Efficient Language Implementation with ALIA4J and EMFText

Forum Demonstration

Christoph Bockisch
Software Engineering group
University of Twente
P.O. Box 217
7500 AE Enschede, the Netherlands
c.m.bockisch@cs.utwente.nl

Andreas Sewe
Software Technology group
Technische Universität Darmstadt
Hochschulstr. 10
64289 Darmstadt, Germany
sewe@st.informatik.tu-darmstadt.de

ABSTRACT

Developing extensions to general-purpose langauges or domain-specific languages with support for new kinds of abstractions is an ongoing trend. Modern language workbenches, such as EMFText of Xtext, support this trend and facilitate implementing langauges in terms of transformations from the new language into an established (intermediate) language. Often, however, the implementation of one element in the source language becomes scattered and tangled in the target language, which makes transformations complex. Furthermore, even though many languages share core concepts, current approaches do not support sharing transformations that implement their semantics; the only possibility of re-using transformations from a language is to extend it syntactically.

We have identified *dispatching* as fundamental to most abstraction mechanisms. With the ALIA4J approach, we provide a meta-model of dispatching to act as rich and extensible intermediate language that allows more direct transformation. The semantics of core language concepts can be modularly implemented as extension of the meta-model. For the execution of the intermediate language, we provide both platform-independent and platform-dependent Java Virtual Machine extensions, the latter of which even allows the modular implementation of machine code optimizations.

In this demo, participants get an overview of advanced dispatching and the ALIA4J approach. By the example of a language for text-based adventure games, they will see the usage of ALIA4J as back-end for a language developed in EMFText. Finally, the implementation of new atomic language concepts and their optimization is demonstrated.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Run-time environments*

Keywords

Advanced dispatching, language implementation, modular optimization

1. PRESENTATION HISTORY

An earlier version of this demo has been presented at the SPLASH conference in 2012 [4]. The demo builds on material that has been used in:

- the academic course "Advanced Programming Concepts" (University of Twente, the Netherlands) in 2011 and 2012,
- the tutorial "Efficient Implementation of Efficient (Domain-Specific) Languages" held at the Brazilian Conference on Software: Theory and Practice (CBSoft) 2010, and
- the journal paper: C. Bockisch, A. Sewe, H. Yin, M. Mezini, and M. Aksit. An in-depth look at ALIA4J. *Journal of Object Technology*, 11(1):1–28, Apr. 2012.

2. DEMONSTRATION OVERVIEW

New programming languages are designed frequently, mostly for two reasons. Firstly, designers seek to improve modularity and other qualities of general-purpose languages. Secondly, domain-specific languages (DSL) are designed to simplify programs in certain business domains. In both cases, the typical approach is to compile the programs into the intermediate code of an established language like Java. Even though many new languages share several concepts, few approaches exist to let them share their implementation.

The main means, by which programming languages provide modularity, respectively by which DSLs are embedded into a host language, is polymorphism. That means, certain locations in the program code, so-called call sites, are late-bound to functionality. Several implementations of a functionality are statically known to be applicable at the call site; which concrete functionality is executed at runtime, is determined based on the dynamic program state when the call is executed. It is possible to add new variants to call sites, override existing functionality or extend the functionality. *Dispatching* is the technology of choosing the appropriate implementation variant at runtime, when a call site is executed.

2.1 Addressed Problems

Typically, implementations of new languages build on the back-ends of established languages, re-using the implementation of the concepts native to that intermediate language. But not all concepts of the new languages map directly to the established intermediate language (e.g., Java bytecode), which was tailored to a different source language (e.g., Java). This task is further complicated when one element in the source program affects the behavior of multiple elements in the intermediate representation (requiring so-called local-to-global transformations).

Compiler frameworks assist in generating low-level code, and even enable to re-use non-trivial code generation logic. Open compilers for aspect-oriented languages, such as the AspectBench Compiler [2] or JastADD [8], even support modularizing local-to-global transformations. But these technologies require the new language to be a syntactic extension of an existing one. Moreover, the knowledge about the source language concepts is lost during the transformation and cannot, e.g., drive specific virtual-machine-level optimizations.

