Virtual Class Support at the Virtual Machine Level

Anders Bach Nielsen Erik Ernst
University of Aarhus, Denmark
{abachn,eernst}@cs.au.dk

Abstract

This paper describes how virtual classes can be supported in a virtual machine. Main-stream virtual machines such as the Java Virtual Machine and the .NET platform dominate the world today, and many languages are being executed on these virtual machines even though their embodied design choices conflict with the design choices of the virtual machine. For instance, there is a non-trivial mismatch between the main-stream virtual machines mentioned above and dynamically typed languages. One language concept that creates an even greater mismatch is virtual classes, in particular because fully general support for virtual classes requires generation of new classes at run-time by mixin composition. Languages like CaesarJ and Object Teams can express virtual classes restricted to the subset that does not require run-time generation of classes, because of the restrictions imposed by the Java Virtual Machine. We have chosen to support virtual classes by implementing a specialized virtual machine, and this paper describes how this virtual machine supports virtual classes with full generality.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features — Classes and objects, Inheritance ; D.3.4 [*Programming Languages*]: Processors — Interpreters, Run-time environments

General Terms Languages, Design

Keywords Object oriented programing language, virtual classes, class combination, virtual machines, gbeta

1. Introduction

Like virtual methods, classes can also be modeled as late-bound features of objects, in which case they are known as virtual classes [20, 10, 11]. A virtual class is obtained by lookup in an object, just like a virtual method implementation is found by lookup. This means that the type analysis of a language that supports virtual classes must be able to handle a form of existential types, because a virtual class is typically only known by an upper bound, and a simple form of dependent types, because the identity of the object where it is looked up is an essential part of the type. However, it is not just the static analysis of virtual classes that requires advanced concepts.

In full generality, the dynamic semantics of virtual classes requires that they can be constructed at run-time by composition of mixins. This creates a significant mismatch between the virtual machine requirements associated with virtual classes, and the actual feature set provided by main-stream virtual machines such as the Java Virtual Machine and the .NET platform.

Several languages that support a form of virtual classes use a main-stream virtual machine (the JVM) as the execution platform, including CaesarJ [3, 22] and Object Teams [15, 16]. However, they support only the subset of virtual classes that does not require run-time generation of classes, because of the restrictions imposed by the Java Virtual Machine. In the source language this implies that it is not possible to inherit from a class unless there is a location in the program from where it is statically known; it is typically not known, e.g., for a superclass on the form `x.C`.

In contrast, we have chosen to support virtual classes in full generality, by implementing a specialized virtual machine, the gvm. This virtual machine differs from the main stream by having an object layout that explicitly models parts of an object as instances of mixins, and by being able to dynamically compute classes and set up the representation of them in objects, in order to support dynamic class lookup of classes. Another area where this virtual machine is different is concerned with object initialization, which may include essentially general execution of code. More traditional languages would initialize all fields to default values (such as null) and run a constructor, but this is not an option when some attributes are class valued and may be used as types—such type attributes must be initialized before user code gets access to the object for the first time.

The main contribution of this paper is that it describes how general support for virtual classes may be realized in a specialized virtual machine, including dynamic computation of classes and type safe initialization of objects.

The rest of this paper is organized as follows: The next section introduces virtual classes and describes how they differ in the language BETA [18, 21] where they were first introduced, in the generalized form present in the language gbeta, and in several other languages. Section 3 is the core part of the paper; it describes the run-time entities and mechanisms that are needed in a virtual machine in order to support the required semantics. Section 4 describes a non-trivial example that demonstrates dynamic creation of classes as well as recursive propagation of the class composition process, also known as deep mixin composition. Finally, section 5 describes the implementation status, section 6 describes related work, and section 7 concludes.

2. History of Virtual Classes

The concept of virtual classes was introduced in the BETA programming language in the late 70s. They were substantially generalized during the development of the gbeta language in the 90s, in particular by allowing them to appear as superclasses and in a number of other contexts where they could not previously be used. Virtual classes have later been adopted and used by a range of lan-

VMIL'09 October 25, 2009, Orlando, FL, USA.

guages, including CaesarJ [3, 22], Object Teams [15, 16], J& [26], Tribe [9], and Newspeak [6, 7]. Moreover, the language Scala is able to emulate virtual classes by means of its abstract type members and path dependent types. This section characterizes the techniques used in the different languages and the extent to which they implement virtual classes.

2.1 BETA

The BETA programming language [18, 21] was the first language to support virtual classes [20]. Actually, they arose as a natural generalization of virtual methods, because methods and classes and a number of other concepts were unified into the concept of a *pattern*. Hence, BETA does not have classes in the traditional sense like Smalltalk or Java, it has patterns. Patterns may be used as methods or as classes, because they have all the features needed in order to play each of these roles; but there is no technical difference between a method call and an object, they are just instances of patterns which are used differently. Consequently, virtual classes are actually virtual patterns in the context of BETA.

The BETA language was realized as the Mjølner BETA compiler, which translates BETA programs to native code and uses the standard linker to create executables. The choice of translating programs to native code gives rise to some considerations, but in the 80s this was the only way to get decent execution speed for translated BETA programs. To get an efficient translation, to produce efficient code, and due to the limited generality of the static analysis, some restrictions were imposed on the language. The two most important restrictions in this context were that 1) virtual patterns could not be used as superpatterns, and as a consequence it was possible to maintain that 2) there was only single inheritance, and no mixin composition.

By having only single inheritance in BETA, the process of finding all contributions to a pattern (all mixins) was made a lot easier, because the superpattern was always a compile-time constant. This made it possible to create a static object layout for each pattern, which again allowed for fast lookup based on fixed offsets to every attribute in the object. There is no need to create classes (i.e., patterns) at runtime with this semantics.

BETA patterns can also be used as methods, and may work similarly to standard method calls with an activation record and a method body that is executed. BETA patterns that contain multiple mixins work like a composition of several methods, somewhat similar to the effect of method combination in CLOS [5], and also similar to the combination of a number of overridden methods in main-stream languages like Java by means of `super` calls. For this method composition, a precise lookup table (similar to a vtable) of mixins can be calculated to represent the entire method. This lookup table is then used to find the next more specific body to call in a method, when evaluating the so-called INNER-statements [13], which are similar to super invocations, but toward the subpattern rather than toward the superpattern.

Virtual patterns can be very useful as superpatterns for this kind of composite method construction, but BETA does not support virtual superpatterns. In fact, with virtual patterns as superpatterns, it would not be possible to have only one object layout for each pattern, but depending on context there would be as many layouts as there are versions of the virtual super pattern, and this set of patterns can not be computed modularly at the location where the subpattern is created. Consequently, BETA programmers must work around the lack of virtual patterns as superpatterns—as shown in the example in Fig. 1.

