Insert here your thesis' task.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

SimpleObjectMachine implementation

Bc. Rudolf Rovňák

Department of Theoretical Computer Science
Supervisor: Ing. Petr Máj

November 19, 2020

Acknowledgements

THANKS (remove entirely in case you do not with to thank anyone)

Declaration

Citation of this thesis

Rovňák, Rudolf. *SimpleObjectMachine implementation.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstrakt

V několika větách shrňte obsah a přínos této práce v českém jazyce.

Klíčová slova Replace with comma-separated list of keywords in Czech.

Abstract

Summarize the contents and contribution of your work in a few sentences in English language.

Keywords Replace with comma-separated list of keywords in English.

Contents

Introduction **1**

1 State-of-the-art **3**

2 Analysis and design **5**

2.1 SOM design and features 5

2.1.1 Data types 5

2.1.2 Classes 5

2.1.3 Class hierarchy 5

2.1.4 Syntax 6

2.2 Interpretation 7

2.2.1 AST interpretation 7

2.2.2 Bytecode interpretation 7

2.3 Optimization 8

2.4 Virtual Machine 8

2.4.1 Garbage collection 8

3 Realisation **9**

Conclusion **11**

Bibliography **13**

A Acronyms **15**

B Contents of enclosed CD **17**

List of Figures

Introduction

Chapter 1

State-of-the-art

Chapter 2

Analysis and design

2.1 SOM design and features

Simple Object Machine (SOM) is a minimal Smalltalk dialect used primarily for teaching construction of virtual machines. Key characteristics according to official website ([1]) are:

- clarity of implementation over performance,
- common language features such as: objects, classes, closures, non-local returns
- interpreter optimizations, threading, garbage collectors are different across various implementations.

2.1.1 Data types

2.1.2 Classes

Classes are the cornerstone of SOM - in Smalltalk, everything is an object and each object has its class. Classes can contain fields and methods. Only single inheritance is supported - given class can only extend one other class. Object methods are dispatched dynamically in SOM and there is a keyword to explicitly call a superclass' method.

SOM implements a non-local return from methods. This gives us the ability to exit the execution of a block (or a closure) to the place where the original method calling the block returns.

2.1.3 Class hierarchy

As SOM is an object oriented language, everything is represented as an object. To enable more convenient work with classes and objects, every class is a subclass of `Object`. This enables som universal interface to be used.

Protocol of an `Object` class is:

- `class` - returns the class of an object,
- `=` - value equality comparison,
- `==` - reference equality comparison,
- `isNil` - check, if the object is `nil`,
- `asString` - converts the object into a string,
- `value` - evaluate (interesting for blocks),
- `print, println` - prints the object,
- `error:` - error reporting,
- `subClassResponsibility` - can be used to indicate the method should be implemented in the subclass of a given class,
- `doesNotUnderstand:arguments:` - can be used for error handling when a method is not implemented.

2.1.4 Syntax

Following example summarizes the syntax of SOM programming language:

```
MyClass = SuperClass (
  | field | "Instance side field"

  foo: arg = primitive "This method is implemented in the VM"

  examples = (
    | aMethodVariable |
    1234 "an integer".
    3.14 "a double".
    'a string'.
    "a comment".
    #aSymbol.
    aVariable := aVariable := 3 + 4.
    field select: [:e | e == #bar] "A message  send with a closure"
    ↑ aVariable "Return"
  )
  ----
  | classField | "Class side field"
)
```

Some notable keywords or tokens are:

- `self` is used to reference the object the method is from, comparable to C++/Java `this`,

- `super` is used to reference the superclass of a class,
- `:=` is used to assign value into a variable/field,
- `^` is used for return,
- quotes are used to delimit a comment.

2.2 Interpretation

Once the source code is parsed, the next step is executing it – this step is called *interpretation*. Interpretation is As per [2], an interpreter for a language L can be defined as a mechanism for the direct execution of all programs from L. It executes each element of the program without reference to other elements.

It is however very rare that any language is interpreted directly. In most cases of non-trivial languages, the interpretation process is preceeded by parsing or compiling into some form of *intermediate representation*. According to [2], this process removes lexical noise (comments, formating), elements can be abstracted/combined (into keywords, operations etc.) and reordered into execution order (for example operators in an algebraic expression).

The choice of intermediate representation is therefore vital. It can determine a lot of aspects of interpretation - from the way of distributing the interpreted program to time and space complexity of the interpreter.

2.2.1 AST interpretation

Abstract syntax tree (AST) is a tree representation of the source code of a computer program that conveys the structure of the source code. Each node in the tree represents a construct occuring in the source code [3].

As the name suggests, AST represents the source code in the form of a tree. During the transformation from the source code to AST, some information is ommitted. Information that is vital for AST's according to [3] is:

- variables – their types, location of their definition/declaration,
- order of commands/operations,
- components of operators and their position (for example left and right operands for a binary operator),
- identifiers and corresponding values.

2.2.2 Bytecode interpretation

Using a form of bytecode. Effective, requires:

- designing the bytecode (instructions, bytecode file formats),

- AST to bytecode translation (AST -¿ bytecode instructions),
- actual bytecode interpretation.

Bytecode interpretation permits easier optimization.

2.3 Optimization

- dead code elimination,
- constant propagation,
- others. . .

2.4 Virtual Machine

Decide on memory hierarchy, garbage collection. . .

2.4.1 Garbage collection

The process of *garbage collection* performed by *garbage collector (GC)* is the process of allocating and freeing memory during application runtime. The main advantage of this mechanics is to prevent *memory leaks* – parts of a program that allocate memory without freeing it when it is not needed [4]. Most modern high-level programming languages implement some form of garbage collection.

Chapter 3

Realisation

Conclusion

Bibliography

[1] SOM. SOM: A minimal Smalltalk for teaching and research on Virtual Machines. 2020, [cit. 2020-11-17]. Available from: `https://som-st.github.io/`

[2] Wolczko, M. Execution mechanisms Part I: Interpretation. [online], 2015, [cit. 2020-10-31]. Available from: `https://www.dropbox.com/s/lfav564dvx20qsw/2%20AST%20Interpretation.pdf`

[3] DeepSource Corp. Abstract Syntax Tree. [cit. 2020-11-4]. Available from: `https://deepsource.io/glossary/ast/`

[4] Boersma, E. Memory leak detection - How to find, eliminate, and avoid. January 2020, [cit. 2020-11-5]. Available from: `https://raygun.com/blog/memory-leak-detection/`

Appendix A

Acronyms

AST	Abstract syntax tree
GC	Garbage collector
SOM	Simple Object Machine
VM	Virtual machine

Appendix B

Contents of enclosed CD

`readme.txt`	the file with CD contents description
`exe`	the directory with executables
`src`	the directory of source codes
`wbdcm`	implementation sources
`thesis`	the directory of LaTeX source codes of the thesis
`text`	the thesis text directory
`thesis.pdf`	the thesis text in PDF format
`thesis.ps`	the thesis text in PS format