Virtual Machines

Yann-Gaël Guéhéneuc
École des Mines de Nantes
4, rue Alfred Kastler – BP 20722
44 307 Nantes Cedex 3
France
yann-gael@gueheneuc.net

Started: October 1st, 2001
Revision: October 16, 2002*

Contents

*The latest version is available at: http://www.yann-gael.gueheneuc.net/Work/Teaching/.

1 Virtual machines

Reference: [15], [17], and [18]

1.1 What?

Computing machine The fundamental of computer science is to provide a universal computational framework to compute algorithms. An algorithm defines a set of data and the sequences of instructions to manipulate that data. The computation of the algorithm consists in following the instructions to transform the data and to obtain a result. The machine performing the computation of an algorithm is a computing machine. There exists four main types of computing machines, depending on their abstractness. In the rest of this course, we follow the example of the Greatest Common Denominator algorithm:

> **Example 1 (Greatest Common Denominator algorithm):**
> Euclid proposed an algorithm to compute the greatest denominator common to two integers. Euclid's algorithm may be iterative:
>
> ```
> Algorithm GCD(x, y)
>     value x, y : integer >= 0;
>     return an integer >= 0;
> begin
>     repeat
>         if(x >= y)
>         then
>             x = x - y;
>         else
>             x = y - x;
>         end if;
>     until(x == 0);
>     return(y);
> end.
> ```
>
> Or Euclid's algorithm may be recursive:
>
> ```
> Algorithm GCD(x, y)
>     value x, y : integer >= 0;
>     return an integer >= 0;
> begin
>     if(y == 0)
>     then
>         return(x);
>     else
>         return(GCD(y, remainder(x, y));
>     end if;
> end.
> ```

Turing machine (1936) The most theoretical computing machine is the Turing machine. Conceptually, a Turing machine has a finite set of states, a finite alphabet (with a blank symbol), and a finite set of instructions. Physically,

it has a head that can read, write, and move on an infinitely long tape divided into cells; each cell has a value of blank or a letter in the Turing machine's alphabet. An instruction is defined as a five-tuple:

{starting state, starting value, new value, movement, new state}

The starting state is the state where the head currently is. The starting value is the value of the cell on which the head is positioned. The new state and the new value replace the starting state and the starting value, respectively. The movement specifies which direction the head moves by one cell. The head halts when it can not find an instruction for the current state or the current cell value. A Turing machine will start at the first non-blank cell. Usually, states are named s_0, s_1, s_2, etc. The alphabet A contains usually: B, for blank; S, for the starting cell; and, *0* and *1*. A number can be represented (but not necessarily) by a series of *1*s with a length of $n + 1$ for a number n. Practically, programming a Turing machine is programming in assembly language, with a very low level of abstraction: It is hard to imagine the equivalent of a procedure call. In a Turing machine, there is no place to store data other than the tape. Therefore, it is hard to process more than one symbol at a time and the number of states grows to huge proportions (as place-holders).

Example 2 (A Turing machine and an infinitely incrementing binary counter):
The following Turing machine demonstrates an infinitely incrementing binary counter. The Turing machine starts is state s_0, which returns to the beginning of the tape and then goes to state s_1. While in state s_1, the machine moves to the right and changes any *1* into a *0*; when the machine encounters a *0* or a B, it changes it into *1* and goes to state s_0. The following table summarizes the two states:

If in state	and on symbol	then write	and move	and go to
s_0	S	—	right	s_1
s_0	A-{S} (B, *0*, or *1*)	—	left	s_0
s_1	*1*	*0*	right	s_1
s_1	*0*	*1*	left	s_0

Exercise 1 (A Turing machine infinitely incrementing a binary counter):
Starting from the graphical representation of a Turing machine in Figure 1, write sequences following the rules defined in Exemple 2.

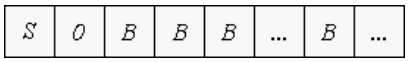

Figure 1: *The starting tape for a binary counter incrementing infinitely.*

Exercise 2 (Turing and the Greatest Common Denominator algorithm):
Implement Euclid's Greatest Common Denominator algorithm for a Turing machine. (Consider representing numbers using integers.)

Exercise 3 (Turing and completeness):
Look for information about the theory of Turing-completeness. In particular, search for information on computability theory and on the halting problem. A good start is `http://www.wikipedia.org/wiki/Turing-complete`.

λ- and π-calculus (1934) The λ-calculus is also a universal computational framework. It has been proposed as a mean to study functions and their applications. More recently, the π-calculus has been introduced to formalize parallel computations. The π-calculus encompasses the λ-calculus. λ- and π-calculus are less low-level than a Turing machine and gave birth to several implementations, the most famous being the family of functional programming languages (such as Lisp, Scheme, or Haskell). The main problems in using λ- and π-calculus is their lack of intuitiveness (perhaps).

Example 3 (Greater Common Denominator algorithm in λ-calculus):
Using λ-calculus and assuming the existence of `if`, `=`, `remainder`, and with `Y` as fixed-point combinator, the Greatest Common Denominator algorithm may be expressed as:

```
Y(λgcdλaλb.if(= b 0) then a else gcd b (remainder a b))
```

Example 4 (Greater Common Denominator algorithm in a functional language):
In a functional language (i.e., Scheme), the Greatest Common Denominator algorithm is:

```
(define (gcd a b)
    (if (= b 0)
        a
        (gcd b (remainder a b))))
```

Register machine A register machine consists in a set of instructions executed serially that operate on registers containing the data. The machine has a set of basic instructions that, theoretically, can be very few (consisting only of conditional jump, comparison, and assignment). Programming is quite easy, because we normally formulate algorithms using registers (i.e., the variables in Example 1). However, the register machine possesses many shortcomings, the main ones being the serialization of instructions and the use of imperative programming. Also, it is difficult to express intuitively recursion using a register machine.

Example 5 (Greater Common Denominator algorithm for a register machine):
The schemas in Figure 2 represent the data and the controller for the Greatest Common Denominator algorithm. The data schema shows which registers are used (`a`, `b`, and `c`), how the values are transferred from one register to another (`a<-b`...), and the instructions producing values (`remainder`). The controller schema shows the order of the instructions (the test, the transfer of the result of `remainder` into register `t`, the transfer of the value of register `b` into register `a`, and the transfer of the value of register `t` into register `b`).

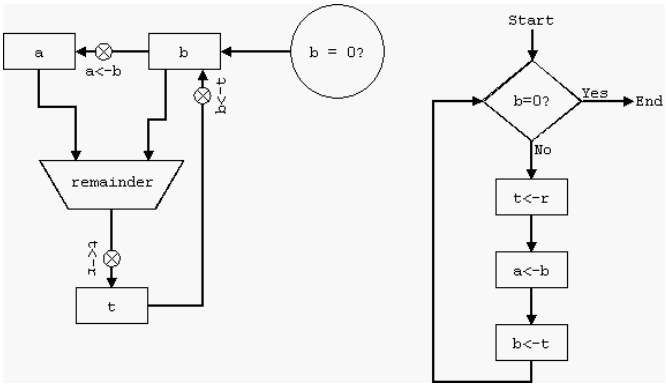

Figure 2: *The data flow and control flow of a register machine for the Greatest Common Denominator algorithm.*

Exercise 4 (Register machine and recursion):
Give a definition of recursion, tail-recusrion, and search on the duality between recursive and iterative algorithms.