Before we dive into the example we need to mention a few unusual syntactic properties of BETA, picking out some details from Fig. 1: A mixin, i.e., the increment between a pattern and a superpattern, is declared by means of a *mainpart*, which is a block enclosed by `(# ... #)`. An argument list may be declared in a mainpart by means of an `enter` clause, a list of returned values may be declared by an `exit` clause, and the statement block that specifies the behavior of the pattern is declared by the keyword `do` followed by a list of statements. Assignment, method call, expression evaluation, and a number of other mechanisms are unified syntactically into *evaluations*, which are binary expressions with the operator `->` in the middle; note that the data-flow is from left to right, just like the arrow, which makes this construct similar to the pipe symbol, '`|`', used on the command line of many operating systems. Finally, `[]` indicates reference evaluation (pointer semantics for assignments etc). For example, `enter theScanner[]` specifies that `scanImpl` accepts one by-reference argument named `theScanner`; `elm[]->current[]` is a reference assignment from `elm` to `current`, and `theElement[]->theScanner` is an invocation of the variable method (i.e., function pointer) `theScanner`, passing `theElement` as an argument, by reference.

```
(#
  container: (#
    element:< object;
    scan: (#
      current: ^element
      do (#
           doINNER: scanner(# do elm[]->current[];
                              INNER scan #)
         do doINNER[]->scanImpl
        #)
    #);
    scanner: (# elm: ^element enter elm[] do INNER #);
    scanImpl:< (# theScanner: ^scanner
      enter theScanner[]
      do INNER
    #)
  #);
  (* dummy implementation: one element only *)
  list: container (#
    theElement: ^element;
    append: (# enter theElement[] #);
    scanImpl::< (# do theElement[]->theScanner #)
  #);
#)
```

Figure 1. BETA example using virtual patterns as both methods and classes.

Figure 1 shows a very simplified piece of the BETA standard library where we define the pattern `container` and the subpattern `list`, describing the standard list data structure with `append` and `scan` methods; the `scan` method is used to iterate over all elements in a list, similarly to the `do:` method on Smalltalk collection classes. Normal usage of such a list structure is shown below, where an object `myList` is created from the pattern `list` where we bind the element type to `string`. Then a string is appended to the list and at last we iterate over the elements of `myList` using the `scan` method to execute the supplied block, which prints each element.

```
(# myList: @list(# element:: string #)
do 'example'->myList.append;
   myList.scan(# do current[]->putline #);
#)
```

Intuitively, the `scan` method in the `container` pattern should be a virtual pattern, in which case the actual implementation in the `list` pattern could be supplied as a further binding of the `scan` method, which would be very simple and direct. However, this would break the example above, because the virtual pattern `scan` would be used as a superpattern, which is not allowed in BETA. Because of this issue, the example in Fig. 1 has a virtual pattern `scanImpl` that is used to specify how to iterate over the data structure, and a non-virtual pattern `scanner` that performs

the copying of the current element, and the delegation to the user supplied block, and then it uses a reference (think: function pointer) in order to pass `theScanner` into `scanImpl` such that it can be called.

2.2 gbeta

Initially, the gbeta programming language was created in order to have an open source version of BETA. Later it turned into a language in its own right, and the basic mechanisms of BETA, including virtual patterns, were fundamentally generalized in the process. With the strong heritage in the BETA language there are many similarities between BETA and gbeta; note however that the syntax of gbeta has been changed quite visibly such that it is in several ways more convenient. The underlying syntactic structures of the two languages are still closely related. The core of this paper describes the virtual machine gvm, which is able to run gbeta programs, so at this point we just introduce gbeta briefly, based on the example in Fig. 2.

As mentioned, the basic structure of gbeta syntax is the same as in BETA. In gbeta, a mainpart is enclosed by braces, { . . . } which may have an argument list and/or return value list associated, `%(<args>|<returns>){ . . . }`, and the arg/return list may also be used on its own if { . . . } is not needed; consequently, the `enter` and `exit` clauses have been eliminated. The evaluation operator is the pipe symbol, '|'. Indications of evaluation mode (mainly: by reference vs. by value) has been moved away from statements and into declarations, which means that the `[]` markers used many places in BETA are now gone. For instance, the anonymous function which accepts an integer and returns two times that integer is declared as `(# i: @integer enter i exit 2*i #)` in BETA, whereas it is `%(i:int|2*i)` in gbeta. The percent symbol, '%', generally indicates communication (receiving arguments, returning results, or looking up features) and must be present whenever communication is to be allowed. Finally, note that the superscript numbers are not part of the syntax, they are just used to refer to specific mainparts later on.

Now, Fig. 2 declares a family of classes, `Lang`, with two virtual classes as members, `Exp` and `Lit`. This amounts to a minimal version of the standard example illustrating the expression problem [28]. This family is specialized into two other families `LangEval` and `LangPrint`, which declare the interface extension needed in order to support evaluation and printing of expressions. The further specialization of these two into `LangEvalImpl` and `LangPrintImpl` adds an implementation of the two interfaces, and finally the program itself (the statements in lines 21–31) assigns the two complete (i.e., implemented) class families to two class valued variables `LangVar1` and `LangVar2`, which implies that the exact class families are not known at compile time when these variables are used. The nested block (lines 25–30) creates an instance of the composition of the two class families, named `F`; the composition must occur at runtime because it is based on variables, not compile-time constant class denotations. Next, it declares a variable `lit` whose type depends on the dynamic family `F`, and creates instances of members of `F`, and calls methods on them. All in all, this example illustrates how dynamic composition of class families with deep mixin composition can be expressed.

2.3 CaesarJ, Object Teams and Scala

Many recent languages are designed to run on existing platforms to thereby utilize software already present in the wild. A popular platform for language designers is the Java virtual machine (JVM). Two recent languages based on the JVM that include support for a kind of virtual classes are CaesarJ and Object Teams. Both languages express virtual classes at the surface level and transform

```
1   ORIGIN 'gbetaenv'
2   -- program:merge --
3   {1
4     Lang: %{2
5       Exp:< object; Lit:< Exp %{10 value: int }
6     };
7
8     LangEval: Lang %{3 Exp:: %{8 eval: %(|i:int)}};
9     LangEvalImpl: LangEval {4
10      Lit:: {11 eval:: { value | i }}
11    };
12
13    LangPrint: Lang %{5 Exp:: %{9 print:%(|s:string)}};
14    LangPrintImpl: LangPrint {6
15      Lit:: {12 print:: { value | int2str | s }}
16    };
17
18    LangVar1: ^#=LangPrint;
19    LangVar2: ^#=LangEval;
20  #
21    LangPrintImpl# | LangVar1#;
22    LangEvalImpl# | LangVar2#;
23
24    {7
25      F: @LangVar1 & LangVar2;
26      lit: ^F.Lit;
27    #
28      F.Lit^ | lit;
29      3 | lit.value;
30      lit.eval | int2str | stdio
31    }
32  }
```

Figure 2. gbeta example where two families are created and the patterns are stored in pattern references. These references are combined to create a new family dynamically. The number annotations are not part of the actual syntax, but used to refer to specific mainparts.

them into regular classes during compilation, along with factory methods that allow for `new` expressions creating instances of a virtual class.

