FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Simple Object Machine implementation in functional programming language
Student:	Filip Říha
Supervisor:	doc. Ing. Filip Křikava, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2023/2024

Instructions

The Simple Object Model (SOM) is a minimal smalltalk dialect used for research in Virtual Machines. It already has nine main implementations in various programming languages. However, they are all done using imperative and object-oriented programming styles. This thesis aims to create a new implementation in a purely functional programming language. The main contribution next to the actual compiler and virtual machine is the comparison of this approach to the imperative ones.

The steps should be:
- Familiarize yourself with SOM and one of its implementation.
- Design and implement a compiler and a virtual machine for SOM.
- Compare the implementations.

In the implementation prefer clarity over raw performance. The implementation should reasonably documented and tested.

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 16 January 2023 in Prague.

Bachelor's thesis

SIMPLE OBJECT MACHINE IMPLEMENTATION IN FUNCTIONAL PROGRAMMING LANGUAGE

Filip Říha

Faculty of Information Technology
Katedra teoretické informatiky
Supervisor: doc. Ing. Filip Křikava, Ph.D.
May 11, 2023

Czech Technical University in Prague
Faculty of Information Technology
© 2023 Filip Říha. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis: Říha Filip. *Simple Object Machine implementation in functional programming language*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Contents

List of Figures

List of Tables

List of Listings

*I would like to thank my supervisor doc. Ing. Filip Křikava, Ph.D.
for helping me with this thesis and for his patience and advices.*

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work for non-profit purposes only, in any way that does not detract from its value. This authorization is not limited in terms of time, location and quantity

In Prague on May 11, 2023 .....

Abstract

This thesis provides an implementation of a Smalltalk programing language dialect called Simple Object Machine (SOM) in Haskell, a purely functional language. It explores the syntax and semantics of a SOM program and analyses already existing implementations. Then it provides the design and implementation details of the virtual machine, that is based on bytecode instructions and a bytecode interpreter. The parts of the VM are individualy explored, which are lexer, parser, compiler, runtime environment and garbage collector.

Keywords Runtime System, Virtual Machine, Object-oriented Programming, Simple Object Machine, Haskell

Abstrakt

Tato práce se zabývá implementaci dialektu programovacího jazyka Smalltak nazývaného Simple Object Machine (SOM), a to v čistě funkcionálním programovacím jazyce Haskell. Práce zkoumá syntaxi a sémantiku SOM programu a analyzuje již existující implementace. Následně je prezentován návrh, design a implementační detaily virtuálního stroje, který je založený na bajtkódových instrukcích a bajtkodovém interpeteru. Jednotlivé části virtuálního stroje jsou jednotlivě popsány, což jsou lexer, parser, překladač, běhové prostředí a garbage collector.

Klíčová slova běhové prostředí, virtuální stroj, objektově orientované programování, Simple Object Machine, Haskell

List of abbreviations

ANTLR	ANother Tool for Language Recognition
AST	Abstract Syntax Tree
BNF	Backus-Naur Form
EBNF	Extended Backus-Naur Form
GC	Garbage collector
GHC	Glasgow Haskell Compiler
JIT	Just-in-time
OOP	Object-oriented Programming
REPL	Read-Eval-Print Loop
SOM	Simple Object Machine
VM	Virtual Machine

Introduction

There has been a trend with many popular programming languages running on some *virtual machine*, as opposed to being compiled to a native binary. Even though this is almost always at a cost of performance when compared to a compiled language, it comes with many benefits including, but not limited to, *write once, run anywhere* architecture, memory safety or faster programmer feedback loop.

Even with this trend, the development and inner workings of virtual machines are still sometimes seen as a black box, especially when compared to other parts of a compiler/interpreter pipeline, namely lexer and parser, as these have been researched extensively. Moreover, implementing a new virtual machine for an already existing programming language is very complex, as they tend to have feature-rich syntax and strong requirements, thus they are not ideal for research and teaching. This is where Simple Object Machine language comes in.

Simple Object Machine (SOM) is a dialect of Smalltalk, a purely object-oriented programming language, used for teaching and researching virtual machines[1], originating from Aarhus, Denmark. Like Smalltalk, it aims to be a minimal language with only a few built-in constructs and a small standard library. This allows the implementations to focus on the design of runtime instead of the lexer and parser. Nonetheless, it is still a very functional language with some real-life applications.

At the time of writing, there are 9 main implementations of SOM, all of which are written in an imperative or OOP language (Java, C, C++, Python, Rust, JavaScript and Smalltalk). This comes naturally because runtime environments are by their definition mutable and therefore managed more easily in an imperative/OOP paradigm.

The goal of this thesis is to design and implement a new compiler and virtual machine for SOM. The language of the implementation is Haskell, a lazily evaluated purely functional programming language. The goal is not to compete with other implementations in the performance of the VM, but to compare the difference in approach to VM implementation in a functional programming paradigm compared to a classical imperative or object-oriented implementation.

Goals

- Design and implement a compiler and VM for SOM in Haskell.
- Benchmark the implementation's speed.
- Compare the implementation details with SOM++ implementation.

Chapter 1

Background

In this chapter, we introduce the Simple Object Machine language, its code structure, built-in concepts and existing implementations. We also briefly introduce the Haskell programming language used for the implementation presented in this thesis.

1.1 SOM language

Simple Object Machine (SOM) is "*a minimal Smalltalk for teaching of and research on Virtual Machines.*"[1] It originates from the University of Aarhus and it was first used in 2001 for teaching object-oriented VMs and runtime systems.[2] The listing 1.1 shows an example of a simple SOM *Hello World* program.

```
1 Hello = (
2     run = (
3         'Hello World!' println.
4     )
5 )
```

Code listing 1.1 A simple SOM Hello world example

It features dynamically loaded classes with fields and methods, single inheritance, dynamic method dispatch, global variables and closures with non-local returns. There is a small official standard library (under 1900 lines of code) that contains the basic definitions for integer and floating point arithmetics, string manipulation, collections (arrays, vectors, dictionaries, hashtables and sets) and provides methods for printing into the standard output and loading text files. It mainly aims to be small enough to be understood, but still big enough to be useful.

SOM has an official syntax specification with an ANTLR grammar[3, specification/SOM.g4], but it lacks a semantic specification. On the other hand, it has several official implementations, so the semantic details can be inferred from these other implementations.

```
Class = className "=" [ superclassName ] "("
    InstanceFields { instanceMethod }
    [ Separator ClassFields { classMethod } ]
    ")" ;

InstanceDefinitions =
    [ "|" { instanceField } "|" ];

ClassDefinitions =
    [ "|"  { classField } "|" ];

Separator = "----" { "-" } ;
```

Code listing 1.2 EBNF grammar for classes

1.1.1 Object model

Being a dialect of Smalltalk, SOM is a *purely object-oriented language*, meaning that *everything is an object* and the computation progress by passing *messages* between objects. SOM employs a class-based system with single inheritance, so each object is an instance of some class.

Objects are not only *instances of classes*, as we may know it from other popular object-oriented programming languages like C++ or JavaScript, but also *classes*, primitives like *numbers*, *booleans* or *strings*, and built-in constructs like closures (called *blocks* in Smalltalk languages) or *arrays*. This provides a nice uniformed way to work with all objects, classes and constructs in SOM and greatly simplifies reflection, but can pose performance hits, as primitive functions like number arithmetics and boolean evaluations are compiled to calling methods.

1.1.2 Class definition

The core of SOM code is the class definition, outlined by the grammar in listing 1.2. Each class is defined in its own file and contains the name of the class, its superclass, instance fields, instance methods, class fields and class methods. If the name of the superclass is omitted, the `Object` superclass is implicitly supplied.

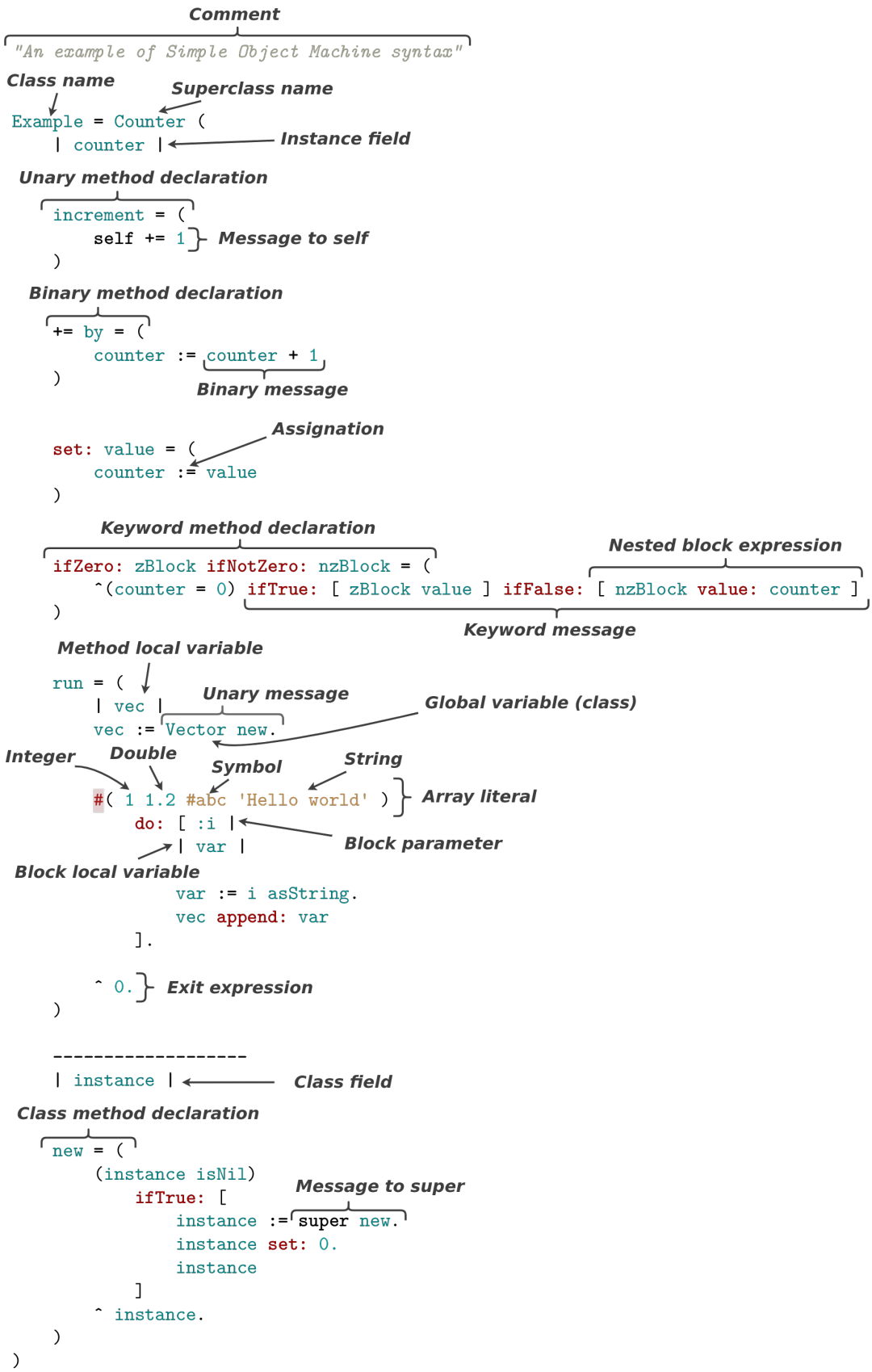
Instance methods contain the code executed on passing the corresponding message to an object that is an instance of the given class, while *class methods* is code executed upon passing the given message to a class.