Stack machine A stack machine is a computing machine that uses a stack for its internal storage. The stack behaves has a LIFO list (the Last value In is the First value Out). It contains the results of the executed instructions. Theoretically, it is of infinite size. A stack machine possesses two basic instructions to manipulate its stack: push (to put a value on top of the stack) or pop (to remove the top value from the stack). Other instructions may indifferently push and pop values.

Example 6 (Greater Common Denominator algorithm for a stack machine):
The schema in Figure 3 represents some of the steps in the calculation of the Greatest Common Denominator algorithm, using a stack machine. The schema follows the iterative Greatest Common Denominator algorithm defined in Example 1.

Virtual machine A virtual machine is a piece of software that goes between a program and the environment into which the program runs (operating system, hardware...), as described in Figure 4. It is both an interpreter and an

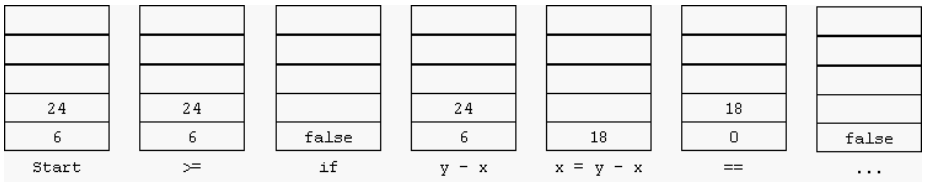

Figure 3: *Steps in the calculation of the Greatest Common Denominator algorithm with a stack machine.*

just-in-time compiler. It interprets an intermediate language obtained from the compilation of a higher-level programming language. It compiles the intermediate language just-in-time to call directly operating-system- or hardware-level instructions.

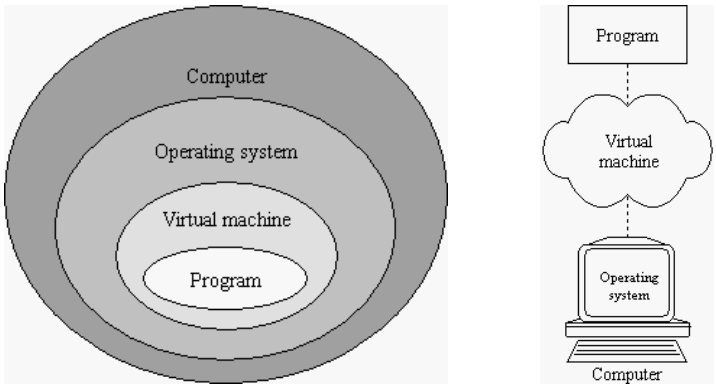

Figure 4: *The imbrication of the program into a virtual machine that runs within an operating system that runs on a computer.*

1.2 Pros?

Compiler writers' point of view The compilation of a programming language for a virtual machine reduces the amount of work needed by the compiler writers. Let consider that we have n programming languages and m operating-systems or pieces of hardware for which we want to compile the programming language, then we need $n * m$ compilers. Now, let consider that we have n programming languages and *1* virtual machine implemented on m operating-systems or pieces of hardware, then we only need $n * 1$ compilers.

Virtual machine writers' point of view The intermediate language is simpler in terms of its instruction set and idioms (for example, the intermediate language may represent all the loops with only two instructions: `if` and `goto`), and it is richer in terms of computational information (for exemple, the

intermediate language may know the types of all the references). Thus, the implementation of a virtual machine requires no type-checking or control-flow computation. It only necessitates the traduction of the intermediate language instructions into operating-system- or hardware-dependent calls.

Maintainer's point of view A virtual machine may provide special-purpose interfaces to ease the work of profilers and maintainers. A virtual machine may offer an interface to debug running programs (such an interface may contain a method to stop a thread, to inspect a stack...), or it may offer an interface to profile running programs (such an interface may contain methods to get the execution of a method or the number of time a method is executed...). Special-purpose interfaces also help the profilers and maintainers by integrating the different aspects of software development in a consistent framework.

Users' point of view A virtual machine may be available on several platforms, this availability allows the user to write and to compile programs only once, and to run them with similar results across several platforms (in terms of execution semantics and execution time). In addition, a virtual machine may propose sophisticated mechanisms: garbage collection, profiling, native call interface, communication interface, debugging... These mechanisms are accessible to any user who possesses a compiler to the appropriate intermediate language, to the appropriate virtual machine.

> **Exercise 5 (Pros of virtual machines):**
> List other arguments or evidences on virtual machine usefulness.

1.3 Cons?

Language designers' point of view When developing a new programming language or when implementing a new compiler, language designers must consider which intermediate language they wish to target. They must consider the trade-offs between the intermediate language, its complexity, its expressiveness, and the virtual machine, its availability, its services. The best virtual machine (in terms of supported platforms and provided mechanisms) may understand an intermediate language that does not fit the programming language (in terms of paradigm, types...). For example, a functional language may be difficult to compile into an imperative intermediate language. The complexity depends on the programming languages and on the virtual machines available.

Virtual machine writers' point of view The implementation of a virtual machine may be difficult or impossible depending on the intermediate language, the proposed mechanisms, and the platform. For example, the intermediate language may offer multi-threading, while the platform only supports single-process programs.

Execution time The virtual machine stands between the program and the operating-system or piece of hardware on which the program runs. It plays the role of mediator between the program and the hardware and, thus, it slows down their communication and the execution time of the program.

> **Exercise 6 (Cons of virtual machines):**
> List other arguments or evidences on virtual machine drawbacks.

2 Java

History The Java Virtual Machine (JVM) is a virtual machine dedicated to the Java programming language. It has been first introduced in 1996. The context was originally house appliances: Sun wanted to offer a general framework for communicating and programming house appliances. Their purpose was to develop a common platform for house appliances (language, run-time). (Another thing about Java is that the Oak team, original code name of Java, looked at already-existing platform. They tried Visual Works, the problem was that Parc Place asked for too much money...)

Then, the context evolved as the World Wide Web (WWW) grew bigger; and, with the WWW, its main limitation: A lack of dynamic interaction with the users. Sun's HotJava web browser was the first to have a truly dynamic potential: It made it possible to embed Java programs in WWW page.

Raison d'être The Java programs embedded in a WWW page run inside a JVM. This offers three great advantages:

- The developers of the web browser may be different from the developers of the JVM. For instance, Sun provides JVM implementations for all the major operating systems and web browsers: This leverages the burden off the web browser developers' shoulders.
- The Java program running in the JVM is isolated from the operating system. Any communication between the Java program and the the web browser or the operating system is filtered by the JVM.
- Www pages have sophisticated interaction and calculation capabilities.

