0
4 }
```

LLVM (from Swift)

```
1 @"$s14elemPtrExample5PointC1ySivgTq" = hidden alias %swift.method_descriptor, getelementptr inbounds (<...
      > @"$s14elemPtrExample5PointCMn", i32 0, i32 16)
2 @"$s14elemPtrExample5PointC1ySivsTq" = hidden alias %swift.method_descriptor, getelementptr inbounds (<...
      > @"$s14elemPtrExample5PointCMn", i32 0, i32 17)
```

Figure 5.2: getElementPtr expression for field member `Point::y`

bound in general.

A real example of such a `getelementptr` alias is shown in Figure 5.2, where the same Point data structure is used. The LLVM code lines show the getter and setter symbol function of the `y` field of the class, which are both a `getelementptr` alias, having a resolution rule of a `getelementptr` instruction with base `s14elemPtrExample5PointCMn` and offset indices 0, and 16 or 17, respectively.

To support `getelementptr` alias symbols in Sulong, which are needed e.g. for field accesses or dynamic binding in Swift-compiled LLVM IR code, implementing the corresponding steps of parsing and efficient symbol resolution was necessary.

5.1.3 Half-precision floating-point type

Besides the more common floating-point types that use 32 (“single precision”/`float`) or 64 bit (“double precision”/`double`), Swift-compiled LLVM can also contain a 16-bit precision

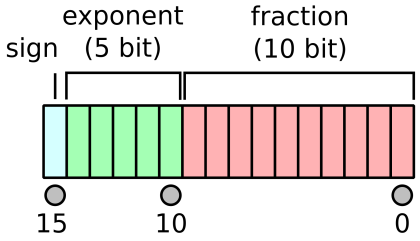


Figure 5.3: IEEE half-precision: bit semantics [30]

floating-point type ("half precision"/`half`), which can be used for certain high-performance computations on appropriate architectures [31]. The bit semantics of the half-precision type are shown in Figure 5.3 and are specified in the IEEE 754 standard [32].

However, since Sulong does not support half-precision floating-point numbers yet, the current solution is to convert the 16-bit numbers into numbers of the existing 32-bit floating-point type instead. Being aware that processing floating-point numbers with a different precision might lead to numerical errors, the current solution is considered as temporary, and can be changed as soon as the 16-bit floating-precision type is fully implemented in Sulong.

5.1.4 Struct return attribute

A possible function parameter attribute in LLVM is the `SRET` (structure return) attribute, which is used if LLVM tuple types are returned by-value. An example of such a usage is shown in Figure 5.4 (C-like pseudo code). The method `foo` returns a struct by-value, which is not possible in LLVM IR in general (depending on the struct members). To overcome this problem, the compiler changes the function signature: the return type (`struct A`) is added as pointer type parameter (`ret_ptr` in Figure 5.4). Now, the caller has to provide a pointer for the returned struct, such that the callee (`foo`) can still return a struct type by-value.

In Sulong, the `SRET` attribute has not been processed correctly, which caused a crash during parsing. Thus, the `SRET` parsing was implemented as specified in the language reference [33].

source level: code returning struct by value

```
1  struct A a = foo(...)
2
3  struct A foo(arg0, ...) {
4          ...
5          struct A a;
6          ...
7          return a;
8  }
```

LLVM level: code returning tuple, using sret attribute