Instance fields are variables available to an instance and *class fields* are variables that can be used in class methods.

A class can be accessed globally by its name. Class methods can be invoked by directly sending the message to the class and by sending the message `new`, we create a new *instance of the class*. For example, in listing 1.3 on line 30, the class `Vector` is accessed and a new instance is created.

1.1.3 Methods

Each class defines methods for a given class and its instances. The body of the method is either composed of expressions or annotated by a `primitive` keyword. The functionality of these primitive methods is provided by the runtime.



Code listing 1.3 Class definition example

```
1 UndefinedBehavior = (
2     foo = ( 'foo1' println. )
3
4     foo = ( 'foo2' println. )
5
6     run = ( self foo. )
7 )
```

Code listing 1.4 Example of undefined behavior

There are three types of method signatures:

- *unary methods*, identified by a single keyword and no parameters (e.g., code listing 1.3 on lines 8–10, the definition of unary method `increment`),
- *binary methods*, identified by a sequence of given special symbols with exactly one parameter (e.g., code listing 1.3 on lines 13–15, the definition of method `+=` with a parameter `by`),
- and *keyword methods* identified by a sequence of keywords, each ending with a colon, and with a parameter after each keyword (e.g., code listing 1.3 on lines 18–20, the definition of method `set:` with a parameter `value`).

A method is identified by its name without arguments and with no spaces, so the signature of the method in example 1.3 defined at lines 23–25 is `ifZero:ifNotZero:`.

Classes inherit all methods of their superclass, meaning we can send the same message to an instance of a class as we can send to an instance of its superclass. If a class defines a method with the same signature as a superclass, we override the inherited method and when called, the body of the overriding method is executed.

SOM has no concept of access modifiers and all methods are public, as opposed to languages like Java or C++, which allow methods to be declared as private or protected.

It is undefined behavior if a class has two method definitions of the same signature. For example, when we execute the code in listing 1.4 in the *SOM++* implementation, it prints `foo1`, whereas the *SOM-RS* prints `foo2`.

1.1.4 Expressions

A method or nested block body consists of zero or more expressions separated by a period.

Exit expression is denoted by the symbol `^` and is followed by another expression. Upon reaching an exit statement, the contained expression is evaluated. The result of the evaluation is used as a return value and the method is exited. Exit expressions have different semantics inside of a nested block, where they behave as non-local returns (discussed later in this chapter).

For *assignation* is used the `:=` operator. Since assignation is also an expression, its result can be used as a value.

Evaluation consists of sending messages to objects. As we have three different method signatures, we also have three different types of messages, each one corresponding to an invocation of a type of method:

- *Unary messages* are sent to an object without any arguments. In this example, we are sending a unary message `not` to an object `true`.

```
true not
```

- *Binary messages* are sent to an object with exactly one argument. Unlike the mathematic notion of binary operators being a function of two arguments, here the first object is the receiver of the message with the second as an argument. In the following example, we are sending a binary message `+` with argument `2` to an object `1`, both instances of `Integer`.

  ```
  1 + 2
  ```

- *Keyword messages* have one argument for each keyword of its signature. In the following example, we are sending a keyword message to an object `Pair`, where `withKey:` and `andValue:` are keywords and `'foo'` and `'bar'` are arguments.

  ```
  Pair withKey: 'foo' andValue: 'bar'
  ```

There is no operator precedence and all messages are left-associative. For example, `5 + 2 / 3` is the same as `(5 + 2) / 3`. In terms of mixing different types of methods, unary methods have the highest precedence, then binary methods, keyword methods, and lastly assignations. Therefore

```
a := '10' asInteger negated to: 1 abs + 3 abs
```

is the same as

```
a := (('10' asInteger) negated) to: ((1 abs) + (3 abs))
```

1.1.5 Variable scoping

Each SOM variable has a scope, which defines how it can be accessed and modified.

- *Local variables*, *method arguments* and *block arguments* are accessible from the method or block they are defined in.
- *Instance fields* are accessible in all instance methods and each instance of a class has its own variables. They are not accessible to the other objects directly, not even to other objects of the same class. This is different from most other programming languages, where for example in Java, two objects of the same class can access and modify each other's private fields. The instance fields of a superclass are also accessible in a method.
- *Class fields* are accessible in all class methods. They cannot be accessed from an instance method, unlike for example static fields in Java and C++.
- *Class objects*, i.e. objects representing a class, can be accessed globally. There is always one class object for one class defined in the current runtime.
- *Global variables* are also accessed globally and they are defined by the virtual machine. New global variables can be defined in the runtime with the `global:put:` instance method of the `System` class. In SOM, there are four global variables by default,
  - `true`, an instance of the `True` class,
  - `false`, an instance of the `False` class,
  - `nil`, an instance of the `Nil` class, representing an empty value,
  - and `system`, an instance of the `System` class, representing the runtime enviornment.

```
Shadowing = (
    | x |

    foo = (
        | x | "Shadows the instance field"
        x := 'Hello'. "Assigns to the local variable"
    )

    bar = (
        x := 'World'
        self foo.
        x println. "Always prints 'World'"
    )
)
```

Code listing 1.5 Variable shadowing example

In a body of a method, the currently executed object is available by the `self` reserved keyword. If we want to execute the method as it is defined in the superclass, we can send it a message via the `super` reserved keyword.

Name shadowing is valid in SOM. One can define a variable or field in a context with the same name that is already in scope and the new definition is used when referring to this name. An example can be seen in listing 1.5, where a local method variable `x` in method `foo` shadows an instance field. The `self` and `super` built-in keywords can also be shadowed this way.

Naming two or more variables with the same name in one scope definition is also valid in SOM and effectively, one of the variables shadows the others, meaning they cannot be accessed normally. There are however primitive methods `instVarAt:` and `instVarAt:put:` that can be used to get and set the value of any instance variable by their index, even the ones that are shadowed.

1.1.6 Built-in constructs

The SOM language has a small set of built-in constructs and little to no syntactic sugar. Like in Smalltalk, the aim is to keep the syntax as minimal as possible.

1.1.6.1 Literal types

Literals allow us to represent a fixed value or constant. In the runtime, there is no difference between a literal type and an instance of an object, as these literals are represented as regular objects. This allows us to call SOM a *purely* OOP language. This is very different from another highly popular OOP language Java, where primitive types are not treated equally to other objects. The types of literals in SOM are

- *strings*, enclosed by single quotes (e.g. `'Hello world!'`),
- *symbols*, prefixed by the hash symbol `#` (e.g. `#hello`),
- *arrays*, enclosed within `#(` and `)` with individual items separated by whitespaces (e.g. `#( 'hello' 'world' 10 ).`),
- *integers*,
- and *floats*.

```
"Block with no parameters"
[ 'Hello world' println ].

"Block with two parameters"
[ :i :j | i + j ].

"Block with two parameters
and one local variable"
[ :i :j |
    | a |
    a := i + j
].
```

(1.6.1) Block syntax example

```
ClosureExample = (
    foo = (
        | block a |
        block := [ a println ]
        a := 'Hello world!'.
        ^ block.
    )

    bar = (
        | a |
        a := 'Not hello'.
        "Prints 'Hello world!"
        (self foo) value.
    )
)
```

(1.6.2) Block closure example

Code listing 1.6 Block examples

1.1.6.2 Blocks

A *block expression* or a *nested block* is a closure with a body composed of expressions. It is very similar to lambdas, arrow functions or anonymous functions in other languages. When a block in a code is reached, it is created, but not executed until invoked. In the runtime, it is represented as an object and as an instance of the class `Block1`, `Block2` or `Block3` depending on the number of parameters (zero, one or two respectively).

A block can define parameters, denoted by a leading colon followed by the variable identifier. Blocks can define local variables, which are lexically scoped to the body of the block. An example of this syntax is in listing 1.6.1.

A block can be executed by sending the `value` method if it has no parameters, with `value:` if it has one parameter and with `value:with:` if it has two parameters. Blocks with three or more parameters are not supported in SOM.

When a new block is defined, it closes over the current lexical scope, traditionally called *home context*[4, p. 306]. All of the non-local variable lookups of a block are done in this home context where it was created, not from the context of block execution, including the keywords `self` and `super`. The home context contains not only the method and object it is created in, but also the exact point in execution, i.e. the current call frame. In a code example 1.6.2, we can see that the local variable in a method `foo` is closed over and later still valid in the method `bar`.

The exit expression inside of a block has a different meaning than inside of a method, as it invokes a *non-local return*. When a non-local return is evaluated, it rolls back the call stack to the method where the block was created and then returns from it with the value of the exit expression. Non-local return can be analogous to an exception, where the "thrown" (returned) value is "caught" in the method of home context and then returned from this point.

This behavior is represented in image 1.1, where a block is created in a method `foo`, and then evaluated in method `bar`. Upon evaluating, the non-local return is executed, exiting both the methods `foo` and `bar`.

When a non-local return is evaluated and the method where the block was created is no longer on the call stack, the method `escapedBlock:` with the current block as an argument is called on the object where the block execution was invoked.

Unlike in a method body, when an exit expression is not used, the implicit return value of the block is not the reference to `self`, but the value of the last expression body.

```
NonLocal = (
    foo = (
        self bar: [ ^'Hello' ].
        ^'world'. "this line is never executed"
    )

    bar: block = (
        block value.
        'Hello from bar' println. "this line is never executed"
    )

    run = (
        (self foo) println. "prints 'Hello'"
    )
)
```

