ALIA4J's [(Just-In-Time) Compile-Time] MOP for Advanced Dispatching

Christoph Bockisch
Software Engineering group
University of Twente
Enschede, The Netherlands
c.m.bockisch@cs.utwente.nl

Andreas Sewe
CASED
Technische Universität Darmstadt,
Gernamy
sewe@st.informatik.tu-darmstadt.de

Martin Zandberg
Software Engineering group
University of Twente
Enschede, The Netherlands
m.d.zandberg@student.utwente.nl

Abstract

The ALIA4J approach provides a framework for implementing execution environments with support for advanced dispatching as found, e.g., in aspect-oriented or predicate dispatching languages. It also defines an extensible meta-model acting as intermediate representation for dispatching declarations, e.g., pointcut-advice or predicate methods. From the intermediate representation of all dispatch declarations in the program, the framework derives an execution model for which ALIA4J specifies an generic execution strategy. The meta-object protocol (MOP) formed by the meta-model and the framework are defined such that new programming language concepts can be implemented modularly. Thereby the semantics can be implemented (1) in an interpretative style (e.g., using reflection) or by describing (2) how to generate corresponding Java bytecode or even (3) machine code. In the latter two cases, the implementation can reason about the current code generation context; this enables sophisticated optimizations. We discuss this optimization facilities by means of two case studies.

1. Introduction

When developing a new (domain-specific) language, initially its syntax and semantics are the primary concern and consequently subject to much experimentation. For this task, a language *designer* is required who is an expert in the business domain to which the language is going to be applied. Furthermore, flexibility in the implementation of the semantics is required to adapt initial designs. For this purpose, an interpreter-based approach to language implementation is most suitable. Once the language design becomes stable, however, performance issues come into focus. Best performance results are typically achieved by compilation-based approaches. But implementing a compiler requires detailed knowledge about the target platform and is typically done by a language *implementer*, a role quite different from that of the language designer. To minimize the effort for both language designers and implementers and to ensure that the semantics of the language with optimizations are the same, an approach for transitioning from the initial design to the final implementation is needed.

In this paper, we propose such an approach for languages based on dispatching, a language feature that has seen much experimentation by language designers in recent years, ranging from predicate dispatching [7] over aspect-oriented programming [12] and context-oriented programming [10] to various domain-specific languages. The *ALIA4J* project[1] therefore provides an approach for implementing such programming languages. As demonstrated in our previous work [5], *advanced dispatching* is a mechanism that subsumes the various styles of dispatching mentioned above.

To make advanced dispatching implementable in a flexible fashion, we employ a meta-object protocol (MOP) together with a generic meta-model of dispatching declarations implemented as abstract Java classes. To define a new language, language designers have to refine this meta-model to include specific concepts of interest; all that is required is extending a handful of Java classes.

In ALIA4J, there are three ways of implementing such a meta-model refinement, each operating on a different level of abstraction.

1. The most abstract way is implementing a plain Java method realizing the semantics in terms of interpretation. ALIA4J's runtime *MOP* passes required input as arguments to that method and handles the output provided as the method's result value. Refinement implementations at this level allow for easy experimentation; thus, this is the level at which language designers will commonly work.
2. More control over the generated code is gained by implementing a Java method that compiles the concept to Java bytecode. ALIA4J's *compile-time MOP* ensures that the required input is provided to the generated bytecode on the operand stack and expects the output on the stack as well. Refinement implementations at this level allow for context-dependent bytecode code generation; this allows language implementers to improve runtime performance in a portable way.
3. The most control is gained by implementing a method that compiles the concept to machine code. ALIA4J's *just-in-time (JIT) compile-time MOP* ensures that the required input and output is provided to respectively collected from the generated machine code in defined memory or register locations. Refinement implementations at this level can make use of all VM internals; while losing platform-independence, this allows language implementers to strive for optimal runtime performance.

