Towards a DSAL for Object Layout in Virtual Machines

- Position Paper -

Stijn Timbermont
Vrije Universiteit Brussel, Belgium
stimberm@vub.ac.be

Bram Adams
Ghent University, Belgium
bram.adams@ugent.be

Michael Haupt
Hasso-Plattner-Institut, University of Potsdam, Germany
michael.haupt@hpi.uni-potsdam.de

Abstract

High-level language virtual machine implementations offer a challenging domain for modularization, not only because they are inherently complex, but also because efficiency is not likely to be traded for modularity. The central data structure used throughout the VM, the object layout, cannot be succinctly modularised by current aspect technology, as provisions for static crosscutting are not fine-grained enough. This position paper motivates the need for a declarative, domain-specific language for handling the tangled object layout concern. Based on observations in real-world VM implementations, we propose such a language, D4OL. It combines a two-level layout mapping, constraints and an engine to divide responsibilities between VM component and VM developers. We consider a domain-specific language like D4OL a necessary complement to behavioural aspect languages in order to modularize VM implementations.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques

Keywords Object Layout, Virtual Machine, Domain-Specific Aspect Language

1. Introduction

High-level language virtual machine (VM) implementations [11] are inherently complex systems, as they provide a wide range of highly interacting abstractions to applications running on top of them. For example, a Java VM provides both automatic memory management and integrated support for multithreading and synchronization such that the Java programmer does not have to take care of possible interactions between them.

The resulting high complexity in VMs is a consequence of the high degree of crosscutting interactions between the different parts of the VM. Tackling these crosscutting concerns by using aspect-oriented programming in the VM is a relatively recent idea, and so is the idea to define a domain-specific aspect language for VM implementations [2, 6]. In this position paper we focus on a particular concern of a virtual machine: the object layout. This is crucial to the VM as it serves as the foundation on which other functionality is built, such as the execution system (interpreter or JIT compiler), automatic memory management (garbage collection), multithreading facilities, etc. Note that this does not only apply to the implementation of object-oriented programming languages; arrays or structures can also be considered as "objects".

We identify three object layout characteristics which, taken together, prevent existing approaches from modularising the object layout. Firstly, the object layout is a highly tangled concern, because it combines contributions of several subsystems of the VM. Secondly, the object layout is a very *structural* concern (as opposed to behavioural), as it simply describes how the concepts provided by the VM are mapped onto the memory of the host machine. Thirdly, in order to make the VM memory efficient, there are a lot of optimizations that can make the object layout implementation much more complicated than its conceptual functionality. We further discuss these characteristics in Section 2.

From these three problems, we distill three requirements (Section 3) for a domain-specific language for object layout and the corresponding interactions between the different subsystems. D4OL, a DSAL for object layout, (Section 4) is an incarnation of these requirements. It enforces distinct roles for the people involved with the VM implementation and provides (semi-)automatic support to refine the layout specification. Section 5 discusses our approach, while we summarize our contributions in Section 6.

2. Object and Layout in Virtual Machines

In this section we explain the three issues which are characteristic for the interactions between object layout and other concerns.

2.1 Object layout is tangled

The object layout plays a central role in a VM implementation: Almost every subsystem uses (i.e. depends on) it. This is not necessarily problematic for modularity, because the object layout could still provide a clean interface using standard functional abstractions. However, the fact that several subsystems also *contribute* to the object layout is problematic. Here are some examples:

- A mark & sweep garbage collector [7] must be able to tell the difference between live objects and garbage. The marking phase traverses the object graph and marks every live object. The sweeping phase traverses the entire heap and adds unmarked objects - garbage - to the free-list. To implement this, every object should reserve (at least) a single bit to keep track whether or not it has been marked as live data.
- A reference counting garbage collector [7] keeps track of the number of references to each object. Every time there is a new reference to an object, the reference count of that object should be incremented. If a reference disappears, the reference count must be decremented. If the reference count reaches zero, the

Workshop DSAL '08 April 1st, Brussels, Belgium.

object is garbage. To implement this, each object must reserve some room to keep the reference count.

- Programming languages which integrate support for concurrency into the object model may require that each object carries some information about its locking state.
- The execution environment (or more generally, the language specification) may require that every object has a hash code.

The object layout contains information about different subsystems and hence breaks abstraction boundaries. It reflects for example the choice of garbage collector such that changing the latter is in general not possible without changing the object layout.

The contributions of several subsystems to the object layout are not always independent: whether or not it is actually necessary to include a hash code in every object depends on the garbage collector. If objects are never moved around, the memory address of an object serves as a good hash code; otherwise, the memory address of an object may change at any point during program execution and is therefore not a good hash code.