```
1  struct A a;
2  foo(&a, ...);
3
4  void foo(/*sret<struct A>*/ struct A* ret_ptr, arg0, ...) {
5          ...
6          struct A a;
7          ...
8          *pa = a;
9          return;
10 }
```

Figure 5.4: Function which returns a tuple on source-level (above) and LLVM level (below), both in pseudo code

5.2 Compiling and executing Swift projects

5.2.1 Single file

To execute a single Swift file on Sulong (without further dependencies), the Swift code has to be converted to LLVM bitcode first. Then, it can be run on Sulong.

For a single file, there are two options:

- bitcode only: The file passed to GraalVM for execution on Sulong only contains the LLVM bitcode. For emitting bitcode, the `-emit-bc` flag can be set when the Swift compiler is invoked, as shown in section 3.2.3.

- embedded bitcode: The file passed to GraalVM for execution on Sulong is basically a binary object file (which could also be run by the operating system directly), containing the necessary LLVM bitcode as a section in the object file. The bitcode can be embedded automatically by setting the flag `-embed-bitcode` (again, as shown in section 3.2.3).

In both of the cases above, to be accepted by both the Swift compiler and Sulong, the Swift

file can be named arbitrarily.

5.2.2 Multiple files

As mentioned in section 3.2.2, it is possible to compile multiple Swift files down to one binary file, given that there is exactly one file called `main.swift` which contains the main method of the program.

For LLVM IR, however, applying the same principle is not possible:

- `-emit-bc`: simply emitting bitcode does not work directly for multiple output files. Instead, the user has to specify a module name.
- Using the `-embed-bitcode` flag, the Swift compiler produces one binary file, including an LLVM section which contains the compiled LLVM IR of all input files. However, the LLVM IR blocks are inserted one after each other, without explicitly being linked. Thus, parsing this LLVM IR section of the generated file leads to errors.

A possible workaround is shown in Figure 5.5 and described below:

1. The binary file can be compiled as described in section 3.2.2 (orange in Figure 5.5).
2. The single source files have to be compiled to LLVM bitcode. By specifying a module name (and setting the flag to emit bitcode), the Swift compiler automatically creates one bitcode file per source code (instead of producing one large LLVM bitcode file which is not linked).
3. The single LLVM bitcode files then can be linked together by the `llvm-link` tool.
4. (optional, green in Figure 5.5) The (linked) bitcode can be added as a section into the binary file of step 1, to get a binary file with an LLVM bitcode section.

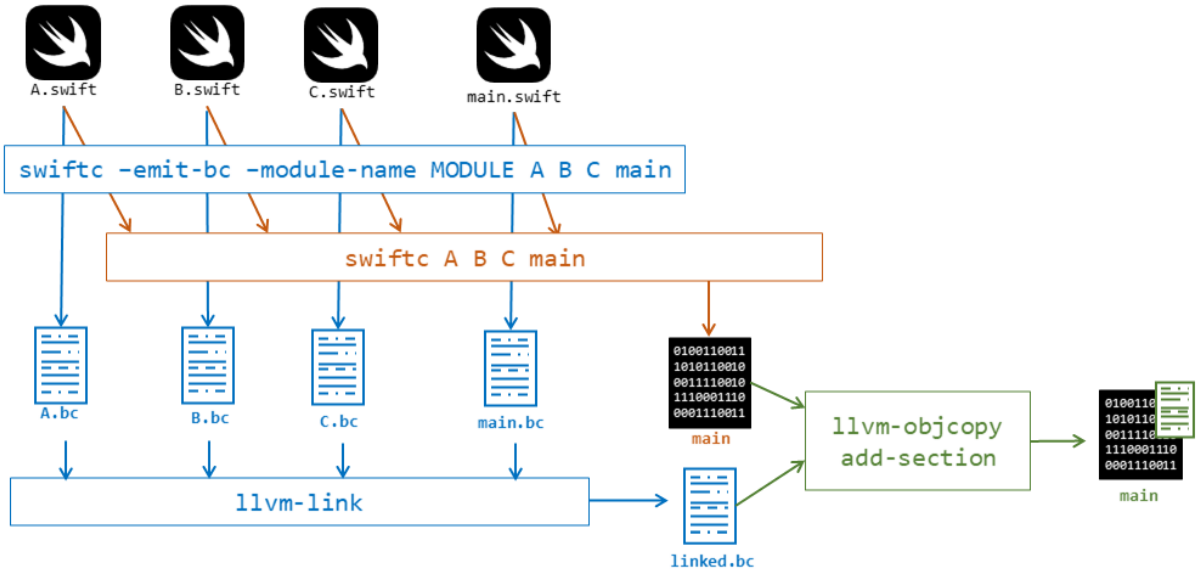


Figure 5.5: Compiling multiple Swift source files to LLVM IR/binary with embedded LLVM IR

5.3 Swift standard library

As described in section 3.3, the Swift standard library consists of different parts, written in C, C++, Swift and Swift-GYB, and is available both – as file in binary format, and as source code. If Swift-compiled LLVM IR which makes use of standard library functions is executed on Sulong, this leads to a problem, as shown in Figure 5.6: Standard library code is still only available in binary form, and cannot be executed on Sulong directly. Therefore, it is necessary to also compile the code of the Swift standard library from source and provide the compiled version (including LLVM IR) to Sulong. In the following, it is shown how this can be applied for the different parts of the standard library.

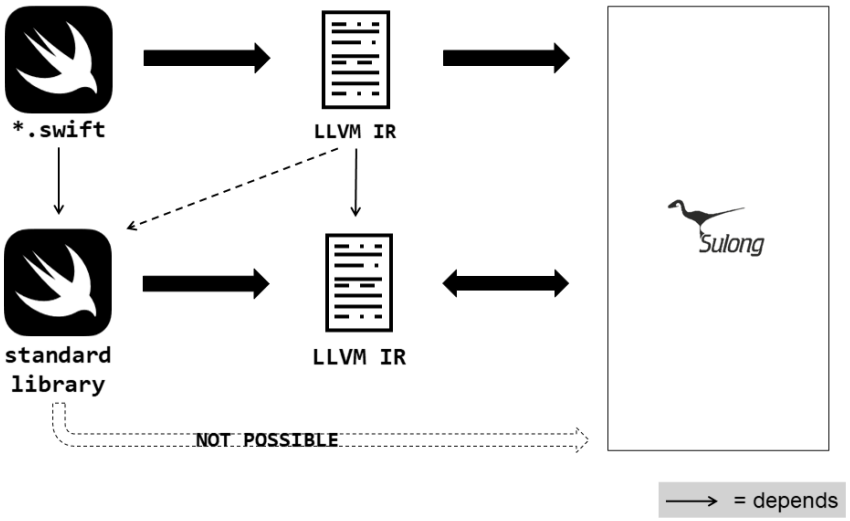


Figure 5.6: Execution of Swift-compiled LLVM code that has standard library dependencies

5.3.1 Compiling Swift and Swift-GYB files

Parts of the Swift standard library other than the runtime module are written in Swift, or the generating language Swift-GYB. These code parts must be compiled in a certain order (due to internal dependencies), which results in non-trivial compilation steps. To compile the standard library from source, however, the official Swift repository also contains a script which does the necessary compilation steps automatically, including generating Swift code from Swift-GYB, and taking care of dependencies [34]. By running this script, the corresponding standard library parts are created automatically as binary object files.

By providing extra arguments to the Swift compiler via the mentioned build script, such as `-embed-bitcode`, the resulting output files contain the necessary Swift code elements as LLVM bitcode.

5.3.2 Compiling C/C++ files

To compile Swift files, providing certain arguments to an existing build script file is enough, as shown above. However, this does not work for C/C++ code, which is used by the runtime module of the Swift library:

If the default compiler for LLVM code generation from C/C++ code (`clang`/`clang++`) is used, the resulting LLVM IR cannot be executed in Sulong. Therefore, there exists the *LLVM toolchain*, which can be seen as a clang wrapper to set certain flags and options accordingly. However, replacing the `clang`/`clang++` operations in the build script with the corresponding LLVM toolchain operation does not lead to the correct result: Due to dependencies and intermediate representations of the workflow specified in the build script, it is not easily possible to create or embed LLVM IR automatically as it has been for Swift code.

Instead, the current solution is to compile the necessary C/C++ files of the Swift standard library manually. Similar to the files written in Swift, manual compilation requires knowledge about the dependencies between the files and only works in correct order. However, it has only been shown that a manual compilation is possible with some of the files. Getting the full Swift standard library runtime module in LLVM does not lead to a further scientific contribution, but goes beyond the time scope of this thesis. Therefore, only provide a proof of concept is provided here.

As a consequence, not all standard library functions are supported if Swift code is executed on Sulong. One such example is any function or method which interacts with the console/standard input and output. Taking print functions to the standard output as a specific example, Swift's interoperability with C (mentioned in section 3.4) can be a possible workaround here. Such an example is shown in section 7.

5.3.3 Linking library files and execution

After applying the strategies mentioned above to compile the Swift standard library to LLVM IR, the output consists of several files, which together form the code of the standard library.

C/C++-compiled LLVM IR files can be linked easily with the standard `llvm-link` tool. In contrast to that, linking C/C++-compiled and Swift-compiled files together with the standard llvm-link tool leads to an error, as Swift-LLVM is not fully compatible with the corresponding (official) LLVM distribution.

However, the `llvm-link` tool of Swift can be used [15], which takes care of differences between the official LLVM distribution and its Swift fork. If Swift's `llvm-link` tool is not available, then

this does not lead to a problem. Sulong accepts more than one library file per execution call, which allows executing code by providing the various LLVM library files as multiple libraries.

5.3.4 Missing parts and future work

For an easy-to-use Swift standard library, several aspects still can be improved, which would result in full compatibility and a better user-friendliness. However, due to limited scientific information gain on the one hand, and due to time and scope limits on the other hand, those aspects which are shortly explained in the following have not further been improved yet:

- Full compatibility of Swift's standard library: As mentioned, currently only parts of the Sulong runtime module are supported. Having the full code range of the standard library available as LLVM IR would considerably increase the amount of executable Swift operations.

- User-friendliness: If Swift files are currently executed, the standard library files have to be provided by hand. On the one hand, reducing the number of different library files would ease execution of Swift code for the user. On the other hand, an automatically provided Swift standard library in Sulong (as it is already the case for C/C++ and their respective standard libraries) would lead to a further simplification for Sulong/GraalVM users.

Although no automated tool has been created to solve the remaining aspects mentioned above, a proof of concept was shown by providing a manual/partial solution. Creating a full and automated solution is thus no fundamental problem as such.

Chapter 6

Extending Truffle's Interoperability in Sulong

Previously, interoperability on Sulong only worked for LLVM IR compiled from C. However, Swift and C++ (both are languages beyond C that can also be compiled to LLVM IR) support language concepts which are not present in LLVM. Thus, those concept differences lead to a major difference between source (Swift/C++) and LLVM code for each of them, which caused incorrect (or even not working) solutions in Sulong's previous implementation.

Therefore, in the following section, the most important changes are described which were/are necessary to correctly process polyglot method calling in Sulong with Swift and C++. The necessary technological background and a description of the (previously) existing interoperability concept have been addressed in section 4.3.1.

6.1 Invoking Swift/C++-compiled LLVM functions from a foreign language

A possible scenario for calling a Swift/C++-compiled LLVM function from a foreign language is shown in Figure 6.1. In the following subsections, it is shown why certain Swift and C++ language concepts led to incorrect executions of examples like Figure 6.1, and what had to be changed in Sulong to resolve these problems.

foreign language, e.g. Python

```
sq = llvmFile.square(3.5)
```

Swift

```
func square(x: Double)-> Double { return x*x }
```

Figure 6.1: Calling a Swift/C++-compiled LLVM function from a foreign language

Swift signature

```
1 class A {
2         func foo(x: Int) -> Int {
3                 return 0
4         }
5 }
```

LLVM (from Swift) signature

```
define hidden swiftcc i64 @"$s9objMeth_S1AC3foo1xS2i_tF"(i64 %0, %T9objMeth_S1AC* swiftself %1) #0 {
```

C++ signature

```
1 class A {
2 public:
3         int foo(int x);
4 };
```

LLVM (from C++) signature