Any of these three strategies can be implemented in the class representing the language concept in our meta-model. Thus, the implementation of a concept's semantics *and* optimization is modular.

[1] The Advanced-dispatching Language-Implement Architecture for Java. See `http://www.alia4j.org/`.

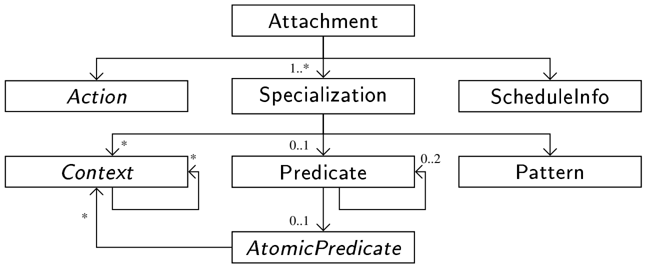

Figure 1. Entities of the Language-Independent Advanced-dispatching Meta-Model (LIAM) as UML class diagram.

Implementations of different strategies can even co-exist; which strategy is used can be chosen at runtime. This is very useful to language implementers who can use the—less efficient but by definition correct—implementation produced by the language designer as a test oracle.

In this paper, we illustrate the three levels of implementing the semantics of language concepts in ALIA4J in two case studies. We furthermore contrast ALIA4J's three-level approach with related work that targets a single level only.

2. A Runtime MOP for Advanced-Dispatching

As outlined in the introduction, we have implemented our approach of implementing programming languages in terms of the ALIA4J project. At its core, ALIA4J contains a meta-model of advanced dispatching declarations, called *LIAM*,[2] and a framework for execution environments that handle these declarations, called *FIAL*.[3]

LIAM defines *categories* of concepts relevant for dispatching and how these concepts can interact. For example, dispatch may be ruled by *atomic predicates* which depend on values in the dynamic *context* of the dispatch. When mapping the concrete advanced-dispatching concepts of an actual programming languages to it, LIAMhas to be *refined*.

Figure 1 shows the meta-entities of LIAM, discussed in detail elsewhere [4, 5], which capture the core concepts underlying the various dispatching mechanisms. They are implemented as abstract classes. The meta-entities *Action*, *Atomic Predicate* and *Context* must be refined to concrete concepts. Refinement is achieved by inheriting from the corresponding abstract class and implementing a so-called "compute" method that realizes the concrete concept's semantics in an interpretative way, e.g., using reflection.

An *Attachment* corresponds to a unit of dispatch, roughly corresponding to a pointcut-advice pair or to a predicate method. An *Action* specifies functionality that may be executed as the result of dispatch, e.g., the body of an advice or predicate method. A *Specialization* defines static and dynamic properties of state on which dispatch depends. A *Pattern* specifies syntactic and lexical properties of the dispatch site. The *Predicate* and *Atomic Predicate* entities model conditions on the dynamic state a dispatch depends on. The *Context* entities model access to values like the called object or argument values. The *Schedule Information* models the time relative to a join point when the action should be executed, i.e., the notions of "before," "after," or "around" familiar from aspect-oriented programming.

Figure 2 shows the relation between ALIA4J's components in more detail. We will now briefly discuss their interaction by discussing the flow of compilation and execution of applications on top of an ALIA4J-based language implementation: First, the compiler starts processing the application's source code and outputs a model ① for the advanced dispatching declarations in the program based on the refined subclasses ② of the LIAM meta-entities ③. Moreover, the compiler produces Java bytecode ④ for the program parts not using advanced dispatching. Then, at runtime, both, the program's model of dispatching declarations and bytecode are passed to a concrete FIAL instantiation ⑤ and subsequently handled by the FIAL framework itself ⑥.

[2] The Language-Independent Advanced-dispatching Meta-model. See `http://www.alia4j.org/alia4j-liam/`.

[3] The Framework for Implementing Advanced-dispatching Languages. See `http://www.alia4j.org/alia4j-fial/`.