The Java programming language and the JVM Before we skim through some features of the JVM, we take a quick look at the Java programming language and the JVM. The Java programming language is an object-oriented programming language. Its syntax is close to the syntax of C and C++. It offers introspection mechanisms (the `java.lang.reflect` package) and multi-threading (the `java.lang.Thread`$\star$ classes and the `java.lang.Runnable` interface). It is possible to interact with the operating system, in a limited way, through the `java.lang.System` class, and the `java.lang.Process` and `java.lang.Runtime` classes. The Java programming language also includes a mechanism of exception.

A typical session with the JVM The JVM executes Java programs. First, the user gives the JVM the name of the main class of the program:

```
java -classpath . emn.course.vm.GCD
```

The JVM starts, it initializes its internals. The JVM creates two threads (`Signal Dispatcher` and `CompileThread0`, in Sun's JVM). The JVM creates

the default instance of class `ClassLoader`. This default class loader loads the class given in the command line. The JVM executes the `main()` method of the loaded class and creates the required threads (for example, `AWT-EventQueue-0`). The program executes. When needed, the JVM calls the garbage collector. Finally, the Java program ends, the JVM dies, and with it all the running threads.

> **Exercise 7 (JVM use):**
> Detail further a typical session with the JVM.

2.1 Java Virtual Machine

Reference: [13]

The JVM is a hybrid of a register machine and of a stack machine. The JVM supports multi-threading. The JVM is composed of:

- Among all threads:
  - A heap. All threads share the heap. The heap is the runtime data area from which the JVM allocate memory for all instances and arrays.
  - A method area. The method area stores per-class structures such as the constant pool, field, method data, and the code for methods and constructors.
  - Native method stacks. The native method stacks are the stacks allocated by the JVM for the native methods, methods written in languages other than Java.
- Per thread:
  - A register. The program counter register contains the address of the JVM instruction currently being executed in the thread.
  - A stack. The stack stores the frames created for each method invocation. It holds local variables and partial results, and plays a part in method invocation and return. The stack may be allocated from the heap.
- Per class or per interface:
  - A constant-pool. The constant-pool contains the constants related to the class or the interface, such as numeric literals or method and field references.
- Per method invocation:

- A frame. The JVM creates a new frame with each new method invocation. It destroys the newly created frame when the method completes. It allocates the frames from the stack of the current thread. Each frame stores local variables, operands (including invocation arguments), and contains a reference to the constant-pool for the type of the current method. The reference to the constant-pool for the type of the current method allows dynamic linking of the method code and on-the-fly class loading.

> **Exercise 8 (Dynamic linking):**
> List the advantages and drawbacks of dynamic linking.

> **Exercise 9 (Exception):**
> Given the organization of the JVM at run-time, explain what happens when an exception is thrown from within a method body.

The JVM reveal an underlying object model. This object model is almost isomorphic to the Java programming language.

The JVM also includes a just-in-time compiler: The Java HotSpot Virtual Machine (JHVM). The HotSpot technology was developed by Animorphics. Animorphics initially developed HotSpot for Smalltalk. They presented it at OOPSLA 1996 (?). Anita Iwing and Allen Wirfs-Brock, from Parc Place-Digitalk, were interested in buying the HotSpot technology. Finally, Sun bought HotSpot and Animorphics adapted it to Java later, with a lot of effort.

The JHVM improves the performances at run-time. The JHVM offers a more efficient memory model, a more efficient garbage collector, an improved thread synchronization mechanism, and just-in-time compilers. The JHVM analyzes the code as it runs to detect critical hot-spots in the program, then it compiles the intermediate language into native code. Thus, the JHVM increases the performance and (in some cases) decreases the memory foot-print of the program.

2.2 Java intermediate language

Reference: [13]

The intermediate language understood by the JVM is called Java byte-code. A Java compiler produces Java byte-code and stores it in a binary file. The binary file, containing the Java byte-code, has a platform-independent format: The class-file format. The class-file format precisely defines the organizational details of the binary file, for example the byte ordering or the set of valid instructions.

The set of valid instructions understood by the JVM is composed of one-byte opocodes. An op-code specifies the operation the JVM must perform, followed by zero or more operands. An operand is an argument or a piece of data used to perform the operation.

Exercise 10 (One-byte op-codes):
List the advantages and drawbacks of the limitation of the op-code to one byte.

Most of the op-code encode type information about the operations they perform. They contain explicit references to the types on which they operate, for exemple the `iload` op-code performs its operation on `int` and, thus, begins with an `i`. All the mnemonic letters are: `i` for `integer`, `l` for `long`, `s` for `short`, `b` for `byte`, `c` for `char`, `f` for `float`, `d` for `double`, and `a` for `reference` operations. Some op-codes do not begin with a mnemonic letter because they operate only on a certain type (for example, `arraylength`) or because they do not operate on typed operands (for example, `goto`).

Exercise 11 (Typed op-codes):
List the advantages and drawbacks of typed op-codes.

Example 7 (Greatest Common Denominator algorithm in Java and in Java byte-codes):
The following Java source code and Java byte-code implement the iterative Greatest Common Denominator algorithm as described in Example 1.

```
public static int computeGCD(int a, int b) {
    if (b == 0) {
        return a;
    }
    else {
        return computeGCD(b, a % b);
    }
}

Method int computeGCD(int, int)
   0 iload_1
   1 ifne 6
   4 iload_0
   5 ireturn
   6 iload_1
   7 iload_0
   8 iload_1
   9 irem
  10 invokestatic #25 <Method int computeGCD(int, int)>
  13 ireturn
```

Exercise 12 (Understanding op-codes):
Manually execute the Java byte-code implementing the Greatest Common Denominator algorithm of Example 7.

2.3 Sandbox

Reference: [10]

A sandbox is usually a place where children play safely, protected from and not disturbing the outside world. The JVM is such a sandbox, from the point of view of the Java program–operating system relationships.

The Java program is isolated from the operating system. It runs in a similar fashion regardless of the operating system. The Java developers do not need to consider operating-system dependent characteristics when developing their programs.

The JVM protects the operating system from any disturbance the Java program might cause. The JVM defines and enforces the security rules that Java programs must comply to. If a Java program does not comply with the JVM rules, the JVM notifies the program using the exception mechanism.

However, the notion of sandbox also prevents sophisticated communication between the Java program and the operating system. Thus, the JVM provides three mechanisms to go around the sandbox principle: Dynamic class loading; Native interface; and, a Security model.

2.4 Dynamic Class loading

Reference: [2] and [6]

The JVM calls the dynamic class loading mechanism when it requires a reference to a class that does not yet exist in the JVM. The class loading mechanism of the JVM has the following characteristics:

- Lazy loading: The JVM loads the classes when it needs them.
- Type-safe linking: The class loading mechanism is safe thanks to the class-verification mechanism.
- User-defined class loading policies: Users may define their own class loading mechanisms.
- Multiple name-spaces: The class loading mechanisms enables multiple name-spaces.

The Java `ClassLoader` class constitutes the core of the class loading mechanism. A class loader is an instance of the `ClassLoader` class and is responsible for loading classes, through the `loadClass()` method: Given the name of a class, the class loader attempts to locate (through the `findClass()` method) or to generate data that constitutes a definition of the class. Once the class loader has found the class-file corresponding to the required Java class, it provides the definition of the class to the JVM, through the `defineClass()` method. (At that point, the JVM has checked the correctness of the Java class-file, cf. Section 2.7.)