Code listing 1.7 Non-local return example

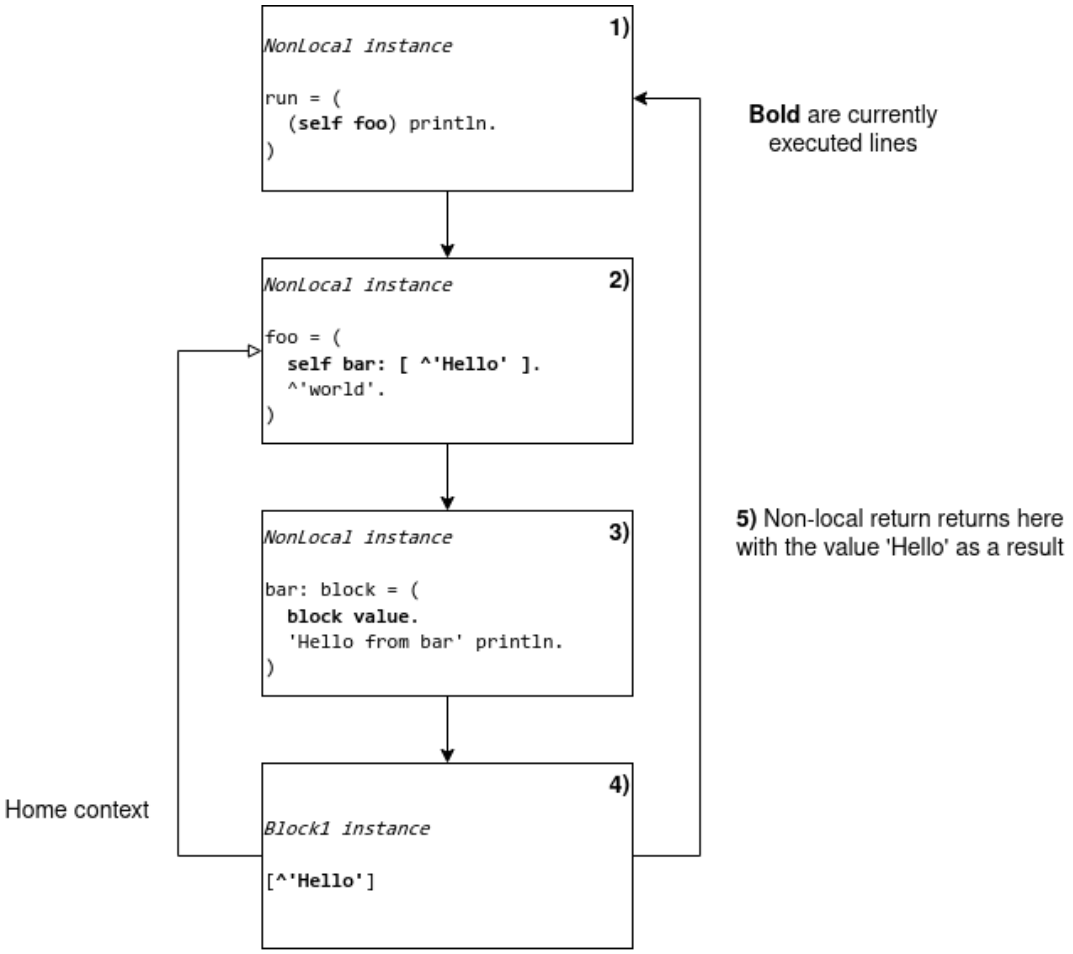


Figure 1.1 State of execution of listing 1.7

```
to: limit do: block = (
    self to: limit by: 1 do: block
)

to: limit by: step do: block = (
    | i |
    i := self.
    [ i <= limit ] whileTrue: [ block value: i. i := i + step ]
)
```

Code listing 1.8 Implementation of iteration methods in Integer class [3, Smalltalk/Integer.som]

1.1.6.3 Control flow

Where most of the other languages have special control structures like `if` or `for`, SOM implements control flow as plain objects and methods. This ties into the "everything is an object" philosophy of Smalltalk languages, while also simplifying the language syntax.

- *Braching*: The `Boolean` class and it's subclasses `True` and `False` define methods `ifTrue:`, `ifFalse:` and `ifTrue:ifFalse:`. The arguments of these methods are blocks that are conditionally executed depending on the class of the boolean they are passed to (`True` method executes the blocks passed to the `ifTrue:` method and `False` executes on `ifFalse:` call).

- *Loops*: Instead of `while` statement, the `Block` class define `whileTrue:` and `whileFalse:` methods. The code `condition whileTrue: runThis` executes the block `runThis` until the block `condition` stops returning `true`, analogous for `whileFalse:`.

- *Iterating*: The `Integer` class defines methods `to:do:` and `to:by:do:`. These capture the basic usage of `for` expression in languages like C and Java, where `i to: limit by: step do: block` repeatedly invokes `block` with the argument in the closed range from `i` to `limit` with the step size of `step`. Internally, this is implemented as a `whileTrue:` loop, as we can see in listing 1.8.

  An example usage of control flow usage can be seen in a simple FizzBuzz code example 1.9.

1.2 Existing SOM implementations

At the time of writing, SOM has 9 main implementations as listed on the official SOM website[1], each one with different goals and approaches to VM implementations. They are either based on AST or bytecode interpretation and have varying levels of optimizations. The following is a brief description of them:

- *SOM* was the first implementation. It is written in Java and is bytecode-based with very basic optimizations. It is the implementation used for teaching at the University of Århus in Denmark.

- *CSOM*, written in C, was the first implementation in another language than Java. It is bytecode-based with a mark/sweep garbage collector and does not have any optimizations. It can be compiled into WebAssembly.

```
FizzBuzz = (
    run = (
        | vec i |
        vec := Vector new.
        i := 1.

        [i <= 100] whileTrue: [
            | str |
            str := ''.
            (i % 3) = 0 ifTrue: [ str := str + 'Fizz' ].
            (i % 5) = 0 ifTrue: [ str := str + 'Buzz' ].
            (str = '')
                ifTrue: [ vec append: i asString ]
                ifFalse: [ vec append: str ].

            i := i + 1.
        ].

        1 to: vec size do: [ :j |
            (vec at: j) println
        ]
    )
)
```

Code listing 1.9 FizzBuzz example

- *PySOM* is both AST and bytecode-based implementation in Python. It is either compatible with RPython with metatracing-based JIT compilation, or pure Python with no dependencies. At first, there were three Python implementations for SOM (PySOM, RPySOM and RTruffleSOM), but since the end of 2020, all of these codebases were merged into one[5, README.md].
- *AweSOM* is a Smalltalk implementation based on bytecode with no optimizations.
- *TruffleSOM* is an AST interpreter written in Java using the *Truffle framework*[6] and *Graal compiler*[7]. It claims that this makes it highly optimized and based on the *Are We Fast Yet benchmarks*[8], it is the fastest SOM implementation.
- *SOM-RS* and *ykSOM* are two implementations in Rust. Both offer a bytecode-based VM. The former has also an AST interpreter. ykSOM is "*partly a test bed to experiment with good ways of structuring Rust interpreters, balancing correctness, performance, and readability*"[9] and is planned to eventually use meta-tracing system Yorick[10] to produce a JIT-compiling VM.
- *JsSOM* is an AST interpreter written in JavaScript that can be run both on Node.js and in browsers. It is the VM behind the REPL shell on the official site.
- *SOM++* is an implemenation in C++. It is bytecode-based with jump bytecodes, meaning the main interpretation loop is implemented as a goto jump based on the current bytecode code. It can be compiled with three different garbage collectors (copying, mark-and-sweep and generational) and optionally supports tagged or cached integers.

There is also an unofficial implementation in C++ written as a part of a master's thesis at the Faculty of Information Technology, CTU. It uses the official ANTLR grammar as the basis for

Name	Type of interpreter	Language	Type of garbage collector
SOM	bytecode	Java	none
CSOM	bytecode	C	mark-and-sweep
PySOM	AST and bytecode	Python	none
AweSOM	bytecode	Smalltalk	none
TruffleSOM	AST	Java (Truffle framework)	none
SOM-RS	AST and bytecode	Rust	none
ykSOM	bytecode	Rust	Rust built-in
JsSOM	AST	JavaScript	none
SOM++	bytecode	C++	copying, mark-and-sweep or generational

Table 1.1 Overview of official SOM implementations as listed on the official SOM website[1]

lexer and parser generation, custom bytecode as a compilation target, a custom virtual machine for executing this bytecode and a simple mark-and-sweep garbage collector.[11]

1.2.1 SOM-based languages implementations

"*SOM has been a platform for research that inspired a range of language implementations.*"[1] These are mostly used for further research of virtual machines and runtime environments.

The biggest has been *SOMns*, a Newspeak implementation based on TruffleSOM, implemented using the *Truffle framework*[6] and using the *Graal compiler*[7] for JIT compilation. This allows SOMns to reach performance that can compete with state-of-the-art VMs for dynamic languages.[12]

Built on top of SOMns is the *Moth*, an interpreter for Grace programming language[13]. It is mostly used as a platform for research on concurrency and tooling.

As an extension to TruffleSOM, *TruffleMATE* has more standard Smalltalk support. It also implements the *Mate approach* to building virtual machines which expose their whole structure and the behavior to the language level.[14]

1.3 Haskell compilers and virtual machines

Haskell is a statically-typed, purely functional language with lazy evaluation. It is a popular choice for lexing, parsing, compiling and transpiling code, mostly because these operations can be more or less described as transformations without side effects and therefore are easily managed in the context of pure functional language such as Haskell.

Notable examples of compilers written in Haskell include *Idris 1*, a general-purpose functional programming language with dependent types that compiles into C or JavaScript[15], *Agda*, a dependently typed programming language and proof assistant with target backend being GHC Haskell or JavaScript[16], or *Elm*, a purely functional language for building web-apps.

While it is popular for compilers, Haskell is not that popular for building interpreters, virtual machines and runtime environments. The main reason for this is performance. As a purely functional language with lazy evaluation, it is hard to compete with languages that offer direct memory access. More details will be provided in later chapters.

Nonetheless, there are some languages whose runtime is implemented in Haskell. The *Write Yourself a Scheme in 48 Hours*[17] is a tutorial book for writing a custom Scheme interpreter in Haskell, while also providing its own reference implementation. There are also some smaller and/or incomplete projects with little to no documentation, like *RType*[18]a Ruby interpreter, or *HST*[19], a Smalltalk interpreter[1]. A more complete list of these can be found at [20].

[1]The only pieces of evidence of this project mention that it exists, but the source code is not available

Chapter 2
Design

This chapter outlines the general design of the implemented virtual machine: the compiler frontend and backend, bytecode instructions, runtime environment and garbage collector.

The whole virtual machine is composed of two main parts, *source code compilation* and *bytecode execution*. Compilation compiles the source codes into a bytecode representation, which is then executed in the runtime environment. The compilation pipeline is illustrated on image 2.1.

This design of the VM being split into individual parts allows us to swap individual parts of the VM pipeline for different ones. Effectively this means that the compiler can target a different VM with a new backend, or we can extract the runtime environment and have a completely different compiler that targets our VM.

Usually, SOM virtual machines are designed with dynamically loaded classes, so that the initial runtime has only basic classes loaded and when the execution comes across a class that is not in the runtime environment, it is only then compiled from the classpath. Our VM is not designed with this dynamic class loading and before the execution of the runtime, all of the classes on the provided classpath are compiled and are introduced to the runtime. This simplifies the execution step but can be a performance issue, mostly because even classes that are never actually used are always compiled and loaded and are taking space on the heap.

2.1 Source files

The SOM classes are defined in plain text source code files with the suffix `.som`. There is always one class defined in one source file and the files follow a convention that its name is the same as the name of the file without suffix (e.g. class `Boolean` is defined in a file named `Boolean.som`). This naming is crucial for dynamic loading, as it allows for loading classes based on the file name without the need to parse the content first. The files do not have to follow any folder structure but can be located inside folders freely.

There are no classes available to the runtime that are not provided as a source code, so even classes from the standard library have to be defined in a `.som` files. The standard library is provided by the official SOM repository [3].

2.2 Compilation

The compilation is composed of a *frontend* and a *backend*. The frontend is the transformation of the source code to an abstract syntax tree (AST), and the backend is the transformation of this AST to the target representation, the bytecode instructions.