```
define i32 @_ZN1A3fooEi(%class.A* nocapture nonnull readnone dereferenceable(1) %0, i32 %1) #0 align 2 !
    dbg !11 {
```

Figure 6.2: Signature differences between source and LLVM for object/receiver methods

6.1.1 Signature differences

Swift and C++ both use the concept of class methods, i.e. functions that belong and have access to the so-called "receiver" (also called "self"/"this"), an object of a specified type. As LLVM does not support methods belonging to a type, every method call/every method declaration in the source language is replaced by an LLVM-compatible pattern.

Therefore, also signatures of Swift/C++ methods and their corresponding LLVM code signatures differ.

As an example, in Figure 6.2, the signature of two methods is shown, both in source (Swift/C++) as well as of the compiled LLVM function. Calling this `foo` method via polyglot programming could e.g. look like aObject.foo(17). In Sulong, the pointer representing `aObject` would now receive the interoperability message `invokeMember(member="foo", args = {17})`,

which could not have been resolved correctly in Sulong's previously existing interoperability handling. Instead, `execute("_ZN1A3fooEi", args = {aObj, 17}` would be necessary for a correct invocation.

To get a correct mapping between the source code signature and the corresponding LLVM function signature, two changes are needed: treating the receiver object on the one hand, and taking care of mangled names on the other hand, which will be described in the following subsections 6.1.1.1 and 6.1.1.2, respectively.

The overall procedure for cross-language member/method invocation resolving is shown afterwards in section 6.1.2.

6.1.1.1 Receiver object as parameter

Methods that are non-static need to have access to their receiving object and its members (often denoted as `"self"`/`"this"` in the source code). Since LLVM does not support receiver objects in functions, the receiving object is therefore added as a parameter. While C++ adds the receiver as the first object, Swift appends it to the end of the parameter list. The basic translation principle is shown in table 6.1 below, an example can be seen in Figure 6.2 above.

source language	**source code**	**LLVM IR**
C++	R A::foo(arg0, ..., argN);	R <fooMangled>(A* receiver, arg0, ..., argN)
Swift	A::foo(arg0, ..., argN) -> R;	R <fooMangled>(arg0, ..., argN, A* receiver)

Table 6.1: Translation of method `foo` in class `A` returning an object of type `R`

By adding the receiver object at the right place in the parameter list, the parameter information of the function signatures from source (Swift/C++) and LLVM can be mapped easily. Besides parameter restructuring, also the function symbol names themselves are different, which is due to name mangling.

6.1.1.2 Name mangling

Names of C code symbols are unique. Therefore, if C code is compiled down to LLVM, symbol names in C code are transferred as is into LLVM. As a consequence, Sulong's previously existing

foreign language, e.g. Python

```
sq = llvmFile.square(3.5)
```

Swift

```
1 func square(x: Int, y: Int) -> Int {
2         return x * x + y * y;
3 }
4
5 func square(x: Double) -> Double {
6         return x * x;
7 }
```

LLVM (from Swift) function signatures

```
define hidden swiftcc i64 @"$s6fnName6square1x1yS2i_SitF"(i64 %0, i64 %1) #0 !dbg !36 {
define hidden swiftcc double @"$s6fnName6square1xS2d_tF"(double %0) #0 !dbg !58 {
```

LLVM debug information section

