An Introduction to the Compilation of Java
Part One: the Byte-Code Machine

Stephan Diehl, Universität des Saarlandes

April 24, 1997

Contents

1 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.

2 Execution of Java programs

Although developed for other purposes (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 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.

Phase3: A method is called, 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. After verifiying the byte-code the JIT translates it into native code of the local machine.

3 Components of the JVM

3.1 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:

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

Class Loader: $\gamma : \textbf{String} \rightarrow \Gamma_K \times \Gamma_F \times \Gamma_M \times \Gamma_P \times N \times N$

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.

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 (p, n, l, sd) where p is a pointer to the byte-code for this method, n is the number of arguments of the method and l the number of local variables. 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.

3.2 Registers

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

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

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

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

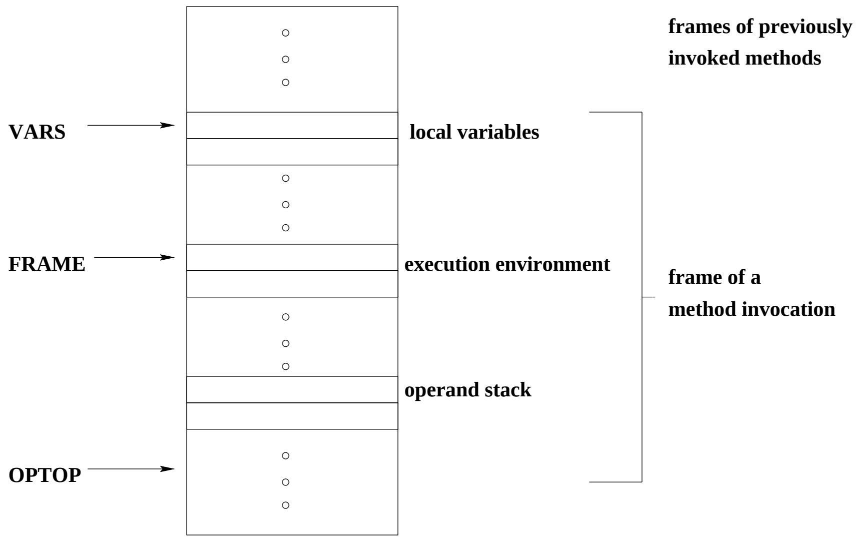

Figure 1: Registers and Stack

3.3 Store

The store is a sequence of cells of size 4 bytes:

$$STORE : N \rightarrow Byte \times Byte \times Byte \times Byte$$

Stack and heap are disjoint areas of the store, i.e.

$$dom(STACK) \cap dom(HEAP) = \emptyset$$

Stack: $STACK : N \rightarrow Byte \times Byte \times Byte \times Byte$
The stack is a sequence of cells of size 4 bytes.

Heap: $HEAP : N \rightarrow Byte \times Byte \times Byte \times Byte$
The heap contains cells of size 4 bytes.

3.4 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.

4 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, arrays. The semantics of the constructs in TassKaf is the same as that of the related constructs in Java. Java constructs which we do not address here include casts, user defined constructors, interfaces and threads. Here is an example program in TassKaf

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

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

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

      e.i = e.i - 1;             // accessing object variables
      C.si = C.si - 1;           // accessing class variables

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

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

   Example m1(Example e) { return e; }
```

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>(Identifier (.)* 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</td></tr>
<tr><td>expression</td><td>→</td><td>expression op expression
| (expression)
| - expression
| (callee) varname ([expression])*
| constant
| methcall
| new</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>(callee) varname = expression ;
| if (expression) { statements } else { statements }
| if (expression) { statements }
| while (expression) { statements }
| do statements while (expression) ;
| for (statement ; expression ; statement) { statements }
| switch (expression)
{ (constant : statements)* (default : statements) }
| break ;
| methcall ;</td></tr>
<tr><td>op</td><td>→</td><td>+|-|*|/|==|!=|≤|...</td></tr>
</table>

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

 void main(java.lang.String argv [])   // main method of program !!!
  { Example obj=new Example();
    Example.m2(this);
    a=new int[20];                     // access field of current object
    obj.m1(10,obj,a);
    obj.m1(obj).i=1;
  }
}
```

5 Compiling TassKaf to Java Byte Code

5.1 Expressions

To translate integer expressions and simple boolean expressions, we need JVM instructions for integer arithmetic, stack manipulation and conditional and unconditional jumps. We represent `true` by 1 and `false` by 0.