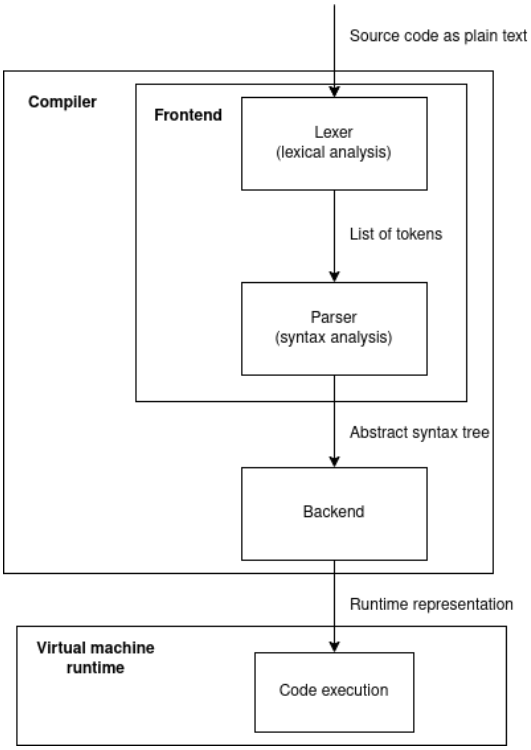


Figure 2.1 Compiler pipeline

There is no analysis done on the outputted AST. If there is a semantic error, like a circular dependency of classes or an undefined class, it is identified during compilation and only then an error is raised.

2.2.1 Compiler frontend

The *compiler frontend* transforms the plain text source code representation of a class to an abstract syntax tree. This transformation is further split into two parts, lexical and syntax analysis. Both of these compilation steps are specified by the ANTLR grammar in the official SOM repository, therefore giving a formal definition of how a source code file should be structured.

Lexical analysis (also called *lexing* or *scaning*) is the process of transforming the plain text of a file into a list of tokens, i.e. individual strings with assigned meaning.

These tokens are then consumed by the *parser* as a *syntax analysis*. Parser consumes the sequence of tokens and transforms them into an intermediate representation of the code. Usually, this representation is an *abstract syntax tree* (including our compiler), but it can be also transformed into a different representation or directly into the target representation, therefore eliminating the need for a backend compiler stage.

2.2.2 Abstract Syntax Tree

The *abstract syntax tree* (AST) is a structural representation of a SOM class. One class definition is transformed into one tree and the whole program is composed of a collection of these ASTs. The diagram outlining the AST structure can be seen in image 2.2.

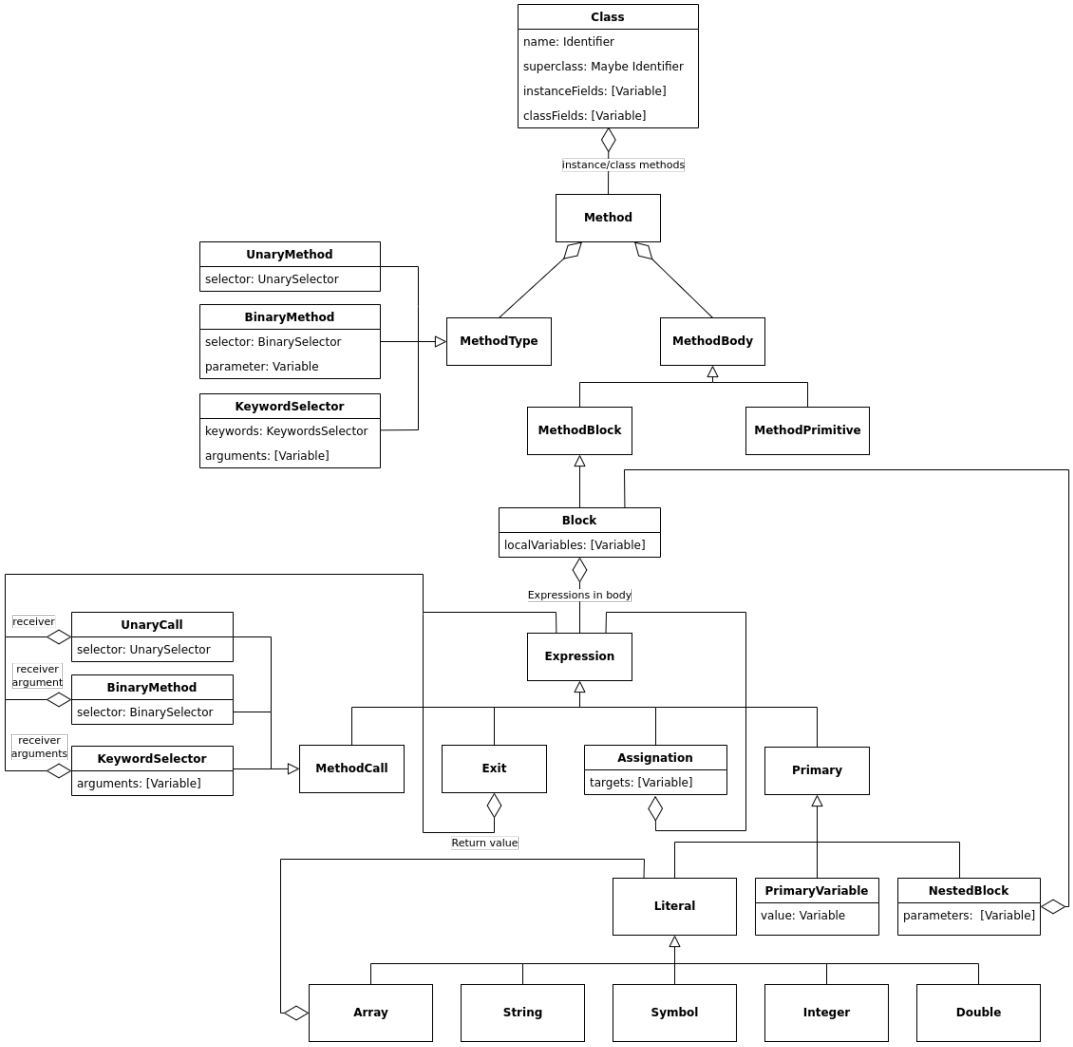


Figure 2.2 Abstract Syntax Tree schema

The individual nodes of an AST are:

- a *class*, the root node of the AST, containing the class name, superclass name (if it has one), instance and class methods definitions and instance and class fields definitions,
- a *method*, composed of a method type (unary, binary or keyword signature), the name of the method, its parameters (if there are any) and a method body, which is either a `primitive` method definition or a block definition,
- a *block definition*, yielding the definitions for local variables and a body composed of a sequence of expressions,
- an *expression*, which is either an exit expression, assignation, a method call or a primary expression (either a variable, nested block or a literal),
- a *nested block*, composed of parameters and a block definition,
- or a *literal*, which is one of five types, an *array* with a list of nested literals, a *symbol*, a *string*, an *integer number* or a *double precision floating point number* (usually shortened to *double*).

2.2.3 Compiler backend

The backend of a compiler generates a *runtime representation* of a class from the AST. For each class given in its AST form, two classes are created, one holding the instance methods and fields definition (called an *instance class*) and the second holding the class methods and fields definition (called a *metaclass*). It also provides these classes represented as an object, which holds the actual value of class fields, as classes themselves are not represented as objects in the runtime. This allows us to insert to have the static representation of class inserted as a value in an object and thus have the methods of an instance closer to the instance object.

If a method is not primitive, it is compiled into a sequence of bytecode instructions. Literals and blocks defined in these methods are brought into the global context.

If a method is defined as `primitive`, its implementation has to be provided by the compiler. If it is not, it is simply not available in the runtime and no error is raised.

2.3 Runtime environment

The runtime environment consists of:

- *Heap*, where all of the currently available objects live. The heap is managed by the garbage collector.
- *Globals*, a collection of global objects, which are either classes or instances of objects. Individual global objects can be accessed with a global index that can be interned from a symbol with *globals interner*.
- *Literals*, a collection of literal objects, which are integers, doubles, strings, symbols and blocks. Literals are accessed with a literal index. Each literal value is given a unique index through *literals interner*. This interning allows us to share one literal index for one literal across multiple classes and methods.
- *Execution stack*, a first-in-first-out collection of objects, where the message receiver and message arguments are passed by.
- *Call stack*, which consists of *call frames*. A call frame contains local variables, that is the currently executed method message receiver (available with the variable `self`), arguments and the local variables. The call frame also contains the currently executed method, the instruction counter in this method and the class of the method, which is used for searching methods in a superclass. If the current execution context is inside of a block, the call frame also contains the home context of the block. This capturing is used for accessing variables defined in the outer contexts as well as non-local returns. A new call frame is pushed on the call stack upon entering a new method (including primitive methods) and it is popped from the call stack after the method is exited.

Each object in the runtime holds its own fields, the class it is an instance of and, if it is a primitive object like Integer or String, it holds its primitive underlying value.

Each class holds the methods it has defined, a superclass (if it has one) and the representation of this class as an object.

2.4 Bytecode

The bytecode represents individual instructions given to the runtime environment. It consists of instructions for manipulating the stack, heap, literals and globals and also instructions for calling and exiting methods.

The instructions work as follows:

- `HALT` exits the runtime environment gracefully without any exceptions.
- `DUP` gets the value on top of the stack and pushes a duplicate of it.
- `POP` discards the top item of the stack.
- `PUSH_LITERAL i` pushes the literal on the given index to stack, transforming it to an object. If the literal is a block, it also captures the current top call frame in the call stack.
- `PUSH_LOCAL i i` pushes a variable from the current local scope on top of the stack. The first index is used for identifying the call frame to look in, 0 meaning the current call frame, 1 the captured call frame, 2 is the captured call frame of the current captured call frame and so on. The second index is then the field index in the given call frame.
- `PUSH_FIELD i` pushes to stack a field on the given index of the current object context (the object accessed as `self`).
- `PUSH_GLOBAL i` gets a global value on a given index and pushes it as an object to the stack. This means that if the global is an instance of a class, it's pushed directly, and if it is a class, it pushes the object representing the class. If no global value of a given index is defined, the runtime exits with an exception.
- `SET_LOCAL i i`, `SET_FIELD i` and `SET_GLOBAL i` pop the top value from the stack and sets the local variable, current context object or a global to this value respectively.
- `CALL i` calls a method of a given identifier on the object on top of the stack, pops the required amount of arguments and pushes a new call frame on the call stack with the popped valued as locals. It also reserves space for locally scoped variables and defaults their value to the `nil` value. This instruction expects that the arguments are ordered on the stack from bottom to top, meaning the last argument is popped next after the message target. This calling convention can be seen illustrated in image 2.3. If no method of the given identifier is defined on the receiving object, a runtime exception is raised and the execution of the virtual machine is halted.
- `SUPER_CALL i` calls a method with the same calling convention as `CALL`, but starts the search for the appropriate method in the superclass of the class where the currently executed method is defined.
- `RETURN` exits the currently executed method, popping the top of the call stack. The return value is passed through the stack.
- `NONLOCAL_RETURN` is an exit expression executed inside of a block. It gets the currently captured call frame and validates that it is still present on the call stack. If it is, the call stack pops the frames until it is reached. If it is not, an exception is raised and the runtime is exited with error.

2.5 Runtime execution

The virtual machine gets always one class as the *main class*. The runtime then expects that this class implements either `run` or `run:` instance method, which will be treated as an entry point into the program execution. This method is equivalent to the `main` function in C-like languages.

