A Formal Introduction to the Compilation of Java

Stephan Diehl
FB 14 - Informatik, Universität des Saarlandes, Postfach 15 11 50, 66041 Saarbrücken, Germany
(email: diehl@cs.uni-sb.de)

SUMMARY
The term *abstract machine* is widely accepted to denote intermediate target languages and related architectures which serve as an intermediate stage to compile programming languages. In this paper we explain how a considerable subset of Java is translated into Byte-Code for the Java Virtual Machine, an abstract machine used as a target for Java compilation. Using formal and precise notation we present the language concepts, the related byte-code instructions and the compilation schemes. Hitherto none of the existing literature on the JVM[1,2] describes how compilation is done, but present the JVM in isolation.

KEY WORDS Java, compilation, abstract machines, object-orientation

Introduction

In 1995 SUN introduced a new programming language called Java. Although originally developed for different purposes, Java became the standard language for sending platform independent programs over the internet and executing them as part of Web pages. SUN has spent much effort and money into marketing Java, building huge libraries for purposes like security or commerce and extending the use of Java into new areas like Smartcards or embedded systems. From a compiler design perspective Java offers a variety of known techniques. A key feature of Java is, that it is translated into an intermediate language, called Java-Byte-Code. Java-Byte-Code programs are then executed by the Java Virtual Machine (JVM). Some of the advantages of using byte-code instead of native code are increased portability and security, and reduced program size (at least for RISC machines). The major disadvantage is the worse performance because of interpretation overhead.
The goal of this paper is to explain the JVM and how Java is translated into Byte-Code. Eventually this should lead to a more in-depth understanding of Java. Furthermore the JVM can also be used as a target machine to translate other programming languages. Our presentation of the JVM follows the technical style introduced by Wilhelm and Maurer[3] to describe other abstract machines[1]. We present the language concepts, the related byte-code instructions and the compilation schemes. None of the existing literature on the JVM[1,2] describes how compilation is done, but present the JVM in isolation.

Restrictions

The specification we present was designed to ease teaching and understanding of the Java Virtual Machine and the compilation of Java to Byte-Code. To achieve this we do not address all instructions of the Java Virtual Machine here, we restrict the complexity of the static information contained in class files and we discuss only the compilation of a substantial subset of Java, which we call TassKaf . We also deviate from the Java Virtual Machine Specification (JVM:1995) in that we make explicit, where local variables are stored and how data structures like arrays are represented. In the Java Virtual Machine Specification such details have been left open, such that an implementor could choose the most efficient implementation on his machine. We also do not describe the Java-Class-File-Format but use a more formal approach. As an alternative we explain how one can use JASMIN,[1] an assembler language for Java-Byte-Code, as a target language and use the JASMIN assembler to produce the related class file.

[1] The term *abstract machine* is widely accepted to denote intermediate target languages and related architectures (like the JVM) which serve as an intermediate stage to compile programming languages.

Execution of Java programs

Although developed for other purposes, namely programming consumer electronics products, Java has become very popular for writing applications which can be loaded over the internet and executed on different platforms. First we look at how the overall process of executing a Java program works.

Phase1: The Java program is translated into Java-Byte Code. Actually a Java program consists of a set of class definitions. For every class defined in the Java program a so-called class file is generated which contains the byte-code and additional information like signatures and access rights.

Phase2: A Class file is loaded and one of its methods is called. If the class file was loaded over the internet, then before the method can be executed, the class file is verified. The verifier checks, whether the byte-codes of each method violate certain security restrictions. The verification is only done, when the class file is loaded, not every time a method is called. Finally the static fields of the class are initialized.

Phase3: A method is called[2], which is not defined in this class but belongs to another class. If the other class has already been loaded, then it is checked, whether the caller is allowed to access the called method. If the class was not yet loaded, it is loaded and verified.

To improve performance Just-In-Time (JIT) compilers can be used, e.g. a JIT compiler developed by Borland is part of the PC version of Netscape 3.0, other companies providing JIT compilers include Silicon Graphics (for IRIX), Symantec, Microsoft and Asymetrix (all for PC). For numerical programs JIT compilers outperform Sun's JDK (which uses an interpreter) by up to a factor of 30. For graphics intensive programs the performance gains are less dramatic (factor of 2), because in both approaches most computation for these programs takes part in the systems graphics routines.[4,5] After verifying the byte-code the JIT translates it into native code of the local machine.

Components of the JVM

Class File Loader

For every class which is accessed when running the machine the class file loader reads the class file, verifies it and initializes its static fields. The information contained in the class file is then made accessible to the rest of the JVM. There is information on constants, fields, methods and attributes of the class:

$$\Gamma_K = N \to \mathsf{Number} \cup \mathsf{String} \cup (\mathsf{Number} \times \mathsf{String}) \cup (\mathsf{String} \times \mathsf{String})$$
$$\Gamma_F = \mathsf{String} \to N \times \{static, dynamic\} \times N$$
$$\Gamma_M = \mathsf{String} \to \mathsf{String} \times N \times N \times N \times \{static, dynamic\} \times N$$
$$\Gamma_P = N \to \mathsf{Byte}$$

Class Loader: $\gamma : \mathbf{String} \to \Gamma_K \times \Gamma_F \times \Gamma_M \times \Gamma_P \times N \times N$

The function γ returns the information contained in the class file. Let c be the name of a class, then $\gamma(c) = (\gamma_K, \gamma_F, \gamma_M, \gamma_P, C, S)$ where

Constant Pool γ_K : In the constant pool there are entries for numbers, strings, class names, method names and types. For array types it contains entries of the form (n, t), where n is the dimension and t the element type of the array. For fields, i.e., object and class variables, it contains entries of the form (c, f), where c is the name of the class and f the name of the field.

[2] This includes the case when an object is created and its constructor method is called.

Field Table γ_F : For each fieldname the table contains an entry (t, sd, a) where t is the type of the field, sd is either $static$ or $dynamic$. In the first case a is the address of the static field otherwise it is an offset.

Method Table γ_M : For each method the table contains an entry (c, p, n, l, sd, ms) where c is the name of the class, the method occurs in, p is a pointer to the byte-code for this method, n is the number of arguments of the method, l the number of local variables and ms the number of stack cells maximally needed by the method. sd is either $static$ or $dynamic$.

Byte-Code Programs γ_P : Byte-Code programs are sequences of cells of size 1 byte. There are a few instructions whose arguments are encoded in the instruction (e.g. `iconst_1`). For all other instructions the instruction and its arguments are stored in consecutive cells.

This Class C : index into constant pool

Superclass S : index into constant pool

In the following we refer with γ_K, γ_F, γ_M and γ_P to the tables of the class which the byte code occurs in. To make the class c explicit we sometimes write γ_K^c, γ_F^c, γ_M^c and γ_P^c.
To compile a TassKaf program means to generate a class file. In our more formal presentation this means one has to compute γ.

Execution Environment

The JVM has 4 registers:

PC : program pointer, points to a position in the program store.

$VARS$: all local variables are addressed relative to this register.

$OPTOP$: points to the topmost cell of the operand stack.

$FRAME$: points to the first cell of the execution environment.

In this presentation we use a fifth register CC. This register is not mentioned in the official specification of the JVM, although internally a similar mechanism must be used. The register CC contains the current class, i.e., the class the method which is executed at the moment is defined in. For example γ_K^{CC} is the constant pool of the current class.
In Figure 1 we show the JVM stack and where the registers point to. The execution environment of a method invocation contains the return address and the values of the registers $VARS$, $FRAME$ and CC of the calling method. These values are needed to restore the registers of the calling method after the invoked method returns.

Store

The store is a sequence of cells of size 4 bytes: $STORE : N \rightarrow Byte \times Byte \times Byte \times Byte$
The stack is a sequence of cells of size 4 bytes. $STACK : N \rightarrow Byte \times Byte \times Byte \times Byte$
The heap contains cells of size 4 bytes: $HEAP : N \rightarrow Byte \times Byte \times Byte \times Byte$
Stack and heap are disjoint areas of the store, i.e. $dom(STACK) \cap dom(HEAP) = \emptyset$

Execution Loop of the JVM

The JVM executes byte code as follows:

```
do forever
  { execute(γP(PC));
  }
```

Each instruction changes the value of PC accordingly. In the following subsections we define the procedure `execute` and thus the semantics of each instruction of the JVM.