2.2 Object layout is a highly structural concern

Rather than defining a lot of behavior and algorithms, the object layout simply describes how the concepts provided by the VM are mapped onto the memory of the host machine. The implementation often consists solely of type definitions with the appropriate getters and setters. Even if the latter involves low-level operations such as bit-shift operations, their logic is rather simple.

The problem is that the data definitions contain elements corresponding to different subsystems in the VM. Several design decisions, for example the choice of garbage collection scheme, are encoded in the type definitions. These cannot be separated with current AOP techniques because these are not geared for such structural concerns.

2.3 Object layout is critical for VM efficiency

The design of an object model for a particular programming language is not an easy task. It has to fulfill the requirements of all the subsystems in the VM, especially the execution system. Some OO programming languages such as Smalltalk [3], prescribe that everything is an object, even numbers. However, actually representing a number as a reference to an object with a header and a class pointer would be extremely memory inefficient. Other dynamically typed languages also have this problem. A common trick to solve this is to use the low-order bit in an object reference as a type tag: if the bit is "0", the value is an actual object reference, but if it is "1", the rest of the value is an immediate signed integer (not on the heap).

The object header is another part of the object layout that must be carefully designed. It contains all the per-object information (e.g. type, size, a mark bit for a mark & sweep garbage collector, etc), which is usually (not always) stored in the object header itself. There are systems [9] that store the mark bit in a so called bitmap. The advantage is that in order to know whether an object is marked, it is not necessary to actually load the object. This approach has the additional benefit that the bitmap is completely separated from the rest of the object layout contributions. However, since there is an additional overhead, this approach may be undesirable for performance reasons.

2.4 Summary

It is clear that the object layout plays a central role in a VM implementation. Based on performance criteria, an implementation approach where each subsystem keeps track of its own contributions might be undesirable; however, an implementation approach that combines all contributions yields a highly complex and monolithic object layout implementation that is difficult to program and to evolve because of structural concern tangling.

3. DSAL requirements

From the three characteristics of the previous section, we distill a number of expected benefits for a domain-specific language geared towards composition of a VM's object layout. A first requirement is modular reasoning. The fact that the object layout contains contributions from different concerns worsens the understanding and independent evolvability of the various VM subcomponents. Developers should be able to declaratively state their object layout requirements independent from the other components in the VM. This tackles the tangling and (partially) structural object layout characteristics.

To obtain an efficient implementation, fine-grained control is needed over the composition of a particular VM's object layout from the various components' requirements. Common object layout and data structure knowledge should be combined with specialised, human expertise to achieve the most optimal object representation satisfying the different components' needs. We encourage a declarative, (semi-)automatic approach for this. The former enables explicit documentation of design choices for the object layout, whereas tool support lets the object layout expert focus on the constraints for a given VM. In addition, it becomes easy to check whether changes in VM components have a positive or negative impact on the object layout, or to experiment with small variations in constraints.

Finally, the resulting object layout code (type definitions and getters/setters) should be generated automatically, as this is quite tedious to implement for each successfully specified object layout. The declarative composition specifications carry enough information to drive this code generation.

4. A DSAL for Object Layout - D4OL

Our proposed approach is depicted in Figure 1. We organise the object layout implementation in two layers:

High-level description Each module (e.g. a reference count collector, a mark & sweep collector, a bytecode interpreter, a JIT compiler) has its own high-level description of the object layout, i.e. describes what it expects from the object layout. Irrelevant information is - by definition - not present. By consequence, a particular high-level object layout description that belongs to a certain module is inherently incomplete: it only contains the contributions of that module to the object layout, of which some are shared with other modules. The high-level description serves more as an *interface* to the eventual object layout than as an actual implementation. It describes what information the module requires from the data objects it operates on. A key idea for this first layer is that there is abstraction from any particular VM.

Low-level mapping The goal of this layer is to describe how to map the incomplete and high-level object layout description onto actual memory. Key to this layer is that there is less abstraction: it is specific to a particular VM. However, there is still modularization. Each module has its own low-level mapping, but the decisions that are made in the mapping may be based on knowledge about the entire VM (e.g. the language that is implemented by the VM might influence certain choices in the memory management object layout). Finally, the different views on the object layout should be merged into one complete implementation of it, starting from the high-level descriptions and the corresponding low-level mappings. It is exactly this step we are trying to automate. However, in order to do so, some in-

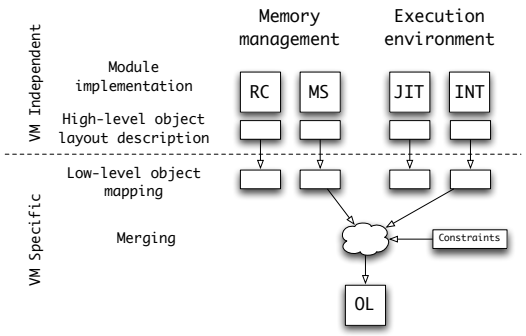

Figure 1. Visualisation of a (partial) VM architecture using D4OL for expressing module specific contributions to the object layout.

formation should be provided that indicates which parts of the different modular definitions refer to the same entity and which do not. This information is provided as a set of constraints between different object layout definitions.