The execution of a SOM runtime is started by calling the method `initialize:` on the global object `system` with an array of arguments, where the first one is the main class to be executed

```
1 Calling = (
2     foo = (
3         self exec: 1 with: 2 and: 'Hello'
4     )
5
6     exec: x with: y and: z = (
7         | a |
8     )
9 )
```

Code listing 2.1 Calling convention code example

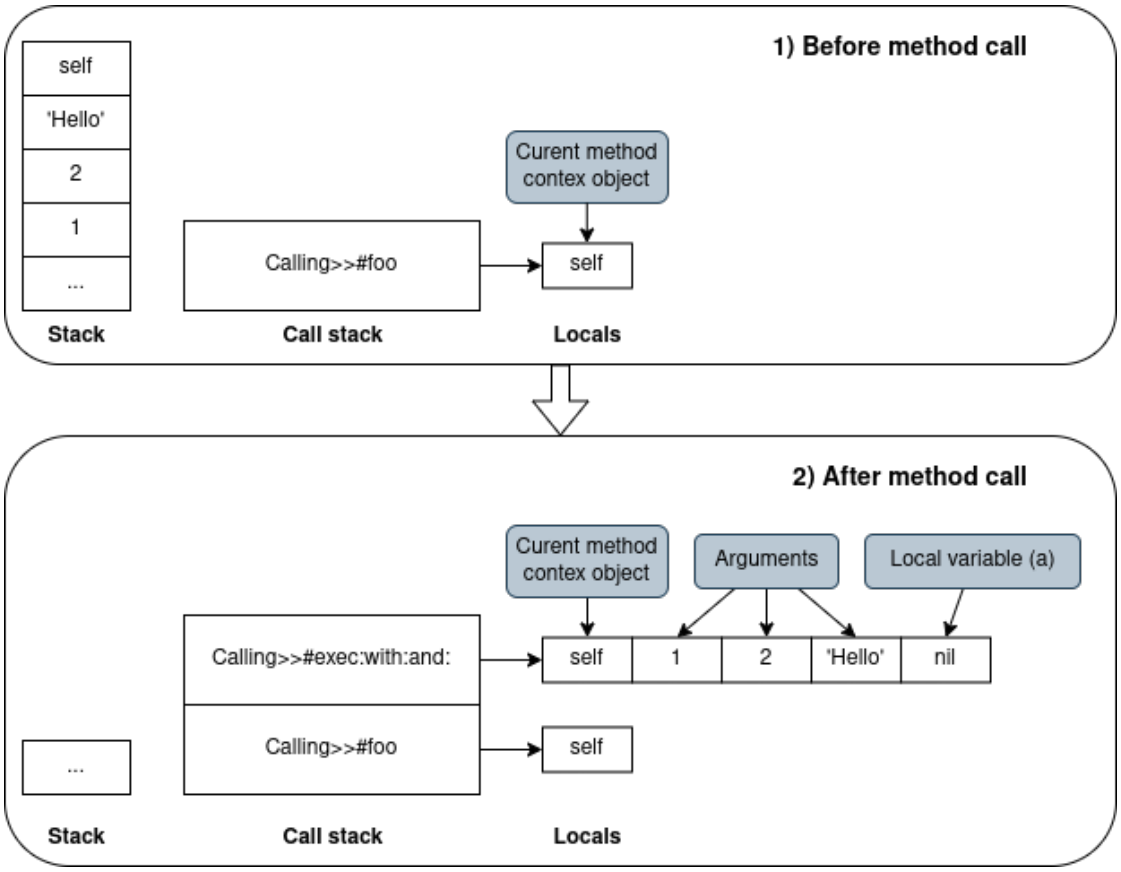


Figure 2.3 Illustration of calling convention of listing 2.1 of method call on line 3

and the rest of the arguments are given on the command line. This method then loads the main class, creates a new instance of it and if it implements the method `run:`, it calls it with the arguments array, otherwise it calls the method `run` without arguments.

2.6 Garbage collector

Since SOM does not provide a mechanism for manual memory management, it expects that it will be managed by the runtime environment. To correctly free up memory that is no longer used, a *garbage collector* is employed. There are many strategies for correctly evaluating which memory can be freed. Our implementation uses a simple *mark-and-sweep* algorithm[21, p. 18], which is composed of two stages:

- *mark* all currently reachable objects (on execution stack, as local variables or as global objects) and subsequent objects reachable from them,
- and *sweep*, i.e. delete unmarked objects and remove marking from marked objects.

The advantage of this algorithm is its simplicity. It is however usually very slow, as it stops the current execution of the runtime and has to go through every object on the heap.

Chapter 3

Implementation

In this chapter, we explore the implementation details of the HaSOM virtual machine: the used technologies and libraries, compilation steps, the intermediate representations and the representation of the runtime state.

The HaSOM virtual machine is written in Haskell, a purely functional programming language. It uses *The Glasgow Haskell Compiler* (GHC) with the *Haskell Stack Toolchain*[22] for compiling, executing, testing and building the documentation. It has no other external dependencies.

The Stack Toolchain is used in combination with *Stackage*, a stable set of Haskell packages. This allows us to specify a snapshot version and have a guarantee of compatibility between external packages. The snapshot used by HaSOM is LTS Haskell 19.33, which uses GHC version 9.0.2, as at the time of creating the project it was the newest snapshot that supported *Haskell Language Server*, a development tool for writing Haskell.

For testing, the *HSpec*[23] testing framework is used in combination with *QuickCheck*[24], a library for randomly testing individual expected properties, and a *hspec-golden*[25] package for executing *golden tests* on the lexer and parser. These golden tests (also sometimes called *snapshot tests*) generate a file with the expected result and on further executions validate that the newly outputted file content is the same as the initial snapshot.

Code documentation is written directly in the Haskell source files and is then generated using *Haddock*[26].

The *Git* versioning system is used for maintaining a code repository and a Git submodule dependency is linked to the official SOM Github repository, as it provides the standard library, various examples and a test suite.

The code is hosted on *GitLab*. It uses the *GitLab CI/CD*, a continuous integration and continuous delivery tool, for automated testing of the code.

3.1 Compiler

The compiler transforms the definitions of classes from plain text to their runtime representation. The compilation process is split into three parts, scanning the input into tokens with lexer, parsing these tokens into an abstract syntax tree and finally transforming all classes from AST to an initial runtime environment.

3.1.1 Lexer

The HaSOM lexer is written in *Alex*[27], a tool for generating lexical analyzers similar to *lex* and *flex* for C/C++. It expects a file with the suffix `.x` and generates a Haskell file with a defined `alexScanTokens` function. The Alex tool is invoked during the Stack build phase.

The Alex tool allows us to use one of many built-in wrappers. We chose a *posn-bytestring* wrapper, as it allows us to have the information about the token position in the source file for better error messages, as well as defining the input as `ByteString`, thus allowing us to have a UTF-8 encoded source files.

The output of the lexer is a list of `PosToken` type. It has two fields, first is the `AlexPosn` data type that defines the position of the token in the source string, and the second is the `Token` data type defined in the module `HaSOM.Lexer.Token` and is the actual token type.

Alex tokens are defined as *regular expressions* and when it is matched, it either yields a token output (defined as a Haskell code enclosed in curly braces) or it is ignored (denoted by a semicolon).

An example of HaSOM tokens definition can be seen in listing 3.1.1. First, we have definitions for `$digit` and `$alpha` macros which define a digits group and a group of alphabetical characters. Next, we have the `tokens` section, where the individual tokens are defined. First is a SOM comment definition on line 5, which is enclosed in double quotes and is ignored by the lexer, as well as a whitespace regular expression on line 6, also ignored in the output. On line 8 we see an Identifier token definition, which is an alphabet character followed by zero or many alphanumerical symbols and/or underscores. When this regular expression is matched, an `Identifier` token is yielded with the matched string as its field.

The Alex tool was chosen because the syntax of the Alex file is very similar to the official ANTLR grammar definition and the transformation from the official definition is pretty straightforward.

3.1.2 Parser

For a parser, we use the *Happy parser generator*[28] that generates a Haskell code from a grammar specification in Backus-Naur form (BNF). It works similarly to the *yacc* tool for C.

A Happy grammar is defined in a `.y` file. We use a convention, where tokens start with an uppercase letter and parser rules start with a lowercase letter. Each production is composed of a non-terminal symbol on the left, separated by a colon and followed by one or more expansions on the right, separated by `|`. Each production then has some Haskell code associated, enclosed in curly braces. In these code snippets, a `$n` holds the value n-th token or non-terminal rule in the expansion.

In the code example 3.1.2 in the rule `classdef`, we see the structure of a class definition. In the first expansion, the constructor `AST.MkClass` is called with the arguments

- `Identifier` (`$1`),
- `superclass` (`$3`),
- `instanceFields` (`$4`),
- the first `methodStar` (`$5`),
- `classFields` (`$7`),
- and the second `methodStar` (`$8`).

The parser runs in a monad `Either Text`, so the result of parsing is either an error message or the constructed abstract syntax tree.

The HaSOM parser definition is mostly based on the official ANTLR grammar definition, but since ANLTR is based on the Extended Backus-Naur form, the transformation is not one-to-one, as repetitions and optional expansions have to be explicitly denoted. For example in the code snippet 3.1.2, the rule `variableStar` corresponds to an ANTLR rule `variable*`, where the `*` symbol is an EBNF notation.

```
    $digit = 0-9
    $alpha = [a-zA-Z]

    tokens :-
        \" (\n | ~\")* \" ;
        $white+ ;

        $alpha [$alpha $digit _]* { tokT Identifier decode }
```

(3.1.1) An example of HaSOM tokens definition in Alex

```
classdef :: { AST.Class }
classdef  : Identifier Equal superclass instanceFields methodStar
            Separator classFields methodStar
            EndTerm
            { AST.MkClass $1 $3 $4 $5 $7 $8 }
            | Identifier Equal superclass instanceFields methodStar
            EndTerm
            { AST.MkClass $1 $3 $4 $5 [] [] }

superclass :: { Maybe AST.Identifier }
superclass : Identifier NewTerm { Just $1 }
           | NewTerm            { Nothing }

instanceFields :: { [AST.Variable] }
instanceFields : {- empty -}        { [] }
               | Or variableStar Or { $2 }

classFields :: { [AST.Variable] }
classFields : {- empty -}        { [] }
            | Or variableStar Or { $2 }

variableStar :: { [AST.Variable] }
variableStar : {- empty -}           { [] }
             | variableStar variable { $2 <:> $1 }
```

(3.1.2) An example of HaSOM parser definition in Happy

Code listing 3.1 Lexer and parser examples

3.1.3 Abstract Syntax Tree

The *abstract syntax tree* generated from the parser is implemented as a plain Haskell algebraic data type with nodes of types `Class`, `Method`, `Block`, `Expression`, `NestedBlock` and `Literal`. It is simplified from the tree generated by the parser, most notably it does not have different node types for expressions, evaluations, message receivers (also called primaries) and literal numbers, but rather groups all of them under a single `Expression` type.

We also define a fold algebra on this AST in the package `HaSOM.AST.Algebra`. Folding on a tree structure is the process of reducing the nodes of a tree via a given set of functions, from leaves to the final root node. These functions do not have to recursively process the children of the node but rather handle them in their transformed representation as the recursion is handled implicitly by the higher-order folding function.

This fold algebra is implemented because it greatly simplifies the transformation of the AST to a different form. It is used by the AST pretty printing function, the bytecode compiler and could be also used to perform analysis and optimizations of the AST.