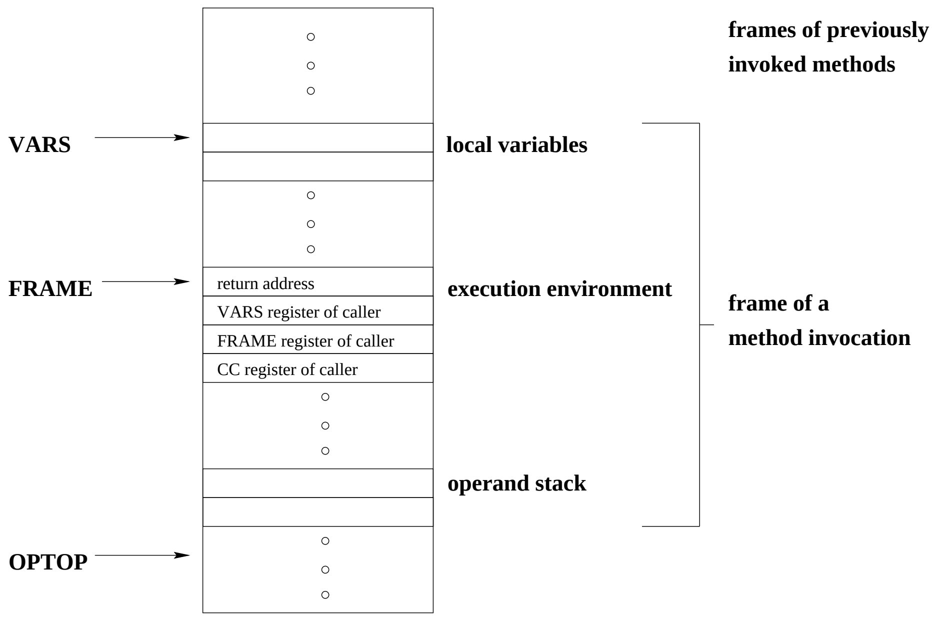

Figure 1: Registers and Stack

Syntax of TassKaf

In Table 1 the syntax of TassKaf is given. TassKaf is the subset of Java we discuss in the rest of this paper. Besides integer arithmetic, simple expressions and a variety of control statements, our subset includes many important aspects of Java, e.g. classes and inheritance, virtual and static methods, method overloading and arrays. The semantics of the constructs in TassKaf is the same as that of the related constructs in Java. Java constructs which are not present in TassKaf include casts, user defined constructors, interfaces, threads and exceptions. Nevertheless we will discuss some of these later. To illustrate some of the features of TassKaf we continue by discussing an example program.

```
class Example extends Object     // inheritance
 { bool b;                       // object fields
   int i=3;                      // initialization of field
   Example ex;                   // reference to object
   Example ex1 = new Example();  // object creation
   int a1[];                     // reference to array
   int a2[][]=new int [20][];    // array creation

```

A class `Example` is defined which inherits from class `Object`. In the class we define some object fields, i.e., every object of this class has a own copy of such a field. Fields can be of type `int`, `bool`, any class type or an array type. In the definition of a field it can be initialized. Objects and arrays are created with `new`.

```
   static int si=3;              // class fields
   static int sa[];
   static int sa2[][]=new int [20][];
   static bool b;
```

A definition of a class field starts with the keyword `static`. In contrast to object fields there is only one copy of a class field.

```
    void m1(int i, Example e, int a [ ])
     { int j=3;                    // local variables
       int a1[];
       int a2[][]=new int [20][];
       bool b;
```

The object method `m1` is defined to be of return type `void`, i.e. the method returns no value. Its three arguments are of different types. Primitive values are passed by value, objects and arrays by reference. Within the method local variables are defined. The object fields `a1` and `a2` are not visible inside `m1`, because it defines local variables with the same names.

```
       e.i = e.i - 1;                 // accessing object fields
       Example.si = Example.si - 1;   // accessing class fields
```

The first two statements in the method are assignments. Here `e` is an argument of the method and its value is an object of class `Example` and `i` is an object field of that class as defined above. In the second assignment the value of the class field `si` of the class `Example` is modified.

```
       if (i>0)
          { e.m1(i-1,this,a1); }            // invoke object method (virtual)
        else                                // special name *this*
          { for i:=1 to 10 do a1[i]=i; od; }
       return;
     }
```

Next the method `m1` of the object in argument `e` is called. One argument of the method call is `this`, which denotes the current object,i.e., the object whose method is executed at the moment.

```

    int m1(int i) { return i-1; }        // m1 is overloaded
    Example m1(Example e) { return e; }
```

Two other versions of method `m1` are defined. They differ from the one defined above both in their result and in their argument types.

```

    static Example m2(Example e)         // class method (static)
     { return e; }
```

With the keyword `static` the method `m2` is defined to be a class method.

```
    static void main(java.lang.String argv [])   // main method of program !!!
     { Example obj=new Example();
       Example.m2(obj);
```

If the class is to be compiled and run as a standalone application, then the class must contain a class method `main` with the above return and argument types. In this example the `main` method creates an `Example` object and calls the class method of `m2` with this object as an argument.

```
       a1=new int[20];                       // access field of current object
       obj.m1(10,obj,a1);
       obj.m1(obj).i=1;
     }
   }
```

Then it changes the value of the object field `a1` of the current object. Since there is no local variable with the name `a1` in the method `main`, the assignment `a1=new int[20];` is short for `this.a1=new int[20];`. In the last assignment the method `m1` of the object in `obj` is called. For an argument of class `Example` the method returns an object of class `Example` again. Then the value 1 is assigned to the object field `i` of the returned object.

Table 1: Syntax of TassKaf

<table>
<tr><td>class</td><td>→</td><td>class classname extends classname {(vardec|methdec)*}</td></tr>
<tr><td>classname</td><td>→</td><td>(LowercaseIdentifier (.))* UppercaseIdentifier</td></tr>
<tr><td>varname</td><td>→</td><td>LowercaseIdentifier</td></tr>
<tr><td>methame</td><td>→</td><td>LowercaseIdentifier</td></tr>
<tr><td>vardec</td><td>→</td><td>(static) lvardec</td></tr>
<tr><td>lvardec</td><td>→</td><td>varspec = expression ;</td></tr>
<tr><td>varspec</td><td>→</td><td>type varname ([])*</td></tr>
<tr><td>methdec</td><td>→</td><td>(static) type methname ((varspec(, varspec)*)) { (lvardec)* statements }</td></tr>
<tr><td>type</td><td>→</td><td>classname | int | void</td></tr>
<tr><td>expression</td><td>→</td><td>expression op expression | (expression) | - expression
| variable | constant | methcall | new | this</td></tr>
<tr><td>variable</td><td>→</td><td>(callee) varname ([expression])*</td></tr>
<tr><td>constant</td><td>→</td><td>(0|1|...|9)+</td></tr>
<tr><td>callee</td><td>→</td><td>(classname|expression) .</td></tr>
<tr><td>methcall</td><td>→</td><td>(callee) methname ((expression (, expression)*))</td></tr>
<tr><td>new</td><td>→</td><td>new classname() | new type ([expression])+ ([])*</td></tr>
<tr><td>statements</td><td>→</td><td>statement*</td></tr>
<tr><td>statement</td><td>→</td><td>stmt1 ; | stmt2</td></tr>
<tr><td>stmt1</td><td>→</td><td>variable = expression | do { statements } while (expression)
| break | methcall | return (expression)</td></tr>
<tr><td>stmt2</td><td>→</td><td>if (expression) { statements }
| if (expression) { statements } else { statements }
| while (expression) { statements }
| for ((stmt1|stmt2) ; expression ; (stmt1|stmt2)) { statements }
| switch (expression) { (case constant : statements)* (default : statements) }</td></tr>
<tr><td>op</td><td>→</td><td>+ | - | * | / | == | != | ≤ | ...</td></tr>
</table>

Compiling TassKaf to Java-Byte-Code

To define how TassKaf is compiled, we define a function *code*. This function takes two arguments: program parts p and an address environment ρ and returns sequences of byte-code instructions. We define $code\ p\ \rho$ by structural induction on p. For now, we assume that ρ maps variable names to addresses. The computation of ρ will be discussed later.

Expressions

To translate integer expressions and simple boolean expressions, we need JVM instructions for integer arithmetic, stack manipulation and conditional and unconditional jumps.

Integer Arithmetics

Instruction	Meaning	Before	After
iadd	$STACK[OPTOP-1] :=$ $STACK[OPTOP-1] + STACK[OPTOP]$; $OPTOP := OPTOP - 1$; $PC := PC + 1$	(N,N)	(N)

analogous: **isub**, **imul**, **idiv**, **iand**, **ior**

Instruction	Meaning	Before	After
ineg	$STACK\,[OPTOP] := -STACK\,[OPTOP]$; $PC := PC + 1$	(N)	(N)

Stack Manipulation

Instruction	Meaning	Before	After
iconst_0	$OPTOP := OPTOP + 1$; $STACK\,[OPTOP] := 0$; $PC := PC + 1$	()	(N)
iconst_1	$OPTOP := OPTOP + 1$; $STACK\,[OPTOP] := 1$; $PC := PC + 1$	()	(N)
bipush b	$OPTOP := OPTOP + 1$; $STACK\,[OPTOP] := b$; $PC := PC + 2$	()	(N)
ldc i	$OPTOP := OPTOP + 1$; $STACK\,[OPTOP] := \gamma_K^{CC}(i)$; $PC := PC + 2$	()	(N)
istore v	$STACK\,[VARS + v] := STACK\,[OPTOP]$; $OPTOP := OPTOP - 1$; $PC := PC + 2$	(N)	()
iload v	$OPTOP := OPTOP + 1$; $STACK\,[OPTOP] := STACK\,[VARS + v]$; $PC := PC + 2$	(N)	()

Jumps

Instruction	Meaning	Before	After
goto a	$PC := a$	()	()

ifeq a	**if** $(STACK\,[OPTOP] = 0)$ **then**
	$PC := a$
	else
	$PC := PC + 3$
	fi;
	$OPTOP := OPTOP - 1$
		(N)	()

if_icmpeq a_1 a_2	**if** $(STACK\,[OPTOP - 1] = STACK\,[OPTOP])$ **then**
	$PC := a_1$
	else
	$PC := a_2$
	fi;
	$OPTOP := OPTOP - 2$
		(N)	()

Other conditional jumps include **ifne** ($\neq 0$), **ifle** (≤ 0),**ifge** (≥ 0), **if_icmpne** ($\neq 0$), **if_icmple** (≤ 0) and **if_icmpge** (≥ 0). In Table 2 we show the translation schemes for boolean expressions (comparison), integer arithmetic expressions and assignments to a local integer variable. We will address assignments to arrays, class and object fields later.