A user-defined class loader always has a parent class loader. The ultimate class loader is the default class loader provided by the JVM. A user-defined class loader may provide a complete implementation of the class `ClassLoader` or delegate calls to its parent. The class loader initiating the loading of a class may be different from the class loader defining the class. This is an important feature when considering name-spaces.

Example 8 (A simple class loader):
Let consider a user-defined class loader `UserCL`. To load the class `GCD`, implementing the Greatest Common Denominator algorithm in Java, the user uses the following Java source code:

```
ClassLoader userCL = new UserCL();
Class gcdClass = userCL.loadClass("emn.course.vm.GCD");
```

The call to `loadClass()` locates the class-file using `findClass()`, and performs a `defineClass()`. The class loader `UserCL` is responsible of the loading and of the definition of the class `GCD`, we write $\langle GCD, UserCL\rangle^{UserCL}$, or $\langle GCD, UserCL\rangle$.

Example 9 (Class loaders and name-space compatibility):
Let consider two class loaders: The default class loader provided by the JVM, and `UserCL`, a user-defined class loader. The `UserCL` class loader delegates the `defineClass()` call on its parent class loader. Let suppose the user requests the class `GCD` twice: Once using `UserCL` class loader, then using the default class loader, then:

```
ClassLoader userCL = new UserCL();
Class gcdClass1 = userCL.loadClass("emn.course.vm.GCD");
ClassLoader jvmCL = ClassLoader.getSystemClassLoader();
Class gcdClass2 = jvmCL.loadClass("emn.course.vm.GCD");
```

The call to `userCL.loadClass()` locates the class-file using `findClass()`, and calls the `defineClass()` method on the default class loader, `jvmCL`. The `UserCL` class loader is responsible of the loading of the class `GCD`, the `JvmCL` class loader is responsible of the definition of the class `GCD`, we write $\langle GCD, JvmCL\rangle^{UserCL}$. The class obtained from the `UserCL` class loader is the same as the class obtained from the default class loader: $\langle GCD, JvmCL\rangle^{UserCL}$ is compatible with $\langle GCD, JvmCL\rangle^{JvmCL}$

Example 10 (Class loaders and name-space non-compatibility):
Let consider two class loaders: The default class loader provided by the JVM, and `UserCL`, a user-defined class loader. The `UserCL` class loader calls the `defineClass()` method. Let suppose the user requests the class `GCD` twice: Once using `UserCL` class loader, then using the default class loader, then:

```
ClassLoader userCL = new UserCL();
Class gcdClass1 = userCL.loadClass("emn.course.vm.GCD");
ClassLoader jvmCL = ClassLoader.getSystemClassLoader();
Class gcdClass2 = jvmCL.loadClass("emn.course.vm.GCD");
```

The call to `userCL.loadClass()` locates the class-file using `findClass()`, and then calls the `defineClass()` on the user-defined class loader. The `UserCL` class loader is responsible of the loading of the `GCD` class and of the definition of the `GCD` class, we write $\langle GCD, UserCL\rangle^{UserCL}$. The class obtained from the `UserCL` class loader is different from the class obtained from the default class loader: $\langle GCD, UserCL\rangle^{UserCL}$ is not compatible with $\langle GCD, JvmCL\rangle^{JvmCL}$

The difficulty when using class loader is to keep the name-space compatibility. When defining class-loaders, the user must avoid using the pseudo-field `class` (for example, `Class gcdClass = GCD.class`) and use instead the method `Class.forName()`
(for example,`Class gcdClass = Class.forName("emn.course.vm.GCD")`).
The user must use the user-defined class loader, through `Class gcdClass = userCL.loadClass("emn.course.vm.GCD")`.

Exercise 13 (Class loader and reference constant):
Let suppose the user adds a static constant of type `Version` (subclass of `Object`) to the `GCD` class, implementing the Greatest Common Denominator algorithm:

```
public final class GCD {
    public static Version version = new Version("YGG", 1.0);
    ...
}
```

From the examples 8, 9, and 10, describe what happens when the user does:

```
versionGCDClass1.getClass() == versionGCDClass2.version.getClass()
versionGCDClass1 == versionGCDClass2
```

After the user has requested the `GCD` class with the two different class loaders. (Where `versionGCDClass1` and `versionGCDClass2` are references on the static field of class `gcdClass1` and `gcdClass2`, respectively.)

2.5 Garbage collection

Reference: [7]

Garbage collection is the process of automatically freeing objects that are no longer referenced by the program. This frees the programmer from having to

keep track of when to free allocated memory, thereby preventing many potential bugs and headaches.

In addition to freeing unreferenced objects, a garbage collector may also reduce heap fragmentation. Heap fragmentation occurs through the course of normal program execution. New objects are allocated, and unreferenced objects are freed such that free blocks of heap memory are left between blocks occupied by live objects.

There are several existing garbage collection algorithms. The simplest garbage collection algorithm is the Reference Counting Garbage Collector (RFGC). The RFGC operates by counting the number of references pointing on an instance of a class. If an instance of a class possesses zero reference pointing to it, the RFGC discards it, because the remaining instances can not access it. The RFGC is simple to understand. However, The RFGC is difficult to implement.

Then, two of the most famous are mark-and-sweep, and copying collector. A mark-and-sweep garbage collector traverses all reachable objects in the heap by following pointers beginning with the *roots*, i.e., pointers stored in statically allocated or stack allocated program variables. All such reachable objects are marked. A sweep over the entire heap is then performed to store unmarked objects into a free list, so that they can be reallocated.

In contrast, a copying collector copies reachable objects to another region of memory as they are being traversed. Provided the traversal is done in breadth-first order, there is a well-known and simple algorithm for performing this traversal without auxiliary storage or recursion. After such a traversal, all surviving objects reside in a contiguous region of memory, and all pointers have been updated to point to the new object locations. The previously used region of memory can then be reused in its entirety. Allocation becomes trivial, since all free space is always contiguous.

Garbage collection is a highly technical topic and is subject to many research and industrial work. The author has no information on what kind of garbage collector the JVM uses.

2.6 Java Native Interface

Reference: [8]

The Java Native Interface (JNI) is a standard programming interface for calling native methods from Java and embedding the JVM into native applications. The primary goal is binary compatibility of native method libraries across all Java virtual machine implementations on a given platform. A secondary goal is to improve the efficiency of Java programs.

> **Exercise 14 (The Java Native Interface):**
> List other reasons for the existence of the JNI.

Primitive types, such as integers, characters... are copied between Java and the native code. Arbitrary Java objects are passed by reference. The JVM keeps track of all objects that have been passed to the native code, so that these objects are not freed by the garbage collector. The native code has a way to inform the JVM that it no longer needs the objects. In addition, the garbage collector must be able to move an object referred to by the native code.

