Technische Universität Berlin
Forschungsberichte der Fakultät IV - Elektrotechnik und Informatik Bericht-Nr. 2007-7
ECOOP Doctoral Symposium 2007

A Rewriting Approach to the Design and Evolution of Object-Oriented Languages

Mark Hills and Grigore Roşu

Department of Computer Science
University of Illinois at Urbana-Champaign, USA
201 N Goodwin Ave, Urbana, IL 61801
{mhills,grosu}@cs.uiuc.edu
http://fsl.cs.uiuc.edu

Abstract. Object-oriented language concepts have been highly successful, resulting in a large number of object-oriented languages and language extensions. Unfortunately, formal methods for defining and reasoning about these languages are still often performed after the fact, potentially resulting in ambiguous, overly complex, or poorly understood language features. We believe it is important to bring the use of formal techniques forward in this process, using them as an aid to language design and evolution. To this end, we propose a set of tools and techniques, making use of rewriting logic, to provide an interactive environment for the design and evolution of object-oriented languages, while also providing a solid mathematical foundation for language analysis and verification.

Key words: object-oriented languages, programming language semantics, language design, rewriting logic, formal analysis

1 Problem Description

Object-oriented languages and design techniques have been highly successful, with OO languages now used for many important applications in academia and industry. Along with commonly-used static languages, such as Java and C++, there has been a resurgence in the use of dynamic languages, such as Python, and domain-specific languages, often built on top of existing OO languages. This has led to a flurry of research activity related both to the design and formal definition of object-oriented languages and to methods of testing and analyzing programs.

Unfortunately, even as object-oriented languages are used in more and more critical applications, formal techniques for understanding these languages are still often post-hoc attempts to provide some formal meaning to already existing language implementations. This decoupling of language design from language semantics risks allowing features which seem straight-forward on paper, but are actually ambiguous or complex in practice, into the language. Some practical examples of this arose in the various designs of generics in Java, where interactions between the generics mechanism, the type system, and the package-based visibility system led to some subtle errors[2]; other ambiguities in Java have also

been documented [3]. Decoupling also makes analysis more difficult, since the meaning of the language often becomes defined by either a large, potentially ambiguous written definition or an implementation, which may be a black box.

With this in mind, it seems highly desirable to provide support for formal definitions of languages during the process of language design and evolution. However, existing formal tools are often not suitable to defining the entirety of a complex language, which leads directly to their delayed use. Language definitions based around structural operational semantics [21] and natural semantics [14] both lead to natural methods for executing programs and creating supporting tools [1], but both face limitations in supporting complex control flow features, modularity, and (with natural semantics) concurrency [19]. Modular Structural Operational Semantics [20] is an attempt to improve modularity of definitions, but is still limited by difficulty in supporting some control flow constructs. Reduction semantics [8] provides strong support for complex control flow, but currently has limited tool support with little focus on language analysis beyond typing. Denotational methods have been created to improve modularity [18], but can be quite complex, especially when dealing with concurrency. Current rewriting-based methods provide powerful tool support [23, 17] and can be conceptually simpler, but can be verbose and non-modular. A formal framework, concise, modular, and broadly usable, providing support for defining even complex language features, while providing tools for language interpretation and analysis, is thus critical to opening up the use of formal techniques during language design.

2 Goal Statement

The goal of our research is to provide an environment for the design and evolution of programming languages that is based on a solid formal foundation. This environment should be flexible enough to define the complexities of real languages, such as Smalltalk, Java, Python, or Beta, with support for the rapid prototyping of new languages and language features. Definitions should be formal, executable, and modular, allowing the user to define new features and immediately test them on actual programs. Analysis should also be easily supported, providing the ability to check existing programs and to ensure that new language features (especially those related to areas like concurrency, which can have unexpected interactions with other language features) work as expected.

Overall, we expect our research to produce: a framework for the modular definition of languages, including a number of pre-defined language modules; definitions of multiple new and existing languages, available as the basis for language extensions and as example definitions; graphical tools for joining together language modules and animating the execution of the semantics (actual program execution, typing, abstract interpretation, etc); and translations into various runtime environments for the execution and analysis of programs using the defined language semantics. We believe this framework, with the associated tools and definitions, can help make formal techniques useful not only for post-hoc studies of languages, but also during the language design process, providing formal definitions as a natural part of language design and evolution.

Technische Universität Berlin
Forschungsberichte der Fakultät IV - Elektrotechnik und Informatik Bericht-Nr. 2007-7
ECOOP Doctoral Symposium 2007

3 Approach

A Framework for Language Definition: Our work on programming languages has focused mainly on the use of rewriting logic [16, 15], a logic of computation that supports concurrency. While we believe rewriting logic, in combination with engines such as Maude [5, 6], is a compelling solution for defining and reasoning about languages [17], the generality of rewriting logic can sometimes lead to verbose, non-modular language definitions [22]. To exploit the strengths of rewriting logic, while ameliorating some of the weaknesses, we have begun work on K, a domain-specific variant of rewriting logic focused on defining programming languages [22]. K is specifically being designed to provide for concise, modular definitions of languages, written in an intuitive style. Initial versions of K have been used in the classroom and to define portions of Java and an experimental object-oriented language named KOOL [4, 11].