A third language which is able to emulate the same subset of virtual classes as CaesarJ and Object Teams is Scala [27, 2]. However, there is no direct support for virtual classes in Scala, so they must be expressed as a rather complicated idiom[1] using traits and abstract type members to express virtual class behavior. Similarly to CaesarJ and Object Teams, classes must in all cases be known precisely, relative to the dynamic type of the enclosing object. Note that this is not the case when inheriting from an object which is looked up (for instance, with this restriction `myList.scan(# . . . #)` as on page 2 is impossible when `scan` is a virtual pattern).

All three languages are compiled directly to Java byte code. There is no method composition, because methods have to be Java-like in order to be executed by the JVM. Hence, only overriding and overloading of methods is supported.

3. The gbeta Runtime System

Traditionally, the compiler and the virtual machine of the gbeta language were combined in one single application. This was convenient for several reasons, including testing and interactive querying about types. However, a virtual machine implemented in a high level language seldom gives good performance, except for meta-circular virtual machines like the Jikes Research VM [8, 1]. When using a high level language the design of the virtual machine is

[1] Design document showing some details of how virtual classes can be expressed in Scala `http://lampsvn.epfl.ch/trac/scala/wiki/VirtualClassesDesign`.

```
1   {1
2     Point: %{2
3       x,y: @int;
4       move:< %(dx,dy:int) {4 inner; x+dx|x; y+dy|y; }
5     };
6     ColorPoint: Point %{3
7       c: @string;
8       move:: {5 2*dx|dx; 3*dy|dy; inner; }
9     };
10    cp:@ColorPoint;
11  };
```

Figure 3. Small gbeta example where the enclosing pattern contains three fields. Two of the fields contain pattern definitions, `Point` and `ColorPoint` and the last field `cp` contains an object created from the pattern `ColorPoint`. The number annotations are not part of the actual syntax, but used to refer to specific mainparts.

often restricted by the typing discipline of the language. We have chosen to implement a new specialized virtual machine, called the gvm, for the gbeta language supporting all its features. The virtual machine is implemented in C++, has a standard Cheney garbage collector [17, page 117] and uses a direct threaded byte code interpreter [4, 12] as its main execution loop. The design of the virtual machine is prepared for execution of native code, but a Just-In-Time compiler is currently future work.

The sections below give an informal description of the basic entities in the gbeta language at both the compilation and run-time level. We discuss the generated intermediate language and will focus on how information related to virtual classes are represented. At last we cover the internals of the gvm and focus on how virtual classes are computed at run-time.

3.1 Basic Entities and Compilation

The compilation process of transforming gbeta programs to the intermediate language is done by a pair of compilers. The first part of the compilation process is performed by the original gbeta compiler [10], which produces textual source code for a specialized bytecode language. This textual format does not represent a suitable input format for the virtual machine, so a second compilation is required. This second part of the compilation process is handled by the gbcc compiler [23], which analyzes and transforms the already compiled program to an optimized binary format, the gbci-file. All the transformations performed by the gbcc compiler could be integrated into the original gbeta compiler, but this would tie the compiler even closer to the virtual machine and could potentially block other possible research directions. We do not go into detail about the static analysis performed, the different optimizations that are done or the other tasks handled by the compilers, but we will focus on explaining only the relevant parts of the compilation process and the basic entities of the language.

The basic entities of the gbeta language can be divided into those that are present at compile-time and those that are present at run-time. We will use the example in Fig. 3, as a running example to illustrate the different aspects of the gbeta language that are important in regards to this paper.

As mentioned, a crucial part of gbeta is the concept of a pattern. The pattern concept unifies the concepts of methods and classes known from other programming languages. A pattern is represented as an array of mixins, and each mixin consists of two elements. The first element is a static description, known as a mainpart, and the second element specifies the dynamic context of the mainpart. These contexts are only available at run-time, and therefore a pattern can only be constructed at run-time. The only compile-time entity within a pattern is the mainpart.

Figure 3 contains one surrounding mainpart denoted by the superscript 1, which has three fields `Point`, `ColorPoint` and `cp`. There are a total of five mainparts in the figure, marked by the superscripts 1 to 5. Future references to mainparts will be of the form mainpart-n. A mainpart is a static description, which contains a list of its fields, the initialization code for each field and an action part. As an example, Mainpart-3 in Fig. 3 has two fields called `c` and `move`. For each field a block of initialization code is generated. For field `c` the initialization code block will create a string object and install it into the first field, and for field `move` the initialization block finds a pattern and installs it into the second field. There is no action part for the mainpart, which means that a default action part containing only an INNER-statement is used.

If enough information is available at compile-time about the mixins that are needed to construct a particular pattern, the compiler may generate a static pattern instead of generating code for building the pattern dynamically. A static pattern is a compile-time entity that describes the mixins needed to construct a particular pattern at run-time. However, if the compiler is unable to determine the exact set of mixins in the pattern, it will generate code for building the pattern at run-time.

Like all other object oriented languages an important run-time entity is the object. An object is created from a pattern and as a pattern is an array of mixins, an object is conceptually a list of part objects. The layout of each part object is described in the mainpart of the corresponding mixin. The actual fields that are described in the mainpart are allocated in the corresponding part objects, and each field is initialized by executing the appropriate initialization block. There are four different kinds of fields in a part object:

- There are immutable fields that contain a pattern.
- There are immutable fields that contain an object.
- There are mutable fields that contain a reference to a pattern.
- There are mutable fields that contain a reference to an object.

Other languages like Java use default values (like null and zero) to initialize all fields in an object. After this initialization the constructor is called, to set up the object. In contrast, an object in gbeta requires full initialization of all immutable fields before user code is allowed to access the object, because these fields can be used in types. This means that if an immutable field contains an object, the object is built, initialized, and installed into the field before the next field is processed. This makes the initialization of objects a complex matter.

During compilation an analysis of each mainpart is performed and an ordering of fields may be found that satisfies the initialization dependencies among the fields. If one field uses another field, in the same object, as the type of the object, which this particular field should contain then there is a dependency between these fields. As an example, look at line 10 in Fig. 3, the field `cp` is immutable and contains an object that is build from the pattern in field `ColorPoint`. To be able to initialize the field `cp`, the pattern in the field `ColorPoint` has to be built first. If there exists an ordering that satisfies all such dependencies among the fields in a mainpart, this ordering can be used to create the array of field initialization instructions that will initialize all fields correctly. If there is no such ordering, the mainpart is marked and a slower on-demand initialization of the fields is used.