> **Example 11 (Greatest Common Denominator algorithm in C):**
> The Java source code in Figure 5 displays the result of a Greatest Common Denominator algorithm implemented in C. The declaration of the native method (line 4) creates a link between Java and the native code. The call to `System.loadLibrary()` loads the dynamic linked library (DLL), where is the native implementation of the native method (lines 5–7). The native method is used as any other Java method (line 10).
>
> The DLL containing the native code is shown in Figure 7. It contains a reference to the C header in Figure 6. The C header is generated from the Java source code with the `javah` tool: `javah emn.course.vm.GCD_C`. The DLL is obtained by compiling the native source code:
>
> ```
> cl /I%JAVA_HOME%\include /I%JAVA_HOME%\include\win32 /GD /LD GCD_C_Impl.c
> ```

```
package emn.course.vm;

public final class GCD_C {
    public native int computeGCD(int a, int b);
    static {
        System.loadLibrary("GCD_C_Impl");
    }
    public static void main(final String[] args) {
        final GCD_C myJNICallToGCD = new GCD_C();
        System.out.println(myJNICallToGCD.computeGCD(6, 18));
    }
}
```

Figure 5: *The Java class calling the native implementation of the Greatest Common Denominator algorithm (file `emn/course/vm/GCD_C.java`)*

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class emn_course_vm_GCD_0005fC */

#ifndef _Included_emn_course_vm_GCD_0005fC
#define _Included_emn_course_vm_GCD_0005fC
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     emn_course_vm_GCD_0005fC
 * Method:    computeGCD
 * Signature: (II)I
 */
JNIEXPORT jint JNICALL Java_emn_course_vm_GCD_1C_computeGCD
  (JNIEnv *, jobject, jint, jint);

#ifdef __cplusplus
}
#endif
#endif
```

Figure 6: *The C header generated from the Java class in Figure 5 (file `emn_course_vm_GCD_0005fC.h`)*

```
#include "emn_course_vm_GCD_0005fC.h"
#include <stdio.h>

jint JNICALL Java_emn_course_vm_GCD_1C_computeGCD(
                JNIEnv *env,
                jobject obj,
                jint a,
                jint b) {

        if (b == 0) {                                                    10
            return a;
        }
        else {
            return Java_emn_course_vm_GCD_1C_computeGCD(env, obj, b, a % b);
        }
}
```

Figure 7: *The C implementation of the Greatest Common Denominator algorithm (file `GCD_C_Impl.c`)*

2.7 Security

Reference: [10]

The Java class-libraries enforce the security policies. The JVM has no concern for the security policies. The JVM only checks if the Java class-files are well-formed and secure with respect to stack management.

The JVM possesses a byte-code verifier to check if the Java class-files are well-formed and secure. The JVM calls the byte-code verifier after the current instance of `ClassLoader` has performed `loadClass()` and while it is performing `defineClass()`. The byte-code verifier ensures that the Java byte-code does not contain erroneous op-codes and does not attempt unsafe operations, such as branch out of bounds, access or modification of a local variable at an index greater than the number of local variables in the method...

> **Exercise 15 (Java class-libraries and byte-code security policies):**
> List the differences between security policies at the class-libraries level and at the JVM level, and the reasons of these differences.

2.8 Java Platform Debug Architecture

Reference: [4] and [9]

Sun develops a complete architecture to ease the debugging of Java programs: The Java Platform Debug Architecture (JPDA).

Debugging a running program consists in collecting information about its state at one point in time. Before the implementation of the JPDA, developers had two solutions to debug Java programs:

- To include a trace in the execution flow of their Java programs. However, including a trace in the program requires either to know where the unfit behavior happens or to trace the whole program and thus to produce overwhelming and uninteresting information.
- To modify or implement a JVM with tracing capabilities. However, modifying a JVM requires a deep understanding of the JVM inner parts; it may break the JVM sandbox mechanism; and, it necessitates to repeat the modifications for each new version of the JVM. Modifying the JVM is even more difficult when debugging remote JVM.

Those are the reasons why Sun develops the JPDA as part of its JVM.

> **Exercise 16 (The Java Platform Debug Protocol):**
> List other reasons why Sun would develop the JPDA.

The JPDA is composed of three different levels, see in Figure 8:

- The Java Virtual Machine Debug Interface (JVMDI): A low-level native interface. JVMDI defines the services a JVM must provide for debugging.
- The Java Debug Wire Protocol (JDWP): The format of information and requests transferred between the debugging process and the debugger front-end.
- The Java Debug Interface (JDI): A high-level Java programming language interface, including support for remote debugging.

As of version 1.4.1, the JVM now uses *full-speed debugging*. In the previous version of the JVM, when debugging was enabled, the program executed using only the interpreter. Now, full performance is available to programs running with debugging enabled. The improved performance allows long running programs to be more easily debugged. It also allows testing to proceed at full speed and the launch of a debugger to occur on an exception.

The JVM also provides an API to substitute modified code for a program in a running JVM. This feature, called HotSwap, allows developers and maintainers to recompile a single class and to replace the old version of the class with the new version.

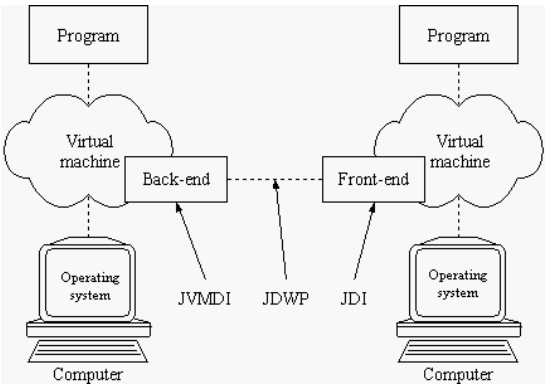

Figure 8: *The three different levels of specification as defined by the Java Platform Debug Architecture.*

2.8.1 Java Virtual Machine Debug Interface

The JVMDI is a native interface implemented by the JVM. It defines the services a JVM must provide for debugging. It includes requests for information (for example, the current stack frame), actions (for example, set a breakpoint), and notification (for example, when a breakpoint has been hit). A debugger may use other JVM information (for example, through the Java Native Interface), but this debugger specific information reduces portability.

The tables in Figure 9 summarize the different functions provided by the JVMDI.

Exercise 17 (JVMDI functions):
Read through all the JVMDI functions and find out their semantics.

Exercise 18 (On the usage of JVMDI functions):
Given the functions provided by the JVMDI, think of original information that could be extracted using the JVMDI. For example, how would you monitor the life-cycle of an instance (creation and garbage collection), using the JVMDI?

Exercise 19 (On other languages):
The JPDA has been extended so that non-Java programming language source, which is translated to Java programming language source, can be debugged in the future. Find out the methods specifically defined to handle debugging foreign source code.

2.8.2 Java Debug Wire Protocol

The JDWP defines the format of information and requests transferred between the debugged process and the debugger front-end. It does not define the transport mechanism (socket, serial line, shared memory...). The specification of the protocol allows the debugged process and the debugger front-end to run under separate JVM and–or on separate platforms. It also allows the front-

Category	Function
Memory Management	Set Allocation Hooks Allocate Memory Deallocate Memory
Thread Execution	Get Thread Status Get All Threads Suspend Thread Resume Thread Stop Thread Interrupt Thread Get Thread Info Get Owned Monitor Info Get Current Contended Monitor Run Debug Thread
Thread Groups	Get Top Thread Groups Get Thread Group Info Get Thread Group Children
Stack Frame Access	Get Thread's Frame Count Get Thread's Current Frame Get Caller Frame Frame Location Notify Frame Pop
Local Variable Access	Get Local Variable Set Local Variable
Breakpoints	Set a Breakpoint Clear a Breakpoint Clear All Breakpoints
Watched Fields	Set a Field Access Watch Clear a Field Access Watch Set a Field Modification Watch Clear a Field Modification Watch
Raw Monitor Support	Create Raw Monitor Destroy Raw Monitor Raw Monitor Enter Raw Monitor Exit Raw Monitor Wait Raw Monitor Notify Raw Monitor Notify All

Category	Function
Class Information	Class Signature Class Status Source File Name Class Access Flags Class Methods Class Fields Implemented Interfaces Is an Interface Is an Array Class Loader
Object Information	Object Hash Code Get Monitor Info
Field Information	Field Name and Signature Field Declaring Class Field Access Flags Is Field Synthetic
Method Information	Method Name and Signature Method Declaring Class Method Access Flags Maximum Stack Local Slots Argument Slots Source Line Numbers Method Location Local Variables Exception Handlers Thrown Exceptions Get Byte-codes Is Method Native Is Method Synthetic
Events	Set Event Hook Enable/Disable Events
Miscellaneous	Get Loaded Classes Get Class-loader Classes Get Version Number Get Capabilities

Figure 9: *The functions provided by the JVMDI.*

end to be written in a language other than Java. Information and requests are roughly at the level of the JVMDI, and include additional information and requests necessitated by bandwidth issues, for example filters.

2.8.3 Java Debug Interface

The JDI is a 100%-Java interface implemented by the front-end. It defines information and requests at a user level. While debugger implementors could directly use the JDWP or the JVMDI, this interface greatly facilitates the integration of debugging capabilities into development environments.

> **Example 12 (A JPDA-based tool to control an implementation of the Greatest Common Denominator algorithm, both in Java):**
> The source codes in Figures 11 and 12 represent a simple Java-implementation of the Greatest Common Denominator algorithm, and a simple JPDA-based tool to control the Java-implementation of the Greatest Denominator algorithm, respectively.
>
> The Java-implementation of the Greatest Common Denominator algorithm creates an instance of `JFrame` (a window) that embeds an instance of `JTextField` (a text field) (lines 5–16). It calls the Greatest Common Denominator algorithm, as presented in Example 7, and displays the result into the instance of `JTextField` (lines 18–32).
>
> The JPDA-based tool does:
>
> - Get an instance of the virtual machine manager through the `Bootstrap` class (line 6: JDI).
> - Get the default instance of the class `LaunchingConnector`. The instance of class `LaunchingConnector` allows the communication between the current virtual machine and the remote virtual machine (line 7: JDWP).
> - Compute the arguments for the remote virtual machine to run the desired class (lines 8–15).
> - Start the remote virtual machine with the previously-built arguments, using the communication channel (instance of class `LaunchingConnector`). All the threads are by default suspended in the remote virtual machine (in Sun's current implementation), they must be resumed (lines 16–17: JDI + JDWP + JVMDI)).
> - Build an instance of `JFrame` (a window) with two instances of `JButton` (two buttons), to suspend/to resume and to stop the remote virtual machine (lines 18–48: JDI + JDWP + JVMDI).
>
> Figure 10 illustrates the JPDA-based control tool and the Greatest Common Denominator algorithm.

Figure 10: *The control tool and the Greatest Common Denominator algorithm, in Java, and in action.*

