Journal of Logical and Algebraic Methods in Programming 85 (2016) 655–680

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Implementing type systems for the IDE with Xsemantics ☆

Lorenzo Bettini

Dipartimento di Informatica, Università di Torino, Italy

A R T I C L E I N F O

Article history:
Received 14 March 2015
Received in revised form 2 October 2015
Accepted 23 November 2015
Available online 27 November 2015

Keywords:
DSL
Type systems
Semantics
Implementation
Eclipse

A B S T R A C T

Xsemantics is a DSL for writing type systems, reduction rules and, in general, relation rules for languages implemented in Xtext (Xtext is an Eclipse framework for rapidly building languages together with all the typical IDE tooling). Xsemantics aims at reducing the gap between the formalization of a language (i.e., type system and operational semantics) and the actual implementation in Xtext, since it uses a syntax that resembles the rules in a formal setting. In this paper we present the main features of Xsemantics for implementing type systems and reduction rules through examples (Featherweight Java and lambda calculus). We show how such implementations are close to the actual formalizations, and how Xsemantics can be a helpful tool when proving the type safety of a language. We also describe the new features of Xsemantics that help achieving a modular and efficient implementation of type systems. In particular, we focus on specific implementation techniques for implementing type systems that are suited for the IDE (in our context, Eclipse), in order to keep the tooling responsive and guarantee a good user experience.

1. Introduction

Xtext [2,7] is a popular Eclipse framework for the development of programming languages and domain-specific languages (DSLs). Starting from a grammar definition it generates a parser, an abstract syntax tree based on EMF [60], and an Eclipse-based editor, with all the typical IDE tooling. Xtext comes with good defaults for all the above artifacts and the language developer can easily customize them all. For all these reasons, Xtext makes language implementation and its integration into Eclipse really easy [27].

Checking that a program is semantically correct in Xtext consists of two mechanisms that have to be customized by the developer: *scoping* (Section 2.1), i.e., resolving and binding symbols (e.g., binding a reference to its definition in the program), and *validation* (Section 2.2), which includes all the other semantic checks that do not deal with scoping (e.g., checking that the type of the returned expression in a method body is conformant to the method's return type). In a statically typed language, these two mechanisms heavily rely on types. The type system and the interpreter/compiler for a language implemented in Xtext are usually implemented in Java. Instead, we want to employ a DSL also for these tasks, in order to implement, in a more direct way, a language that has been formalized with a type system and an operational semantics. Furthermore, this will allow us to have the typing and reduction rules in a compact form, similar to the actual formalization, easy to understand and maintain, and then have the corresponding Java code automatically generated. Finally, the gap between the formalization of a language and its actual implementation in Xtext will be reduced.

☆ Work partially supported by MIUR (2010LHT4KM, proj. CINA), Ateneo/CSP (proj. SALT), and ICT COST Action IC1201 BETTY.
E-mail address: bettini@di.unito.it.

http://dx.doi.org/10.1016/j.jlamp.2015.11.005
2352-2208/© 2015 Elsevier Inc. All rights reserved.

For all the above reasons, we implemented Xsemantics,[1] a DSL for writing rules for languages implemented in Xtext, in particular, the *static semantics* (type system), the *dynamic semantics* (operational semantics) and relation rules (subtyping). A system definition in Xsemantics is a set of rules that act on the elements of the Abstract Syntax Tree (AST) of a program. Xsemantics will generate Java code that can be used to implement the scoping and the validation for the language.

The very first prototype of Xsemantics was introduced in [9], where it was used, together with other frameworks, in an analysis of several approaches for implementing type systems for Xtext DSLs. Then, in [8], the first release of Xsemantics was presented. At that time, Xsemantics was still a proof of concept for studying prototype language implementations starting from existing language formalizations. Since then, Xsemantics has started to be adopted also in industry (as illustrated in more details in the following), and this required to rewrite many of its parts so that complex type systems could be effectively and efficiently implemented. Its adoption in real-world type system implementations dictated the introduction of new features in the Xsemantics DSL to make its specifications modular, reusable and completely interoperable with Java. The runtime of Xsemantics had to be extended as well in order to improve the performance of the implemented type systems.