Table 2: Translation of Expressions

Function		Restriction
$code(e_1 == e_2)\ \rho$	$= code\ e_1\ \rho; code\ e_2\ \rho;$**isub**;	$Type(e_1) = Type(e_2) = N$
	ifeq l_1; **iconst_0**; **goto** $l_2; l_1$: **iconst_1**; l_2 :	
$code(e_1 != e_2)\ \rho$	$= code\ e_1\ \rho; code\ e_2\ \rho;$**isub**;	$Type(e_1) = Type(e_2) = N$
	ifeq l_1; **iconst_1**; **goto** $l_2; l_1$: **iconst_0**; l_2 :	
$code(e_1 < e_2)\ \rho$	$= code\ e_1\ \rho; code\ e_2\ \rho;$ **isub**;	$Type(e_1) = Type(e_2) = N$
	iflt l_1; **iconst_0**; **goto** $l_2; l_1$: **iconst_1**; l_2 :	
$\vdots$		
$code(e_1 + e_2)\ \rho$	$= code\ e_1\ \rho; code\ e_2\ \rho;$**iadd**	$Type(e_1) = Type(e_2) = N$
$code(e_1 - e_2)\ \rho$	$= code\ e_1\ \rho; code\ e_2\ \rho;$ **isub**	$Type(e_1) = Type(e_2) = N$
$code(e_1 * e_2)\ \rho$	$= code\ e_1\ \rho; code\ e_2\ \rho;$ **imul**	$Type(e_1) = Type(e_2) = N$
$code(e_1 / e_2)\ \rho$	$= code\ e_1\ \rho; code\ e_2\ \rho;$ **idiv**	$Type(e_1) = Type(e_2) = N$
$code(-e)\ \rho$	$= code\ e\ \rho;$ **ineg**	$Type(e) = N$
$code\ x\ \rho$	$=$ **iload** $\rho(x)$	x local variable of type N
$code\ c\ \rho$	$=$ **ldc** $\gamma_K^{-1}(c)$	c constant of type N
$code(x = e)\ \rho$	$= code\ e\ \rho;$ **istore** $\rho(x)$	$Type(e) = T$, x local variable

Table 3: Translation of Control Flow Statements

$code($**if** e **then** st_1 **else** st_2 **fi**$)\ \rho$	$= code\ e\ \rho;$ **ifeq** l_1; $code\ st_1\ \rho$; **goto** l_2; l_1: $code\ st_2\ \rho$; l_2:
$code($**if** e **then** st **fi**$)\ \rho$	$= code\ e\ \rho;$ **ifeq** l; $code\ st\ \rho$; l:
$code$ (**while** e **do** st **od**) ρ	$= l_1$: $code\ e\ \rho$; **ifeq** l_2; $code\ st\ \rho$; **goto** l_1; l_2 :
$code$ (**do** st **while** e) ρ	$= l$: $code\ st\ \rho$; $code\ e\ \rho$; **ifne** l
$code\ (st_1;st_2)\ \rho$	$= code\ st_1\ \rho;\ code\ st_2\ \rho$

Statements

Using conditional and unconditional jumps translation schemes for basic control flow statements like **if** and **while** are defined in Table 3. The table does not contain an entry for the **for** loop, because the **for** loop can be translated into a **while** loop:

$$\textbf{for}(init;condition;step)\ body \ \Rightarrow\ init;\textbf{while}\ condition\ \textbf{do}\ body;step\ \textbf{od}$$

By expanding the individual translation schemes we get a translation scheme:

$code$ (**for**($init;condition;step$) $body$) ρ =
$code\ init$; l_1: $code\ condition\ \rho$; **ifeq** l_2; $code\ body\ \rho$; $code\ step\ \rho$; **goto** l_1; l_2:

To translate **switch** statements the JVM offers a very powerful instruction called **lookupswitch**.

Instruction	Meaning	Before	After
lookupswitch $d\ n$ $i_1\ a_1$ ... $i_n\ a_n$	**if** $(STACK[OPTOP] = i_j)$ (for smallest $1 \le j \le n$) **then** $PC := a_j$; **else** $PC := d$; **fi**; $OPTOP := OPTOP - 1$	(N)	()

To translate nested **switch** statements, each occurrence of a **break** label must be mapped onto a label at the end of the code produced for the directly embracing **switch** statement. To achieve this we use a special name λ which we bind to the corresponding label.

$$code \left(\begin{array}{l} \textbf{switch}\ (e)\ \{ \\ \quad \textbf{case}\ i_1\ :\ st_1; \\ \quad \vdots \\ \quad \textbf{case}\ i_k\ :\ st_k; \\ \quad \textbf{default}\ :\ st \\ \} \end{array} \right) \rho\ = \begin{array}{rl} & \textbf{lookupswitch}\ l_d\ k\ i_1\ l_1 \ldots\ i_k\ l_k\ ; \\ l_1 : & code\ st_1 \rho'; \\ & \vdots \\ l_k : & code\ \ st_k\ \ \rho'; \\ l : & code\ \ st\ \ \rho'; \\ b : & \end{array}$$

where $\rho' = \rho[\lambda \mapsto b]$.

When translating a **break** statement within one of the st_i or st, the corresponding label can be found in ρ.

$code\ \ \textbf{break}\ \rho\ \ = \textbf{goto}\ \rho(\lambda)$

Arrays

In TassKaf arrays can be muli-dimensional and their elements can be primitive types or objects. In the JVM multi-dimensional arrays are implemented as one-dimensional arrays whose elements pointers to multi-dimensional arrays.

Creation of Arrays

Instruction	Meaning	Before	After
newarray t	$STACK\,[OPTOP]$ $:= newarray(STACK\,[OPTOP], t);$ $PC\ := cpc + 2$	(N) $t \in \{int, bool\}$	(A)
anewarray i	**let** $t = \gamma_K^{CC}\,(i)$ **in** $STACK\,[OPTOP]$ $:= newarray(STACK\,[OPTOP], t);$ $PC\ := PC\ + 3$ **end**	(N) t must be a class	(A)
multianewarray $i\ d$	**let** $t = \gamma_K^{CC}\,(i)$ $(d_0, t_0) = \gamma_K^{CC}\,(t)$ **in** $OPTOP\ := OPTOP\ - d;$ $STACK\,[OPTOP] := alloc(1, d, d_0, t_0);$ $PC\ := PC\ + 4$ **end**	(N,...,N) d times	(A)

Now we have to define the auxiliary function $alloc$:

$alloc(s, d, d_0, t_0)=$
    if $s = d_0$ **then return** $newarray(STACK\,[OPTOP\ + s - 1], t_0);$
    $r := newarray(STACK\,[OPTOP\ + s - 1], ArrayPointer);$
    if $s < d$ **then**
        for $i := 0$ **to** $STACK\,[OPTOP\ + s - 1] - 1$ **do**
            $STORE\,[r + i] := alloc(s + 1, d, d_0, t_0);$
    return r;

The function $newarray(s, t)$ returns the address a of an area of $1 + s * sizeof(t)$ consecutive free cells in $HEAP$, where $sizeof(t)$ yields the number of 4 Byte cells needed to store one instance of type t,

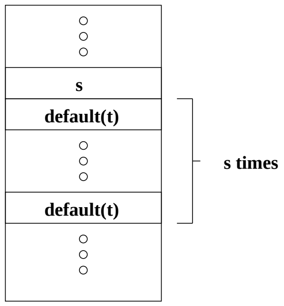

Figure 2: Allocation of Arrays

if t is a scalar type, or a pointer to an object, if t is an object type. The area is initialized as follows, where $default(t)$ is a fixed default value of type t, e.g. 0 for integers and `false` for booleans:
Arrays are allocated with **new** in expressions. In Java it is not neccessary to provide values for all dimensions of an array:

$$code\ (\mathbf{new}\ t\ [e_1]\ldots[e_k]\ \underbrace{[]\ldots[]}_{m-times}\ )\ \rho = code\ e_1\ \rho;\ldots code\ e_k\ \rho; \mathbf{multianewarray}\ i\ k$$

where $\gamma_K(i) = (k+m, t)$.