```
// Package and import declarations not shown for conciseness.

public final class GCD {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Euclid's GCD algorithm");
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        frame.setLocation(400, 100);
        frame.setSize(300, 100);
        JTextField text = new JTextField("Results");
        frame.getContentPane().add(text);
        frame.setVisible(true);

        int a = 6;
        int b = 18;
        StringBuffer buffer = new StringBuffer();
        while (true) {
            buffer.setLength(0);
            buffer.append(text.getText());
            buffer.append("\n\rGCD(");
            buffer.append(a);
            buffer.append(", ");
            buffer.append(b);
            buffer.append(") = ");
            buffer.append(computeGCD(a, b));
            text.setText(buffer.toString());
            a = a + 1;
            b = b + 3;
        }
    }

    public static int computeGCD(int a, int b) {
        if (b == 0) {
            return a;
        }
        else {
            return computeGCD(b, a % b);
        }
    }
}
```

Figure 11: *Euclid's Greatest Common Denominator algorithm in Java.*

```
// Package and import declarations not shown for conciseness.

public final class GCDDebugger {
    public static void main(String[] args) {
        try {
            final VirtualMachineManager vmManager = Bootstrap.virtualMachineManager();
            final LaunchingConnector launchingConnector = vmManager.defaultConnector();
            final Map arguments = launchingConnector.defaultArguments();
            final Iterator iterator = arguments.values().iterator();
            while (iterator.hasNext()) {
                final Argument argument = (Argument) iterator.next();
                if (argument.name().equals("main")) {
                    argument.setValue("emn.course.vm.GCD");
                }
            }
            final VirtualMachine vm = launchingConnector.launch(arguments);
            vm.resume();
            final JFrame frame = new JFrame("Euclid's GCD algorithm debugger");
            frame.setLocation(100, 100);
            frame.setSize(300, 100);
            frame.addWindowListener(new WindowAdapter() {
                public void windowClosing(WindowEvent e) {
                    System.exit(0);
                }
            });
            frame.getContentPane().setLayout(new BorderLayout());
            final JButton suspendResumeButton = new JButton("Suspend");
            suspendResumeButton.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    JButton button = (JButton) e.getSource();
                    if (button.getText().equals("Suspend")) {
                        button.setText("Resume");
                        vm.suspend();
                    } else {
                        button.setText("Suspend");
                        vm.resume();
                    }
                }
            });
            frame.getContentPane().add(suspendResumeButton, BorderLayout.CENTER);
            final JButton stopButton = new JButton("Stop");
            stopButton.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    vm.exit(0);
                }
            });
            frame.getContentPane().add(stopButton, BorderLayout.EAST);
            frame.setVisible(true);
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Figure 12: *A tool to control Euclid's Greatest Common Denominator algorithm, both implemented in Java.*

2.9 Java Virtual Machine Profiler Interface

Reference: [11]

Sun is defining a standard profiling interface for the JVM, the Java Virtual Machine Profiler Interface (JVMPI). Sun provides the current implementation and documentation for the benefit of tool vendors who have an immediate need for profiling hooks in the JVM. The JVMPI continues to evolve, based on the feedback from customers and tool vendors.

The JVMPI defines a set of events and a set of data structures. The JVMPI-based tool tells the JVM the events it is interested in, and gives the JVM a pointer on the function to call when such events occur; the JVM calls this function and passes to it the data structure corresponding to the event.