At the syntactic level there are two kinds of virtual pattern declarations, the initial binding of a virtual pattern (`:<`) and the further bindings of the virtual pattern (`::`). There cannot be a further binding of a virtual pattern if there is no initial binding for it, and initial and further bindings are handled separately at both compile- and run-time.

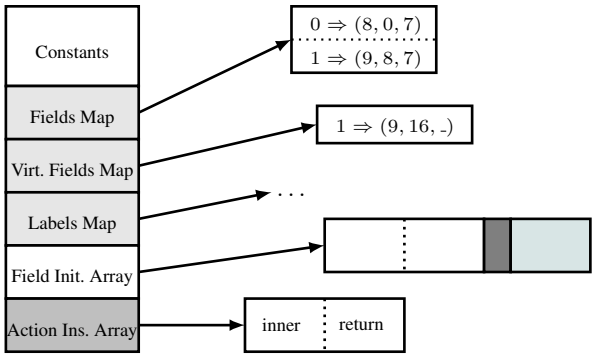

Figure 4. The structure of mainpart 3 in Fig. 3 and its elements. For the sake of simplicity the structure of the labels map is left out.

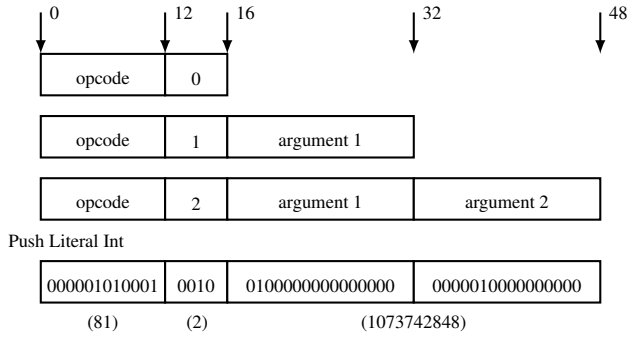

Figure 5. Structure of the byte code instructions with op-code, number of arguments and the actual arguments. As example is shown the push literal integer instruction.

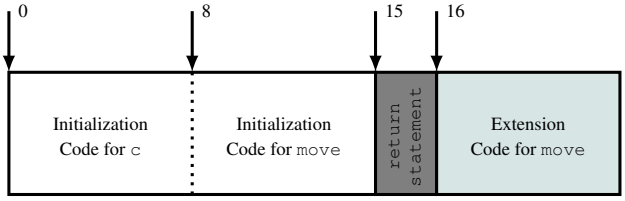

Figure 6. Layout of the field initialization array from Fig. 4 with initialization code for two fields, a return statement separating the field initialization code and the virtual expansion blocks and one expansion block for a further binding of a virtual pattern.

The code generated for an initial binding and for further bindings of a virtual pattern is very different. Where the initial binding of a virtual pattern drives the construction of the pattern, the further binding relies on the initial binding to have built the complete pattern.

The field initialization block for an initial binding of a virtual pattern contains three parts. The first part is the construction of the initial pattern, which can be any number of instructions involving the creation of a pattern. The second part is what makes initial bindings special, in that it contains the search instruction *next_virtual* and the third part is the installation of the complete pattern into a specific field in the part object.

As an example look at line 4 in Fig. 3. This is the initial binding of the virtual pattern `move`. It describes that the initial pattern has one mixin with mainpart-4.

A further binding of virtual pattern will generate an extension code block, in addition to the field initialization block. The extension block contains code for extending an already existing pattern, and the field initialization block is the installation of an already built pattern from the initial binding of the virtual pattern.

As an example look at line 8 in Fig. 3. This declaration is a further binding of the virtual pattern `move` and describes that one mixin with mainpart-5 should be added to the final pattern. At run-time this field will also contain the complete virtual pattern of `move`.

3.2 Intermediate Language

The binary gbci-file, which is the output of the gbcc compiler and the input for the virtual machine, is a compact description of a compiled and linked gbeta program containing only the compile-time entities described in the previous section.

The layout of a gbci-file is relatively simple and can be divided into three parts. The first part of the gbci-file is comprised of several tables. Each table is mapping an integer value to a symbol, a string or the textual name of a mainpart. The second part of the gbci-file is a list of mainparts, which we will return to, and the third part is a list of static patterns. We will not discuss static patterns any further in this paper, because they represent patterns that are fully known at compile time, which is similar to the case that is already well-known from other virtual machines. In this paper we will focus on the cases where the compiler can not do this analysis and needs to delay the construction of the pattern to run-time.

The mainpart is the most essential building block in the gbeta programming language. At run-time, it is possible to generate new patterns from existing patterns [24], but it is not possible to generate new mainparts. As shown in Fig. 4 a mainpart can be divided into four parts, as indicated by the coloring. The first part contains constants, which include stack size and temporal array size. The second part contains three maps, one for fields, one for virtual fields, and one for labels. The last two parts of a mainpart are both arrays of byte code instructions and contains the field initialization blocks and action part, respectively. The array containing the field initialization blocks also contains the virtual pattern extensions. The action part is only used when a pattern is used as a method.

The intermediate language is the descriptions of all the static patterns and the mainparts that a program contains. Besides these elements, the intermediate language describes several tables for symbols and strings, various maps in each mainpart, and byte code instructions in the field initialization and action arrays. There are a little more than 200 byte code instructions, and each instruction consists of an op-code, the number of arguments the op-code uses, and the actual arguments. The byte code instructions are segmented into 16 bit blocks, and the op-code and the number of arguments are packed into one 16 bit block. The arguments are located in the subsequent 16 bit blocks. An overview of this structure is shown in Fig. 5 where the instruction layout of op-codes with zero, one and two arguments are shown. At the bottom of Fig. 5 the 'push literal integer' instruction is shown in its binary form. This instruction has op-code 81, uses 2 argument slots of 16 bits each, and will push the raw integer (in this case the number 1073742848) onto the evaluation stack.

Of the three maps in each mainpart, only two are important for field initialization. For the on-demand initialization of field to work the fields map is used to find the correct piece of initialization code for a given field. The virtual fields map is used only when searching for extensions to a virtual pattern.