This demonstration targets designers and implementers of programming languages expressible as a dispatching problem, which is frequently possible as we have shown for predicate dispatching [9] (e.g., JPred [14]), pointcut-advice [13] (e.g., AspectJ [12]), inter-type member declarations (e.g., AspectJ [12]), policy enforcement (e.g., ConSpec [1]), as well as a domain-specific language extension enforcing the Decorator pattern.

2.2 Technology of Our Solution

The ALIA4J[1] architecture realizes our approach to implementing programming languages with advanced dispatching. At its core sits a meta-model of advanced dispatching declarations, called LIAM, and a framework for execution environments that handle these declarations, called FIAL. LIAM hereby defines a *language-independent meta-model* of atomic concepts relevant for dispatching. For example, dispatch may be ruled by predicates which depend on values in the dynamic context of the dispatch. When mapping the concrete advanced-dispatching concepts of an actual programming language to it, LIAM either has to be *refined* with the *language-specific* semantics or suitable, existing refinements have to be re-used. The dispatch declarations defined in terms of this meta-model, are partially evaluated by the FIAL framework and automatically re-written into an execution model for the dispatch sites in the program.

Furthermore, we provide multiple execution environments conforming to our execution model: a portable implementation based on interpretation (NOIRIn), a portable implementation performing bytecode optimization (SiRIn), a virtual-machine-integrated implementation applying dynamic native-code optimization (STEAMLOOM$^{\text{ALIA}}$), etc. In all cases the execution semantics for atomic language concepts have been implemented modularly in ALIA4J in terms of plain-old Java. Moreover, bytecode-level and dynamic native-code-level optimizations are implemented modularly as well.

We have realized several existing and one new programming language using the ALIA4J architecture, showing that the dispatching mechanism is sufficient to realize a multitude of different languages and paradigms. Overall, ALIA4J contains more than 75 re-usable language concepts, which we have developed while realizing the above mentioned languages. Most of these concepts are used in more than one language realization. The concept implementations can be used out-of-the-box and the majority of them offers at least bytecode-level optimizations.

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

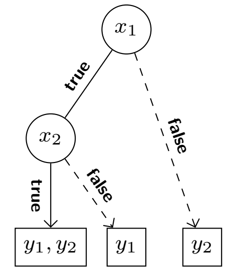

Figure 1: A dispatch site's execution model.

2.3 Uniqueness in Design and Implementation

From the whole disptching declarations, at runtime our framework generates one execution model for each dispatch site in the program. Thereby it implicitly creates dispatching declarations for the Java operation (such as a method call or field access) underlying each dispatch site. Therefore, this default operation and additional operations specified by dispatching declarations are expressed uniformly, and it is possible to reason about them jointly.

The execution model of a dispatch site is also called its *dispatch function* and is represented as a graph that forms a binary decision diagram (BDD) [6]. This graph has an explicit root node and its inner nodes represent the decisions to be made during dispatching. Decisions are defined in terms of so-called *atomic predicates.* Each node representing a decision has two directed, out-going edges, one for each possible outcome of the atomic predicate. The leaf nodes of this graph ultimately store descriptions of how to perform actions as result of the dispatch.

When a dispatch site is encountered at runtime, its dispatch function is evaluated, beginning at the root node. For a decision node, first the corresponding atomic predicate is evaluated in the current execution context. Depending on the outcome, one of the edges is traversed and the next node is evaluated until eventually a leaf node is reached. For a leaf node, the execution environment performs the actions that it specifies. Figure 1 shows an example of a dispatch function in which x_1 and x_2 represent atomic predicates, and y_1 and y_2 represent descriptions of actions.

The atomic predicates and the descriptions of actions are elements from the LIAM meta-model. Thus, the intermediate representation and the execution model share the same atomic concepts. In ALIA4J, there are three ways of modularly implementing the semantics of such atomic language concepts and optimizations thereof. As our architecture employs factories for creating instances of LIAM meta-entities, the different implementations can be chosen flexibly and transparantly to the client.

1. The most abstract way is implementing a plain Java method that realizes the semantics of an atomic language concept through *interpretation.* This allows easy experimentation and is targeted at designers of the semantics of language concepts.

2. Control over the generated code is gained by implementing a Java method that compiles the concept to *Java bytecode*, allowing context-dependent bytecode 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*. While losing platform-independence, this allows to achieve optimal runtime performance.