The events known by the JVM are:

- Method enter and exit;
- Object allocate, move, and free;
- Heap arena create and delete;
- GC start and finish;
- JNI global reference allocate and free;
- JNI weak global reference allocate and free;
- Compiled method load and unload;
- Thread start and end;
- Class file data ready for instrumentation;
- Class load and unload;
- Contended Java monitor wait to enter, enter, and exit;
- Contended raw monitor wait to enter, enter, and exit;
- Java monitor wait and waited;
- Monitor dump;
- Heap dump;
- Object dump;
- Request to dump or to reset profiling data;
- JVM initialize and shutdown;

Exercise 20 (JVMPI events):
Identify the role and give an example of each event.

Example 13 (A profiling-tool for the Greatest Common Denominator implemented in Java):
The source code in Figure 13 shows a simple profiling-tool for the JVM, based on the JVMPI. This profiling-tool shows the names of the threads running in the JVM, and displays the names of the loaded classes (`java` and `sun` packages excluded).

First, the profiling-tool gets a pointer on the JVMPI (line 37–40). Then, it tells the JVM of the function to call when an event arrives: It passes to the JVM a pointer on the `NotifyEvent()` function (line 43). The profiling-tool notifies the JVM of the events it is interested in: It calls the `EnableEvent()` function with the appropriate JVMPI constants (lines 46–47).

When an interesting action occurs in the JVM (an action in which the profiling-tool is interested (lines 46–47)), the JVM calls the `NotifyEvent()` function defines lines 7–30. The `NotifyEvent()` function displays information according to the event received: The name of the loaded classed (`java` and `sun` packages excluded) or the name of the started threads.

The profiling-tool, named PCVM, profiles the Greatest Common Denominator algorithm implemented in Java when the JVM is called with the following syntax:

```
java -XrunPCVM emn.course.vm.GCD
```

In the console, the profiling-tool displays the following information:

```
Initializing the profiler for the Course on VMs.
Initialization done.

> Thread started: Signal Dispatcher
> Thread started: CompileThread0
> Class loaded: com.sun.rsajca.Provider
> Class loaded: com.sun.rsajca.Provider$1
> Class loaded: emn.course.vm.GCD
> Thread started: AWT-EventQueue-0
> Thread started: SunToolkit.PostEventQueue-0
> Thread started: AWT-Windows
> Class loaded: emn.course.vm.GCD$1
> Thread started: TimerQueue
> Thread started: Thread-0
```

Exercise 21 (The Java Virtual Machine Profiler Interface):
List some uncommon applications of the JVMPI.

```
#include <jvmpi.h>

// Define global JVMPI interface pointer
static JVMPI_Interface *jvmpi_interface;

// Define the function for handling event notification
void NotifyEvent(JVMPI_Event *event) {
    const char *class_name;
    const char *thread_name;
                                                                        10
    switch(event->event_type) {
        case JVMPI_EVENT_CLASS_LOAD:
            class_name = event->u.class_load.class_name;
            if (!((class_name[0] == 'j' &&
                   class_name[1] == 'a' &&
                   class_name[2] == 'v' &&
                   class_name[3] == 'a')
                  ||
                  (class_name[0] == 's' &&
                   class_name[1] == 'u' &&                              20
                   class_name[2] == 'n'))) {
                fprintf(stderr, "> Class loaded: %s\n", class_name);
            }
            break;
        case JVMPI_EVENT_THREAD_START:
            thread_name = event->u.thread_start.thread_name;
            fprintf(stderr, "> Thread started: %s\n", thread_name);
            break;
    }
}                                                                       30

// Profiler agent entry point
JNIEXPORT jint JNICALL JVM_OnLoad(JavaVM *jvm, char *options, void *reserved) {
    fprintf(stderr, "Initializing the profiler for the Course on VMs.\n");

    // Get JVMPI interface pointer
    if (((*jvm)->GetEnv(jvm, (void **)&jvmpi_interface, JVMPI_VERSION_1)) < 0) {
        fprintf(stderr, "Initialization error in obtaining JVMPI interface pointer.\n");
        return JNI_ERR;
    }                                                                   40

    // Initialize the JVMPI interface
    jvmpi_interface->NotifyEvent = NotifyEvent;

    // Enable "class load"- and "thread start"-event notification
    jvmpi_interface->EnableEvent(JVMPI_EVENT_CLASS_LOAD, NULL);
    jvmpi_interface->EnableEvent(JVMPI_EVENT_THREAD_START, NULL);

    fprintf(stderr, "Initialization done.\n\n");
    return JNI_OK;                                                      50
}
```

Figure 13: *A profiling-tool for the JVM, based on the JVMPI.*

2.10 A brief comment

The Java programming language and the JVM offer a great potential and numerous interesting functionalities.

However, the Java programming language and the JVM do not reify most of the JVM functionalities. This lack of reification defeats the purpose of the object-oriented paradigm, because the execution mechanism, the object creation mechanism... are external to the Java language and the JVM.

3 Other languages in a nutshell

There exist several programming languages and environments that use virtual machines. In this section, we present four programming languages and their virtual machines in a nutshell. The objective is not to give an in-depth presentation of the virtual machines, the objective is to give some ideas on the wide range on virtual machines implementations and usages.

> **Exercise 22 (A world of virtual machines):**
> List several languages with specific virtual machines, especially, take a look at the virtual machines dedicated to functional programming languages, such as Lisp and Scheme.

3.1 Pascal

Reference: [14]

The Pascal programming language was the first to use an intermediate language and a virtual machine: The UCSD p-System.

The UCSD p-System is a portable operating system that was popular in the early days of computer science, in the late 1970s and early 1980s. It is based on a virtual machine with a standard set of low-level, machine-language-like p-Code instructions that are emulated on different hardware. The most popular language for the p-System is UCSD Pascal. The p-System operating system itself is written in UCSD Pascal: This makes the entire operating system relatively easy to port across platforms.

3.2 C♯

Reference: [3] and [5]

The C♯ programming languages is the main entry point to the Microsoft .Net programming platform. The .Net platform defines an intermediate language, an object-model, a set of base class-libraries, an execution framework, and several run-time services. The material available on the .Net platform is still enormous and fast-growing. The following presentation of the .Net platform from the run-time point of view is brief and misses important points.

From the run-time perspective, the .Net platform decomposes into several parts:

- MSIL: The Microsoft Intermediate Language.
- CLR: The Common Language Runtime.
- BCL: The Base Class-Libraries.

- COM: The .NET platform includes native support for the Component Object Model (COM).
- Web services: The .NET platform integrates the definition and the usage of Web services.

> **Exercise 23 (The .Net platform):**
> Search information on the features of .NET platform.

The .NET platform defines a general object model, the Virtual Object System (VOS). This object model supports several kind of method dispatch, it limits inheritance to single class inheritance and multiple abstract class (interface) inheritance. At run-time, data exists as scalars, as references, and as instances of value classes. The VOS allows two different method invocation mechanisms: Virtual dispatch and non-virtual dispatch. The VOS may pass parameters by value or by reference.

The instructions defined by the MSIL, in opposition to the JVM op-codes, have no specific data types.

> **Exercise 24 (The .Net platform run-time and multi-languages):**
> Assess how the .NET platform run-time features enable the .NET platform to support different languages and different programming paradigms.