5.1.1 Integer Arithmetics

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

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

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

5.1.2 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	$OPTOP:=OPTOP+1$; $STACK[OPTOP]:=\gamma_K(i)$; $PC:=PC+2$	()	(N)
istore v	$STACK[VARS+v]:=STACK[OPTOP]$; $OPTOP:=OPTOP\text{-}1$; $PC:=PC+2$	(N)	()
iload v	$OPTOP:=OPTOP+1$; $STACK[OPTOP]:=STACK[VARS+v]$; $PC:=PC+2$	(N)	()

5.1.3 Jumps

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

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

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

5.2 Arrays

Arrays are allocated with **new** in expressions. Thus identifiers which are bound to array references get initialized in the declaration of a field or local variable or when we pass an array as a parameter. Furthermore the value of such an identifier can be changed by an assignment.

Instruction	Meaning	Before	After
aload v	$OPTOP := OPTOP+1$; $STACK\,[OPTOP]{:=}STACK\,[VARS+v]$; $PC := PC+2$	()	(A)
aaload	$OPTOP := OPTOP$ -1; **if** $(STACK\,[OPTOP]{=}\mathrm{NULL})$ **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	$OPTOP := OPTOP$ -1; **if** $(STACK\,[OPTOP]{=}\mathrm{NULL})$ **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)	(N)
astore v	$STACK\,[VARS+v]{:=}STACK\,[OPTOP]$; $OPTOP := OPTOP$ -1; $PC := PC+2$	(O)	()

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

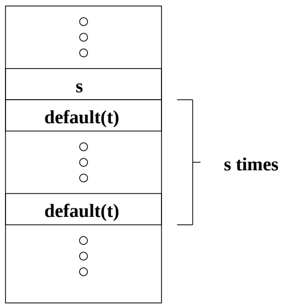

Figure 2: Allocation of Arrays

5.3 Classes

Before a class can be compiled some static information has to be computed. Here we divide the compilation of a class into three phases:

- computation of an address environment: $elab\_def$
- simple type inference for expressions: $infer\_type$
- code generation: $code$

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

$elab\_def\ (t\ x{=}init)\ \rho\ nav\ nas\ loc$ =
  $(\rho[\,(t, nav)/x\,], nav + sizeof(t), nas, loc + 1)$

$elab\_def$ (**static** $t\ m(t_1\ p_1,\ldots,t_k\ p_k)\ \{b\}$) $\rho\ nav\ nas\ loc$ =
  let $(\rho', nav', nas', loc') = elab\_def\ b\ \emptyset\ 0\ 0\ 0$
  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 $(\rho', nav', nas', loc') = elab\_def\ b\ \emptyset\ 0\ 0\ 0$
  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 is added into the constant pool such that $\gamma_K(c/m(t_1 \ldots t_k)t) = \pi$ where c is the current class and π is a pointer into the method table. In addition the method table is extended by an entry $\gamma_M(\pi) = (p, static, n, l)$, where p is the address of the byte code for m, n the number of its arguments and l the number of its local variables. For virtual methods similar entries are added to the constant and method table. The only difference is, that in this case we have $\gamma_M(\pi) = (p, dynamic, n, l)$.
For field declarations **static**$tx = e$ an entry is added into the constant pool such that $\gamma_K(c/x) = \pi$ where c is the current class and π is a pointer into the field table. In addition the field table is extended by an entry $\gamma_F(\pi) = (t, static, a)$, where a is the address of the field. For object fields $tx = e$ we add an entry $\gamma_F(\pi) = (t, dynamic, a)$ where o is the offset of the field.