There are two special cases for one-dimensional arrays. For primitive types we use **newarray** and for class types **anewarray**:

$code\ (\mathbf{new}\ t\ [e])\ \rho = code\ e\ \rho; \mathbf{newarray}\ i\ k$ where $t \in \{int, bool\}$ and $\gamma_K(i) = t$.

$code\ (\mathbf{new}\ c\ [e])\ \rho = code\ e\ \rho; \mathbf{anewarray}\ i\ k$ where c is a classname and $\gamma_K(i) = c$.

Access to Arrays

There are several instructions for accessing arrays and their elements. Some of these instructions can also be used for objects. The instructions **aload** loads loads a reference to an object or array from local a variable on top of the stack, the instruction **astore** stores the reference on top of the stack into a local variable. Given a reference to an array and an index on top of the stack, the instructions **aaload** and **iaload** load the reference or integer value of the given array cell onto the stack. Finally, the instructions **aastore** and **iastore** store the value on top of the stack into the given array cell.

Instruction	Meaning	Before	After
aload v	$OPTOP := OPTOP + 1;$ $STACK[OPTOP] := STACK[VARS + v];$ $PC := PC + 2$	() or (A)	(O)
aaload	$OPTOP := OPTOP - 1;$ **if** $(STACK[OPTOP] = NULL)$ **or** $(STACK[OPTOP + 1] < 0)$ **or** $(STACK[OPTOP + 1] > STORE[STACK[OPTOP]])$ **then** error(); **fi**; $STACK[OPTOP] :=$ $STORE[STACK[OPTOP] + STACK[OPTOP + 1] + 1];$ $PC := PC + 1$	(A,N)	(O) or (A)
iaload	same as **aaload**	(A,N)	(N)
astore v	$STACK[VARS + v] := STACK[OPTOP];$ $OPTOP := OPTOP - 1;$ $PC := PC + 2$	(O) or (A)	()
aastore	$OPTOP := OPTOP - 3;$ **let** $i = STACK[OPTOP + 2]$ **in** **if** $(STACK[OPTOP + 1] = NULL)$ **or** $(i < 0)$ **or** $(i > STORE[STACK[OPTOP + 1]])$ **then** error(); **fi**; $STORE[STACK[OPTOP + 1] + i + 1] :=$ $STACK[OPTOP + 3];$ **end** $PC := PC + 1$	(A,N,O) or (A,N,A)	() ()
iastore	same as **aastore**	(A,N,N)	()

If a local variable is not of type integer, but contains an array or a reference to an object, we use the following translation scheme: $code$ x ρ= **aload** ρ(x) where x is a local variable of array or object type. To access the value of a multidimensional array of integers we generate the following code:

$$
\begin{array}{ll}
code\ \ x[i_1]\ldots[i_k]\ \rho = & \textbf{aload}\ \rho(x); \\
& code\ i_1\ \rho; \\
& \textbf{aaload}; \\
& \vdots \\
& code\ i_{k-1}\ \rho; \\
& \textbf{aaload}; \\
& code\ i_k\ \rho; \\
& \textbf{iaload};
\end{array}
$$

If k is smaller than the dimension of the array stored in x, then we have to use **aaload** instead of **iaload**. We also use **aaload** if x contains an array of objects.

Classes

Before a class can be compiled some static information has to be computed. This information is used to define the functions γ_K, γ_F and γ_M for the class. Here we divide the compilation of a class into three phases:

- computation of an address environment: $elab\_def$

- type inference for expressions and resolution of overloading: *infer_type*
- code generation: *code*

Computation of Address Environment

In the first phase we compute the types and offsets of local variables, object and class fields. For every method we compute its result type, the number of its local variables and its local environment. Every method is identified by its name and its parameter types.

$elab\_def$ (**class** c **extends** s $\{b\}$) $= elab\_def\ b\ \emptyset\ 0\ 0\ 0$

$elab\_def\ c_1;c_2\ \rho\ nav\ nas\ loc =$
let $(\rho', nav', nas', loc') = elab\_def\ c_1\ \rho\ nav\ nas\ loc$
in $elab\_def\ c_2\ \rho'\ nav'\ nas'\ loc'$ **end**

$elab\_def$ (**static** $t\ x{=}init$) $\rho\ nav\ nas\ loc =$
$(\rho[\,(t, nas)/x\,], nav, nas + sizeof(t), loc)$

within a class:
$elab\_def\ (t\ x{=}init)\ \rho\ nav\ nas\ loc = (\rho[\,(t, nav)/x\,], nav + sizeof(t), nas, loc)$

within a method:
$elab\_def\ (t\ x{=}init)\ \rho\ nav\ nas\ loc = (\rho[\,(t, loc)/x\,], nav, nas, loc + sizeof(t))$

$elab\_def$ (**static** $t\ m(t_1\ p_1,\ldots,t_k\ p_k)\ \{b\}$) $\rho\ nav\ nas\ loc =$
let $(\rho', nav', nas', loc') = elab\_def\ b\ \rho[\,(t_i, i-1)/p_i\,]\ 0\ 0\ k$
in $(\rho[\,(t, loc', \rho')/m(t_1,\ldots,t_k)\,], nav, nas, loc)$ **end**

$elab\_def\ (t\ m(t_1\ p_1,\ldots,t_k\ p_k)\ \{b\})\ \rho\ nav\ nas\ loc =$
let $(\rho', nav', nas', loc') = elab\_def\ b\ \rho[\,(t_i, i)/p_i\,]\ 0\ 0\ k+1$
in $(\rho[\,(t, loc', \rho')/m(t_1,\ldots,t_k)\,], nav, nas, loc)$ **end**

For all other cases we have: $elab\_def\ (\_)\ \rho\ nav\ nas\ loc = (\rho, nav, nas, loc)$

For **static** $t\ m(t_1,\ldots,t_k)$ an entry i is added into the constant pool such that $\gamma_K(i) = "c/m(t_1 \ldots t_k)t"$ where c is the current class. In addition the method table is extended by an entry $\gamma_M("c/m(t_1 \ldots t_k)t") = (c, p, n, l, static, ms)$, where p is the address of the byte code for m, n the number of its arguments, l the number of its local variables and ms the number of stack cells maximally needed by this method. The value of ms can be computed by analyzing the byte-code for the method because every byte-code instruction allocates or deallocates a fixed number of stack cells and if there are several paths to a byte-code instruction, the same number of stack cells must have been allocated on every path before reaching this instruction. This analysis is also performed by the byte-code verifier.
For virtual methods similar entries are added to the constant and method table. The only difference is, that in this case we have $\gamma_M("c/m(t_1 \ldots t_k)t") = (c, p, n, l, dynamic, ms)$.
For field declarations **static** $t\ x = e$ an entry i is added into the constant pool such that $\gamma_K(i) = "c/x"$ where c is the current class. In addition the field table is extended by an entry $\gamma_F("c/x") = (t, static, a)$, where a is the address of the field. For object fields $t\ x = e$ we add an entry $\gamma_F("c/x") = (t, dynamic, a)$ where o is the offset of the field.

Scope, Visibility and Inheritance

The scope of a method or field is the class and the scope of a local variable is the method it occurs in. Within a method m in class c we have the following rules of visibility.

- local variables of m < fields of class c < fields of the superclass of c < fields of the superclass of the superclass of $c < \ldots$
- methods of c < methods of the superclass of c < methods of the superclass of the superclass of $c < \ldots$
- in Java: classes in the same package < imported classes

Fields or methods can be made visible by using a prefix notation like `A.y` or `super.m()` in constructors in Java.

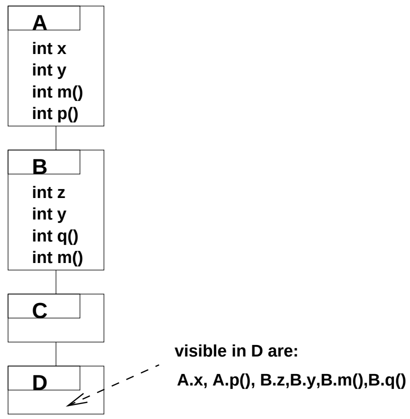

Figure 3: Visibility and Inheritance

In JVM instructions like `invokevirtual`, `putstatic` or `getfield` one uses the signatures of the visible, defining occurrences as an argument. For example if we consider the inheritance hierarchy in Figure 3 and define `C` as

```
class D extends C { int v=C.m(); }
```

Then the above call to m is translated into (in JASMIN syntax):

```
invokestatic B/m()V
```

As a consequence the compiler has to compute the visible, defining occurrence for each using occurrence of a field or method. First we introduce some notation for the inheritance relation on classes:

- $c \downharpoonleft d$ iff c inherits directly from d
- $c \downharpoonleft^0 c$
- $c \downharpoonleft^1 d$ iff $c \downharpoonleft d$
- $c \downharpoonleft^n e$ iff $c \downharpoonleft^{n-1} d$ and $d \downharpoonleft e$
- $c \downharpoonleft^* d$ if $c \downharpoonleft^n d$
- $c \downharpoonleft^+ d$ if $c \downharpoonleft^n d$ and $n > 0$
- $methods(c) = \{m(t_1 \ldots t_k)t \mid$ class c contains a definition of a method with signature $m(t_1 \ldots t_k)t\}$
- $fields(c) = \{(v, t) \mid$ class c contains a definition of a field v of type $t\}$

Overriding of methods is restricted. A method may override an inherited method only if it has the same name and the same number and types of arguments. In this case it must also have the same result type:

- $\forall m(t_1 \ldots t_k)t \in methods(c) : \forall d : c \downarrow^+ d : \forall m(t_1 \ldots t_k)t' \in methods(d) : t = t'$

Note, that there is no similar restriction on fields, i.e. a field needn't have the type of the field it overrides.