```
1 !36 = distinct !DISubprogram(name: "square", linkageName: "$s6fnName6square1x1yS2i_SitF", scope: !5, file:
      !1, line: 1, type: !37, scopeLine: 1, spFlags: DISPFlagDefinition, unit: !0, retainedNodes: !2)
2 !58 = distinct !DISubprogram(name: "square", linkageName: "$s6fnName6square1xS2d_tF", scope: !5, file: !1,
      line: 5, type: !59, scopeLine: 5, spFlags: DISPFlagDefinition, unit: !0, retainedNodes: !2)
```

Figure 6.3: Different mangled names for the same source-language name

Truffle interoperability system assumed names to be fixed – objects declared with a certain name in the C source code can be found by exactly the same name in the resulting LLVM IR. However, languages like Swift and C++ support method overloading, which violates the symbol uniqueness property (i.e. two different methods with e.g. different signatures can exist in the same scope with the same name, as in the Swift part of Figure 6.3).

To remove this name ambiguity in lower-level code (such as LLVM), compilers intentionally change symbol names and usually include information of the declaring scope or parameters. This step of changing symbol names is called *name mangling* and, as metioned, produces unique symbol names throughout the compilation unit and beyond. As a consequence, names are not preserved during the compilation process, i.e. the mangled symbol names in LLVM IR language names differ from their source language names (specified in Swift/C++).

An example of name mangling can be found in Figure 6.3, where the two `square` functions in Swift are mangled to `$s6fnName6square1x1yS2i_SitF` and `$s6fnName6square1xS2d_tF`. The information which mangled (e.g. `$s6fnName6square1xS2d_tF`) and demangled (e.g. `square`) names belong together is stored in the debug information section (cf. lower part of Figure 6.3) of the LLVM file.

When Sulong receives an invoke message with the source language function name (i.e. `square`),

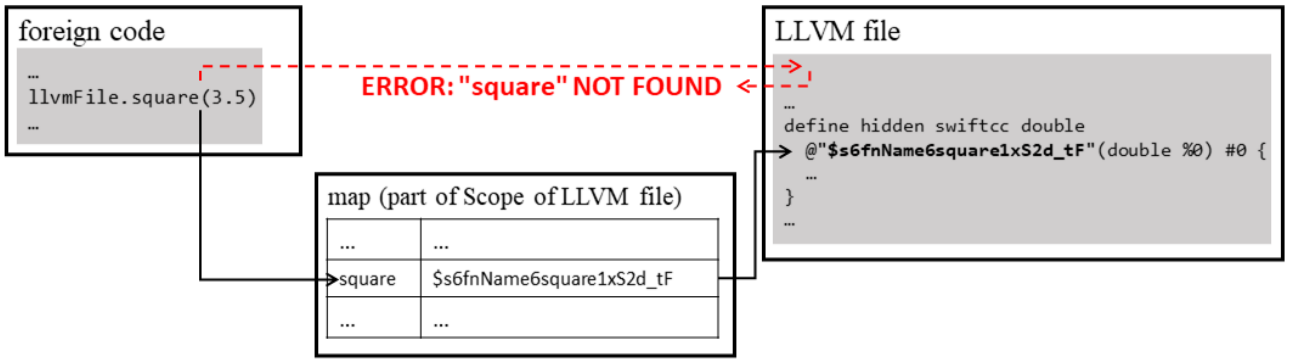


Figure 6.4: Name mangling resolution via a cross-language interoperability call

a direct look-up in the LLVM Scope/bitcode file does not lead to the correct function, as shown by the red and dotted arrows in Figure 6.4. Instead, a map is needed where every function is registered with its mangled name. This map can then be used to find the corresponding mangled name to the source language name, if an invoke message is received. To continue with the previously mentioned example, in Figure 6.4, a polyglot call of `square` leads to the mangled name `$s6fnName6square1xS2d_tF` via the look-up table. Now, the mangled symbol name can be found in the LLVM file scope, which leads to the corresponding `square` function implemented in LLVM.

The remaining task is now to create this necessary mapping table between source language names and their mangled versions. As mentioned above, the necessary information for that is stored in the debug information section, which has to be parsed to obtain the mappings.

However, parsing is done lazily: Sulong reads the function signature, and resolves the rest of the function information (including debug information) only when needed. For getting the (correct) mangled name of a function (needed for interoperability message processing), Sulong needs to parse the debug information section. Therefore, an additional parsing step was added, which only parses the necessary debug information, while everything else is skipped. The mapping is then stored in the corresponding scope in Sulong.

6.1.2 Method resolving

Getting the correct method/function signature, as discussed in the previous section, is a necessary step for cross-language interoperability method calls. However, it does not guarantee

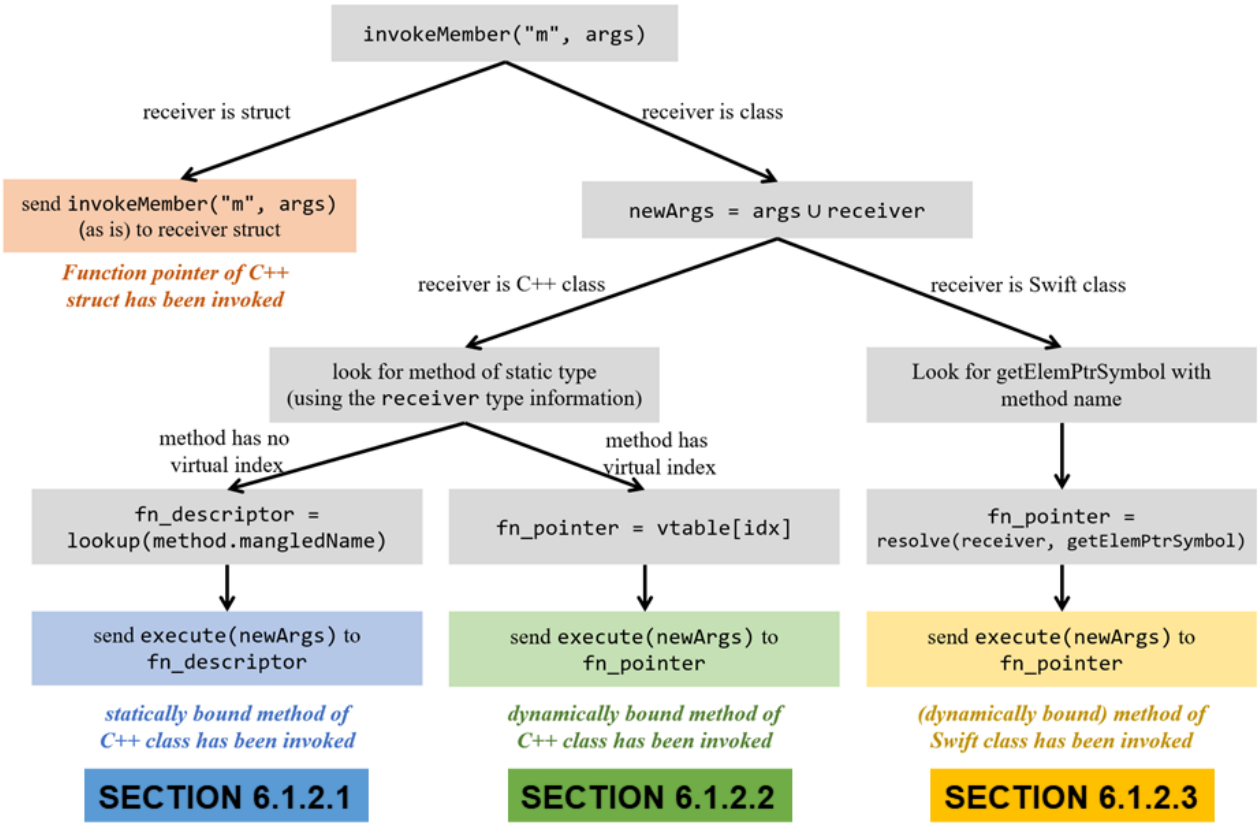


Figure 6.5: Processing of `invokeMember` messages in Sulong

a correct execution according to the semantics of the source language. The other important concept, which has to be taken into account, is method resolving.

Methods can be resolved statically or dynamically, which has already been explained in section 2.3.1. The approaches and extensions made in Sulong to enable a correct method resolution will be described below. Since the static/dynamic binding concept of C++ and Swift methods follow different approaches in LLVM IR, there are separate subsections, depending on the source language and resolution kind.

Figure 6.5 shows a general overview how `invokeMember` messages are processed in Sulong, i.e. the process of determining which strategy to apply. The detailed procedure can be found in the following corresponding subsections. As invoking a function pointer of a C++ struct has already been working in Sulong's previous implementation (cf. top left in Figure 6.5), no further explanation is provided for that.

6.1.2.1 C++/LLVM: Static binding

Static method resolving is relatively easy to achieve, as LLVM and C++ are both statically typed languages. Therefore, pointer objects in Sulong (*LLVMPointer*) (can) have their static type attached.

The first extension in Sulong for static binding is to extend Sulong's interoperability symbol table by a `class` type. This type is an extension of the existing `struct` type, which can also store the corresponding class methods as functions. During parsing of the debug information section, class and method information are automatically read in and stored correctly for later accesses. As soon as class and method information is available from the corresponding object in the symbol table, calling a statically bound method becomes straight-forward, which will be described in the following.

If a C++ method is called from a foreign language, the LLVM pointer representing the receiver object receives an `invokeMember(methodName, args)` message:

1. The static class type `C` of the receiver (attached to the `LLVMPointer` object from debug information) is extracted.

2. Due to the signature change mentioned in section 6.1.1, the receiver object is added to the arguments as the first one:
   `args = [arg0, ..., argN]` → `argsNew := [receiver, arg0, ..., argN]`.

3. Sulong iterates over the available methods which have been declared in `C` and selects the method that fits:

   - The demangled method name has to equal the `methodName` of the `invokeMember` message.

   - The parameter count has to fit, in case the called method has overloaded implementations (which is the case in the name mangling example in Figure 6.3).
     Note: Sulong does not perform a type check whether the foreign actual parameters match the formal parameters, as the foreign language is not necessarily typed.

- If there is a method fulfilling these requirements, the (unique) mangled name is available in the corresponding method object. Lookups by this mangled name lead to its function descriptor object. If no method with the correct signature is found, an error is thrown.

4. An interoperability `execute` message is sent to the function descriptor found in step 3 with `argsNew` as argument list, which leads to the desired method/function call.

6.1.2.2 C++/LLVM: Dynamic binding

The basic principle how a dynamic method call in C++ is done in LLVM has been shown in section 2.3.1. By following that principle, every `invoke` message for a dynamically bound method has to be translated to a table lookup.

For resolving, a vtable-like structure for Sulong's interoperability type handling had to be implemented, where every class object has access to its vtable information. Every virtual method is added to the corresponding class together with its method index during parsing. Since not every class type has its own implementation for every virtual method, classes also have access to their superclass types. Therefore, if a virtual method `m` is implemented in a parent class only, the vtable of the child class does not have an entry for `m`.

If a virtual C++ method is called from a foreign language, the LLVMPointer representing the receiver gets an `invokeMember` message. In fact, at this point in time, it is not known whether the method to be called is statically or dynamically resolved. Thus, the first three steps of a polyglot method invocation are the same as described for statically bound methods in section 6.1.2.1: Extracting the static class type, adding the receiver object, and finding a method that fits. If the found method is statically bound (i.e. there is no virtual index information), resolving is done fully as described above (steps 1 to 4). If there is virtual index information available for the found method, it is dynamically bound, thus resolving proceeds as follows (i.e. procedure as shown in Figure 2.3/6.6):

4. Given that a C++ class has virtual methods (which is automatically true when dynamic binding has to be applied), the first element stored in the type is always the pointer to the vtable (`vptr`, cf. Figure 6.6). Thus, `vptr` can be obtained by a read operation at

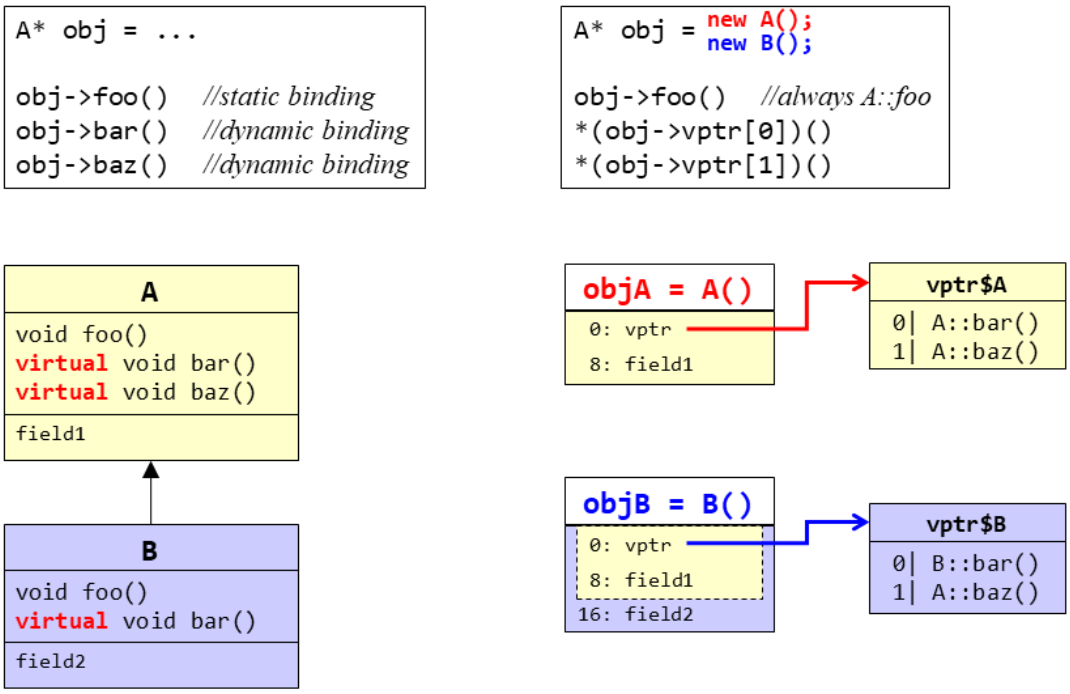


Figure 6.6: [identical with Figure 2.3] Method resolving: C++ (left) and LLVM (right)

index 0.

5. After having found the address of the beginning of the vtable (`vptr`), the function pointer can be computed as follows: Starting from the vtable address (`vptr`), the corresponding method offset (i.e. the virtual method index) is added to get the address of the needed function pointer. Afterwards, the actual function pointer value can be easily obtained by a `read` message to this address.
   In C semantics, the described computation can be written as `fn_ptr = *(vptr[idx])`.

6. In contrast to static binding (where an interoperability `execute` message is sent to the function descriptor) the `execute` message is now sent to the previously computed function pointer instead.

In this scenario, the C++ class `A` can serve as an interface, while `B` contains the actual implementation. As typical for object-oriented programming, `B`'s implementation can even be invisible from the JavaScript code, while JavaScript code can still invoke it (indirectly).

Swift code – the function `process` is calling `A::foo`