The layout of the fields initialization instruction array is shown in Fig. 6. At the beginning of the array all the initialization instructions for the fields are located in sequence. If the dependency analysis found an ordering of the fields this sequence can be executed to initialize all fields in the part object. However, if that or-

Blocks of field initialization code in mainpart 2

Field `x` (I)

```
push_ptn_integer
new_stk
ins_ptn 0
```

Field `y` (II)

```
push_ptn_integer
new_stk
ins_ptn 1
```

Field `move` (III)

```
push_ptn_object
add_mixin_t 4
next_virtual
ins_ptn 2
```

Blocks of field initialization code in mainpart 3

Field `c` (V)

```
push_ptn_string
new_stk
ins_ptn 0
```

Field `move` (VI)

```
rtps_upp 1
rtps_lpo 2
ins_ptn 1
```

Blocks of field extension code in mainpart 3

Field `move` (IV)

```
push_ptn_object
add_mixin_t 5
ptn_merge
next_virtual_return
```

Figure 7. Field initialization code blocks for fields in the mainparts 2 and 3 in Fig. 3. The roman numerals indicate the evaluation order when initializing the object in field `cp`.

dering was not found then the instructions for a given field need to be copied and executed on a field-by-field basis. The last part of the field initialization instruction array is used for the virtual pattern extensions. There is no need for a separator between the extension blocks because all blocks end with the search instruction: *next_virtual_return*.

Of all the byte code instructions, some of the more interesting instructions in relation to this paper, can be seen in Fig. 7. Most of the instructions are straight forward and the behavior can be guessed from the instructions name. There are however, the instructions *next_virtual*, *next_virtual_return* and the range of *rtps_* instructions, seen in block VI in Fig. 7.

In gbeta there is a concept of a run-time-path that can be traversed at run-time to find another object or pattern. The *rtps_* instructions are used to traverse such a run-time-path one step for each instruction. Each *rtps_* instruction comes in two versions, where the ones that end with *pp* are steps from part object to part object and the ones that end with *po* and from a part object to an object or pattern and the object or pattern is pushed onto the evaluation stack. The two instructions shown in Fig. 7 are *rtps_upp 1* and *rtps_lpo 2*, which respectively goes up one step to a more general part object and a lookup of the object stored in the second field in the part object and pushed that object found in the field to the evaluation stack.

The two search instructions that are used to find extensions to a virtual pattern are the *next_virtual* and the *next_virtual_return* instructions. The *next_virtual* search instruction is only generated in the initialization blocks of initial bindings of virtual patterns. This instruction will search the more specific part objects of the object to find an extension to the virtual pattern. If an extension is found a larger pattern will be created. The *next_virtual_return* search instruction is generated as the last instruction in the extension block of further bindings of virtual patterns and work similar to the first search instruction.

3.3 Virtual Machine

Mainparts and static patterns are the compile-time entities, which the virtual machine loads from the gbci-file. All compile-time entities are currently allocated in a static code space in the virtual machine, where there is no garbage collection.

The most important run-time entities are objects, patterns and evaluation frames. There are more run-time entities in the virtual machine, but these are the most important ones. All run-time entities are heap allocated and are subject to garbage collection. We have already given a hint about the structure of objects and patterns in section 3.1, but at this point we will go into greater detail.

As a point of reference, consider Java class files [14, 19]. Among other things, Java class files contain information about static fields and methods, instance variables and instance methods related to a class. There are no static fields or methods in gbeta, so no corresponding element is needed. Next, patterns are arrays of mixins that form classes or methods, so they correspond to both classes and methods in Java. Each mixin contains a mainpart, which roughly corresponds to the information about one class in Java. It may contain references to other mainparts used in the methods of the class—that is, nested patterns. A mixin also contains a reference to its dynamic context, the enclosing part object, and finally each part object is created from a mixin.

Given that every part object has a mixin which has an enclosing part object, it may seem like a never ending chain. However, during start-up the virtual machine creates the top most part object of the chain and installes this part object into a global position. This part object is called *predef*, because all the predefined patterns are located as fields in this *predef* part object. It is special in that it does not have a mixin or an enclosing part object, and it is enforced at compile time that no attempts are made to look up these non-existing things.

This strong relationship between mixins and part objects would make it natural to model an object as an array of part objects just like a pattern is an array of mixins. However, unlike mixins, part objects do not have a fixed size, so using an array is not possible. A natural way to model an object would be as a doubly linked list of part objects, which would enable the traversal of part objects in both the more general and more specific direction. However, the object would be split into several smaller part objects in the heap and linked together by pointers. The current design of an object in the virtual machine is to contiguously allocate the object header, an offset table and all the part objects in one memory cell. This design enables a more compact memory layout, which permits the garbage collector to move an entire object in one copy operation, instead of several copy operations and pointer updates to move each part object separately. With the design choice of having object contiguously allocated, came the limitation that a part object could not be referenced directly. Therefore are all part object references actually a pair of an object reference and an offset into the object. With the object reference and the offset into the object we are able to find the desired part object within the contiguously allocated object and using the offset table in the object we are able to traverse the object structure.

The evaluation frame is the most frequently used run-time entity in the virtual machine, though not the most visible entity. Evaluation frames cannot be described or accessed by programmers like patterns and objects. An evaluation frame is created by the virtual machine every time a computation is needed, e.g., when initializing an object, evaluating a method, or calculating the contributions of a virtual pattern. We have chosen to put the evaluation stack into the evaluation frame along with an array of temporals, a reference to the current evaluation context (a part object), the program counter, and a reference to the old evaluation frame. The usage of evaluation frames are similar to the activation records in Java or C.

Virtual patterns are not special run-time entities, they are just patterns with a slightly more complex construction semantics. Figure 7 shows the byte code instructions of the field initialization blocks for mainpart-2 and mainpart-3 from Fig. 3. These initialization blocks are used when the virtual machine executes line 10 in Fig. 3, where an object is created from the pattern in the field `ColorPoint`, and installed into the field `cp`.