3.1.4 Bytecode compiler

The *bytecode compiler*, defined in the package `HaSOM.Compiler` transforms a list of abstract syntax trees and transforms them into their runtime representation. The result of a compilation is a global variables collection, literals collection, the collection of core classes and the initial heap structure, that is objects to be added to a heap with the index to put them on.

As defined in chapter 2.2.3, each abstract syntax tree is transformed into an instance class, metaclass and objects representing these classes.

For a class to be compiled, its superclass has to be compiled first, because its fields have to be known. If the superclass is not defined or a cyclical inheritance is detected, the compilation halts and an error message is returned. Otherwise, the compiler takes the field definitions of the superclass and proceeds to run a compiling fold algebra on the AST.

If a method is defined as primitive, the compiler tries to find the primitive function in the global context. If none is found, the method is silently dropped and not present in the runtime.

When a literal or a nested block is encountered, it is added to the `literals` table and it is represented by the `LiteralIx` type. When a global variable name is encountered, it is interned by the `globals` table and then represented as the `GlobalIx` data type.

3.2 VM Runtime

The execution of the program is initialized by the `doExecute` function in package `HaSOM.Run`. It takes the compilation result from the compiler, initializes an empty garbage collector heap to the expected state and creates a new empty execution stack and empty call stack. Then it runs the `bootstrap` function, which effectively puts a `system` global variable on the execution stack with the command line arguments as an array and sends the `initialize:` message to it. The execution is then passed to the interpreter, described later in this chapter.

3.2.1 Runtime state

The state of the HaSOM runtime is composed of a heap, a garbage collector, an execution stack, a call stack, a collection of literals and a collection of global variables.

Each type of runtime object (SOM object, global variable, literal, object field and local variable) has its separate associated index type denoted by the suffix `Ix`. These are defined as *newtypes* of `Int` and defined in the package `HaSOM.VM.Object.Ix`. A newtype in Haskell creates a new data type that cannot be implicitly converted to other types, but via the *deriving*

mechanism can inherit properties of the contained type. This allows us to strongly separate different index types as we cannot for example query a global variable by a literal index.

Most of the internal objects of the VM are parametrized by the type of primitive function they use. Because we want the primitive function to be typed by the runtime state and then the primitive functions are fields inside the state, not being parametrized would create a circular dependency. This could be solved by defining all of the runtime objects in one package, but it is not a good programming practice. Nonetheless, we create type synonyms in the package `HaSOM.Universe`, where these objects are specialized to the used type of primitive function. They are named with the `Nat` suffix.

3.2.1.1 Object representation

A *SOM object* in the runtime has two representations, either an index `ObjIx` that is unique for each reachable object or the `VMObject` type. The `ObjIx` works similarly to pointers, as when one object is in the runtime in multiple places, it is represented by this index and needs unwrapping through the heap in order to get the `VMObject` representation. This approach was chosen because Haskell does not provide a simple way to work with pointers to values. It has the `IORef` and `STRef` types that work similarly to pointers in the `IO` and `ST` monads respectively, but these also require explicit unwrapping and are generally harder to optimize by the Haskell compiler as they are not pure functions, thus may also lead to a slower execution.

The `VMObject` is a Haskell algebraic data type. Each type of built-in object has an associated constructor, as they all define their underlying fields. All objects created by the method `new` on an `Object` class are constructed with the `InstanceObject` constructor.

We see an example of this in code listing 3.2. Here, the `ClassObject`, represents a class as an object and has a `classOf` field that is an index this object is created from. The `IntObject`, an instance of `Integer` class, has a field that holds the integer primitive value. All objects also hold the class they are an instance of and an array of their fields.

```
1  newtype Fields = MkFields {runFields :: Arr.VMArray FieldIx ObjIx}
2
3  data VMObject f
4    = InstanceObject
5        { clazz :: VMClass f,
6          fields :: Fields
7        }
8    | ClassObject
9        { clazz :: VMClass f,
10         fields :: Fields,
11         classOf :: GlobalIx
12       }
13   | IntObject
14       { clazz :: VMClass f,
15         fields :: Fields,
16         intValue :: Int
17       }
```

Code listing 3.2 Partial definition of the `VMObject`

All objects live on the *heap*, which is managed by the *garbage collector*. The heap then allows us to create new objects (get a unique `ObjIx` and have it represent a given `VMObject`), dereference the `ObjIx` and modify an object on given `ObjIx`.

3.2.1.2 Global object and classes representation

All of the *global objects* are in a `VMGlobals` collection which stores two types of values, an object on heap represented as `ObjIx` or a class represented by the `VMClass` type. Global objects are passed as `GlobalIx` for the same reasons we use the `ObjIx`. The `VMGlobals` also exposes a function for interning a global index from its string representation, used when a global variable is accessed by its name.

In listing 3.3, we see the definition of a `VMClass`. It holds the name of the class, the definitions of its instance fields, its superclass as a `GlobalIx` (if it has one), instance methods and the `asObject` field that represents the class as an object.

```
data VMClass f = MkVMClass
  { name :: Text,
    instanceFields :: VMArray FieldIx Text,
    superclass :: Maybe GlobalIx,
    asObject :: ObjIx,
    methods :: VMMethods f
  }
```

Code listing 3.3 Definition of `VMClass`

The runtime also contains a collection of *core classes*, which are needed for constructing primitive objects.

3.2.1.3 Method representation

A *method* in the runtime is represented by the `VMMethod` algebraic data type (defined in listing 3.4). It has a signature (name of the method and the class where it is defined), the number of parameters and local variables and the body of the method. If it is a primitive (also called a *native method*), the body is defined as a pure Haskell function and has the type it is parametrized on. Otherwise, the body is a sequence of bytecode instructions, contained in the data type `Code`. Primitive methods do not use the local fields as bytecode methods do, so the `localCount` value would always be zero, thus this field is omitted from the `NativeMethod` constructor.

```
data VMMethod f
  = -- | Method represented in bytecode
    BytecodeMethod
      { signature :: Text,
        body :: Code,
        parameterCount :: Int,
        localCount :: Int
      }
  | -- | Method represented by Haskell function
    NativeMethod
      { signature :: Text,
        nativeBody :: f,
        parameterCount :: Int
      }
```

Code listing 3.4 Definition of `VMMethod`

3.2.1.4 Literal representation

The *literals collection* contains all numbers, strings, symbols, arrays and nested blocks used by the compiled methods. Other bytecode-based implementations usually define a separate collection of constants and blocks for each method, as they are indexed by a byte and therefore limited to 256 unique values, which could be less than needed if a global collection was used. Because we are indexing literals by an `LiteralIx` type that internally uses `Int`, we are not constrained by this limitation and can use a global literals collection, therefore reducing duplication of literals and increasing memory effectiveness.

3.2.1.5 State of execution

For storing the information about currently executed methods, a *call stack* is needed. It is a first-in-first-out data structure composed of individual *call frames*. Each call frame is then either a pure value, or if it has been captured by some block as its *home context*, it is encapsulated in `IORef`. This wrapping allows the block to share the values of the variables at the time of execution, not at the time of creation.

A *call frame*, represented by the type `CallFrame` and defined in listing 3.5, has the values of local variables, the currently executed method and the class that holds this method (for executing a call to superclass). It then has the height of the execution stack at the time of creating the call frame, which is used when a `restart` is called on the block, as it reset the values on the execution stack. If the call frame is a context for a block execution, it also holds the captured home context of the block.

The local values are sorted as follows: first is the object context of the method (the value of `self`), then the values of arguments in the order they are defined and lastly the values of local variables.

```
1  data CallFrame f
2    = MethodCallFrame
3        { methodHolder :: VMClass f,
4          method :: VMMethod f,
5          pc :: InsIx,
6          locals :: VMArray LocalIx ObjIx,
7          objStackHeight :: Int
8        }
9    | BlockCallFrame
10       { methodHolder :: VMClass f,
11         method :: VMMethod f,
12         pc :: InsIx,
13         locals :: VMArray LocalIx ObjIx,
14         objStackHeight :: Int,
15         capturedFrame :: IORef (CallFrame f)
16       }
```

Code listing 3.5 Definition of `CallFrame`

For passing arguments to a method, the *execution stack* is needed. It is implemented as a first-in-first-out collection with elements of type `ObjIx`. The calling convention for passing arguments is described in the chapter 2.4.

The runtime state also contains a *garbage collection flag* of type `GCFlag`. It signals if the garbage collection should be run. Contained is also the *runtime start time* for measuring the execution time.

3.2.1.6 State management

Since Haskell is a purely functional language, the state of a program cannot be represented with a mutable variable. We can explicitly pass the state as an input and output to individual functions, but this leads to a lot of boilerplate code and increases the complexity of the code. A typical way to overcome this is encapsulating a computation in a *State monad* that passes the state around implicitly and defines `get` and `set` functions for manipulating the value of the state inside this monad. This way, the computation is still a pure function with immutable values, but the code looks like it is using mutable variables and is much more concise.

The problem then becomes the usage of multiple monads, as they do not compose well and when used in their pure form they need a lot of boilerplate code to individually handle the nested monads. This constraint is usually overcome with *monad transformers*, as they allow the composition of multiple monads in a more accessible way. The package usually used for this is *transfomers*[29] in combination with *mtl*[30]. The drawback of monad transformers is a static ordering of the monads, meaning that we specify the ordering of monads and this has to stay the same across all functions using these monads. It also does not allow for multiple monads of the same type (e.g. Reader, State) to be used more than once.

For our implementation, we chose an alternative to monad composing, a library called *Extensible effects*[31]. It is based on the idea of *effects*, where each function specifies which effects are used (e.g. which monads need to be instantiated for it to be run) but not how they are ordered. It also allows us to split the state of the runtime environment into individual states and specify which parts of the environment are used by which function. We use the `State` monads in combination with the `Reader` monad, a type of state monad that can be read, but not changed, the `Exc` exception effect to raise an error that halts the execution of runtime, as well as use the `IO` monad for manipulating the command line output, the file system and the `IORef` data type. The state of the whole runtime (called the *Universe*) is then defined as a composition of these effects in the package `HaSOM.VM.Universe` as `UniverseEff`.

As an example, the type of a function to transform a global index into a `VMClassNat` data type is

```
getClass :: (GlobalsEff r, Member ExcT r) => GlobalIx -> Eff r VMClassNat
```

where `GlobalsEff r` is an alias to `Member (State VMGlobalsNat) r` and `ExcT` is a specification of exception monad as `Exc Text`. From the type of the function, we can see that this function is accessing the globals collection and that it can fail with a `Text` error message.

A drawback of using Extensible effects is the need for explicit type declarations, as the Haskell compiler cannot infer the types of used effects.

3.2.2 Garbage collector

The *garbage collector* holds the heap and a stack of currently available indices. The heap is represented as an array indexed by the `ObjIx` type. When a new object is created, it is first checked if the free capacity (size of the stack with indices) is under 10% of the overall capacity. If this is true, the state of `GCFlag` is set to `RunGC` as a signal to the interpreter that collection should be run, as described later in this chapter. Then the GC pops the top index from the indices stack and initializes that object to the `nil` value. If the stack is empty, the heap is expanded and the new indices are made available to allocate.