```
class A {
        func foo(x: Int) -> Int {...}
}
func process(a: A) -> Int {
        return a.foo(x: 4)
}
```

LLVM (from Swift): Function descriptor of `A::foo` as a getelementptr expression alias

```
@"$s8methCall1AC3foo1xS2i_tFTq" = hidden alias %swift.method_descriptor, getelementptr inbounds (<{ i32, i32, i32, i32, i32, i32, i32, i32, i32, i32, i32, %swift.method_descriptor, %swift.method_descriptor }>, <{ i32, i32, i32, i32, i32, i32, i32, i32, i32, i32, i32, i32, i32, %swift.method_descriptor, %swift.method_descriptor }>* @"$s8methCall1ACMn", i32 0, i32 13)
```

LLVM (from Swift): LLVM IR call of `A::foo`

```
  %2 = getelementptr inbounds %T8methCall1AC, %T8methCall1AC* %0, i32 0, i32 0, i32 0
  %5 = getelementptr inbounds i64 (i64, %T8methCall1AC*)*, i64 (i64, %T8methCall1AC*)** %4, i64 13
  %7 = call swiftcc i64 %6(i64 4, %T8methCall1AC* swiftself %0)
```

Figure 6.7: Caller structure of a Swift method in Swift and LLVM

6.1.2.3 Swift/LLVM

A simple scenario where a Swift method is invoked can be seen in Figure 6.7. The method `A::foo` is called from the `process` function. The function descriptor for `A::foo`, also shown in Figure 6.7, is a getElementPointer symbol with the class descriptor of `A` as a base, and the offsets 0 and 13. In embedded Swift code (bottom part of Figure 6.7), the offsets are then encoded directly.

Similar to C++, also Swift uses offsets/indices for dynamically bound method resolution. However, in contrast to C++/LLVM, these offsets are not stored in the debug information section. Instead, the method offset is encoded in a `getElementPointer` expression, which itself is connected to the corresponding LLVM function descriptor.

Given that, interoperability `invokeMember` messages on Swift receiver instances are processed as follows:

1. As there is no direct information available that could reveal the receiver type of the method to be invoked, a method with `foo` as its demangled name is looked up in the current scope. If such a method is found, the (unique) mangled name can be obtained easily.

2. Given the mangled name (`"<mangled>"` for simplicity), the function descriptor object can be found in the scope by appending the suffix `Tq` to the mangled name (i.e. `"<mangled>Tq"`), as it is specified in the Swift name mangling documentation [35]. The returned function descriptor object is structured as a getElementPointer alias symbol, which has been explained in section 5.1.2.

3. As mentioned above, the getElementPointer symbol contains the necessary method offset information, similar to the C++ vtables. Like for dynamically bound C++-compiled methods, the method offset (together with the base object) can be used to get the function pointer.

4. As explained in section 6.1.1, the receiver object has to be appended to the end of the argument list (similar to C++, where the receiver has been added as the first instead of the last argument).

5. Finally, an interoperability `execute` message is sent to the previously determined function pointer (with the modified argument list).

6.2 Invoking a foreign language implementation from LLVM

6.2.1 C++-compiled LLVM

To call methods of a foreign object from C++-compiled LLVM via Truffle's cross-language interoperability, there are two possibilities:

- Using Sulong's existing polyglot API function `polyglot_invoke_member(...)`. This call automatically forwards the arguments to the foreign object in the form of a corresponding `invokeMember` message. The drawback with this workaround is that `polyglot_invoke_member(...)` is a helper function where C++ code has to be modified for polyglot programming.

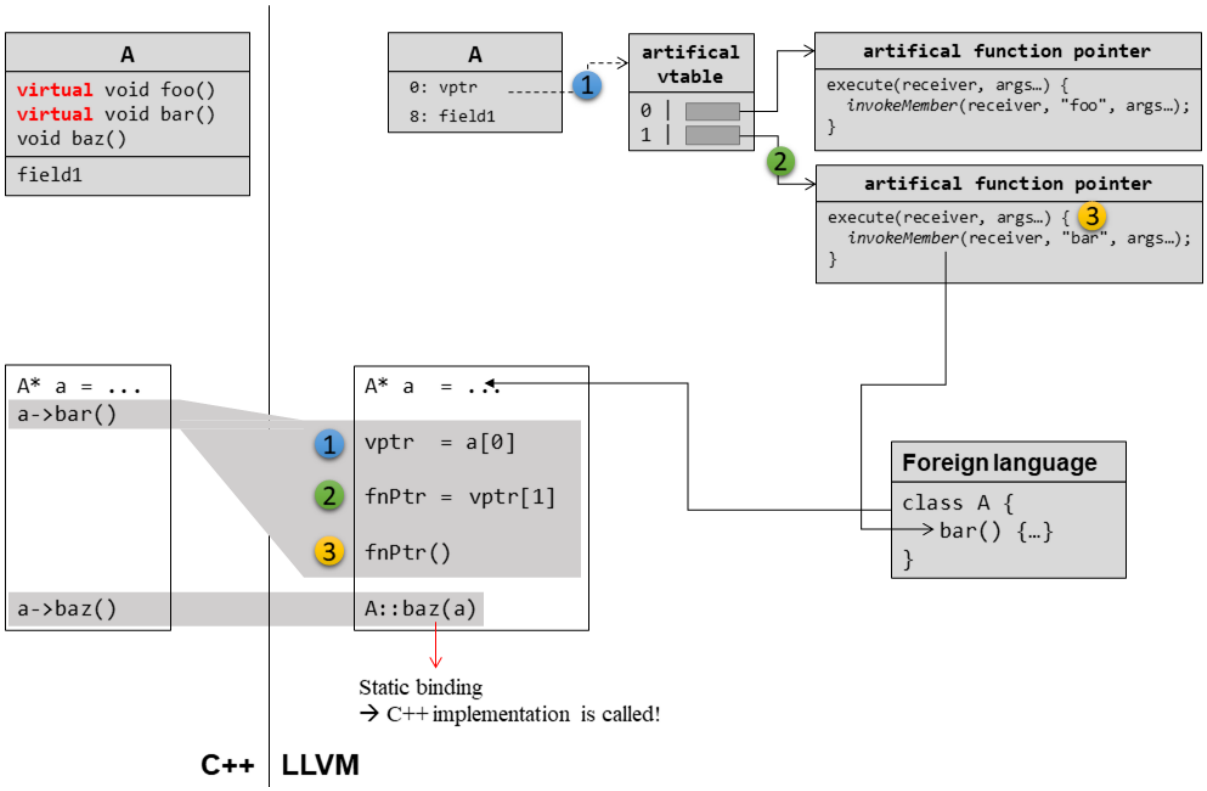


Figure 6.8: Calling a foreign method object from LLVM

- The other possibility is to declare the corresponding method as virtual. If the underlying receiver of the virtual method is a foreign object, then (by definition of dynamic binding), also the foreign implementation has to be invoked.

As an example, how a virtual foreign object/method can be called from LLVM, is shown with the `a->bar` call in the `accept` method in the C++ code in Figure 6.8. If the `A* a` object provided to the `accept` method is foreign, then its own (foreign) implementation should be called, since `bar` is declared as virtual.

Thus, the compiler replaces every call of that method by a dynamic method lookup as explained above, and as it can be seen in Figure 6.8 in the LLVM code block. If now the receiver object (in this case `A* a`) is foreign, then this foreign object receives a `readMember` message on index 0, as Sulong would expect the vtable pointer here.

To enable foreign interoperability calls from LLVM, Sulong was extended in the following way:

1. For every `readMember` message on a foreign object, it is checked whether the read index

is equal to 0. If so, a look at the static type of the receiver object is taken (the static type information is available in Sulong, even if the foreign language typing is not known).

2. If the static type (in this example: `A`) of the receiver object has virtual methods, `readMember[0]` would return the vtable pointer `vptr` of the object. Thus, an artificially created Truffle object (1) is returned, which will be called *artificial vtable* in the following. It is designed as a Truffle object and thus also supports interoperability messages. Instead of being structured like a vtable (array of function pointers), this artificial vtable contains an array of artifical Truffle objects themselves, each representing a virtual method of the class `A`. In the following, these objects are called *artificial function pointers*.

3. As soon as the vtable Truffle object (representing the vtable of `A`) receives an interoperability `read` message with a certain index, LLVM would return the function pointer of the corresponding function in LLVM code. In this case, the corresponding Truffle object representing the function pointer (2) is returned. Since the ordering of these methods belongs to their vtable indices specified in the debug information section (cf. section 6.1.2.2), a read operation with index `i` also leads to the artifical function pointer representing the method with the same virtual index `i`.

4. The artificial function pointers have been designed as Truffle objects, where their behavior for receiving certain interoperability messages has been set. When such an artificial function pointer receives an execute message (which happens in case of a method call), it is implemented in a way that the execute message is forwarded as an `invoke` message to the foreign object. In this example, when the artificial pointer belonging to `A::bar` receives an `execute(receiver, args...)` message, the message `invokeMember("bar", args...)` is sent to the (foreign) `receiver` object (3).

