The ALIA4J Approach to Efficient Language Implementation

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

New programming languages are frequently designed to improve upon other languages or to simplify programs through domain-specific abstractions. They are often implemented as transformations to an established (intermediate) language (IL). But while many new languages overlap in the semantics of their core concepts, re-using the corresponding transformations is limited by existing compiler implementation frameworks. In the ALIA4J approach, we have identified *dispatching* as fundamental to most abstraction mechanisms and provide a meta-model of dispatching as a rich, extensible IL. Based on this meta-model, the semantics of new atomic language concepts can be implemented in a modular and portable fashion. For the execution of the IL, we provide both platform-independent and platform-dependent Java Virtual Machine extensions, the latter of which 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 a modern Language Workbench. 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. The Demonstrated Technology

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 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.

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,

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

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.

Uniqueness in Design and Implementation In ALIA4J, there are three ways of modularly implementing a meta-model refinement (and as such an atomic language concept): (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.

Interesting Details The STEAMLOOM$^{\text{ALIA}}$ 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 assure 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 more than 10 peer-reviewed journal, conference, and workshop papers.

2. Content of the Demo

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. Presenter

Christoph Bockisch is an assistant professor at the University of Twente, the Netherlands. He received his doctoral degree from Technische Universität Darmstadt, Germany in 2008. To provide optimizations of new language mechanisms, Christoph researches extensions to high-performing Java virtual machines based on just-in time compilation. He initiated the ALIA4J project as part of his PhD studies and is now one of two project supervisors and lead programmers. He is co-founder and co-organizer of the workshop series on Virtual Machines and Intermediate Languages (VMIL) and Free Composition (FREECO), both held at the SPLASH conference.

4. Presentation History

The demo builds on material that has been used in:

- the academic course "Advanced Programming Concepts" (University of Twente, the Netherlands) in 2010/11 and 2011/12,
- the tutorial "Efficient Implementation of Efficient (Domain-Specific) Languages" held at the Brazilian Conference on Software: Theory and Practice (CB-Soft) 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.