Scalan MetaDSL Framework

Alexander Slesarenko Alexey Romanov
Shannon Laboratory, Huawei Technologies, Moscow, Russia
{alexander.slesarenko, alexey.romanov}@huawei.com

Categories and Subject Descriptors D.3.3 [*Programming languages*]: Language products and features

Keywords DSL, domain-specific languages, high-performance computing, Scalan

Abstract

While high-level abstractions greatly simplify program development, they ultimately need to be eliminated to produce high-performance code. This can be done using generative programming; one particularly usable approach is Lightweight Modular Staging [1, 2].

We present Scalan, a framework which enables compilation of high-level object-oriented-functional code into high-performance low-level code. It extends the basic LMS approach by making rewrite rules and compilation stages first-class and extending the graph IR with object-oriented features.

Rewrite rules are represented as graph IR nodes with edges pointing to a *pattern graph* and a *replacement graph*; whenever new nodes are constructed, they are compared with the pattern graphs of all active rules and in case a match is found, the corresponding replacement graph is generated instead.

Compilation stages are represented as graph transformers and together with the final output generation stage assembled into a compilation pipeline. This allows using multiple backends together, for example generating C/C++ code with JNI wrappers for the most performance-critical parts and Spark code which calls into it for the rest.

We show how object-oriented programming is supported by staging class constructors and method calls (including "factory" methods on companion objects) as part of the IR, thus exposing them to rewrite rules like all other operations. JVM mechanisms allow treating symbols as typed proxies for their corresponding nodes. Now it becomes necessary to eliminate such nodes at some compilation stage to avoid virtual dispatch in the output code (or at least minimize it for OO target languages).

In the simple case when the receiver node of a method is a class constructor, we can simply delegate the call to the subject at that stage. The more interesting case when the receiver node is the result of a calculation is handled by *isomorphic specialization*, as previously described in [3].

We will also demonstrate how the use of a Scala compiler plugin further simplifies development by avoiding the explicit use of the `Rep` type constructor and how our framework can handle effects using free monads.

We will finish by discussing future plans for Scalan development.

References

[1] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM*, 55(6):121–130, 2012.

[2] T. Rompf, K. J. Brown, H. Lee, A. K. Sujeeth, M. Jonnalagedda, N. Amin, G. Ofenbeck, A. Stojanov, Y. Klonatos, M. Dashti, C. Koch, M. Püschel, and K. Olukotun. Go meta! A case for generative programming and DSLs in performance critical systems. In *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA*, pages 238–261, 2015.

[3] A. Slesarenko, A. Filippov, and A. Romanov. First-class isomorphic specialization by staged evaluation. In *Proceedings of the 10th ACM SIGPLAN workshop on Generic programming, WGP 2014, Gothenburg, Sweden, August 31, 2014*, pages 35–46, 2014.