6.2.2 Swift-compiled LLVM

The basic procedure how to invoke a foreign method implementation from LLVM has the same concept for C++-compiled LLVM as for Swift-compiled LLVM, but is different in the implementation. Due to the scope of this thesis, this concept was only implemented for C++ in Sulong. For Swift, the steps to be applied are then similar: For a certain read access on the vtable pointer (also at index 0 in Swift), artificial Truffle objects are returned, which

simulate the corresponding Swift method data structures up to the direct function invocation. Following this principle, no fundamental problems during implementing this feature also for Swift-compiled LLVM in Sulong are expected.

Chapter 7

Case study

In this section, a short case study for running Swift-compiled LLVM IR code on Sulong is provided. First, the general structure (section 7.1), and how the code is compiled and executed (section 7.2) is shown. Afterwards, results will be presented and discussed (sections 7.3 and 7.4, respectively). The source code used for this case study is attached at the end of this chapter (section 7.5).

7.1 Overview

As a computation example, digits of π in its hexadecimal representation will be computed. This is done according to the BBP (Bailey–Borwein–Plouffe) formula [36], which allows a digit-independent computation of π in hexadecimal base.

The general code structure can be seen in Figure 7.1 and is as follows:

- `computeNthHexDigit` computes the n^{th} hexadecimal digit of π.
- `computeHexDigits` calls the `computeNthHexDigit` function n times to compute the first n (in this case 10000) hexadecimal digits of π. One call of `computeHexDigits` corresponds to one iteration.
- The `main` function calls `computeHexDigits` several (in this case 100) times, where the

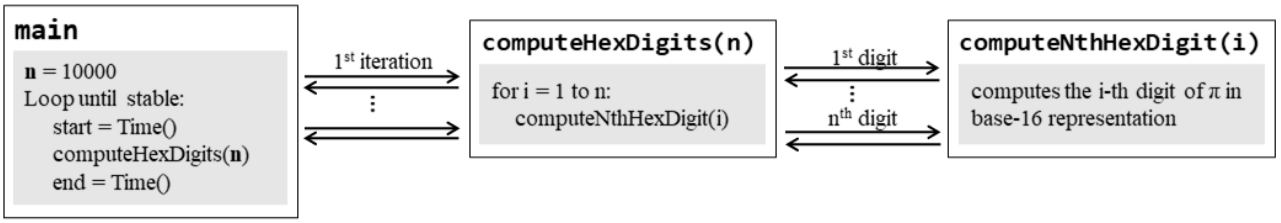


Figure 7.1: Basic structure

scenario	**main function**	`computeHexDigits`	`computeNthHexDigit`
"Swift"	JavaScript	Swift/LLVM	Swift/LLVM
"mixed"	JavaScript	JavaScript	Swift/LLVM
"JS"	JavaScript	JavaScript	JavaScript

Table 7.1: Computation languages in different scenarios

time between call and return is measured (each call computes the same 10000 digits of π). For evaluation, the first (in this case 30) iterations are ignored due to optimizations during the warm-up phase. Thus, the last 70 iterations are taken for evaluation.

Concerning different language levels, three scenarios were evaluated to compare performance, which are listed in Table 7.1:

- Swift: In this scenario, most of the code is used from Swift source. After starting time measurement in JavaScript, the Swift implementation of the `computeHexDigits` function is called once via interoperability. This measurement serves as an evaluation for Swift-compiled LLVM code on GraalVM.

- mixed: The mixed scenario serves as an overhead measurement for cross-language interoperability use. In contrast to the previous scenario, `computeHexDigits` is taken from JavaScript code. Thus, for every single of the 10000 digits, the Swift/LLVM function is called via interoperability.

- JS: For this scenario, only JavaScript code is run, which serves as a reference comparing Swift-compiled LLVM to another Truffle language.

7.2 Compilation flow and execution settings

To run the example described in this section, two steps are necessary:

1. Compiling the Swift code to LLVM IR: This has been done with the command `swiftc -O -embed-bitcode -g pHD_swift.swift`:
   The `-O` flag forces the Swift compiler to optimize at a higher level that usual.
   `-embed-bitcode` is set to embed the LLVM bitcode into the object file, as described in section 3.
   `-g`, the flag for emitting debug information, has to be set in order to include name mangling information into the output file.
   The resulting file is named `pHD_swift` by the Swift compiler.

2. Running code, i.e. starting the main function: Since the main function is written in JS, *GraalJS* is the primary language runtime. The main file with its dependencies can be run with the command
   `js --polyglot --experimental-options --llvm.C++Interop=true <file>.js`:
   `js` calls the GraalJS implementation.
   `--polyglot` is necessary to enable polyglot programming.
   `--experimental-options --llvm.C++Interop=true` triggers the additional parsing step which is necessary for name mangling (as shortly discussed in section 6.1.1.2).

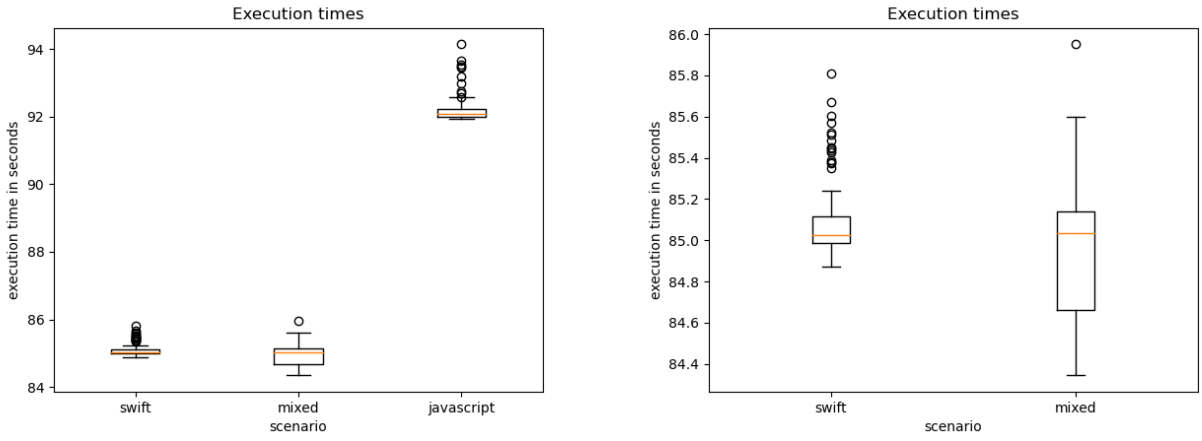
Note: For checking whether the iterations are already stable (i.e. no further dynamic compilation by the Graal compiler), the flags `--engine.TraceCompilationDetails` and `--engine.TracePerformanceWarnings=all` can be set to print ongoing operations on the interpreted/executed code.

7.3 Results

For the following results, each of the three scenarios described in Table 7.1 has been run. As mentioned, the first 30 iterations of each scenario are ignored, since the Graal compiler performs dynamic optimization during these executions. Thus, the remaining 70 iterations are considered for evaluation.

metric	Swift	mixed	JS
min (0%)	84.95	84.53	91.97
Q1 (25%)	84.99	84.91	92.01
mean (50%)	85.02	85.08	92.07
Q3 (75%)	85.10	85.19	92.22
max (100%)	85.67	85.95	93.64
arithmetic avg.	85.10	85.08	92.21

Table 7.2: Results (time in seconds)



(a) All three scenarios

(b) Swift and mixed scenario

Figure 7.2: Results

The results for all three scenarios can be seen in Table 7.2 and in Figure 7.2a. The JavaScript scenario took the longest in the performed run, whereas the Swift and mixed version are in the same order of magnitude. Figure 7.2b shows the results of the Swift and mixed scenario in more detail.

All run examples were computed correctly, i.e. the results of each of the three scenarios were mutually equal.

7.4 Discussion

As the task of this thesis has been mainly to make Swift-compiled LLVM IR code run on Sulong (which did not work before), the focus has not been laid at time performance. However, the results show that running Swift code on GraalVM can be done without having the case of significantly longer execution times. In this case, Swift-compiled LLVM code on Sulong was faster than JavaScript code on GraalJS, which can be expected, since Swift is a statically typed language, while JS is dynamic. As the Swift and mixed scenario both achieve similar results (cf. Table 7.2 and Figure 7.2), it can be concluded that cross-language interoperability calls do not have a significant impact on performance and are optimized away.

7.5 Code

Swift code (scenarios *Swift* and *mixed*)