Our framework furthermore implements an infrastructure for dynamic deployment of dispatching declarations. Even for this dynamic deployment operation, differently efficient implementations exist. The platform-independent SiRIn execution environment, for example, uses the class redefinition feature from the Java Virtual Machine Tools Interface (JVMTI); the STEAMLOOM^{ALIA} execution environment provides a more efficient, though platform-dependent, implementation [3]. The provided infrastructure also treats all the delicate interactions between dynamic deployment and Java's dynamic class loading.

2.4 Interesting Details

The STEAMLOOM^{ALIA} JVM extension, which enables the machine code optimization, is an extension of the Jikes Research VM (RVM), a high-performance Java VM. It can bypass bytecode generation for LIAM entities to directly generate native machine code for them, using the two JIT compilers of the Jikes RVM, the baseline compiler and the optimizing compiler. The generation can access all VM internals and rich information about the generation context to produce the most specific machine code.

The implementation of a concept's semantics *and* optimization is modular. Implementations of different strategies can even co-exist; the best strategy is picked at runtime. This is very useful to implementers of optimizations who can use the—less efficient but by definition correct—implementation produced by the language designer as a test oracle. Overall, we provide re-usable implementations of more than 75 atomic language concepts, the majority of which offers at least bytecode-level optimizations.

We use an extensive integration test suite to ensure the high quality of ALIA4J. The integration tests use our intermediate representation as interface; thus, all FIAL-based JVM extensions are subject to the same test suite, which ensures compatibility between different execution environments. Almost all of the 4,083 tests are systematically generated to cover all relevant variations of dispatch sites and LIAM entities. Our build process is fully automated with the Maven build manager and our integration test suite is automatically executed using the Jenkins continuous integration server.

Over the past years, four PhD projects and more than 20 master and bachelor student projects have contributed to ALIA4J. The technologies applied in ALIA4J are presented in more than 10 peer-reviewed journal, conference, and workshop papers.

2.5 Relevance to the Community

The AOSD conference has a tradition in gathering innovative research on programming languages for increased modularity, including, both, new language concepts and new optimization techniques. This demo targets designers and implementers of programming languages which are expressible as a dispatching problem. In particular, the demo is relevant for two groups:

1. *Designers* who want to focus on designing a source level-language and who want to quickly prototype an implementation of their language. While ALIA4J allows language designers to implement the semantics of their own language concepts purely with Java means, possibly existing optimizations of pre-implemented language concepts are automatically re-used, as well.

2. *Optimizers* who want to implement sophisticated, possibly dynamic, optimizations for established language concepts. For this purpose, ALIA4J can expose the internals of Java bytecode or Java Virtual Machines, which are normally hidden from language implementers. Since optimizations can be implemented modularly all languages using an optimized concept benefit.

2.6 Integration with Language Workbenches

The ALIA4J approach is suitable for implementing languages which are either extensions of the general-purpose language Java, or domain-specific languages (DSL) which in some way control the execution of an underlying Java program. To showcase the process of developing a language with ALIA4J, we choose the second case in this demonstration. We will show the development of a small DSL for defining text-adventure games. A simple, generic game engine acts as underlying base program. This engine reads user commands from the console, tracks the position of items, persons, etc. and allows simple interactions between them.

The DSL whose implementation will be demonstrated is for specifying adventures that can be played. Besides, defining the layout of the world, and the items and persons that exist in it, the DSL also facilitates to define the influence of items on the world. For instance, certain items can, when used by a person, render that person invisible. Such effects are implemented by changing the behavior of the basic commands provided by the engine. In the example of turning a person invisible, the behavior of the "look about" command it changed to not print out the names of invisible persons.

In this demo, we will explain this language in detail and show the implementation of the grammar and parser in the EMFText language workbench [11]. We will show how to compile a DSL in our ALIA4J approach by demonstrating the development of a component that transforms the abstract syntax tree (produced by the EMFText parser) into our LIAM meta-model. The resulting dispatching declarations influence the execution of the game engine.

2.7 Related Work

Several alternative execution environments exist that can be suitable backends for implementing aspect-oriented languages. Examples are Nu [7], Reflex [16], delMDSOC [15], or JAMI [10]. From the existing alternatives, ALIA4J offers the most fine-grained abstractions and, thus, facilitates the highest degree of re-use in language implementations. Furthermore, ALIA4J is the only approach that allows language implementers to modularly implement different optimization strategies for atomic language constructs. These different strategies can be freely interchanged in the execution environment in a way that is transparent to the executed program.