We further explain these layers in the following sections, in terms of the example of Figure 1.

4.1 High-level object layout description

Suppose we are building a VM for Smalltalk [3]. As garbage collector, we use mark & sweep. The execution system consists of a bytecode interpreter. These two subsystems have their own view on the object layout. During the mark phase, the heap is traversed in order to look for live data. During the sweep phase, every object not encountered during the mark phase is collected. For the object layout, this means that each object can be either "free" or "marked". Furthermore, the mark phase must be able to traverse an object in order to find other live data. The interpreter has a rather different view: in Smalltalk each value is an object, instance of a particular class. We would like to make an exception for integers, in order to reduce overhead, but neither the interpreter nor the garbage collector should not have to care about this.

The high-level description should be as declarative as possible, and should only contain that information which is actually required by the module at hand. Inspired by algebraic data types, the code for expressing the high-level description for the mark & sweep view on the object layout might look as follows:

```
data Object = FreeObject [Reference]
            | MarkedObject
type Reference = PointerTo Object
```

This definition:

- simply states that an object is either a marked object or a free object, and that a free object has a collection of references to other objects which might be traversed;
- does *not* say that an object can only have references to objects and not integers or other data; it just states that the mark & sweep module is only interested in object references; similarly, a marked object still has object references, but they are not needed anymore by the garbage collector;
- does *not* specify how the distinction between a free and a marked object should be made: with an extra bit inside the object or with a separate data structure which keeps track of this.

The interpreter's view might be expressed as follows:

```
data Object = Object ClassPointer [Value]
type Value = Reference
```

This definition says that each value is a reference to an object that contains a class pointer and some fields (which are values). However, it does not specify that integers should be treated differently, because the interpreter does not care about this.

4.2 Low-level mapping

Before we try to combine these independent, incomplete and rather different definitions, we add the refinement that integers should be treated differently. This should be possible with the same degree of declarativeness as the high-level description:

```
data Value = Reference Address
           | Number Int31
type Address = Word31
type ClassPointer = Address
```

Note that we specify the amount of space used for the address of an object reference or for the immediate data in the number, namely 31 bits. This is not surprising, as we need one bit to tag the value. Because this impacts the valid range of integers, this knowledge is included in the specification of the VM. We also specify that the class pointer should consist of 31 bits. This is a good example of the difference between the high-level descriptions and the low-level mappings: the bit that becomes available in the interpreter object layout is used as "mark bit" for the mark & sweep object layout.

4.3 Merging

Now it is time to combine the different definitions. This is automated as much as possible. The final result is an actual implementation for the object layout which fulfills the view of every subsystem of the VM. There are two main steps:

- the high-level constructs used in the definitions should be automatically converted to more concrete, lower level constructs (not to be confused with the low-level mapping described earlier). E.g. the "|" operator can be translated in an extra bit that serves as a tag to distinguish between the two cases. In general, this step deals with the high declarativeness of D4OL.
- merging the separate "modular" definitions into one "tangled" definition. However, in order to allow this, we have to specify which parts of the different definitions are shared between modules, and which parts are unique. In other words, we express constraints which resolve accidental name clashes or differences in the high-level descriptions. As long as these are satisfied, the generator is free to for example arrange fields or bits differently. In general, this step deals with the tangling in the object layout.

The following code shows the constraints for merging the object layout definitions of mark & sweep and the interpreter,

```
MS.Object = INT.Object
MS.Object.[Reference] = INT.Object.[Value] where
    Value @ (INT.Reference INT.Address)
```

Again in a highly declarative fashion, it says that the definitions for `Object` in the mark & sweep module and the interpreter module refer to the same entity, and that the set of references in the mark & sweep version of `Object` corresponds to the set of values in the interpreter definition. The `@` operator should be read as "matches with" and serves as a kind of filter: the set of references in `MS.Object` consists of those values in `INT.Object` that are actual object references and not numbers.