With the above notation we can now define the visible, defining occurrences within a class:

- $vis\_meths(c) = \{d/m(t_1 \ldots t_k)t \mid m(t_1 \ldots t_k)t \in methods(d),\ c \downarrow^n d$ and n is minimal $\}$
- $vis\_fields(c) = \{(d/v, t) \mid (v, t) \in fields(d),\ c \downarrow^n d$ and n is minimal $\}$

Type Inference

When generating code for a method call, we have to infer the types of its argument expressions first. Then we can use the method name and these types to identify the method and look up its result type in the environment. Type inference computes the type of every expression in the program. It proceeds bottom-up, i.e. the types of variables and constants are known.

$infer\_type\ c\ \rho = t$	where c is a constant of type t
$infer\_type\ x\ \rho = t$	where x is a local variable and $(t, a) = \rho(x)$

If we know the types of its subexpressions we can compute the type of a compound expression.

$infer\_type\ (e_1\ op\ e_2)\ \rho = t_1 \tilde{op} t_2$	where $t_1 = infer\_type\ e_1\ \rho$, $t_2 = infer\_type\ e_2\ \rho$ and $\tilde{op}$ is the abstract operation on types induced by op.
$infer\_type\ (e[e_1] \ldots [e_k])\ \rho = t'$	where e yields an array and $(m, t) = infer\_type\ e\ \rho$, and $t' = \begin{cases} t & \text{if } m = k \\ (m - k, t) & \text{if } m > k \end{cases}$

For fields we need to know, in what class to look for its type. The field must not be defined in that class, but it must be visible. For method calls we first have to infer the types of their arguments, then we can look up the result type.

$infer\_type\ o.x\ \rho = t$	where x is field, $infer\_type\ o\ \rho = t_0$ and $(d/x, t) \in vis\_fields(t_0)$
$infer\_type\ x\ \rho = t$	as above, but $t_0 = c$ where c is the current class
$infer\_type\ c.x\ \rho = t$	as above, but $t_0 = c$
$infer\_type\ o.m(e_1, \ldots, e_k)\ \rho = t$	where $t_0 = infer\_type\ o\ \rho$, $t_1 = infer\_type\ e_1\ \rho$, ..., $t_k = infer\_type\ e_k\ \rho$ and $d/m(t_1 \ldots t_k)t \in vis\_meths(t_0)$
$infer\_type\ m(e_1, \ldots, e_k)\ \rho = t$	as above, but $t_0 = c$ where c is the current class
$infer\_type\ c.m(e_1, \ldots, e_k)\ \rho = t$	as above, but $t_0 = c$

Using the result of type inference we can extend our translation scheme for multidimensional arrays:

$$
\begin{array}{ll}
code\ \ e[i_1] \ldots [i_k]\ \rho = & code\ e\ \rho; \\
& code\ i_1\ \rho; \\
& \mathbf{aaload}; \\
& \vdots \\
& code\ i_{k-1}\ \rho; \\
& \mathbf{aaload}; \\
& code\ i_k\ \rho; \\
& \mathbf{iaload};
\end{array}
$$

Where $infer\_type(e) = (m, t)$, $t = integer$ and $m = k$. If $m > k$ we use **aaload** instead of **iaload**. We also use **aaload** if t is a classname and $m = k$, i.e. e yields an an array of objects.
So far we didn't consider casting, but assumed that the types of the arguments in a method call exactly match those of a defined method. In Java primitive datatypes and objects can be explicitly casted into other datatypes or objects:

```
double pi=3.14;
int    x = (int) pi;

class A { }
class B extends A
 { int x=3;
   int m() { A obj=new B();
             int z=obj.x;       <------ error
             int z=((B)obj).x;  <------ ok
           }
 }
```

In the above example we applied a narrowing conversion. Java performs implicit casting, when the actual arguments of a method call do not exactly match those of parameters of the defined method. In this case a widening conversion is applied to the actual parameters. Let d be a subclass of c, then

widening: an instance of d can be used where an instance of c is expected. An explicit cast is not necessary.

narrowing: an instance of c can be used as an instance of d if it is explicitly casted.

Method Overloading In TassKaf methods can be overloaded. It is legal for a class to have several methods with the same method name but different argument types and result types. The only restriction is, that there are not two methods with the same name and the same argument types but different result types (In Ada it is legal that two methods only differ in their result types, as a consequence overloading resolution is slightly more difficult (an additional top-down pass in the type inference). In fact overloading resolution in Java is more difficult than that. If there is no method with exactly those argument types inferred, but the arguments in the method call can be converted into the types of one or more existing methods, then one of these methods has to be chosen.
In a preliminary version of the "Java 1.0 Language Specification" this choice was based on conversion costs. This was a very problematic approach to overloading resolution, because it is very difficult to handle by the Java programmer. In the final version of the "Java 1.0 Language Specification"[6] this approach was replaced by one based on maximally specific methods.

Although maximally specific methods are easier to identify by the programmer, many Java programmers use casts, i.e. explicit type conversion, to make sure that the right method is called. For TassKaf we decided to use a simpler form of overloading, i.e. we only allowed method calls, for which the inferred argument types exactly match those of a defined method, see the above definition of $\mathit{infer\_type}\ o.m(e_1, \ldots, e_k)\ \rho$.

Overloading Resolution in Java

If the parameter types of the defined method do not exactly match those of the arguments in a call, we have to compare these types. Thus we define the relation $\sqsubseteq$ (called "more specific") as follows:

- $t \sqsubseteq t'$ iff t, t' are classes and $t \downarrow^* t'$
- $t \sqsubseteq t'$ iff t, t' are primitive types and entities of type t can be converted into entities of type t' without data loss.
- $m(t_1 \ldots t_k) \sqsubseteq m(t'_1 \ldots t'_k)$ iff $t_i \sqsubseteq t'_i$ $(1 \leq i \leq k)$

Now we define the set of applicable methods for a given class and method:

- $\mathit{applicable}(c/m(t_1 \ldots t_k)) = \{c'/m(t'_1 \ldots t'_k)t' \in \mathit{vis\_meths}(c) \mid m(t_1 \ldots t_k) \sqsubseteq m(t'_1 \ldots t'_k)\}$

Given a set of applicable methods A we can now define the most specific method

- $most\_specific(A) = c/m(t_1 \ldots t_k)t \in A$ such that $\forall c'/m(t'_1 \ldots t'_k)t' \in A : m(t_1 \ldots t_k) \sqsubseteq m(t'_1 \ldots t'_k)$

With these definitions the type of a method call in Java is computed as:
$infer\_type\ o.m(e_1, \ldots, e_k)\ \rho = t'$ where $t_0 = infer\_type\ o\ \rho$, $t_1 = infer\_type\ e_1\ \rho$, $\ldots$, $t_k = infer\_type\ e_k\ \rho$ and $d/m(t'_1 \ldots t'_k)t' = most\_specific(applicable(t_0/m(t_1 \ldots t_k)))$

If $applicable(t_0, m(t_1 \ldots t_k)))$ is not empty but there exists no most specific method, we have an ambiguous method call. The following example shows such a case:

```
class A {}
class B extends A {}
class C extends B
 { void m(A x, C y) { }
   void m(B x, A y)
       { m(new B(), new C());
       }
 }
```

For the arguments of the method call we infer types B and C and thus get $applicable(C, m(B, C)) = \{m(A, C), m(B, A)\}$ because $m(B, C) \sqsubseteq m(A, C)$ and $m(B, C) \sqsubseteq m(B, A)$. Unfortunately neither is $m(A, C) \not\sqsubseteq m(B, A)$ nor $m(B, A) \not\sqsubseteq m(A, C)$ and thus the call is ambiguous[3].

Methods

Every method is uniquely determined by its signature. The signature consists of the name of the class c where the method was defined, the name m of the method, the types t_i of its parameters (and as a result implicitly the number n of its parameters) and the result type t. We will write the signature as $c/m(t_1, \ldots, t_n)t$. This signature is stored as a string in the constant table. Using the signature we find further information on the method in the method table.

Method Declaration

$code\ \ (\textbf{static}\ t\ m(t_1\ p_1, \ldots, t_k\ p_k); body)\ \rho\ = code\ body\ \rho;$

$code\ \ (t\ m(t_1\ p_1, \ldots, t_k\ p_k); body)\ \rho\ = code\ body\ \rho;$

Local Variables

Declarations of local variables are translated as follows:

$code\ \ (t\ x{=}init)\ \rho\ = code\ init\ \rho;$ **istore** ρ(x) where $t \in \{int, bool\}$.