Example 14 (Greatest Denominator algorithm in the Microsoft Intermediate Language):
The following code represents the MSIL code required to implement the Greatest Common Denominator algorithm as described in Example 7.

```
.method public hidebysig static int32  ComputeGCD(int32 a,
                                                   int32 b) cil managed
{
  // Code size       21 (0x15)
  .maxstack  3
  .locals (int32 V_0)
  IL_0000:  ldarg.1
  IL_0001:  brtrue.s   IL_0007

  IL_0003:  ldarg.0
  IL_0004:  stloc.0
  IL_0005:  br.s       IL_0013

  IL_0007:  ldarg.1
  IL_0008:  ldarg.0
  IL_0009:  ldarg.1
  IL_000a:  rem
  IL_000b:  call       int32 GCD::ComputeGCD(int32,
                                             int32)
  IL_0010:  stloc.0
  IL_0011:  br.s       IL_0013

  IL_0013:  ldloc.0
  IL_0014:  ret
} // end of method GCD::ComputeGCD
```

3.3 Prolog

Reference: [19]

Logic programming languages, such as Prolog, stirred up much research works on their implementation. The common implementation of Prolog uses the Warren Abstract Machine (WAM). The WAM is an abstract machine consisting of a memory architecture and an instruction set tailored to Prolog. It can be realized efficiently on a wide range of hardware, and serves as a target for portable Prolog compilers. It has now become accepted as a standard basis for implementing Prolog.

The WAM is a stack machine with registers. The WAM adopts *structure copying* to represent Prolog terms. A heap exists, but the WAM reduces its use to the maximum. The WAM uses the heap to store variables and structures. The WAM includes a special stack, called the Push-Down-List, to memorize unifications. The WAM also has another stack to keep trace of choice points and of environment frames. Finally, the WAM possesses a special stack, the

trail. The trail is organized as an array of addresses of the (stack or heap) variables that must be reset (set to *unbound*) upon backtracking.

The WAM does not provide debugging or profiling possibilities in itself. On one hand, the WAM is a theoretical work on virtual machine implementation for logic programming. On the other hand, some authors have proposed formal definition of the WAM and have proved the correctness of compilers: This is a very formal and theoretical approach to virtual machines.

> **Exercise 25 (Warren Abstract Machine):**
> List the differences between the Warren Abstract Machine and the Java Virtual Machine. Compare the execution of a Prolog program with the execution of a Java program.

3.4 Squeak

Reference: [12]

The Squeak programming environment is a direct descendant of the original Smalltalk programming environment. The Squeak programming environment includes the Squeak Object Engine.

The Squeak Object Engine (SOE) encompasses both the Smalltalk low-level system code (the `Context`, `Process`, `Number`, `InstructionStream`, and `Class` classes) and the virtual machine. The SOE contains a stack machine. The SOE is written in a subset of Smalltalk. The SOE does provide memory handling, meta-programming capability, accessible threads and control structures, message sending, object creation... Theoretically, it is possible for a SOE to run into another SOE.

The SOE is implemented with simplicity in mind. It offers a simple and extensible architecture. It does not offer direct debugging or profiling capabilities, however, it must be possible to debug or to profil Squeak application by running the application within a SOE running itself within a SOE.

> **Exercise 26 (Squeak implementation of the Squeak virtual machine):**
> List the advantages and drawbacks of the Squeak virtual machine being implemented in Squeak. Study and explain the interface between the Squeak virtual machine and the operating system.

4 Other ideas in a nutshell

4.1 Proof-carrying code

Reference: [1]

Proof-Carrying Code (PCC) is the technique to guarantee that a program does not access unauthorized resources, read private data, and overwrite valuable data. With PCC, the run-time environment defined a safety policy, which tells under what conditions a word of memory may be read or written or how much of a resource may be used (such as CPU cycles). The program must provide a program-verification proof that it satisfies the safety policy. With PCC, the proof is performed on the native machine code and it may be fully automatically built. The PCC can use both types and data flow to prove safety.

4.2 Strongly-typed intermediate languages

Reference: [16]

Statically typed intermediate languages are effective tools for staging the compilation of high-level languages. Types express invariants that help programmers understand their programs, and strongly typed languages prevent many common programming errors. Compiler writers can use these properties to debug sophisticated program transformations such as closure conversion and optimizations like data-type specialization. Types not only help check the correctness of transformations but enable analyses or optimizations that are extremely difficult without them.

References

[1] Andrew W. Appel and Amy P. Felty ; *A Semantic Model of Types and Machine Instructions for Proof-Carrying Code* ; Proceedings of POPL, January 2000.

[2] Sheng Liang and Gilad Bracha ; *Dynamic Class Loading in the Java Virtual Machine* ; Proceedings of OOPSLA, October 1998, pages 36–44.

[3] K. John Gough ; *Stacking them up: a Comparison of Virtual Machines* ; Proceedings of the Australian Computer Systems and Architecture Conference, February 2001.

[4] Yann-Gaël Guéhéneuc and Narendra Jussien and Rémi Douence ; *No Java without Caffeine – A Tool for Dynamic Analysis of Java Programs* ; Proceedings of ASE, September 2002, pages 117–126.

[5] Peter Drayton and Ben Albahari and Ted Neward ; *C♯ in a Nutshell* ; March 2002, O'Reilly, ISBN: 0-596-00181-9.

[6] *Java ClassLoader* ; Available at:
`java.sun.com/j2se/1.3/docs/api/java/lang/ClassLoader.html`

[7] *Java Garbage Collector* ; Available at:
`www.javaworld.com/javaworld/jw-08-1996/jw-08-gc.html`

[8] *Java Native Interface* ; Available at:
`java.sun.com/products/jdk/1.2/docs/guide/jni/`

[9] *Java Platform Debug Interface* ; Available at:
`java.sun.com/products/jpda/`

[10] *Java Security* ; Available at:
`www.securingjava.com/chapter-two/`

[11] *Java Virtual Machine Profiler Interface* ; Available at:
`java.sun.com/j2se/1.3/docs/guide/jvmpi/`

[12] *Squeak at Its Core* ; Available at:
`coweb.cc.gatech.edu/squeakbook/35`

[13] *The Java Virtual Machine Specification* ; Available at:
`java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html`

[14] *The UCSD p-System Museum* ; Available at:
`www.threedee.com/jcm/psystem/`

[15] *Turing Machine* ; Available at:
`cgi.student.nada.kth.se/cgi-bin/d95-aeh/get/umeng`

[16] *Types Assembly Language* ; Available at:
`www.cs.cornell.edu/talc/default.html`

[17] *Virtual Turing Machine* ; Available at:
`www.nmia.com/ soki/turing/`

[18] *Virtual Virtual Machine* ; Available at:
`www-sor.inria.fr/projects/vvm/papers.html`

[19] *Warren Abstract Machine* ; Available at:
`www.isg.sfu.ca/ hak/documents/wam.html`

Acknowledgments

The author would like to thank Hervé Albin-Amiot, Xavier Alvarez, Pierre-Charles David, Andrés Farías, Jacques Noyé, and Marc Ségura-Devillechaise for their help and comments on these lecture notes.

The author does not claim to have produced all the content of these lecture notes and would gladly accept any comment, suggestion, and correction that a kind reader could suggest.