This paper extends [9] and [8] in many respects. First of all, the main features of Xsemantics are described in more details (e.g., auxiliary functions were not shown in the previous papers), still using as an example Featherweight Java (FJ) [39], a lightweight functional version of Java, which focuses on a few basic Object-Oriented features. Another example is presented, a λ-calculus DSL, to show how to implement type inference, including generic types, with unification. Although these examples are not real-world programming languages, still they feature the main characteristics of Java-like and functional languages. The above mentioned new features are described in details with examples. The internals of Xsemantics are also described, in particular, concerning the generated Java code. Moreover, general issues related to efficient implementations of type systems are discussed, especially aiming at improving the IDE experience of the implemented language.

Note that an efficient implementation of a type system is crucial in the context of an IDE: being able to quickly type a program, while the program is being edited, provides the user with an earlier error reporting. Furthermore, by using types, we can provide better code completion in the IDE (e.g., in an OOP language, we can propose the methods invokable on an expression if we know its type). Moreover, if we implement a type system for inferring types (as in Section 5), the IDE can automatically insert the inferred types in the text editor, or suggest more generic types. In any case, the type system has to be implemented efficiently, in order to keep the IDE responsive. Error recovery is another important aspect when implementing type systems. The implementation should be able to type as many parts of a program as possible, even in the presence of type errors. Similarly to error recovery in parsing, the type system should avoid to throw many errors due to previous failed type computations. This is already important in a command line compiler, but in the IDE it is even more crucial: the portions of the edited file with error markers should be kept to the minimal. This way, the user can easily spot where the problems in the program really are. To deal with this issue, as we will see in the paper (Section 7), it is required to separate type computation from type checking.

A specification DSL, like Xsemantics, must provide IDE support so that type system specifications can be easily edited with the help of all the typical tooling mechanisms that we mentioned above. Moreover, we think that such specifications should be completely interoperable with Java so that the language implementors are not forced to use Xsemantics for all the tasks and they can write some parts in Java. In that respect, Xsemantics itself is implemented in Xtext, thus it provides a complete Eclipse IDE. In particular, Xsemantics uses Xbase [24] to provide a rich Java-like syntax for defining premises in rules. Xbase is a reusable Java-like expression language that is interoperable with Java. Thanks to Xbase, from an Xsemantics definition we can refer to any existing Java library, and we can even debug Xsemantics code (Section 6.4). This also means that, when using Xsemantics, we can still implement parts of the type system directly in Java and we are not forced to use Xsemantics for everything. In an existing language implementation, this also allows for an easy incremental or partial transition to Xsemantics.

Xsemantics has proved to be mature and powerful enough to be used in real-world languages in industry.[2] Notably, it is employed to implement a full-featured type inference system for a Javascript dialect implemented in Xtext. This Javascript dialect is a super set of JavaScript with modules and classes as proposed in [3] with a static type system on top of it, which combines the type systems provided by Java, TypeScript [36] and Dart [21]. It features primitive types, declared types such as classes, interfaces, roles and union types [38] and it supports generic types and generic methods, including wildcards, requiring the notion of existential types [15]. Moreover, an industrial experience report has been recently presented at XtextCon, showing how Xsemantics helped to highly simplify the implementation of the type system in several DSLs (which have to deal with large models), making the type system implementation clearer, cleaner and easier to maintain [35].

Since Xsemantics generates readable Java code, it is possible to use all the Java tools and frameworks for code quality analysis on the generated code, e.g., Jacoco and Findbugs. Xsemantics also provides Maven integration, so that projects using Xsemantics can be built in a Continuous Integration system with build tools like Maven and Gradle. All these features have been used in the real-world industry implementations that we have just mentioned.

Finally, although Xsemantics does not aim at providing mechanisms for formal proofs, it can still be a helpful tool when proving the type safety of a language. In particular, the generated Java code keeps track of the trace of applied rules, and this

[1] Xsemantics is available as an open source project at http://xsemantics.sf.net. Sources can be found at https://github.com/LorenzoBettini/xsemantics.

[2] The author is aware of at least two commercial products using Xsemantics, having been directly involved in both of them. Currently, these languages, being commercial products, are closed-source.