Defining and Evolving Object-Oriented Languages: An important test of our techniques is to define and experiment with actual object-oriented languages. This has been done both using K and directly in rewriting logic. One result has been the KOOL language, a dynamic, concurrent, object-oriented language [12], designed specifically to experiment with a variety of language extensions. Other work has resulted in an initial rewriting logic definition of Beta [9] and definitions of Java [7] and JVM bytecode [7]. We plan to extend this work, not only defining languages but also building libraries of language features, while using the feedback from this process to improve the K framework.

Tool Support: To make the K framework and associated language definition techniques widely useful, we plan to develop tools to support language design, analysis, and execution. We have done some initial work on how language design decisions impact analysis performance [13], and our work on KOOL [12] provided an interesting example of how analysis can be used to check that language features work as expected. Other initial work has demonstrated the promise of translating language definitions in K into executable form [10]. Additional work will involve developing a user interface for working with K definitions and animation tools to visualize the workings of the language semantics, with further work on translations directly from K to Maude and various target languages.

References

1. P. Borras, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: the system. In *Proceedings of SDE 3*, pages 14–24. ACM Press, 1988.
2. G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: adding genericity to the Java programming language. In *OOPSLA '98*, pages 183–200, New York, NY, USA, 1998. ACM Press.
3. J.-T. Chan, W. Yang, and J.-W. Huang. Traps in Java. *J. Syst. Softw.*, 72(1):33–47, 2004.

4. F. Chen, M. Hills, and G. Roşu. A Rewrite Logic Approach to Semantic Definition, Design and Analysis of Object-Oriented Languages. Technical Report UIUCDCS-R-2006-2702, University of Illinois at Urbana-Champaign, 2006.
5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.
6. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 System. In *Proceedings of RTA'03*, volume 2706 of *LNCS*, pages 76–87. Springer, 2003.
7. A. Farzan, F. Chen, J. Meseguer, and G. Roşu. Formal Analysis of Java Programs in JavaFAN. In *Proceedings of CAV'04*, volume 3114 of *LNCS*, pages 501–505. Springer, 2004.
8. M. Felleisen and R. Hieb. A Revised Report on the Syntactic Theories of Sequential Control and State. *Theoretical Computer Science*, 103(2):235–271, 1992.
9. M. Hills, T. B. Aktemur, and G. Roşu. An Executable Semantic Definition of the Beta Language using Rewriting Logic. Technical Report UIUCDCS-R-2005-2650, University of Illinois at Urbana-Champaign, 2005.
10. M. Hills, T. F. Şerbănuţă, and G. Roşu. A rewrite framework for language definitions and for generation of efficient interpreters. In *Proceedings of WRLA'06*, ENTCS. Elsevier, 2007. To appear.
11. M. Hills and G. Roşu. KOOL: A K-based Object-Oriented Language. Technical Report UIUCDCS-R-2006-2779, University of Illinois at Urbana-Champaign, 2006.
12. M. Hills and G. Roşu. KOOL: An Application of Rewriting Logic to Language Prototyping and Analysis. In *Proceedings of RTA'07*, LNCS. Springer, 2007. To appear.
13. M. Hills and G. Roşu. On Formal Analysis of OO Languages using Rewriting Logic: Designing for Performance. In *Proceedings of FMOODS'07*, volume 4468 of *LNCS*, pages 107–121. Springer, 2007.
14. G. Kahn. Natural semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Sciences*, LNCS, pages 22–39. Springer, 1987.
15. N. Martí-Oliet and J. Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, 285:121–154, 2002.
16. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
17. J. Meseguer and G. Roşu. The rewriting logic semantics project. *Theoretical Computer Science*, to appear, 2007.
18. E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Computer Science Dept., University of Edinburgh, 1989.
19. P. D. Mosses. The varieties of programming language semantics. In D. Bjørner, M. Broy, and A. V. Zamulin, editors, *Ershov Memorial Conference*, volume 2244 of *LNCS*, pages 165–190. Springer, 2001.
20. P. D. Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61:195–228, July-December 2004 2004.
21. G. D. Plotkin. Lecture notes DAIMI FN-19: A Structural Approach to Operational Semantics. Dept. of Computer Science, University of Aarhus, 1981.
22. G. Roşu. K: a Rewrite Logic Framework for Language Design, Semantics, Analysis and Implementation. Technical Report UIUCDCS-R-2006-2802, University of Illinois at Urbana-Champaign, 2006.
23. M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the ASF+SDF compiler. *ACM TOPLAS*, 24(4):334–368, 2002.