A detailed comparison to related work can be found in our paper: C. Bockisch, A. Sewe, H. Yin, M. Mezini, and M. Aksit. An in-depth look at ALIA4J. *Journal of Object Technology*, 11(1):1–28, Apr. 2012.

2.8 Description of the Demo Content

The demo contains an explanation of the predicate dispatching and aspect-oriented programming paradigms, which have shaped the ALIA4J approach, followed by an introduction to ALIA4J's meta-model and its execution semantics. By the example of a domain-specific language for defining text-based adventure games, it is demonstrated how an EMFText-based language implementation can use ALIA4J as an execution back-end. The participants will see how the example language is transformed into ALIA4J's intermediate representation by re-using provided atomic language concepts; and how to implement the execution semantics of new, specific language concepts. The concepts will be implemented in a platform-independent, high-level way and supplanted by bytecode and machine code optimizations.

3. ACKNOWLEDGMENTS

We would like to thank everyone who has contributed to ALIA4J in the past few years (in alphabetical order): Matthew Arnold, Remko Bijker, Tom Dinkelaker, Sebastian Eifert, Sarah Ereth, Pascal Flach, Michael Haupt, Michael Hausl, Jannik Jochem, Sebastian Kanthak, Michael Krebs, Andre Loker, Markus Maus, Suraj Mukhi, Heiko Paulheim, Nico Rottstädt, Christian Rüdiger, Jan Sinschek, Kai Stroh, Zied Trabelsi, Nathan Wasser, and Martin Zandberg. This work was supported by the Center for Advanced Security Research Darmstadt (`http://www.cased.de/`www.cased.de).

4. REFERENCES

[1] I. Aktug and K. Naliuka. ConSpec: A formal language for policy specification. In *Proceedings of REM*. Elsevier Science Publishers B. V., 2008.

[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. In *Transactions on Aspect-Oriented Software Development I*, number 3880 in Lecture Notes in Computer Science, pages 293–334. Springer Verlag, Berlin/Heidelberg, Germany, 2006.

[3] C. Bockisch, M. Arnold, T. Dinkelaker, and M. Mezini. Adapting virtual machine techniques for seamless aspect support. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, 2006.

[4] C. Bockisch and A. Sewe. The alia4j approach to efficient language implementation. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, SPLASH '12. ACM, 2012.

[5] C. Bockisch, A. Sewe, H. Yin, M. Mezini, and M. Aksit. An in-depth look at ALIA4J. *Journal of Object Technology*, 11(1):1–28, Apr. 2012.

[6] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35, 1986.

[7] R. Dyer and H. Rajan. Supporting dynamic aspect-oriented features. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(2):7:1–7:34, 2010.

[8] T. Ekman and G. Hedin. The JastAdd system – modular extensible compiler construction. *Elsevier Science of Computer Programming*, 69(1-3):14–26, 2007.

[9] M. Ernst, C. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. In *Proceedings of ECOOP*. Springer Verlag, 1998.

[10] W. Havinga, L. Bergmans, and M. Akşit. Prototyping and composing aspect languages: using an aspect interpreter framework. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 180–206, Berlin/Heidelberg, Germany, 2008. Springer Verlag.

[11] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende. Derivation and refinement of textual syntax for models. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, ECMDA-FA '09, Berlin, Heidelberg, 2009. Springer-Verlag.

[12] 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*, pages 327–353, Berlin/Heidelberg, Germany, 2001. Springer Verlag.

[13] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *Proceedings of ECOOP*. Springer Verlag, 2003.

[14] T. Millstein, C. Frost, J. Ryder, and A. Warth. Expressive and modular predicate dispatch for Java. *ACM Transactions on Programming Languages and Systems*, 31(2), 2009.

[15] H. Schippers, D. Janssens, M. Haupt, and R. Hirschfeld. Delegation-based semantics for modularizing crosscutting concerns. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 525–542, New York, NY, USA, 2008. ACM.

[16] É. Tanter. An extensible kernel language for AOP. In *Proceedings of the Workshop on Open and Dynamic Aspect Languages*, 2006.