The garbage collector uses the *mark-and-sweep* algorithm mentioned in the chapter 2.6. When a garbage collection is invoked, the runtime collects all currently available object indices from call frames, globals, execution stack and other objects into a `HashMap`. It then passes the collected object into the GC for the collection phase.

The runtime cannot run the garbage collection instantly when the low capacity is detected, as some of the objects may not be reachable in the *mark* phase, but still needed by the runtime to be valid.

3.2.3 Primitive functions

The type of a *primitive function* is `(UniverseEff r, Lifted IO r) => Eff r (Maybe Int)`. This means that primitive functions have access to the whole runtime state and the `IO` monad for printing into the standard output and accessing the file system. The return value is a flag signifying halting the runtime, see the next chapter.

We also define a helper function `pureNativeFunction` that takes a function with more restricted access to the state, where all of the `UniverseEff` effects are available, except the execution stack and call stack. This restricted function then gets its specified amount of arguments and the `self` object, and has to return a return value of type `ObjIx`, which is pushed to the execution stack. After executing this restricted function, the current call frame is popped from the call stack.

This helper allows us to work with most of the primitive functions in a uniform and safe way so that we execute all of the necessary operations like validating the arguments count and leaving the call stack and execution stack in the desired state.

In the listing 3.6, we can see the usage of this helper function. We expect two arguments to this method, denoted by the type parameter `@N2`. Then we pass the `pureNativeFunction` a lambda function, whose first argument is `self` and the second is the list of two arguments named `g` and `val`. On line 3 we then cast the `ObjIx` to a `Symbol` object, extracting the underlying string representation of the symbol. Then we intern this string to a `GlobalIx` on line 4 and set a global object on this index to the value of argument `val` on line 6. Lastly, the primitive function returns the `self` object.

```
1 globalPut :: NativeFun
2 globalPut = pureNativeFun @N2 $ \self (g :+: val :+: Nil) -> do
3   symbol <- castSymbol g
4   idx <- internGlobalE symbol
5
6   setGlobalE idx (ObjectGlobal val)
7   pure self
```

Code listing 3.6 Implementation of primitive method `global:put:` in class System

3.2.4 Interpreter

The *interpreter* is the main entry point to the runtime execution. It is defined in the package `HaSOM.Interpreter` as the function `interpret` (see the listing 3.7). This function defines the used effect `Lifted IO`, so the individual instructions and primitive function have access to the `IO` monad, a `UniverseEff` effect containing the state of the runtime described earlier in this chapter, and a `TraceEff` containing flags that signify if tracing should be written into the standard output.

When interpreting, we first extract the top of the current call stack (on line 3). We then pattern match on the contained method (on line 4). If the current method is a *primitive function* (case of `NativeMethod`), its body is executed. Otherwise, it is a *bytecode method* (case of `BytecodeMethod`), it fetches an instruction from this method on the current program counter (lines 7–10), advances the program counter by one (line 12), and executes the instruction (line 13).

The result type of both instruction execution and native function is a `Maybe Int`. If the result is some integer value, the runtime exits with this value as an exit status. Otherwise, the interpreter loop continues. This pattern matching is done by the function `maybe` on line 19.

The function `mbyRunGC` executes a garbage collection if the state of `GCFlag` is set to `RunGC`. It is invoked if the executed instruction was `CALL`, `SUPER_CALL` or a primitive function.

```haskell
interpret :: (Lifted IO r, UniverseEff r, TraceEff r) => Eff r Int
interpret = do
  cf <- getCurrentCallFrame

  r <- case method cf of
    BytecodeMethod {signature, body} -> do
      ins <-
        throwOnNothing
          ("Index " <+ showT (pc cf) <+ " fell out of code block")
          (getInstruction (pc cf) body)

      advancePC
      res <- executeInstruction ins
      mbyRunGC
      pure res
    NativeMethod {signature, nativeBody} -> do
      runNativeFun nativeBody

  maybe interpret pure r
```

Code listing 3.7 Definiton of the `interpret` function, simplified without a tracing

Upon entering the `executeInstruction`, defined as in listing 3.8, the current instruction is pattern matched. If it is the `HALT` instruction, the virtual machine exits gracefully with the exit code 0. Otherwise, a specific action is called with the arguments from the bytecode instruction. These actions are defined in the module `HaSOM.VM.Universe.Instructions`.

As an example of an action, we use the instruction `SET_GLOBAL` and its associated action `doSetGlobal` in listing 3.9. From the signature of the function, we see that it accesses the execution stack (`ObjStackEff`), a globals collection (`GlobalsEff`) and that it can fail (`MemberExcT`). It has a type of `GlobalIx` as a parameter. The action pops a `ObjIx` from the execution stack (function `popStack`), passes it into `ObjectGlobal` constructor, and sets a global variable on the given index to this value.

We omit the implementation of the helper functions to keep this example small and provide only their type. Still, we can see that the `popStack` function is accessing the execution stack, can fail and returns a `ObjIx`, whereas the `setGlobalE` modifies the globals collection with a given `ObjIx` and a global object definition.

```
executeInstruction ::
  (Lifted IO r, UniverseEff r, TraceEff r) =>
  Bytecode ->
  Eff r (Maybe Int)
executeInstruction HALT = pure $ Just 0
executeInstruction bc = do
  case bc of
    DUP -> doDup
    POP -> doPop
    PUSH_LITERAL li -> doPushLiteral li
    PUSH_LOCAL env li -> doPushLocal env li
    PUSH_FIELD fi -> doPushField fi
    PUSH_GLOBAL gi -> doPushGlobal gi
    SET_LOCAL env li -> doSetLocal env li
    SET_FIELD fi -> doSetField fi
    SET_GLOBAL gi -> doSetGlobal gi
    CALL li -> doCall li >> mbyRunGC
    SUPER_CALL li -> doSupercall li >> mbyRunGC
    RETURN -> doReturn
    NONLOCAL_RETURN -> doNonlocalReturn
  pure Nothing
```

Code listing 3.8 Definition of the `executeInstruction` function

```
-- In package HaSOM.VM.Universe.Instructions
doSetGlobal :: (ObjStackEff r, GlobalsEff r, Member ExcT r) => GlobalIx -> Eff r ()
doSetGlobal gi = popStack >>= setGlobalE gi . ObjectGlobal

-- In package HaSOM.VM.Universe.Operations
popStack :: (ObjStackEff r, Member ExcT r) => Eff r ObjIx
setGlobalE :: (GlobalsEff r) => GlobalIx -> VMGlobalNat -> Eff r ()
```

Code listing 3.9 Definition of `SET_GLOBAL` instruction execution, with the signatures of helper functions

Chapter 4

Assessment

In this chapter, we evaluate the correctness of our implementation of Simple Object Machine and compare the implementation with SOM++.

4.1 Correctness

As previously mentioned, the Simple Object Machine language has specified syntax with an ANTLR grammar but does not have a semantics specification. The strategy we chose for implementing most primitive methods and runtime execution cases was to follow the official implementations, mostly SOM++. Still, there are some cases where our VM works differently from other implementations:

Various *SOM runtime errors* are handled differently. These include non-local return in a block that has escaped its home context, reacting to an object not understanding the message and encountering an unknown global variable. On each of these errors, a method corresponding to the error type (`escapedBlock:`, `doesNotUnderstand:arguments:` or `unknownGlobal:`) should be called on the object where this error has occurred, allowing potential error recovery. Our implementation instead prints an error message and halts the execution of the VM.

HaSOM uses internally the `Int` Haskell data type for representing Integer values, which is guaranteed to be able to hold values as a 32-bit signed integer, as specified by the Haskell2010 report[32, p. 181]. The SOM language does not set any expectations, but other implementations represent them with at least a 64-bit signed integer or an arbitrary precision integer.

The implementation of method `objectSize` in class `Object` always returns the integer `-1`, as measuring the size of an object in Haskell proved to be challenging. Methods `inspect` and `halt` in `Object` are not implemented and they currently only print a message to standard output.

4.1.1 Test suite

The SOM repository provides a complex test suite[3, TestSuite]. It is written directly in SOM and tests most of the language constructs and primitive functions.

The test suite cannot be executed directly with our VM, because of the different behavior described above. Our implementation provides a custom SOM test runner that ignores these crashing tests. Despite this restriction, our compiler can compile all of the classes in the test suite.

In total, out of the 205 provided tests, our implementation passes 195 of them, where 7 tests are ignored because of the crashing behavior described previously.

4.2 Comparison with SOM++

The biggest difference between HaSOM and SOM++ is the implementation language. SOM++ is written in an object-oriented paradigm in C++, whereas HaSOM contains purely functional Haskell code apart from the lexer and parser definitions. The SOM++ repository contains almost 13000 lines of code, whereas the HaSOM code base is less than 6500 lines of code, both including tests.

Both HaSOM and SOM++ are bytecode-based, but SOM++ employs a variety of optimizations. These include directly compiling `ifTrue:` and `ifFalse:` methods to C++ `if` statements, optional caching of integers or usage of tagged integers.

4.2.1 Compiler

The SOM++ compiler is written in pure C++ and is composed of a lexer and parser. It does not generate an abstract syntax tree, but rather directly compiles into the target representation. The parser is written in a recursive-descend style.

The SOM++ compiler allows the array to be only defined with number literals elements, both integers and doubles, whereas the HaSOM implementation allows for any literal to be nested inside of the array, including another array.

The C++ implementation also implements the dynamic loading of classes, so a class is not parsed and compiled until it is needed or explicitly called.

4.2.2 Bytecode

The bytecode instructions of SOM++ are very similar to the ones implemented by HaSOM but have still some differences. Whereas local variables and arguments in HaSOM are grouped into one array, SOM++ has different instructions for manipulating these, namely `POP_LOCAL`, `POP_ARGUMENT`, `PUSH_LOCAL` and `PUSH_ARGUMENT`. SOM++ has also special instructions `JUMP`, `JUMP_IF_TRUE` and `JUMP_IF_FALSE`. These are used in the compiler as optimized jumps in the method execution.

The code of a SOM++ method is an array of bytes. If a byte in this array is interpreted as an instruction, its operands are followed directly after. On the other hand, HaSOM bytecode instructions are an algebraic data type and each instruction holds its operands. This means that HaSOM instructions are more type-safe, as an instruction cannot be interpreted as an operand and vice versa, but it is less cache efficient, as the Haskell data types are boxed and internally represented as pointers.

4.2.3 Runtime representation

Both interpreters have a similar structure of the interpreter loop, but where the Haskell implementation fetches an instruction, does pattern matching on it, extracts the operands from it directly and advances the program counter by one, the C++ implementation does a `goto` jump based on the byte value of the instruction, then has to fetch individual operands separately and advance the program counter differently on each instruction, depending on the number of operands.

The big difference between the two implementations is in object representation. Whereas in HaSOM, objects are represented with `ObjIx`, a simple wrapper around `Int` that needs to be passed to the global garbage collector in order to get the actual `VMObject` data type, SOM++ uses two main types of pointers, `GcOop*` and `VmOop*`. The former is a heap value, a pointer to an object that needs to be first handled with a read barrier when a generational garbage collector is used, and the latter is a value pointer, an object that has been handled and its inner structure

can be accessed. The heap contains all pointers to the objects but does not need to be called when an object is dereferenced.

The fields in a C++ object are also managed differently. When a new object is being created, additional space is allocated directly after the newly created object which is then used for storing object fields. This is faster than having a separately allocated array of fields, as they are as close to the actual object as possible.