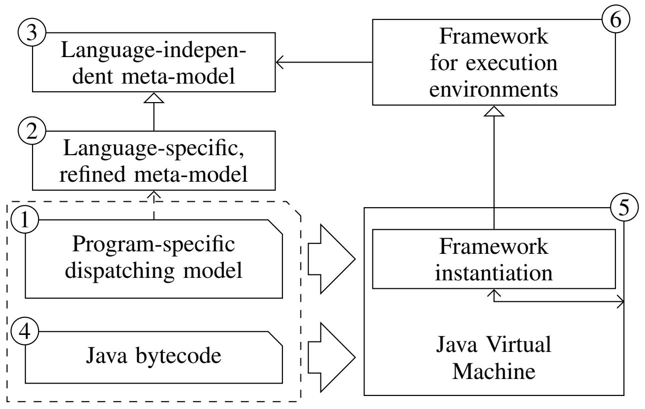
3
Language-independent meta-model
6
Framework for execution environments
2
Language-specific, refined meta-model
1
Program-specific dispatching model
5
Framework instantiation
4
Java bytecode
Java Virtual Machine

Figure 2. Overview of the application life cycle in ALIA4J-based language implementations.

By transforming the advanced dispatching declarations, FIAL generates an execution model for each dispatch site (join-point shadow in aspect-oriented terminology) in the program, containing the model entities which are refinements of Context, Atomic Predicate, and Action. For simplicity, they are called LIAM entities or just entities throughout this paper. ALIA4J's MOP defines the control and data flow depending on the results of evaluating the LIAM entities. The evaluation protocol for an individual entity is as follows:

1. Retrieve the Java object representing the LIAM entity.
2. If the entity depends on context values, first perform these steps for all required contexts.
3. Invoke the "compute" method on the entity object passing the values retrieved in the previous step as arguments.
4. The value returned in the previous step is the result value of this entity.

3. From Runtime MOP to (Just-in-Time) Compile-Time MOP

A FIAL instantiation is realized as an extension of an existing Java Virtual Machine. For the purpose of this paper, its most important functionality is to execute the declarative execution model derived by the FIAL framework, either directly by interpreting the execution model or indirectly by generating bytecode or machine code for it. When bytecode or machine code is generated, the FIAL instantiation may generate a call to the "compute" method (step 3 in the above protocol). To make use of ALIA4J's (JIT) compile-time MOP, the FIAL instantiation can alternatively hand over control over code generation to the LIAM entity. We illustrate the implementation of this protocol by the example of two FIAL instantiations in the following subsections.

3.1 The Compile-Time MOP in SiRIn

SiRIn[4] is a fully portable FIAL instantiation implemented using a Java 6 agent; it does not require a native component. SiRIn wraps every dispatch site into a special method and generates bytecode for these "reified" dispatch sites using the ASM bytecode engineering library.[5] Each wrapper method contains code derived from the execution model.

For many entities more efficient alternatives to reflection exist, e.g., to load a value from the local execution context. Especially for those entities that participate in regular, non-advanced dispatch, such bytecode generation strategies are obviously more efficient than a reflectively implemented "compute" method. In fact, often bytecode can be generated which is equivalent to code a Java compiler would generate for the selfsame functionality.

The bytecode building method gets passed a description of the syntactic context in which the dispatch takes place. This can be used by the implementation to choose between different code generation strategies.

3.2 The (JIT) Compile-Time MOP in Steamloom$^{\mathrm{ALIA}}$

Steamloom$^{\mathrm{ALIA}}$ is a FIAL-based execution environment built on the Jikes Research Virtual Machine (RVM) [1], a high-performance Java VM. As such, it is a re-design of an earlier execution environment, the Steamloom VM [3], built among others by the first author. Unlike its predecessor but like SiRIn, Steamloom$^{\mathrm{ALIA}}$ wraps call-sites into a special method-like construct. As these "reified" call-sites are completely under the control of the execution environment, generating code for them is straight-forward. Steamloom$^{\mathrm{ALIA}}$ thus does not employ a bytecode engineering library; rather, we implemented a light-weight code generation engine ourselves.