$code\ \ (t\ x{=}init)\ \rho\ = code\ init\ \rho;$ **astore** ρ(x) where t is a reference type.

[3] Using the resolution based on costs the first method would be chosen, because conversion from B to A is cheaper than conversion from C to A.

Method Return

Instruction	Meaning	Before	After
return	$PC := STACK[FRAME];$ $OPTOP := VARS - 1;$ $VARS := STACK[FRAME + 1];$ $FRAME := STACK[FRAME + 2];$ $CC := STACK[FRAME + 3]$	()	...
ireturn	$PC := STACK[FRAME];$ $STACK[VARS] := STACK[OPTOP];$ $OPTOP := VARS;$ $VARS := STACK[FRAME + 1];$ $FRAME := STACK[FRAME + 2];$ $CC := STACK[FRAME + 3]$	(N)	...
areturn	$PC := STACK[FRAME];$ $STACK[VARS] := STACK[OPTOP];$ $OPTOP := VARS;$ $VARS := STACK[FRAME + 1];$ $FRAME := STACK[FRAME + 2];$ $CC := STACK[FRAME + 3]$	(O)	...

At the end of every method a **return** statement must be executed. If the programmer forgot to put such a statement at the end of the method, then the compiler has to insert one automatically.

$code$ **return** ρ = **return**
$code$ (**return** e) ρ = $code\ e\ \rho$; **ireturn** if $infer\_type(e) \in \{int, bool\}$
$code$ (**return** e) ρ = $code\ e\ \rho$; **areturn** if $infer\_type(e)$ yields a reference type

Method Call

Instruction	Meaning	Before	After
invokestatic i	**let** $(c, p, n, l, static, ms) = \gamma_M^{CC}(\gamma_K^{CC}(i))$ **in** $STACK[OPTOP + 1 + l] := PC + 3;$ $STACK[OPTOP + 2 + l] := VARS;$ $STACK[OPTOP + 3 + l] := FRAME;$ $STACK[OPTOP + 4 + l] := CC;$ $VARS := OPTOP - n;$ $FRAME := OPTOP + 1 + l;$ $OPTOP := OPTOP + 4 + l;$ $CC := c;$ $PC := p$ **end**	(V,...,V) n times	...

In TassKaf objects are always reference-parameters and parameters of scalar type are always passed by value.

For static methods we only have to pass the arguments to the method:

$$\begin{aligned} code\ \ c.m(e_1,\ldots,e_n)\ \rho \ =\ \ & code\ \ e_1\ \rho\ ; \\ & \vdots \\ & code\ \ e_n\ \rho\ ; \\ & \textbf{invokestatic}\ i \end{aligned}$$

where c is a class name, $\gamma_K(i) = "c'/m(t_1,\ldots,t_k)t"$, $t_i = infer\_type(e_i)$ and c' is the first superclass of c (or c itself) where the method is defined, i.e. $c'/m(t_1,\ldots,t_k)t \in vis\_meths(c)$. If c is not explicitly given we use the name of the current class instead.

Instruction	Meaning	Before	After
invokevirtual i	**let** $(c, p, n, l, dynamic, ms) = \gamma_M(\gamma_K(i))$ **in** $STACK[OPTOP + 1 + l] := PC + 3;$ $STACK[OPTOP + 2 + l] := VARS;$ $STACK[OPTOP + 3 + l] := FRAME;$ $STACK[OPTOP + 4 + l] := CC;$ $VARS := OPTOP - n;$ $FRAME := OPTOP + 1 + l;$ $OPTOP := OPTOP + 4 + l;$ $CC := c$ $PC := p$ **end**	(V,...,V) n times	...

For virtual methods we have to pass a reference of the object:

$$
\begin{array}{lll}
code\ \ e.m(e_1,\ldots, e_n)\ \rho & = & code\ \ e\ \rho\ ; \\
 & & code\ \ e_1\ \rho\ ; \\
 & & \vdots \\
 & & code\ \ e_n\ \rho\ ; \\
 & & \textbf{invokevirtual}\ i
\end{array}
$$

where c is the type inferred for the expression e, $\gamma_K(i) = "c'/m(t_1,\ldots,t_k)t"$, $t_i = infer\_type(e_i)$ and c' is the first superclass of c (or c itself) where the method is defined, i.e. $c'/m(t_1,\ldots,t_k)t \in vis\_meths(c)$. If there is no expression e, then we generate **aload** 0 to get a reference to the current object and let c be the name of the current class.

Method Calls in Java

In Java arguments in a method call can be casted into the types of an applicable method.
For static methods we only have to pass the arguments to the method:

$$
\begin{array}{lll}
code\ \ c.m(e_1,\ldots, e_n)\ \rho & = & code\ \ e_1\ \rho\ ; \\
 & & \vdots \\
 & & code\ \ e_n\ \rho\ ; \\
 & & \textbf{invokestatic}\ i
\end{array}
$$

where c is a class name, $\gamma_K(i) = "c'/m(t_1,\ldots,t_k)t"$
and $c'/m(t_1,\ldots,t_k)t \in most\_specific(applicable(c, m(t_1 \ldots t_k)))$. If c is not explicitly given we use the name of the current class instead. For virtual methods the signature of the method to be called is computed in an analogous way.

Objects

Initialization of Fields

The initialization of class and object fields is done by two special methods $\langle$`clinit`$\rangle$ and $\langle$`init`$\rangle$. The body of the method $\langle$`clinit`$\rangle$ contains for each declaration **static** $t\ x = e$ an assignment of the form $x = e$. The body of the method $\langle$`init`$\rangle$ contains for each declaration $t\ x = e$ an assignment of the form $x = e$. The method $\langle$`clinit`$\rangle$ is called once when the class is loaded to initialize its static fields. The method $\langle$`init`$\rangle$ is called when an object is created and initializes its object fields.

this

In each execution frame of a virtual method a reference to the object of the method is stored in the first frame element ($STACK[VARS]$).
$code$ **this** ρ = **aload** 0

Creation of Objects

Instruction	Meaning	Before	After
new i	$STACK[OPTOP] := newObject(\gamma_K^{CC}(i))$; $PC := PC + 3$	()	(O)

Figure 4: Definition of $newObject$

$newObject(c)$=
let $size = \sum_{\substack{i \in dom(\gamma_F^c) \\ (t, \text{dynamic}, a) = \gamma_F^c(i)}} sizeof(t)$
$obj = newMem(1 + size)$
in
$STORE[obj] := \gamma_M^c$
for every $i \in dom(\gamma_F^c)$ **do**
let $(t, sd, a) = \gamma_F^c(i)$ **in**
if $sd = dynamic$ **then** $STORE[obj + 1 + a] := default(t)$;
end
od
return obj;
end

The auxiliary function $newObject$ is defined in Figure 4. The function $newMem(n)$ returns the address of the first cell of a free area of size n in the heap. After an instance of an object is created with the **new** instruction, it has to be initialized by calling the $\langle$init$\rangle$ method of the class:
$code$ **new** $c()$ ρ = **new** i; **dup**;**invokespecial** j where $\gamma_K(i) = c$ and $\gamma_K(j) = c/\langle\texttt{init}\rangle\ ()V$.

Assignments

Instruction	Meaning	Before	After
getstatic i	**let** $"c/f = \gamma_K^{CC}(i),$ $\quad (t, static, a) = \gamma_F^c("c/f")$ **in** $\quad OPTOP := OPTOP + 1;$ $\quad STACK[OPTOP] := STORE[a];$ $\quad PC := PC + 3$ **end**	()	(V)
putstatic i	**let** $"c/f" = \gamma_K^c cf(i),$ $\quad (t, static, a) = \gamma_F^c("c/f")$ **in** $\quad STORE[a] := STACK[OPTOP];$ $\quad OPTOP := OPTOP - 1;$ $\quad PC := PC + 3$ **end**	(V)	()
getfield i	**let** $"c/f" = \gamma_K^{CC}(i),$ $\quad (t, dynamic, a) = \gamma_F^c("c/f")$ **in** $\quad STACK[OPTOP] := STORE[STACK[OPTOP] + a];$ $\quad PC := PC + 3$ **end**	(O)	(V)
putfield i	**let** $c/f" = \gamma_K^{CC}(i),$ $\quad (t, dynamic, a) = \gamma_F^c("c/f")$ **in** $\quad STORE[STACK[OPTOP - 1] + a] := STACK[OPTOP];$ $\quad OPTOP := OPTOP - 2;$ $\quad PC := PC + 3$ **end**	(O,V)	()

In TassKaf we can assign values to local variables, object fields and class fields. For object and class fields the expression on the left of the assignment operator can be a qualified name, which contains a reference to the object or the name of a class.

$code(c.x = e)\ \rho$=$code\ e\ \rho$; **putstatic** i;
where x is a class field, $\gamma_K(i) = "c'/x"$ and c' is the first superclass of c (or c itself) where the field x is defined,i.e., $(c'/x, t) \in vis\_fields(c)$.