5.3.2 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.

The implementation of a type inference for TassKaf will be part of project 3.

5.3.3 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 this is legal, as a consequence overloading resolution is slightly more difficult (an additional bottom-down pass in the type inference). In fact overloading resolution in Java is more difficult than what we discuss here. If there is no method with exactly those argument types infered, but the arguments in the method call can be converted into the types of one or more existing methods, then the method is chosen with the least conversion costs. This is a very problematic approach to overloading resolution, because it is very difficult to handle by the Java programmer.

```
class A { }
class B extends A { }

class Overload
{ static void m(A x, B y) { System.out.println("first"); }
  static void m(B x, A y) {  System.out.println("second"); }

  public static void main (String args[])
  { m(new B(), new B()); }
}
```

In this case the conversion costs for both alternatives are equal and a Java compiler will produce an error message, that the call is ambiguous. If we write

```
static void m(A x, A y) { ... }
static void m(B x, A y) {  ... }
```

then the second alternative is chosen. In Java many 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 allow method calls, for which the infered argument types exactly match those of a defined method.

5.3.4 Method Invocation

Instruction	Meaning	Before	After
return	$PC := STACK[FRAME]$;	()	...
	$OPTOP := VARS$-1;		
	$VARS := STACK[FRAME+1]$;		
	$FRAME := STACK[FRAME+2]$		
ireturn	$PC := STACK[FRAME]$;	(N)	...
	$STACK[VARS] := STACK[OPTOP]$;		
	$OPTOP := VARS$;		
	$VARS := STACK[FRAME+1]$;		
	$FRAME := STACK[FRAME+2]$		
areturn	$PC := STACK[FRAME]$;	(O)	...
	$STACK[VARS] := STACK[OPTOP]$;		
	$OPTOP := VARS$;		
	$VARS := STACK[FRAME+1]$;		
	$FRAME := STACK[FRAME+2]$		
invokestatic m	**let** (p,n,l,static)=$\gamma_M(\gamma_K(m))$ **in**	(V,...,V)	...
	$STACK[OPTOP+1+l] := PC+3$;	n times	
	$STACK[OPTOP+2+l] := VARS$;		
	$STACK[OPTOP+3+l] := FRAME$;		
	$VARS := OPTOP$-n;		
	$FRAME := OPTOP+1+l$;		
	$OPTOP := OPTOP+3+l$;		
	PC:=p		
	end		
invokevirtual m	**let** (p,n,l,dynamic)=$\gamma_M(\gamma_K(m))$,	(V,...,V)	...
	$\gamma'_M = STORE[STACK[OPTOP$-n$]]$	n times	
	(p',n',l',dynamic)=$\gamma'_M(\gamma_K(m))$		
	in		
	$STACK[OPTOP+1+l'] := PC+3$;		
	$STACK[OPTOP+2+l'] := VARS$;	$n = n'$	
	$STACK[OPTOP+3+l'] := FRAME$;		
	$VARS := OPTOP$-n';		
	$FRAME := OPTOP+1+l'$;		
	$OPTOP := OPTOP+3+l'$;		
	PC:=p'		
	end		

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

For virtual methods we have to pass a reference of the object as an implicit first parameter. The current object is always stored as the 0-th local variable (**aload** 0).

For static methods we only have to pass the arguments to the method and no reference to the current object.

5.4 Objects

5.4.1 this

In each execution frame of a virtual method a reference to the object of the method is stored in the first frame element ($STACK[VARS]$).