Steamloom$^{\mathrm{ALIA}}$ uses two JIT compilers: The *baseline* compiler is very fast but produces un-optimized machine code only. In contrast, the *optimizing* compiler is slow but produces highly optimized machine code. Depending on the configuration, Jikes RVM may choose between these compilers for each method separately depending on the estimated performance impact of the method. Beyond the ability to generate specific bytecode that realizes the semantics of a LIAM entity, Steamloom$^{\mathrm{ALIA}}$ also offers the ability to tailor the machine-code generation of both JIT compilers.

The machine-code building method gets passed a description of the current JIT compilation context. This includes all the syntactic information about the currently compiled dispatch. Moreover, in the case of the optimizing compiler, the compilation context may contain additional information. When the optimizing compiler, for instance inlines a method not only is information about the immediate caller of the dispatch available, but also about part of the chain of calls leading to the dispatch in question.

4. Case Study: Compile-Time MOP

Our implementation of ALIA4J contains more than 70 refinements of Context, Atomic Predicate and Action; this gives rise to numerous re-use opportunities [5]. For all refinements we provide an interpretative implementation, using ALIA4J's runtime MOP facilities. Many of the entities reflect either runtime values from the local context of a dispatch or primitive arithmetic and logical operations.

As an example, consider the dispatch of a method invocation. Because dispatch is logically wrapped in *site* methods in ALIA4J, values like the receiver (or *callee*) object as well as the argument values are available as local variables. Thus, the bytecode-support version of the *CalleeContext* can simply emit a bytecode instruction loading the corresponding variable. In case the called method is static and there is no receiver, the bytecode generation emits the instruction for producing the constant **null**. Since the generation method gets a description of the dispatch site, it can recognize whether the called method is static or not and can produce the correct instruction. The distinction is made at bytecode-generation time; the resulting bytecode does not contain any conditional instructions.

5. Case Study: JIT Compile-Time MOP

An example of languages that can be realized with the ALIA4J approach are aspect-oriented languages. In AspectJ [11], for instance, *aspects* are types and extend the concept of a class in several ways: Besides conventional members like methods and fields, an aspect can contain *pointcut-advice* pairs. A pointcut selects events at which the advice functionality must be executed, with advice containing statements like methods. But advice are invoked implicitly in contrast to the explicitly-invoked methods. Just like methods, advice must be executed in the context of an instance of their defining type. When a pointcut matches—or rather when dispatch decides to perform an advice action—an appropriate aspect instance has therefore to be discovered in order to invoke the advice on it. Different strategies exist to discover such an instance.

In ALIA4J, a pointcut-advice pair can be defined as an Attachment, where the Action corresponds to advice and the Specializations correspond to pointcuts. The strategies for discovering aspect instances are represented by a Context entity.

5.1 Semantics of the PerTupleContext

We have devised a generalized model that covers most of the instantiation strategies used in present aspect-oriented languages and implemented it as a refinement of Context: the so-called *PerTupleContext*. A LIAM Attachment may specify that a PerTupleContext provides the receiver object to the Action, which then simply calls a method corresponding to the advice functionality.

The PerTupleContext specifies a list of Contexts which are relevant for discovering the desired aspect instance: When it is evaluated, a tuple containing the values to which the required Contexts have been evaluated is passed to the PerTupleContext. Based on these inputs, the PerTupleContext discovers the appropriate aspect instance, whereby it can operate in one of two modes: In the *explicit* instantiation mode, it may only return an instance when there is already one associated with the input tuple. In the *implicit* instantiation mode, it may either return an instance already associated or, if no association exists so far, create a new aspect instance. In the latter case, the newly created instance is associated with the input tuple and returned the next time the PerTupleContext is evaluated with the same input.