The overall structure of both runtimes is very similar, employing a universe with heap and garbage collection, a call stack and a collection of global objects. In HaSOM, there is a global collection of literals, that includes all of the currently available strings, symbols, integers, doubles and blocks. SOM++ has a global collection of symbols, but strings, symbols, number constants and blocks are uniquely stored in each method. The stack for passing arguments and callable objects that is global in HaSOM is encapsulated for each call frame in the call stack in SOM++. SOM++ also differentiates between local fields and arguments, as they are stored separately.

4.2.4 Garbage collector

SOM++ can be compiled with different types of garbage collecting strategies, at the time of writing being copying GC, generation GC and mark-and-sweep GC. HaSOM has only one type of garbage collector that uses a mark-and-sweep algorithm. Both mark-and-sweep collectors function the same, the HaSOM mark phase is marking objects by collecting them into a set data structure, and SOM++ marks an object pointer as marked by changing its member variable.

Both implementations run garbage collection after a method call if the collection was triggered. SOM++ also runs the collection after pushing a block or a global variable on the execution stack.

4.2.5 Primitive functions

When a primitive method is executed in SOM++, it gets passed an `Interpreter` and the current *call frame* (which also holds all previous call frames) and it has to manually manipulate these primitives. Primitive method execution does not get a new call frame inserted on the call stack as it is executed instantly.

On the other hand, HaSOM implements a variety of helper functions to handle most of the primitive functions. These helpers include the manipulation of `self` field, the arguments count and values, return value and the casting of objects to their correct type. A primitive function also gets a new call frame in the call stack which it has to pop because its execution is handled in the interpreter loop.

4.2.6 Execution speed

For comparing the two implementations, we used the *Are We Fast Yet* benchmarks[8], which are also used by the official SOM site to compare individual implementations. We used the included micro benchmarks, each executed with the default parameters. Unfortunately, we were not able to run the *Mandelbrot* and *NBody* benchmarks, as these returned wrong results on both implementations and thus were not able to finish.

As we can see in the table 4.1, the gap between the speed of the implementations is significant. This is to be expected, as SOM++ is among the fastest SOM implementations[1]. It is still interesting to see HaSOM is at least a hundred times slower.

The largest disparity is in the *Storage* test, where the Haskell implementation is over 1500 times slower. We assume that this is because the test is focused on stressing the garbage collector and since our garbage collector does not reduce the size of the heap once it is expanded, it slows down the garbage collection.

Benchmark name	Execution time in SOM++	Execution time in HaSOM++
Bounce	71ms	13 637ms
List	106ms	38 086ms
Permute	105ms	10 038ms
Queens	85ms	13 997ms
Sieve	90ms	15 318ms
Storage	66ms	104 689ms
Towers	104ms	11 023ms

Table 4.1 Comparision of individual Are We Fast Yet micro benchmarks

These benchmarks should not be taken as concrete values, as factors such as cold starts and longer running programs were not taken into consideration, but serve only to illustrate the preformance difference between the two runtimes.

Conclusion

This thesis gives an overview of a Simple Object Machine (SOM), its syntax and semantics. It explores the official existing implementations and then provides a new implementation written in Haskell.

The implementation is very basic but provides a working virtual machine for the compilation and execution of SOM programs. All of the language constructs are available, including classes, objects and literals, message passing and execution of non-local returns. It implements all of the primitive methods of the standard library, excluding the methods `objectSize`, `inspect` and `halt`.

Our implemented VM is based on bytecode instructions and the runtime implements a simple garbage collector based on the mark-and-sweep algorithm. It does not implement any optimizations.

The code base greatly uses the features of Haskell, including a strong type system, no side effects (only explicitly typed effects), lazy evaluation and immutable data. It allows for a definition of a framework for manipulating the individual parts of runtime without the need to redefine this behavior. This comes at the cost of speed, as the design of Haskell does not allow direct manipulation of memory and therefore abstractions and less optimized constructs have to be used.

Compared to other implementations, our virtual machine deviates in error handling, where instead of calling methods from the standard library, built-in constructs stop the execution and print an error message.

When compared to the C++ implementation of SOM, our implementation exhibits a considerable performance gap, most notably when the garbage collector is stressed. On the other hand, the code base is much more concise, as many functions are reusable.

Future work

The speed performance of our virtual machine is poor, especially when compared to other implementations. There are a few ways that it could be improved, by rewriting the runtime using a mutable state and with strict evaluation or by using a more advanced and robust garbage collector. Still, it is unlikely that the execution speed could be comparable to other SOM implementations, due to the limitations posed by Haskell.

Another approach could be taken, extracting only the implemented compiler and targeting another runtime. Using the abstract syntax tree and its fold algebra, there is already a framework for implementing optimization on the AST. Also, by employing the Haskell strong type system and pattern matching, further optimizations could be implemented easily.

Bibliography

1. *SOM: Simple Object Machine* [online]. 2022. [visited on 2023-03-29]. Available from: `https://som-st.github.io/`.
2. MARR, Stefan. Another Decade of SOM Language Implementation. 2019. Available also from: `https://stefan-marr.de/2019/04/simple-object-machine/`.
3. *SOM - Simple Object Machine* [online]. GitHub, [n.d.] [visited on 2023-04-08]. Available from: `https://github.com/SOM-st/SOM`.
4. BERGEL, Alexandre; CASSOU, Damien; DUCASSE, Stéphane; LAVAL, Jannik. *Deep Into Pharo*. Lulu. com, 2013.
5. *PySOM - The Simple Object Machine Smalltalk* [online]. GitHub, [n.d.] [visited on 2023-04-16]. Available from: `https://github.com/SOM-st/PySOM`.
6. WIMMER, Christian; WÜRTHINGER, Thomas. Truffle: A Self-Optimizing Runtime System. In: *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*. Tucson, Arizona, USA: Association for Computing Machinery, 2012, pp. 13–14. SPLASH '12. ISBN 9781450315630. Available from DOI: `10.1145/2384716.2384723`.
7. ORACLE. *GraalVM* [online]. [N.d.]. [visited on 2023-05-08]. Available from: `https://www.graalvm.org/`.
8. MARR, Stefan; DALOZE, Benoit; MÖSSENBÖCK, Hanspeter. Cross-language compiler benchmarking: are we fast yet? *ACM SIGPLAN Notices*. 2016, vol. 52, no. 2, pp. 120–131.
9. *ykSOM* [online]. GitHub, [n.d.] [visited on 2023-04-16]. Available from: `https://github.com/softdevteam/yksom/`.
10. *yk* [online]. GitHub, [n.d.] [visited on 2023-04-16]. Available from: `https://github.com/ykjit/yk`.
11. ROVŇÁK, Rudolf. *Implementace Virtualniho Stroje SOM*. 2021. MA thesis. České vysoké učení technické v Praze. Vypočetní a informační centrum.
12. *SOMns - A Simple Newspeak Implementation* [online]. GitHub, [n.d.] [visited on 2023-04-16]. Available from: `https://github.com/smarr/SOMns`.
13. *A Graceful Blog* [online]. [N.d.]. [visited on 2023-04-16]. Available from: `http://gracelang.org/applications/home/`.
14. CHARI, Guido; GARBERVETSKY, Diego; MARR, Stefan; DUCASSE, Stéphane. Towards Fully Reflective Environments. In: *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* Pittsburgh, PA, USA: Association for Computing Machinery, 2015, pp. 240–253. Onward! 2015. ISBN 9781450336888. Available from DOI: `10.1145/2814228.2814241`.

15. BRADY, Edwin. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*. 2013, vol. 23, pp. 552–593. ISSN 1469-7653. Available from DOI: 10.1017/S095679681300018X.

16. AGDA GITHUB COMMUNITY. *Agda 2* [online]. GitHub, [n.d.] [visited on 2023-03-29]. Available from: https://github.com/agda/agda.

17. WIKIBOOKS. *Write Yourself a Scheme in 48 Hours — Wikibooks, The Free Textbook Project* [online]. 2020. [visited on 2023-04-16]. Available from: https://en.wikibooks.org/w/index.php?title=Write_Yourself_a_Scheme_in_48_Hours&oldid=3664801.

18. YOSHIDA, Yuichi. *RType: Yet Another Ruby Interpreter, written in Haskell* [online]. 2014. [visited on 2023-04-16]. Available from: http://research.nii.ac.jp/~yyoshida/rtype.html.

19. LSTEPHEN. *Completing the Spike* [online]. 2007. [visited on 2023-04-16]. Available from: http://research.nii.ac.jp/~yyoshida/rtype.html.

20. HASKELLWIKI. *Applications and libraries/Compilers and interpreters — HaskellWiki* [online]. 2020. [visited on 2023-04-16]. Available from: https://wiki.haskell.org/index.php?title=Applications_and_libraries/Compilers_and_interpreters&oldid=63265.

21. JONES, Richard; HOSKING, Antony; MOSS, Eliot. *The garbage collection handbook: the art of automatic memory management*. CRC Press, 2016.

22. COMMERCIAL HASKELL SIG. *stack: The Haskell Tool Stack* [online]. Hackage, 2022 [visited on 2023-05-07]. Available from: https://hackage.haskell.org/package/stack.

23. HENGEL, Simon. *hspec: A Testing Framework for Haskell* [online]. Hackage, 2023 [visited on 2023-05-07]. Available from: https://hackage.haskell.org/package/hspec.

24. CLAESSEN, Koen. *QuickCheck: Automatic testing of Haskell programs* [online]. Hackage, 2020 [visited on 2023-05-07]. Available from: https://hackage.haskell.org/package/QuickCheck.

25. STACK BUILDERS. *hspec-golden: Golden tests for hspec* [online]. Hackage, 2023 [visited on 2023-05-07]. Available from: https://hackage.haskell.org/package/hspec-golden.

26. MARLOW, Simon; WAERN, David. *haddock: A documentation-generation tool for Haskell libraries* [online]. Hackage, 2022 [visited on 2023-05-07]. Available from: https://hackage.haskell.org/package/haddock.

27. DORNAN, Chris; MARLOW, Simon. *alex: Alex is a tool for generating lexical analysers in Haskell* [online]. Hackage, 2023 [visited on 2023-05-07]. Available from: https://hackage.haskell.org/package/alex.

28. GILL, Andy; MARLOW, Simon. *happy: Happy is a parser generator for Haskell* [online]. Hackage, 2023 [visited on 2023-05-07]. Available from: https://hackage.haskell.org/package/happy.

29. ANDY GILL, Ross Paterson. *transformers: Concrete functor and monad transformers* [online]. Hackage, 2023 [visited on 2023-05-07]. Available from: https://hackage.haskell.org/package/transformers.

30. GILL, Andy. *mtl: Monad classes for transformers, using functional dependencies* [online]. Hackage, 2022 [visited on 2023-05-07]. Available from: https://hackage.haskell.org/package/mtl.

31. KISELYOV, Oleg; SABRY, Amr; SWORDS, Cameron; FOPPA, Ben. *extensible-effects: An Alternative to Monad Transformers* [online]. Hackage, 2019 [visited on 2023-05-07]. Available from: https://hackage.haskell.org/package/extensible-effects.

32. MARLOW, Simon et al. Haskell 2010 language report. 2010.