```
1  public class PidigitGenerator {
2          public func computeHexDigits(_ length: Int) -> Int {
3                  for i in 0...(length-1) {
4                          let d = requestNthHexDigit(i)
5                          if(d<0 || d>15) {
6                                  return 0
7                          }
8                  }
9                  return 1
10         }
11
12         //returns the n-th hexadecimal digit of pi
13         public func requestNthHexDigit(_ n: Int) -> Int {
```

```
14              let sum = 4*subsum(n: n, j: 1) - 2*subsum(n: n, j: 4) - subsum(n: n, j: 5) - subsum(n: n,
                    j: 6)
15              let fractional = fract(x: sum)
16              let digit: Int = Int(fractional * 16)
17              return digit
18      }
19
20      //only keeps the fractional part of the number
21      func fract(x: Double) -> Double {
22              let f = x - Double(Int(x))
23              let g = f+1
24              return g - Double(Int(g))
25      }
26
27      func subsum(n: Int, j: Int) -> Double {
28              var sum: Double = 0
29              for k in 0...n {
30                      sum += Double(modPow(base: 16, exp: n-k, mod: 8*k+j)) / Double(8*k+j)
31              }
32              return sum
33      }
34
35      //returns (base^exp)%mod
36      func modPow(base: Int, exp: Int, mod: Int) -> Int {
37              if(exp==0) {
38                      return 1
39              } else if(exp%2 == 0) {
40                      let m = modPow(base: base, exp: exp/2, mod: mod)
41                      return (m*m)%mod
42              } else {
43                      return (modPow(base: base, exp: exp-1, mod: mod)*base) % mod
44              }
45      }
46  }
47
48  public class ObjectCreator {
49      public static func requestGenerator() -> PidigitGenerator {
50              return PidigitGenerator()
51      }
52  }
```

Listing 7.1: Swift code for scenarios *Swift* and *mixed*

JS code of Swift scenario

```
1  length   = 10000;
2  llvmFile = Polyglot.evalFile("llvm", "pHD_swift");
3  swiftGen = llvmFile.ObjectCreator.requestGenerator();
4  durations = []
5
6  for(it=0;it<100;it++) {
7          //print("iteration "+it)
```

```
8           startJS = Date.now();
9           result = swiftGen.computeHexDigits(length);
10          duration = Date.now() - startJS;
11          if(result!=1) {
12                  print("WRONG RESULT!");
13          }
14          durations[it] = duration
15          print(it+" "+duration)
16  }
17  print(durations)
```

Listing 7.2: JavaScript code for scenario *Swift*

JS code of mixed scenario

```
1   function computeHexDigits(generator, length) {
2           pi_array = [];
3           for(i=0;i<length;i++) {
4                   pi_array[i] = generator.requestNthHexDigit(i)
5           }
6           return pi_array
7   }
8
9   length   = 10000;
10  llvmFile = Polyglot.evalFile("llvm", "pHD_swift");
11  swiftGen = llvmFile.ObjectCreator.requestGenerator();
12  durations = []
13
14  for(it=0;it<100;it++) {
15          //print("iteration "+it)
16          startJS = Date.now();
17          jsArray = computeHexDigits(swiftGen, length);
18          jsDuration = Date.now() - startJS;
19          durations[it] = jsDuration
20          print(jsDuration)
21  }
22  print(durations)
```

Listing 7.3: JavaScript code for scenario *mixed*

JS code of JS scenario

```
1   function requestNthHexDigit(n) {
2           sum = 4*subsum(n, 1) - 2*subsum(n, 4) - subsum(n, 5) - subsum(n, 6)
3           fractional = fract(sum)
4           digit = Math.floor(fractional * 16)
5           return digit
6   }
7
8   //only keeps the fractional part of the number
```