The associations are stored in a table, not unlike a relational database. The values from the execution context are stored in the first columns and the associated instances are stored in the last column. Looking up an instance in this table is also called *performing a query*, whereby we again distinguish two cases: First, we can perform *exact queries*, which means that an input value is provided for each column. As a consequence, an exact query can yield either zero or one result. Second, we can perform *range queries*, where we can pass wildcards for columns potentially leading to multiple results.[6]

[4] The Site-based Reference Implementation. See `http://www.alia4j.org/alia4j-sirin`.

[5] See `http://asm.ow2.org/`.

[6] Above it was explained that an aspect instance acts as the receiver of a method call, but the PerTupleContext can evaluate to zero or more instances. Therefore, the result is passed to a special Action which iterates over the provided instances and invokes the actual advice functionality on each of them.

This general model of aspect instantiation is motivated by—but more general than—*association aspects* [14], whose purpose is to support behavioral relations amongst a group of objects. It can be used to, for example, keep the state between a group of objects synchronized. In our unified model, association aspects correspond to explicit instantiation with range queries. Our model is also able to support the less complex instantiation strategies of AspectJ. The isSingleton strategy, for example, always uses the same aspect instance throughout the entire program execution. Thus, the instance does not depend on the execution state. In terms of our model, this can be expressed using implicit instantiation and an empty tuple of required Contexts. The perThis and perTarget strategies provide separate aspect instances for each caller respectively callee object at a join point. They can be expressed by implicit instantiation with a 1-tuple of required Contexts. The tuple then consists either of the CallerContext or the CalleeContext.

5.2 Optimizations: An Outlook

Above we have described the semantics of the PerTupleContext and have outlined its plain Java implementation. In this section, we outline possible optimizations that are specifically possible when generating machine code. For our optimizations, four properties of the associations of a PerTupleContext are relevant:

1. Is the tuple size of required input values zero or more? In the former case, the query does not depend on input, and we call the PerTupleContext *context insensitive*; otherwise it is called *context sensitive*.
2. Are the associated values instantiated *explicitly* or *implicitly*?
3. Are the associations fixed, or may new associations be added in the future? Here, it is required that the programmer marks a PerTupleContext explicitly as *non-changing* after an initialization phase.
4. Is the query a *range* or *exact* query?

We now discuss three scenarios in which the above questions have been answered in different ways. Each of these scenarios gives rise to a possible optimization.

Scenario 1 The first optimization we discuss here is applicable whenever we have a *context insensitive*, *explicit*, *non-changing*, and *exact* association. In this case, it is known that the evaluation of the PerTupleContext always leads to the same, constant result, which is already known at JIT-compile time.

In Java, however, no notion of object constants exists. Therefore, neither the reflective implementation of PerTupleContext nor a bytecode-generating implementation can take advantage of this knowledge. In Jikes RVM and hence in Steamloom$^{\mathrm{ALIA}}$, however, objects are represented as pointers and, in principle, constant pointers can exist as immediate values in the generated machine code. In practice, this is not easily possible because the RVM performs garbage collection and may move objects in memory; then a constant pointer would become invalid. The machine code we generate for the PerTupleContext in this case re-uses the mechanism put in place by Jikes RVM for so-called root objects, i.e., globally accessible objects. Such objects can be accessed with just one memory look-up.

Scenario 2 For *context sensitive*, *explicit*, *non-changing*, and *range* associations, things are more complex: As the query depends on input that can be different at each evaluation, the result is also variable rather than constant. Nevertheless, since we have specified that the association does not change anymore, we can cache results depending on the input values. The required operations are relatively complex and we do not see significant opportunities for optimizations when generating these operations directly in machine code. Therefore, we have implemented this behavior in plain Java and generate a simple method invocation.