Based on the high-level descriptions, the low-level mappings and the merging constraints, it should be possible to obtain the object layout implementation. Both "|" operators can be translated into an additional bit, based on the low-level information that certain data only uses 31 bits. By specifying which definitions correspond, we can merge the definitions and generate a tangled object layout implementation starting from modular definitions. The object layout which the D4OL engine should produce might look:

```
data Object = Object Word31 Bit [Value]
data Value = Value Word31 Bit
```

This definition is clearly tangled: the definition of `Object` contains both a 31-bit word for the class pointer, something that was required by the interpreter, and a single bit to distinguish between free and marked objects, something that was required by the mark & sweep object layout. This definition is also more low-level, as a value consists of a 31-bit word together with a single bit to distinguish between immediate integers and actual object references. In other words, the D4OL engine helps us with building a concrete object layout, starting from high-level descriptions and some additional information on how these descriptions should be mapped.

4.4 Integration with base language

The final point to discuss is how the generated code should look like, or how the modules that use the object layout can actually be implemented. Recall the high-level object layout description for mark & sweep: the only available information is that a reference is a pointer to an object and that an object is either a free object with a set of references, or a marked object. As this is the only information available to the mark & sweep module, the generated code should provide operations that correspond to this high-level definition. The following code snippet (in C) illustrates some of the available constructs:

```
void mark() {
  Reference current;
  Object obj;
  ...
  obj = * current;
  free_or_marked(obj, {   // The object is free
    iterate(references, {
      ... reference ... } );
    }, {                  // The object is marked
    ... } );
}
```

Obtaining an object through a reference is done with the dereference operator. Distinguishing a free from a marked object is possible by means of the `free_or_marked` macro, which takes an object and two code blocks. If the object is free, the first block is executed in the context of an extra variable `references`; if marked, the second block is executed. The references can be traversed with the `iterate` macro. Another possibility is to provide an iterator, with corresponding functions `hasNext` and `next`. The generated code heavily uses typedefs and macros in order to provide the desired interface. We expect the compiler to be able to remove any unnecessary data accesses introduced by these constructs.

5. Discussion

Is This AOP? Aspect-oriented programming [8] is widely equated with its *pointcut-and-advice* (PA) flavour [10]. We deliberately adopt the wider notion that AOP is about modularizing crosscutting concerns. We are applying aspect-oriented abstraction to the domain of VM implementations, or, rather to its sub-domain of object model implementations.

The work described herein is but a small part of a definitely more large-scale effort with the goal of supporting modularization in VM implementations in general [2, 6, 5]. At various levels of abstractions, various AOP mechanisms are employed.

Related Approaches The approach we present here seems to be related to subject-oriented programming [4] and open classes [1]. Like in subject-oriented programming, we take the approach where each module defines its own view on the object layout in a decentralized way: there is no common base class or base object layout. In open classes, this is the case.

6. Conclusion

In this position paper we motivate the need for a declarative and domain-specific language for object layout in VMs, as this is a tangled and structural concern. The language should enable modular reasoning about the object layout for the different subsystems of the VM without sacrificing performance. We present an architecture where a VM independent layer, a VM specific layer and a set of constraints allow the (semi-)automatic generation of the optimal object layout implementation which fulfills the object layout interfaces for all VM subsystems.

References

[1] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. Multijava: modular open classes and symmetric multiple dispatch for java. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 130–145, New York, NY, USA, 2000. ACM.

[2] Yvonne Coady, Celina Gibbs, Michael Haupt, Jan Vitek, and Hiroshi Yamauchi. Towards a domain-specific aspect language for virtual machines. First Domain-Specific Aspect Languages Workshop, October 2006.

[3] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[4] William Harrison and Harold Ossher. Subject-oriented programming: a critique of pure objects. *SIGPLAN Not.*, 28(10):411–428, 1993.

[5] Michael Haupt, Bram Adams, Stijn Timbermont, Celina Gibbs, Yvonne Coady, and Robert Hirschfeld. Disentangling virtual machine architecture. Extended version of [6], currently under review, 2007.

[6] Michael Haupt, Celina Gibbs, and Yvonne Coady. Disentangling virtual machine architecture. 4th Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT). Co-located with ECOOP 2007, Berlin, Germany, July 2007.

[7] Richard Jones and Rafael Lins. *Garbage collection: algorithms for automatic dynamic memory management*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[8] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[9] YoungMin Kwon, Sameer Sundresh, Kirill Mechitov, and Gul Agha. Actornet: an actor platform for wireless sensor networks. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1297–1300, New York, NY, USA, 2006. ACM Press.

[10] H. Masuhara and G. Kiczales. Modeling Crosscutting Aspect-Oriented Mechanisms. In *Proc. ECOOP 2003*, 2003.

[11] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.