$code(e_1.x = e_2)\ \rho$=$code\ e_1\ \rho; code\ e_2\ \rho$; **putfield** i;
where x is an object field, $Type(e_1) = c$, $\gamma_K(i) = "c'/x"$ and $(c'/x, t) \in vis\_fields(c)$.

$code(x = e)\ \rho$=$code\ e\ \rho$; **istore** $\rho(x)$; where x is a local variable and $Type(e_1) \in \{int, bool\}$.

$code(x = e)\ \rho$=$code\ e\ \rho$; **astore** $\rho(x)$;
where x is a local variable and $Type(e_1)$ is a reference type (array or object).

In assignments to object fields of the current object, it is legal to omit the reference to the object:

$code(x = e)\ \rho$=**aload** 0;$code\ e\ \rho$; **putfield** i;
where x is an object field of the current class c, $\gamma_K(i) = "c'/x"$ and and $(c'/x, t) \in vis\_fields(c)$.

$code(x = e)\ \rho$=$code\ e\ \rho$; **putstatic** i;
where x is a class field of the current class c, $\gamma_K(i) = "c'/x"$ and $(c'/x, t) \in vis\_fields(c)$.

The translation scheme for assignments to arrays is a little bit more complicated. First we look at the translation scheme for an assignment to a static fields containing an array:

$$
\begin{aligned}
code(e_1.x[d_1]\ldots[d_k] = e_2)\ \rho = \quad & code\ e_1\ \rho;\ \mathbf{getstatic}\ i;\\
& code\ d_1\ \rho;\ \mathbf{aaload};\\
& \vdots\\
& code\ d_{k-1}\ \rho;\ \mathbf{aaload};\\
& code\ d_k\ \rho;\ code\ e_2\ \rho;\ \mathbf{aastore};
\end{aligned}
$$

where $Type(e_1) = c$ and $\gamma_K(i) = "c'/x"$ and c' is the first superclass of c (or c itself) where the field x is defined, i.e. $(c'/x, t) \in vis\_fields(c)$.

For object fields we use **getfield** and for local variables **aload** $\rho(x)$ instead of **getstatic**. If the type of the expression e_2 is *int* or *bool* we use **iastore** instead of **aastore**. Finally if the assignment is to an object field of the current object (i.e. there is no e_1), we use **aload** 0 instead of the code for e_1.

Optimizations at Runtime instead of at Compiletime

Instruction	Meaning	Before	After
invokestatic i	**let** $(c, p, n, l, static, ms) = \gamma_M^{CC}\ (\gamma_K^{CC}\ (i))$ **in** $STACK[OPTOP + 1 + l] := PC + 3;$ $STACK[OPTOP + 2 + l] := VARS;$ $STACK[OPTOP + 3 + l] := FRAME;$ $STACK[OPTOP + 4 + l] := CC;$ $VARS := OPTOP - n;$ $FRAME := OPTOP + 1 + l;$ $OPTOP := OPTOP + 4 + l;$ $\longrightarrow$ $\gamma_P^{CC}\ (PC) :=$ **invokestatic_quick** $(c, p, n, l, static, ms)$; $CC := c;$ $PC := p$ **end**	(V,...,V) n times	...
invokestatic_quick π	**let** $(c, p, n, l, static, ms) = \pi$ **in** $STACK[OPTOP + 1 + l] := PC + 3;$ $STACK[OPTOP + 2 + l] := VARS;$ $STACK[OPTOP + 3 + l] := FRAME;$ $STACK[OPTOP + 4 + l] := CC;$ $VARS := OPTOP - n;$ $FRAME := OPTOP + 1 + l;$ $OPTOP := OPTOP + 4 + l;$ $CC := c;$ $PC := p$ **end**	(V,...,V) n times	...

The JVM can optimize the byte code at runtime. At first it might seem awkward to wait until runtime, but the security mechanisms for code loaded over the network induce significant overhead. So far, we did not describe the security checks involved with some of the machine instructions. Consider the execution of the instruction **invokestatic** i with $\gamma_K^{CC}\ (i)$=$c/m(t_1 \ldots t_k)t"$. The JVM first checks, whether the class c has been loaded before, if not the class is loaded. Then it looks up the pointer value π associated with m in the method table ($\pi = \gamma_M^{CC}\ (\gamma_K^{CC}\ (i))$). Finally the JVM tests whether the calling object is allowed to access the method and if this is the case, it executes the byte code of the method. Now assume that the instruction is contained in a loop and executed again. All these tests would have to be done again. But after the first execution of the method we know that the class is loaded, we know the value π and that the object is allowed to call the method. To enforce, that these tests are not done again, the JVM replaces **invokestatic** i" after its first execution by **invokestatic_quick** π, which does less tests. In the JVM there are **quick** variants of many other byte code instructions, e.g **invokevirtual**, **putfield** and **anewarray**

Some other Java Concepts

Exceptions

In the class file contains an exception table for every method. The table contains entries of the form (s, e, h, c) where s,e and h are addresses in the program code. h is the address of an exception handler and c the name of a class, which must be a subclass of `Throwable`.
If an exception, which is an instance of of class c' is thrown by the JVM instruction **athrow** at byte-code position PC, then the handler h of the first entry (s, e, h, c) in the exception table is executed for which we have that $PC \in [s, e[$ and $c \downarrow^* c'$.
If there is no matching entry in the table of the current method, the frame of the method is popped from the stack and the exception is rethrown in the context of the calling method.

$\mathit{code}\ (\mathbf{throw}\ e)\ \rho = \mathit{code}\ e\ \rho; \mathbf{athrow}$ where e yields an object of a subclass of `Throwable`.

Now we look at how the **try** statement is translated. Within the code of the current method the compiler produces code for the body of the statement:
$\mathit{code}\ (\mathbf{try}\ \{\ \mathit{body}\ \}\ \mathbf{catch}\ (c\ e)\ \{\ \mathit{handler}\ \})\ \rho = l_s : \mathit{code}\ \mathit{body}\ \rho; l_e :$

Separately it produces $l_h : \mathit{code}\ \mathit{handler}\ \rho; \mathbf{goto}\ l_e$ and appends it at the end of the code produced for the current method. In addition it stores an entry (l_s, l_e, l_h, c) in the exception table of the method.

Threads

The JVM only has two special instructions for threads: **monitorenter** and **monitorexit**. All other thread-specific functionality is provided by the Java-API (in `java.lang.Thread`). **monitorenter** obtains an exclusive lock on the object referenced on top of the stack, whereas **monitorenter** releases such a lock.
For methods, which are declared to be synchronized, the instruction **invokevirtual** internally performs an equivalent to **aload_0**;**monitorenter** and **monitorexit** is called implicitly when the method returns.
For the **synchronized** statement the compiler has to produce code which makes use of the above instructions:

$$\mathit{code}\ (\mathbf{synchronized}\ (e)\ \{\ \mathit{body}\ \})\ \rho = \mathit{code}\ e\ \rho; \mathbf{dup}; \mathbf{monitorenter}; \mathit{code}\ \mathit{body}\ \rho; \mathbf{monitorexit}$$

Native Methods

Native methods are methods which are not implemented in Java, but in a platform-dependent language like C, C++ or assembler. The Java Native Interface handles communication between these languages, e.g. conversion of types, mapping of names and access to the garbage collector. It is possible for the native code to call Java methods and vice versa.

Explicit Casts

For primitive types there are special instructions, which convert values of one type into values of another type, e.g. **i2l** converts the integer value on top of the stack into a long value. Assume, that the function $conv(t_1, t_2)$ yields the name of an JVM instruction converting values of type t_1 into values of type t_2, e.g. $conv(integer, long) = \mathbf{i2l}$, $conv(long, integer) = \mathbf{l2i}$, $conv(integer, float) = \mathbf{i2f}$, $conv(float, integer) = \mathbf{f2i}$, etc. Then casts of primitive types are translated as follows:

$code\ ((t)\ e)\ \rho = code\ e\ \rho; conv(t', t)$ where $t' = infer\_type\ e\ \rho$.

For reference types there are no conversion instructions, but an instruction which tests at runtime, whether an object is a subclass of a given class. The instruction **checkcast** i where $\gamma_K^{CC}(i) = c$ throws an `ClassCastException`, if the topmost element on the stack is NULL or a reference to an object, which is not a subclass of c. Otherwise the next instruction is executed. Thus for casts of reference types we have the following translation scheme:

$code\ ((c)\ e)\ \rho = code\ e\ \rho; \mathbf{checkcast}\ i$ where $\gamma_K(i) = c$.

Implementing a Compiler with JASMIN

Implementation of a compiler for TassKaf is a much simpler task, when we produce a textual representation which can be converted by the JASMIN assembler into a file in Java-Class-File format. JASMIN takes care of the generation of the constant pool, so that one can write number and string constants, method signatures and types as strings in JASMIN files. Furthermore the mapping of labels to addresses in the byte-code is done by JASMIN, so that one can use strings as labels in jumps, e.g., `goto MyLabel`.