The two scenarios above discussed explicit instantiation. This is, e.g., applied by the association aspects language. The AspectJ language, in contrast, always applies implicit instantiation. Furthermore, queries are always exact. With implicit instantiation, the association is necessarily changing as new input values can occur.

Scenario 3 For *context insensitive*, *implicit*, *changing*, and *exact* associations, things are similar to the first scenario. However, we have to distinguish two cases, namely whether the singleton instance is already initialized or not when code is generated for the PerTupleContext. In the latter case, we create code that initializes the association and returns the just-created instance.[7] In the former case, we apply the similar optimization as was discussed for the first scenario.

But in this case, we can do even better: Since the allocation of the object is under the control of the PerTupleContext, the instance is allocated in a specific memory area where garbage collection does not move objects. Thus, we actually can use a constant pointer to the instance and completely save indirections.

5.3 Performance

Current performance measurements are encouraging. They show that our implementation outperforms that of the compiler for association aspects in several cases. Also, in scenario 3 the AspectJ compiler is outperformed by a factor of ten in the best case. However, our benchmarking results are too premature to be published in detail here. For example, we do not measure different structures of the *input tuple to instance association*. These are relevant, however, as many factors influence the measurements: How many elements does the input tuple have? How many entries are stored in the association? Does a range query match many or few entries?

6. Related Work

In the following, we discuss several MOPs that each target one of the three levels addressed by ALIA4J (cf. Section 3). None of presented approaches, however, supports all three levels of MOPs; in particular, no systematic approach is provided for transitioning from one level to another while switching from the language design to the optimization phase.

Runtime MOPs for Dispatching The new invokedynamic Java bytecode instruction and its supporting API [13], as specified by JSR 292, also constitute a MOP for advanced-dispatching. On the one hand, this meta-model is more low-level than ours; in particular, it forces language implementers to derive the dispatch logic themselves using primitives like guardWithTest. On the other hand, the JSR 292 meta-model is a pure runtime MOP that completely hides the code generation level; in particular, the implementers of a language runtime are barred from the optimization opportunities offered by user-defined bytecode or even machine code generation.

That being said, an implementation of ALIA4J's runtime MOP on top of JSR 292 is certainly viable. Such an implementation would benefit from the proliferation of JSR 292 support in production JVMs, while at the same time offering language implementers a more convenient, higher-level meta-model of advanced dispatching. A detailed analysis of the trade-offs involved in such an implementation, in particular in the light of invokedynamic-specific optimizations [15], is subject to future work.

[7] In practice, this case is slightly more complex: at the time the dispatch site is executed, the association may already be initialized, e.g., because the dispatch site has been executed before. Therefore, we insert a test ensuring that the association is not initialized twice.

Compile-Time MOPs for Dispatching The AspectBench Compiler (abc) [2] is an extensible compiler for the AspectJ language. Its parser front-end builds an abstract syntax tree (AST) of the source program. To implement a language extension, the parsed grammar is adapted and as a consequence new AST node types are added. After parsing, the abc framework transforms AST nodes representing aspect-oriented concepts to instances of abc-specific classes, which are similar to ALIA4J's meta-model. As is the case in ALIA4J, the abc-specific classes have to be refined for new language concepts. Such a refinement must implement a method that can attempt to statically evaluate the concept (e.g., whether a pointcut designator will always or never be satisfied) and implement a Java bytecode generation strategy if this is not possible. As there is no directly supported way of implementing the semantics in an interpretative style, a language designer developing a prototype with abc is always confronted with low-level implementation details.

Just-in-Time Compile Time MOPs Implementers of just-in-time compilers often need to open up the JIT compiler to customizable compilation strategies. This is mainly for two reasons: First, virtual machines may be meta-circular, i.e., written in the language whose execution they manage, which may prohibit operations like direct memory access required by the virtual machine implementation Second, virtual machine implementers want to provide a means of extending runtime support operations.