The object created has two part objects. The first part object (2) has fields described in mainpart-2 and the second part object (3) has fields described in mainpart-3. After creating the object, the fields have to be initialized. The virtual machine starts by initializing the most general part object (2). In the left column of Fig. 7 the initialization byte codes for the three fields `x`, `y` and `move` are shown. The evaluation order of the byte code blocks are indicated by the roman numerals. The virtual machine will first initialize field `x`, then field `y` and then field `move`. The first two fields contain an integer object built in blocks I and II. The third field is the initial binding of the virtual pattern `move`. The byte codes for initializing field move, as seen in block III in Fig. 7, will push the empty pattern onto the evaluation stack and add one mixin to this pattern with the current evaluation part object as context and mainpart-4 from Fig. 3 as its static structure. This pattern is the initial pattern of the virtual pattern `move`. The next instruction evaluated by the virtual machine is the `next_virtual` instruction. It will search the more specific part objects for extensions to the virtual pattern, and if found it will create an evaluation frame to evaluate the code of the extension block. In the example in Fig. 3 there is a further binding of the virtual pattern in mainpart-3, so the virtual machine builds an evaluation frame, moves the initial pattern to the evaluation stack in the new evaluation frame, and begins the execution of the extension code. The extension code can be seen in block IV in Fig. 7. This code builds a small pattern and extends the initial pattern with the small pattern and leaves the result of the merge on the evaluation stack. The last instruction in the extension is the `next_virtual_return` instruction, which, like the `next_virtual` instruction, will search for more extensions to the virtual pattern. However, the behavior is slightly different: If the `next_virtual_return` instruction finds an extension, the evaluation frame is reused to execute the new extension code. If no extension is found, the pattern on the evaluation stack is moved to the old evaluation frame and the current evaluation frame used to build the extensions is removed. In the example from Fig. 3 there are no more extensions, so the evaluation frame is removed and the pattern is put back onto the evaluation stack of the old evaluation frame. Evaluation thereby returns to the `ins_ptn 2` instruction in block III. After initializing all fields in the first part object (2) the virtual machine will initialize the fields in the next part object (3). The field initialization blocks can be seen in the right column of Fig. 7. The first field to be initialized is field `c`, which contains a simple object created from the string pattern. The second field is the further binding of the virtual pattern `move`. The complete pattern is already built and installed into the third slot in part object 2. The instructions in block VI will traverse the object and lookup the pattern in the third field in part object (2). This pattern is then pushed to the evaluation stack and installed into the second field in the part object (3).

As seen in this little example the calculation of a virtual pattern is a complex matter. We will go through a larger example in section 4, where the dynamic process is shown again.

4. Example: Family Combination

In this section we will extend on the discussion about how the virtual machine constructs virtual patterns, and how the structures are related to one another at run-time. To illustrate this we will use the example from Fig. 2 throughout this section, and we will put special emphasis on line 25 in the example.

Before going into details about the example, we will give a short overview of the intention of the program in Fig. 2. The program expresses one super family of patterns called `Lang` and two derived families of this super family called `LangEval` and `LangPrint`. Obviously, this example is the expression problem, but it could be a real life example where a large company with several divisions had a common set of classes and each division had its own extensions of this set of classes. Each of the two sub-families are split into an interface extension part and an implementation extension part. The family `LangEval` extends the virtual pattern `Exp` with an `eval` method, and the family `LangEvalImpl`, which extends the `LangEval`, adds the implementation of the `eval` method. The families `LangPrint` and `LangPrintImpl` are built in a similar way.

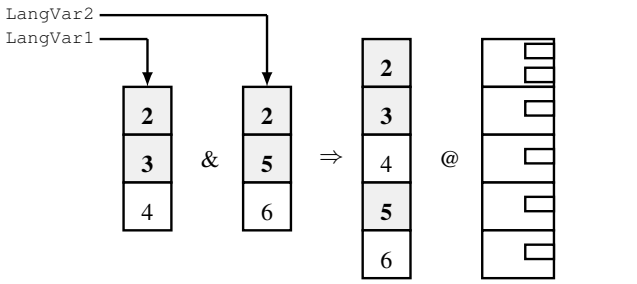

Figure 8. Diagram showing the pattern merge, where the numbers in each mixin correspond to the mainpart annotations in Fig. 2. The context pointer is the same for all mixins and is therefore omitted. The rightmost figure shows the structure of an object created as an instance of this pattern.

Gbeta patterns are first class values, so they can be referenced and passed around. In the example program at lines 18–19, we create two pattern references `LangVar1` and `LangVar2`. The reference `LangVar1` can point to a pattern of type `LangPrint` and the reference `LangVar2` can point to a pattern of type `LangEval`. It is not uncommon in gbeta programs to pass a reference to a pattern around. For instance, a method may accept a pattern as an argument, and use it as a superpattern in a method call, i.e., it may call the given method with a locally defined specialization. In this case we pass patterns around in variables in order to build a pattern dynamically. We assign the pattern `LangPrintImpl` to the pattern reference `LangVar1`, and `LangEvalImpl` to the pattern reference `LangVar2`. These are valid assignments because the patterns that are assigned are subtypes of the bounds of the references.

In the rest of this section we will focus on line 25 in the example in Fig. 2, where the pattern variables are used. All the relevant mainparts in the example have been given a number as usual.

The semantics of the declaration of the object `F` in line number 25 in Fig. 2 contains a series of smaller actions that need to be performed in order to initialize this field `F`. First, a pattern needs to be constructed from the merge between the pattern held in the pattern reference `LangVar1` and the pattern held in the pattern reference `LangVar2`. Second, from the constructed pattern a new object is created, and finally the new object must be initialized before it can be installed into the field `F`. The first two parts of the process is shown in Fig. 8, where the patterns held in the two pattern references are merged to a new pattern. We will not go into details about how patterns are merged as this is the subject of another paper [24]. Figure 8 shows the two patterns that are held in the pattern references, and the highlighted mixins are the statically known ones. If all mixins were highlighted the compiler would have been able to create a static pattern that could have been used instead of doing the dynamic merge. Doing the merge of these two patterns will result in the pattern shown in the middle of Fig. 8 and from that pattern, a new object is created. The newly created object has one part object per mixin, and the fields in each part object are marked as little boxes in the part object.