```
9  function fract(x) {
10         f = x - Math.floor(x)
11         g = f+1
12         return g - Math.floor(g)
13 }
14
15 function subsum(n, j) {
16         sum = 0
17         for (k=0;k<=n;k++) {
18                 sum += modPow(16, n-k, 8*k+j) / (8*k+j)
19         }
20         return sum
21 }
22
23 //returns (base^exp)%mod
24 function modPow(base, exp, mod) {
25         if(exp==0) {
26                 return 1
27         } else if(exp%2 == 0) {
28                 m = modPow(base, exp/2, mod)
29                 return (m*m)%mod
30         } else {
31                 return (modPow(base, exp-1, mod)*base) % mod
32         }
33 }
34
35 function computeHexDigits(length) {
36         pi_array = [];
37         for(i=0;i<length;i++) {
38                 pi_array[i] = requestNthHexDigit(i)
39         }
40         return pi_array
41 }
42
43 length   = 10000;
44 durations = []
45
46 for(it=0;it<100;it++) {
47         //print("iteration "+it)
48         startJS = Date.now();
49         jsArray = computeHexDigits(length);
50         jsDuration = Date.now() - startJS;
51         durations[it] = jsDuration
52 }
53 print(durations)
```

Listing 7.4: JavaScript code for scenario *JS*

Chapter 8

Conclusion and Future Work

In this thesis, it was investigated how to execute Swift code on GraalVM, which works via the approach of running Swift-compiled LLVM IR on Sulong, a component which serves as the LLVM runtime of GraalVM and which is used to run C-/C++-compiled LLVM IR. Although LLVM IR can be emitted by the Swift compiler and run by Sulong, running Swift-compiled LLVM IR on Sulong was not possible without further modification.

Thus, Sulong was extended and adapted such that the Swift-compiled LLVM IR is now found by Sulong in the correct section of the object file. Also, the handling of LLVM instructions that have not been needed by Sulong before then was complemented. To be able to run Swift code which makes use of dependencies (e.g. from the standard library), a proof of concept was shown how make the standard library functionality accessible for Sulong. Concerning Truffle's cross-language interoperability, it was explained how different concepts of higher languages (e.g. dynamic binding) are mapped to lower-level LLVM instructions. Based on the findings, Truffle's existing cross-language interoperability concept has been extended for a selection of C++/Swift features. These include e.g. making sure that names are mangled and demangled correctly, or that static/dynamic binding is applied according to the corresponding language semantics. Finally, it was proven that an existing Swift program can be run on Sulong, including a possible application of the cross-language interoperability concept.

However, full compatibility for the Swift language in general, as well as for cross-language interoperability, has not been achieved yet. The following issues have not been solved yet completely:

- Availability of all Swift standard library functions: In section 5, only a part of the Swift standard library is made available to Sulong when Swift-compiled LLVM IR is run. For further integration, more standard library files have to be compiled from source, with the need of considering existing inter-file dependencies.

- Calling a foreign method from Swift via Truffle's interoperability concept: If a method on a foreign object is invoked in Sulong (from Swift-compiled LLVM IR), then the callee is not chosen correctly. To make this way of method resolution work, read operations to the corresponding method resolving table have to be caught and forwarded to the foreign language correctly. After implementing foreign method calling for C++-compiled LLVM and investigating Swift-compiled LLVM IR, it can be concluded that the solution process is similar for Swift-compiled LLVM IR.

- Field access of a struct/class via Truffle's interoperability concept: If Swift-compiled LLVM IR accesses a type member (e.g. a class field or struct member) of a foreign object or vice versa, the instructions are not performed correctly. Similar to the issue above, this task has been implemented for C++-compiled LLVM IR. For Swift, however, fields are dynamically bound and thus accessed via implicit getter and setter methods. The first step to implement a correct solution for field accesses is therefore a correct method invocation handling via Truffle's interop messages (previous bullet point). Given that, reads and writes to fields have to be converted to the corresponding getter and setter calls, and vice versa.

- Storing structs: In Sulong's native execution mode, managed objects cannot be stored in memory. However, Swift-compiled LLVM IR very likely contains such store operations, depending on the code complexity and structure. Unfortunately, Swift code which tries to save managed objects does not work with Sulong's native mode.

Nevertheless, a proof of concept was given how to fix each but the last of these mentioned issues without having to face further fundamental problems.

Acknowledgement

This work is written to receive the "Diplom-Ingenieur" degree in Computer Science at the Johannes Kepler University Linz. The author would like to thank his technical supervisor, Roland Schatz, for guiding him through the work very well and for always being available when help was needed! Also thanks to the primary supervisor of this thesis, Herbert Prähofer, and to Hanspeter Mössenböck for their regular feedback. Concerning the implementation, the author would like to thank the Sulong team (especially Paley Li and Josef Eisl), and Jacob Kreindl, who – along from giving very useful advice and being also working on Sulong – has been (and still is) a great office neighbor too. Last, but not least, thanks go to all the scientists for their preliminary work, and all the persons who are there for the author outside of his working life, especially friends, family and colleagues in free-time activities. Although the author is a single person, this finished thesis would not have been possible without the support of several others.

List of Figures

List of Tables

Listings

Bibliography

[1] Swift language. `https://swift.org/`. Accessed: 2021-10-20.

[2] Swift the fastest-growing programming language, as it breaks into top 10. `https://9to5mac.com/2018/03/09/swift-ranking-programming-languages`. Accessed: 2022-02-25.

[3] Pypl popularity of programming language. `https://pypl.github.io/PYPL.html`. Accessed: 2022-02-25.

[4] The 10 most popular programming languages to learn in 2022. `https://www.northeastern.edu/graduate/blog/most-popular-programming-languages`. Accessed: 2022-02-25.

[5] Graalvm. `https://www.graalvm.org/`. Accessed: 2021-12-01.

[6] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[7] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Georg Richards, Doug Simon, and Mario Wolczko. One vm to rule them all. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming and software*, pages 187–204, New York, NY, USA, 2013. ACM.

[8] The truffle language implementation framework. `https://github.com/oracle/graal/tree/master/truffle`. Accessed: 2021-12-01.

[9] Manuel Rigger, Roland Schatz, René Mayrhofer, Matthias Grimmer, and Hanspeter Mössenböck. Sulong, and thanks for all the bugs: Finding errors in c programs by abstracting from the native execution model. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 377–391, New York, NY, USA, 2018. ACM.

[10] Llvm bitcode file format. `https://llvm.org/docs/BitCodeFormat.html`. Accessed: 2021-12-01.

[11] Llvm source level debugging. `https://www.llvm.org/docs/SourceLevelDebugging.html`. Accessed: 2022-01-12.

[12] Klee. `https://klee.github.io/`. Accessed: 2021-11-08.

[13] Swift – resources. `https://developer.apple.com/swift/resources/`. Accessed: 2022-02-14.

[14] Swift compilation process. `https://swift.org/swift-compiler/`. Accessed: 2021-10-28.

[15] Swift: Github repository. `https://github.com/apple/swift`. Accessed: 2022-02-08.

[16] Swift standard library. `https://www.swift.org/standard-library/`. Accessed: 2022-02-16.

[17] Swift standard library developer information. `https://developer.apple.com/documentation/swift/swift_standard_library/`. Accessed: 2022-02-16.

[18] Importing objective-c into swift. `https://developer.apple.com/documentation/swift/imported_c_and_objective-c_apis/importing_objective-c_into_swift`. Accessed: 2022-02-08.

[19] Swift – c interoperability. `https://developer.apple.com/documentation/swift/swift_standard_library/c_interoperability`. Accessed: 2022-02-20.

[20] Swift and c++ interoperability workgroup announcement. `https://forums.swift.org/`

t/swift-and-c-interoperability-workgroup-announcement/54998/10. Accessed: 2022-02-27.

[21] Interoperability between swift and c++. https://github.com/apple/swift/blob/main/docs/CppInteroperability/CppInteroperabilityManifesto.md. Accessed: 2022-03-11.

[22] Interoperability between swift and c++. https://github.com/apple/swift/blob/main/docs/CppInteroperability/GettingStartedWithC%2B%2BInterop.md. Accessed: 2022-03-11.

[23] Polyglot on the jvm with graal. https://www.slideshare.net/InfoQ/polyglot-on-the-jvm-with-graal-79014042. slide 15. Accessed: 2021-12-01.

[24] Hanspeter Mössenböck and Matthias Grimmer. Truffle–a self-optimizing language implementation framework. *5th Int. Conf. on Mathematics and Informatics*, 2015.

[25] Y. FUTAMURA. Partial evaluation of computation process-an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[26] Sulong on github. https://github.com/oracle/graal/tree/master/sulong. Accessed: 2021-12-01.

[27] Manuel Rigger, Matthias Grimmer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. Bringing low-level languages to the jvm: Efficient execution of llvm ir on truffle. In *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages*, pages 6–15, 2016.

[28] Sulong: Source-level debugging. https://github.com/oracle/graal/blob/master/sulong/docs/contributor/DEBUGGING.md. Accessed: 2022-01-12.

[29] Graalvm llvm reference manual: Interoperability. https://www.graalvm.org/reference-manual/llvm/Interoperability/. Accessed: 2022-01-04.

[30] Codekaizen. The memory format of an ieee 754 half precision floating

point value. https://commons.wikimedia.org/wiki/File:IEEE_754r_Half_Floating_Point_Format.svg. CC BY-SA 4.0. Accessed: 2022-02-17.

[31] Ahmad Abdelfattah, Stanimire Tomov, and Jack Dongarra. Towards half-precision computation for complex matrices: A case study for mixed precision solvers on gpus. In *2019 IEEE/ACM 10th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*, pages 17–24, 2019.

[32] Ieee 754-2019 – ieee standard for floating-point arithmetic. https://standards.ieee.org/ieee/754/6210/. Accessed: 2022-02-17.

[33] Llvm language reference. https://llvm.org/docs/LangRef.html. Accessed: 2022-02-16.

[34] Contributing to swift – how to set up an edit-build-test-debug loop. https://github.com/apple/swift/blob/main/docs/HowToGuides/GettingStarted.md. Accessed: 2022-02-16.

[35] Swift name mangling. https://github.com/apple/swift/blob/main/docs/ABI/Mangling.rst. Accessed: 2022-01-12.

[36] David Bailey, Peter Borwein, and Simon Plouffe. On the rapid computation of various polylogarithmic constants. *Mathematics of Computation*, 66(218):903–913, 1997.

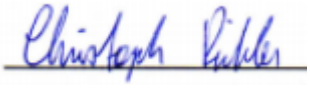
Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Sworn declaration

I hereby declare under oath that the submitted Master's Thesis has been written solely by me without any third-party assistance, information other than provided sources or aids have not been used and those used have been fully documented. Sources for literal, paraphrased and cited quotes have been accurately credited.

Linz, March 17, 2022



Christoph Pichler