The first reason is the main motivation for the `org.vmmagic` framework by Frampton et al. [8] applied in Jikes RVM and for the Klein virtual machine presented by Ungar et al. [17]. This `org.vmmagic` framework achieves extensibility by so-called *intrinsic functions* not directly expressible in Java or Java bytecode. The implementation of Jikes RVM can contain calls to corresponding methods following a special naming convention. The Java implementation of those methods, however, is merely a dummy; when the just-in-time compiler encounters a call to such a method, it directly inserts the corresponding machine code.

The Klein virtual machine is based on the Self bytecode. Self includes the similar concept of *primitives* which are messages that can be sent but that do not have an implementation in the application code. Instead, the virtual machine must provide an implementation. Klein provides several ways of implementing primitives based on either generating corresponding machine code or implementing the primitive in Self. When generating machine code, the VM implementer can make use of an API which factors out the generation of code sequences realizing common functionality. When implementing a primitive in Self, a lower-level variant of Self can be used which restricts the available primitives thus avoiding infinite recursions.

In contrast to the `org.vmmagic` framework, in ALIA4J, the code to generate is not hard-wired in the JIT compiler or bytecode generation, but is implemented in a separate module. Thus, adding a code generation strategy for a language concept entity does not require a re-compilation of the JIT compiler. Klein's first option of implementing primitives is similar. Supporting both, implementing a code-generation strategy and a high-level method to which a callout is generated in Klein is similar to ALIA4J's distinction of the JIT compile-time MOP and the MOP. In ALIA4J, though, modularity and extensibility of language concepts are supported better. Sharing code generation strategies for common functionality is similar to what we aim to support with our fine-grained meta-model. Naturally, a meta-model is more extensible than an API.

Examples mainly driven by the goal of modularizing the implementation of the JIT compiler and other virtual machine components, are the Open Runtime Platform (ORP) by Cierniak et al. [6] and the XIR language of Titzer et al. [16]. ORP supports multiple source languages and multiple platforms and targets at avoiding redundancies in the implementation of the JIT compiler that might arise due to the different combinations. The implementation of runtime support operations is called *stub* in ORP. Stub implementations are not part of the JIT compiler but part of the component (e.g., the garbage collector) whose function they realize; thus, they are a means to hide details about other components from the JIT compiler. Stubs can be implemented in the C language, but ORP also includes a domain-specific language, called LIL [9] for this purpose. LIL is more abstract than assembler and in fact architecture-independent. At the same time, it embodies primitives specific to managed runtime environments, for example thread-safe operations.

The XIR language of Titzer et al. [16] was designed to improve compiler-runtime separation in Java virtual machines. With it, the developer of the runtime communicates the implementation strategy for essential runtime features like field accesses or method calls to the developer of the (just-in-time) compiler in the form of so-called snippets. Each snippet is formulated using an assembler-oriented interface provided by the compiler.

Like ORP and XIR, ALIA4J's MOP was designed as a clear interface, albeit not between implementers of runtime and compiler but between implementers of high-level language and language runtime. As such, ALIA4J by default exposes the language implementer to less low-level detail; it is easily possible to realize a concept's semantics in a reflective, interpretative style only. If, however, more control over code generation for LIAM entities is desired, a language like XIR or LIL could be employed to good effect. As this level of control is typically exerted by the developer of the language runtime, who integrates FIAL with a given runtime and compiler, we believe that abstracting from the details of code generation is, while definitely useful, not essential.

The presented approaches mainly present different ways of defining the code which is to be generated for realizing runtime support operations. These approaches do not put forward the idea of a JIT compile-time MOP which reflects upon the current compilation context and generates different code in specific situation. Nevertheless, approaches using an API for emitting code (`org.vmmagic` and Klein) can in principle conditionally emit different sequences of machine code.