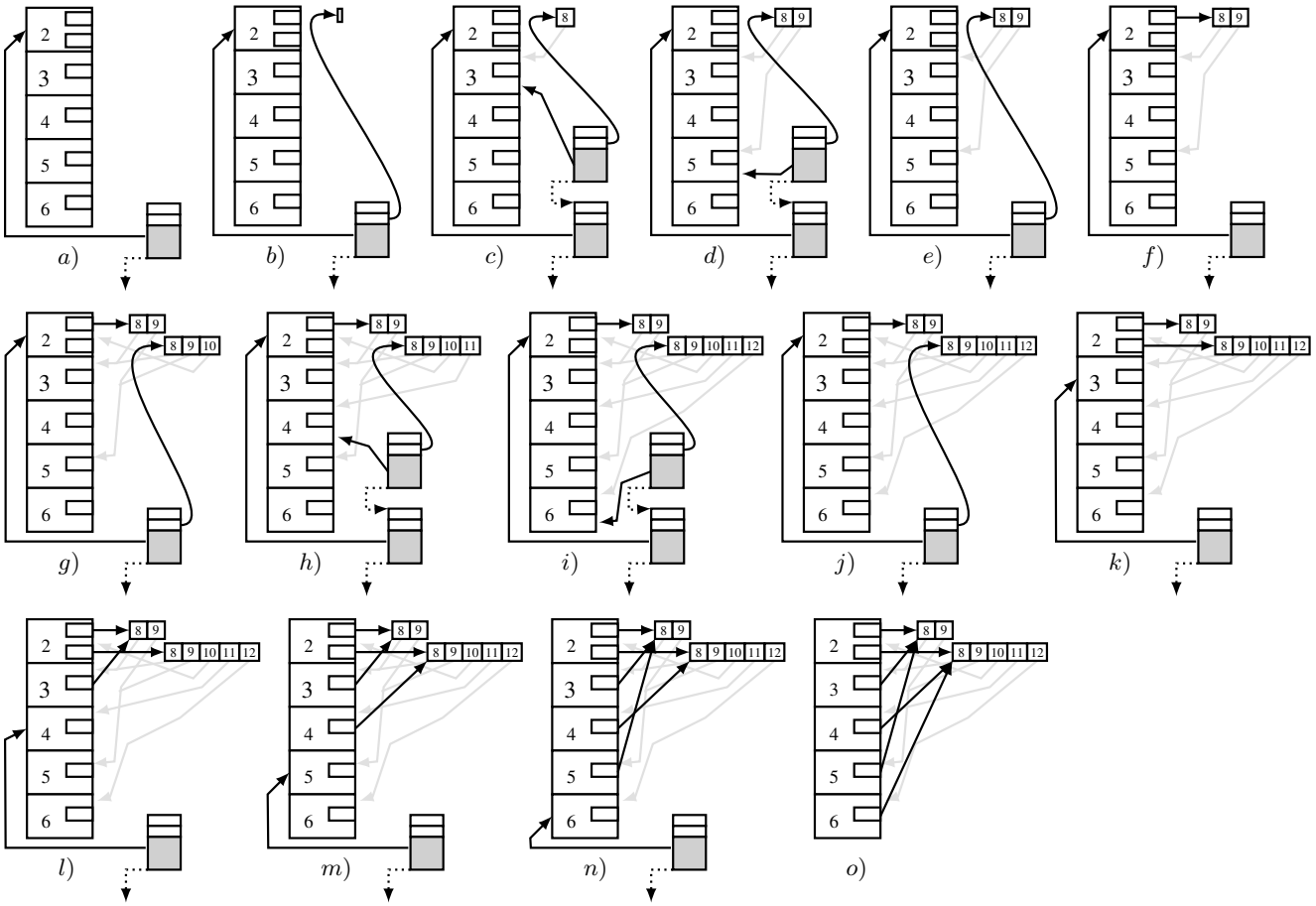

Figure 9. The figure shows the step-by-step process of initializing the object created in Fig. 8. The object has several part objects and the first part object contains the initial binding of two virtual patterns that are further bound in the more specific part objects.

Before the newly created object can be installed into the field `F`, it has to be initialized. We will now describe how the virtual machine will initialize this object. The entire process is shown in Fig. 9 and split up into steps from a) to o).

In step a) in Fig. 9 the initialization of the object is about to begin. The virtual machine has created an evaluation frame and the evaluation context is set to the most general part object marked by the mainpart id 2. The evaluation frame has a reference to its old evaluation frame, marked by the dashed line, in which the initialization of the field `F` is executing. At this point no initialization code for any of the fields has been executed yet.

The first field to be initialized is the field `Exp`. In step b) the initial pattern for the field `Exp` has been built and placed on the evaluation stack. This is the pattern `object`, which has zero mixins. After building the initial pattern, the virtual machine searches down the object to find any extensions to the virtual pattern `Exp`. In step c) the virtual machine has found an extension in the part object marked with mainpart id 3, has set up a new evaluation frame, and has extended the pattern with one mixin. This mixin has mainpart with id 8 (see Fig. 2 for reference) and it has a context pointer to the current evaluation context. The context pointer in the mixin is marked as a gray arrow going from the mixin to the part object in Fig. 9. After extending the pattern, the virtual machine again searches for more extensions to the virtual pattern `Exp`. In step d) the virtual machine has found another extension in the part object marked by mainpart id 5. This time, however, the evaluation frame is reused and only the context pointer is redirected to the new part object. The extension code is run and a new larger pattern in placed on the evaluation stack before the virtual machine again searches for more extensions. This time there are no more extensions in any of the more specific part object of the object, so the evaluation frame is removed and the pattern is moved to the evaluation stack in the old evaluation frame, as seen in step e). Evaluation has now returned to the field with the initial binding of the virtual pattern and the last instruction is to install the pattern from the evaluation stack into the first field of the part object marked with mainpart id 2. This finishes the initialization of the first field in the part object, and the state of the virtual machine is shown in step f).

In step g) the virtual machine has built the initial pattern of the second field, `Lit`. As field `Lit` has the pattern in field `Exp` as its super pattern, the pattern in field `Exp` is pushed onto the evaluation stack and a new pattern is created that extends it with one mixin. This is an example why it is important that virtual patterns are built completely from the initial binding. The pattern on the evaluation stack is the initial pattern of field `Lit` and the virtual machine will search for extensions in the more specific part objects. Like with the pattern for field `Exp` the virtual machine finds two extensions to the virtual pattern `Lit`. This is shown in steps h) and i). When there are no more extensions to the virtual pattern the construction is complete, and the evaluation frame can be removed. The pattern is then moved to the evaluation stack on the old evaluation frame, as

seen in step j). The complete pattern for the field `Lit` is now built, and in step k) it is installed into the second field of the first part object. All fields in the first part object are now initialized and the virtual machine will proceed with initializing the second part object marked with mainpart id 3. The field in the part object marked with mainpart id 3, is a further binding of the field `Exp`. The virtual machine then finds the complete pattern for the virtual pattern `Exp` in the first part object and installs the pattern into the field. This is shown in step l). In the steps m), n) and o) the remaining fields in the part objects are initialized and the complete object is shown in step o).

The two fields that were the initial bindings of the virtual patterns `Exp` and `Lit`, were the only places where the virtual machine had to do any construction of patterns. All the other fields are just aliases to the same patterns. However, the construction process involves the traversal of the entire object to find all extensions to a virtual pattern. Note that it is easy to generate code that gives rise to an object layout that does not duplicate fields, but there is a time/space trade-off because it is then necessary to navigate through more part objects in order to perform lookup. It is part of future work to further explore this trade-off.

5. Implementation Status

All the tools described in the paper including the gbeta compiler, the gbcc compiler and the gvm virtual machine are implemented and available for download [25]. The gbeta language and compiler has existed for over 10 years and is continuously further developed. The version of the compiler that is available for download above, is a snapshot of the current development towards the next stable release. The compiler is implemented in 82.000 lines of BETA code and features a new syntax together with many language extensions. The gbcc compiler and the gvm virtual machine are developed as part of the first author's PhD project. The gbcc compiler is implemented in 4200 lines of Python code and the gvm virtual machine is implemented in 6700 lines of C++ code. Along with the gvm virtual machine there are around 1000 lines of C++ test cases that exercise the various parts of the virtual machine. The gbcc compiler and the gvm virtual machine are able to handle almost all of the gbeta language, which includes the main features like pattern composition, virtual patterns, use of static pattern information, and more.