Naming Conventions

The notation for signatures in JASMIN and in Java-Class files differs slightly from the one presented above. For the type `integer` the abbreviation `I` is used, for the type `void` the abbreviation `V`. Array types have a `[` as a prefix and class types a `L` as a prefix and a `;` as a suffix. Due to these conventions signatures can be written without a comma:

```
LTest;                      % type of an object: new Test
Test/foo(I)V                % void foo(int x)
Test/foo(I)I                % int foo(int x)
Test/foo(II)I               % int foo(int x, int y)
Test/foo(LTest;I)LTest;     % Test foo(Test o, int x)
Test/foo(LTest;LTest;)I;    % int foo(Test o1, Test o2)
[I                          % one-dimensional array of integers: new int [ ]
[[I                         % two-dimensional array of integers: new int [ ] [ ]
[LTest;                     % one-dimensional array of objects: new Test [ ]
Test/foo([I[[II)[LTest;     % Test[] foo(int x[], int y[][])
```

We define a function ξ, which given a type t returns the JASMIN equivalent of t:

$$\xi(t) = \begin{cases} \texttt{I} & t = int \\ \texttt{V} & t = void \\ \texttt{B} & t = bool \\ \texttt{L}t\texttt{;} & t \text{ is a classname} \\ \texttt{[}\xi((n-1,t')) & t = (n,t) \end{cases}$$

JASMIN Syntax

In the rest of this section we list those cases where the syntax of code, which has to be generated for JASMIN, diverges from the one presented previously in this paper.

Expressions

Using JASMIN we don't have to use indices to the constant pool in our code, but instead we write:

$code$ c ρ= **ldc** c where c is a constant of array or object type.
$code$ x ρ= **getfield** $\xi(x)$ $\xi(t)$ where x is object variable of type t.
$code$ x ρ= **getstatic** $\xi(x)$ $\xi(t)$ where x is class variable of type t.

Classes

In JASMIN there are special directives (like .class or .field) to declare information which has to be entered into the appropriate parts of the class file. Using JASMIN as a target language declarations of classes are translated as follows:

$code$ (**class** c **extends** s $body$) ρ = .class public c
.super s
$code_{Fields}$ $body$ ρ
.method public <init>()V
  aload_0
  invokespecial s/ <init>()V
  $code_{InitObjectFields}$ $body$ ρ
  return
.end method
.method public <clinit>()V
  $code_{InitClassFields}$ $body$ ρ
  return
.end method
$code$ $body$ ρ;

In the above translation scheme we have three auxiliary functions which traverse the $body$ of the class declaration and process field definitions (i.e. class or object fields) in a special way:

$code_{Fields}$ $(t\ x = e)$ ρ = .field public x $\xi(t)$
$code_{Fields}$ $(static\ t\ x = e)$ ρ = .field public static x $\xi(t)$
$code_{InitObjectFields}$ $(t\ x = e)$ ρ = $code$ $(x = e)$ ρ
$code_{InitClassFields}$ $(static\ t\ x = e)$ ρ = $code$ $(x = e)$ ρ

Method Declarations

Declarations of methods are translated as follows:

$$code\ (\textbf{static}\ t\ m(t_1\ p_1, \ldots, t_k\ p_k); body)\ \rho \ = \ \begin{array}{l}\text{.method public static } m(\xi(t_1)...\xi(t_k))\xi(t)\\ \quad code\ body\ \rho;\\ \text{.end method}\end{array}$$

$$code\ \ (t\ m(t_1\ p_1, \ldots, t_k\ p_k); body)\ \rho \ = \ \begin{array}{l}\text{.method public } m(\xi(t_1)...\xi(t_k))\xi(t)\\ \quad code\ body\ \rho;\\ \text{.end method}\end{array}$$

Method Call

Instead of **invokestatic** i where $\gamma_K(i) = c/m(t_1 \ldots t_k)t$ the JASMIN syntax allows to write **invokestatic** $c/m(\xi(t_1)\ldots\xi(t_k))\xi(t)$. The same holds for **invokevirtual**.

Creation of Objects

An instance of an object is created with the **new** instruction and has to be initialized by calling the $\langle$`init`$\rangle$ method of the class:
code **new** $c()$ ρ = **new** c; **dup**; **invokespecial** $c/\langle$`init`$\rangle$ $()V$

Assignments

In the code produced for assignments to object fields and class fields we can now use their names directly:
$code(c.x = e)\ \rho{=}code\ e\ \rho$; **putstatic** $c/x\ \xi(t)$; where x is a class fields and t its type.
$code(e_1.x = e_2)\ \rho{=}code\ e_1\ \rho; code\ e_2\ \rho$; **putfield** $c/x\ \xi(t)$;
where x is an object fields and $Type(e_1) = c$ and t the type of x.

Also translation of assignments to arrays becomes more simple:

$$code(e_1.x[d_1]\ldots[d_k] = e_2)\ \rho{=} \ \begin{array}{l} code\ e_1\ \rho;\ \textbf{getstatic}\ c_1/x\ \xi(t);\\ code\ d_1\ \rho;\ \textbf{aaload};\\ \vdots\\ code\ d_{k-1}\ \rho;\ \textbf{aaload};\\ code\ d_k\ \rho;\ code\ e_2\ \rho;\ \textbf{aastore};\end{array}$$

where $Type(e_1) = c_1$ and $Type(e_2)$ is a reference type and t is the type of x.

For object fields we use **getfield** and for local variables **aload** instead of **getstatic**. If the type of the expression e_2 is *int* or *bool* we use **iastore** instead of **aastore**. Finally if the assignment is to an object field of the current object (i.e. there is no e_1), we use **aload** 0 instead of **getstatic**.

Implementing TassKaf as a Project

At the University of Saarland we use the book[3,7] to teach compiler design. Parallel to the lecture students have to implement a compiler for a considerable subset of Java. In this project they learn standard techniques to generate modules for lexical and syntactical analyses and to implement other compiler phases by hand. Moreover they also learn implementation details of a modern object-oriented language: static and dynamic binding, method lookup, type inference and resolution of overloading. This is in contrast to our course "Introduction to the WWW" based on[8] where we teach Java from a programmers point of view. Students report that understanding Java compilation helps a lot to write Java programs.

Parts of the Project

The project consists of three parts. A reader familiar with compiler design might be surprised that the students have to implement code generation first. But it turned out that covering code generation first helps the teacher to motivate the other compiler phases. The student knows right at the beginning what information about a program he will need later to generate code.

Part One: We provide a frontend which given a Java program produces an intermediate representation, the decorated abstract syntax tree. The students have to implement a code generator which traverses the tree and produces programs for the Java-Virtual Machine (JVM). In this part of the project the students learn what the intermediate representation looks like, how code generation works and how the virtual machine operates. The task in the other parts of the project will be to write the frontend.

Part Two: In this part the students have to write a parser which reads a Java program and produces an abstract syntax tree. To get a running parser they first have to write a lexer which reads the program character by character and produces the tokens of the language. In this part of the project the students learn to use well-established tools like `flex` and `bison`. These tools have been used to implement a variety of software systems other than compilers.

Part Three: In the last part the students have to implement a semantic analyzer for Java. In part two they realized that their parser accepted many ill-formed programs. Now they have to implement a type inference for Java to detect errors and compute information which is stored in the decorated abstract syntax tree needed by the code generator written in part one. In this part of the project the students learn how structural induction helps to formally describe the static semantics of a language and at the same time serves as a basis to implement a static analyzer. The students also become aware of the delicate difference of static binding (used in Java for fields) and dynamic binding (used in Java for methods) and the advantages and disadvantages of overloading.

Experience with the Project

To enable our students to implement all this in just one term, we have to provide definitions of the Java-Virtual-Machine, the syntax of Java, its static semantics and define how code generation works. For our purposes Sun's official specifications of Java (830 pages) and the Java-Virtual-Machine (470 pages) are too informal and do not provide any details on the compilation of Java. So we wrote our own specifications which became part of this document. Using the JVM as a target machine and providing concise documentation writing a compiler for a considerable subset of Java can be a valuable project. It helps students both to gain deeper insights and to develop practical skills. In our course 30 independent teams of up to two students successfully wrote runnable compilers.

Acknowledgement

An early draft of this document was used in a student project for a compiler design course at the University of Saarland at Saarbrücken. Independently over 30 different, runnable TassKaf compilers have been implemented by the students of this course. Their feedback helped to improve this document.

References

[1] Jon Meyer and Troy Downing. *Java Virtual Machine*. O'Reilly, 1997.

[2] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[3] Reinhard Wilhelm and Dieter Maurer. *Compiler Design: Theory, Construction, Generation*. Addison-Wesley, 2nd printing edition, 1996.

[4] Richard V. Dragan and Larry Seltzer. Java speed trials. *PC Magazine*, 15(18), October 1996. http://www8.zdnet.com/pcmag/features/pctech/1518/java05.htm.

[5] Jonathan Hardwick. Java microbenchmarks. http://www.cs.cmu.edu/ jch/java/benchmarks.html, 1997.

[6] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[7] Reinhard Wilhelm and Dieter Maurer. *Übersetzerbau – Theorie, Konstruktion, Generierung*. Springer Verlag, Berlin, Heidelberg, 2te auflage edition, 1997.

[8] Stephan Diehl. *Java & Co.* Addison-Wesley, Bonn, 1997.