7. Conclusion and Future Work

In this paper we have presented the ALIA4J architecture as a meta-object protocol (MOP) for controlling advanced dispatching. The MOP's implementation allows one to modularly implement concepts relevant for dispatching, including different kinds of optimizations: Besides an implementation of the concept's semantics in a reflective style, it is also possible to generate optimized bytecode or even machine code. We have demonstrated this feature in terms of two case studies: One of them shows that the user-defined bytecode generation can lead to compilation results similar to those of a conventional compiler. The second case study shows that specific machine-code generation makes feasible optimizations that are out of reach for bytecode optimization. The different implementation strategies can be freely mixed and it is possible to implement the semantics of a language concept at the most appropriate level. For instance, a language implementer is not forced to generate machine code for a concept that is already supported by bytecode.

In future work, we will improve the interface between ALIA4J execution environments and the bytecode respectively machine-code building methods. This includes most importantly the passed information about the dispatch itself. The different execution environments do not yet provide information in the same way which hinders reusing concept implementations across execution environments. Furthermore, we will improve the implementation of the PerTupleContext realizing aspect instantiation strategies. The current implementation only supports the baseline compiler of Jikes

RVM; an implementation for the optimizing compiler will be our next step. We also still have to perform a systematic performance evaluation of this optimization.

Acknowledgments

We would like to thank Andre Loker, who has developed the unified model of aspect instantiation strategies underlying the PerTupleContext.

This work was supported by the Center for Advanced Security Research Darmstadt (www.cased.de).

References

[1] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen. Implementing Jalapeño in Java. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1999.

[2] P. Avgustinov, A. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: An extensible AspectJ compiler. *Transactions on Aspect-Oriented Software Development I*, 3880:293–334, 2006.

[3] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual machine support for dynamic join points. In *Proceedings of the Conference on Aspect-Oriented Software Development (AOSD)*, 2004.

[4] C. Bockisch, S. Malakuti, M. Aksit, and S. Katz. Making aspects natural: Events and composition. In *Proceedings of the Conference on Aspect-Oriented Software Development (AOSD)*, 2011.

[5] C. Bockisch, A. Sewe, M. Mezini, and M. Akşit. An overview of ALIA4J: An execution model for advanced-dispatching languages. In *Proceedings of the Conference on Objects, Models, Components, Patterns (TOOLS)*, 2011.

[6] M. Cierniak, M. Eng, N. Glew, B. Lewis, and J. Stichnoth. The Open Runtime Platform: a flexible high-performance managed runtime environment. *Concurrency and Computation: Practice & Experience*, 17:617–637, 2005.

[7] M. Ernst, C. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1998.

[8] D. Frampton, S. M. Blackburn, P. Cheng, R. J. Garner, D. Grove, J. E. B. Moss, and S. I. Salishev. Demystifying magic: high-level low-level programming. In *Proceedings of the Conference on Virtual Execution Environments (VEE)*, 2009.

[9] N. Glew, S. Triantafyllis, M. Cierniak, M. Eng, B. T. Lewis, and J. M. Stichnoth. LIL: An architecture-neutral language for virtual-machine stubs. In *Virtual Machine Research and Technology Symposium*, 2004.

[10] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), 2008.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2001.

[12] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2003.

[13] J. R. Rose. Bytecodes meet combinators: invokedynamic on the JVM. In *Proceedings of the 3rd Workshop on Virtual Machines and Intermediate Languages (VMIL)*, 2009.

[14] K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura, and S. Komiya. Design and implementation of an aspect instantiation mechanism. *Transactions on Aspect-Oriented Software Development I*, pages 259–292, 2006.

[15] C. Thalinger and J. Rose. Optimizing invokedynamic. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java (PPPJ)*, 2010.

[16] B. L. Titzer, T. Würthinger, D. Simon, and M. Cintra. Improving compiler-runtime separation with XIR. In *Proceedings of the 6th International Conference on Virtual Execution Environments (VEE)*, 2010.

[17] D. Ungar, A. Spitz, and A. Ausch. Constructing a metacircular virtual machine in an exploratory programming environment. In *Companion to the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2005.