All the examples shown in the paper can also be found in the download above. There are descriptions in the download package on how to run the examples on a linux based system.

6. Related Work

Many aspects of related work were already discussed in section 2. As mentioned, CaesarJ [3, 22] and Object Teams [15] embody a similar notion of virtual classes as gbeta, though with restrictions derived from the decision to have only classes that can be computed at compile-time. A similar restriction applies to Scala [27, 2].

The language Newspeak [6], which is dynamically typed and in several ways related to the Smalltalk family of languages, allows classes to be features of objects and uses late binding to perform class lookup. This makes the classes virtual because they are late-bound, but there is no notion of deep mixin composition and no mechanism to ensure that an overriding definition of a class is actually a subclass. Hence, dynamic creation of classes is supported in Newspeak just like in other image based, dynamically typed environments, but it does not support the specialized features targeted at virtual classes. Newspeak actually uses a specialized version of the Squeak virtual machine, but the modified parts are concerned with foreign method calls, not classes.

7. Conclusion

We have described how virtual classes in full generality can be supported in a specialized virtual machine, the gvm. The reason why special support is needed is that general virtual classes require the construction of new classes at run-time through mixin composition, which conflicts with the design decisions behind main-stream virtual machines, such as the Java Virtual Machine and the .NET platform. Moreover, we have also shown how the dynamic class composition process continues recursively when an instance of a dynamic class is created, because of the computation of the nested virtual classes. This process is also known as deep mixin composition, and it is supported by the gvm. To the best of our knowledge, the gvm is the first virtual machine to support dynamic creation of virtual classes as an integrated part of the language semantics, thereby enabling full support for virtual classes.

Acknowledgments

We would like to thank the anonymous reviewers for valuable feedback and suggestions to make this a better paper.

References

[1] B. Alpern, C. R. Attanasio, A. Cocchi, S. F. Hummel, D. Lieber, M. Mergen, J. C. Shepherd, and S. Smith. Implementing jalapeno in java. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '99)*, pages 314–324. ACM Press, 1999.

[2] P. Altherr and V. Cremet. Inner classes and virtual types. Technical Report 2005013, EPFL, March 2005.

[3] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development I*, volume 3880/2006 of *Lecture Notes in Computer Science*, pages 135–173. Springer Berlin / Heidelberg, 2006.

[4] J. R. Bell. Threaded code. In R. Morris, editor, *Communications of the ACM*, volume 16. ACM, June 1973.

[5] B. Bobrow, D. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, and D. Moon. *Common Lisp Object System Specification*. Document 88-002R. X3J13, June 1988.

[6] G. Bracha. Executable grammars in Newspeak. *Electron. Notes Theor. Comput. Sci.*, 193:3–18, 2007.

[7] G. Bracha, P. Ahe, V. Bykov, Y. Kashai, W. Maddox, and E. Miranda. Objects as modules in Newspeak. Available at http://bracha.org/newspeak-modules.pdf, 2009.

[8] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño dynamic optimizing compiler for java, 1999.

[9] D. Clarke, S. Drossopoulou, J. Noble, and T. Wrigstad. Tribe: a simple virtual class calculus. In B. M. Barry and O. de Moor, editors, *Proceedings of the 6th International Conference on Aspect-Oriented Software Development, AOSD 2007, Vancouver, British Columbia, Canada, March 12-16, 2007*, volume 208 of *ACM International Conference Proceeding Series*, pages 121–134. ACM, 2007.

[10] E. Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark, 1999.

[11] E. Ernst, K. Ostermann, and W. R. Cook. A virtual class calculus. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 270–282, New York, NY, USA, 2006. ACM Press.

[12] M. Ertl and D. Gregg. The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism*, 5:1–25, 2003.

[13] D. S. Goldberg, R. B. Findler, and M. Flatt. Super and inner: together at last! *SIGPLAN Not.*, 39(10):116–129, 2004.

[14] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, May 2005.

[15] S. Herrmann. Object Teams: Improving modularity for crosscutting collaborations. In *PROCS. OF NET.OBJECTDAYS*, pages 248–264. Springer, 2002.

[16] S. Herrmann, S. Herrmann, C. Hundt, C. Hundt, K. Mehner, and K. Mehner. Translation polymorphism in Object Teams. Technical report, Technical University Berlin, 2004.

[17] R. Jones and R. Lins. *Garbage Collection - Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1996.

[18] B. B. Kristensen, O. L. Madsen, and B. Møller-Pedersen. The when, why and why not of the BETA programming language. In B. G. Ryder and B. Hailpern, editors, *HOPL*, pages 1–57. ACM, 2007.

[19] T. Lindholm and F. Yellin. *Java(TM) Virtual Machine Specification*. Prentice Hall PTR, 2 edition, 1999.

[20] O. L. Madsen and B. Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings OOPSLA'89, ACM SIGPLAN Notices*, volume 24, 10, pages 397–406, Oct. 1989.

[21] O. L. Madsen, K. Nygaard, and B. Møller-Pedersen. *Object-Oriented Programming in The* BETA *Programming Language*. Addison-Wesley, 1993.

[22] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 90–99, New York, NY, USA, 2003. ACM Press.

[23] A. B. Nielsen. Efficient virtual machine support for dynamic patterns. Master's thesis, Department of Computer Science, University of Aarhus, 2008.
Available at http://www.cs.au.dk/∼abachn/publications.html.

[24] A. B. Nielsen and E. Ernst. Optimizing dynamic class composition in a statically typed language. In R. F. Paige and B. Meyer, editors, *TOOLS EUROPE 2008: Objects, Components, Models and Patterns*, volume 11 of *Lecture Notes in Business Information Processing*, pages 161–177. Springer, June/July 2008. Available at http://www.cs.au.dk/∼abachn/publications.html.

[25] A. B. Nielsen and E. Ernst. Download of the gbeta compiler and virtual machine, 2009. Available at http://www.cs.au.dk/∼abachn/vmil-2009.tar.gz.

[26] N. Nystrom, X. Qi, and A. C. Myers. J&: nested intersection for scalable software composition. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 21–36, New York, NY, USA, 2006. ACM.

[27] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In L. Cardelli, editor, *ECOOP 2003 - Object-Oriented Programming, 17th European Conference, Darmstadt, Germany, July 21-25, 2003, Proceedings*, volume 2743 of *Lecture Notes in Computer Science*, pages 201–224. Springer, 2003.

[28] M. Torgersen. The expression problem revisited - four new solutions using generics. In *In Proceedings of the 18th European Conference on Object-Oriented Programming*, pages 123–143. Springer-Verlag, 2004.