5.4.2 Initialization of Fields

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

5.5 Machine Instruction

Instruction	Meaning	Before	After
new c	STACK[OPTOP]:= $newObject(c)$; PC:=PC+3	()	(O)
getstatic i	**let** $(c,f)=\gamma_K(i)$, $(\gamma'_K,\gamma'_F,\gamma'_M,\gamma'_P,C',S')=\gamma(c)$, $(t,sd,a)=\gamma'_F(f)$ **in** $OPTOP:=OPTOP+1$; $STACK[OPTOP]:=STORE[a]$; $PC:=PC+3$ **end**	()	(V)
putstatic i	**let** $(c,f)=\gamma_K(i)$, $(\gamma'_K,\gamma'_F,\gamma'_M,\gamma'_P,C',S')=\gamma(c)$, $(t,sd,a)=\gamma'_F(f)$ **in** $STORE[a]:=STACK[OPTOP]$; $OPTOP:=OPTOP$-1; $PC:=PC+3$ **end**	(V)	()
getfield i	**let** $(c,f)=\gamma_K(i)$, $(\gamma'_K,\gamma'_F,\gamma'_M,\gamma'_P,C',S')=\gamma(c)$, $(t,sd,a)=\gamma'_F(f)$ **in** $STACK[OPTOP]:=STORE[STACK[OPTOP]+a]$; $PC:=PC+3$ **end**	(O)	(V)
putfield i	**let** $(c,f)=\gamma_K(i)$, $(\gamma'_K,\gamma'_F,\gamma'_M,\gamma'_P,C',S')=\gamma(c)$, $(t,sd,a)=\gamma'_F(f)$ **in** $STORE[STACK[OPTOP\text{-}1]+a]:=STACK[OPTOP]$; $OPTOP:=OPTOP$-2; $PC:=PC+3$ **end**	(O,V)	()

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

$newObject(c)=$
let $(\gamma'_K,\gamma'_F,\gamma'_M,\gamma'_P,C,S)=\gamma(\gamma_K(c))$

$$size = \sum_{\substack{i \in dom(\gamma'_F) \\ (t, \text{dynamic}, a) = \gamma'_F(i)}} sizeof(t)$$

obj=$newMem$(1+$size$)
in
$STORE$[obj]=γ'_M
for every $i \in dom(\gamma'_F)$ **do**
let $(t, sd, a) = \gamma'_F(i)$ **in**
if sd=dynamic **then** $STORE$[obj+1+a]:=$default$(t); **end**
od
return obj;
end

The function $newMem(n)$ returns the address of the first cell of a free area of size n in the heap.

5.6 Assignments

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.

In assignments to object fields of the current object, it is legal not to explicitly reference the object:

The translation schemes for assignments to arrays is a little bit more complicated.

6 Optimizations at Runtime instead of at Compiletime

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** "$c/m(t_1 \ldots t_k)t$". The JVM first checks, whether the class c has been loaded before, if not the class is loaded. Then it looks up the pointer π associated with m in the constant pool and the related entry in the method table ($\gamma_M(\pi)$). 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 it. To enforce, that these tests are not done again, the JVM replaces **invokestatic** "$c/m(t_1 \ldots t_k)t$" 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**

Instruction	Meaning	Before	After
invokestatic m	**let** π=$\gamma_K(m)$ (p,n,l,static)=$\gamma_M(\pi)$ **in** $STACK$[$OPTOP$+1+l]:=PC+3; $STACK$[$OPTOP$+2+l]:=$VARS$; $STACK$[$OPTOP$+3+l]:=$FRAME$; $VARS$:=$OPTOP$-n; $FRAME$:=$OPTOP$+1+l; $OPTOP$:=$OPTOP$+3+l; $\gamma_P(PC)$="invokestatic_quick π" PC:=p **end**	(V,...,V) n times	...
invokestatic_quick π	**let** (p,n,l,static)=$\gamma_M(\pi)$ **in** $STACK$[$OPTOP$+1+l]:=PC+3; $STACK$[$OPTOP$+2+l]:=$VARS$; $STACK$[$OPTOP$+3+l]:=$FRAME$; $VARS$:=$OPTOP$-n; $FRAME$:=$OPTOP$+1+l; $OPTOP$:=$OPTOP$+3+l; PC:=p **end**	(V,...,V) n times	...