Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 117 (2005) 393–416
www.elsevier.com/locate/entcs

Modular Rewriting Semantics in Practice

Christiano Braga

Universidade Federal Fluminense, Niterói, Brazil

José Meseguer

University of Illinois at Urbana-Champaign, USA

Abstract

We present a general method to achieve modularity of semantic definitions of programming languages specified as rewrite theories, so that semantic rules do not have to be redefined in language extensions. We illustrate the practical use of this method by means of two language case studies: two different semantics for CCS, and three different semantics for the GNU bc language.

Keywords: Rewriting logic, programming languages semantics, modularity

1 Introduction

From its early stages, rewriting logic has been understood as a semantic framework particularly well suited for defining the mathematical and operational semantics of programming languages [22,27,19]. A semantic definition for a programming language $\mathcal{L}$ takes the form of a rewrite theory $\mathcal{R}_{\mathcal{L}}$. The mathematical semantics of $\mathcal{L}$ is then provided by the initial model $\mathcal{T}_{\mathcal{R}_{\mathcal{L}}}$, and $\mathcal{L}$'s operational semantics is provided by *deductive inference* in rewriting logic within the theory $\mathcal{R}_{\mathcal{L}}$. That giving a rewriting semantics to a programming language is in practice an easy way to develop executable formal definitions of programming languages, which can then be subjected to different tool-supported formal analysis, is by now a well-established fact [40,2,41,37,36,25,38,8,35,39,14,15].

The rewriting logic semantics of programming languages is related to both algebraic semantics and to structural operational semantics, in the sense of

1571-0661/$ – see front matter
doi:10.1016/j.entcs.2004.06.019

combining and extending both [28]. Since equational logic is a sublogic of rewriting logic, rewriting semantics is a natural generalization of *algebraic semantics* (see, e.g., [42,18,5] for early papers, [31] for the relationship with action semantics, and [17] for a recent textbook) where the semantics of a programming language $\mathcal{L}$ is axiomatized as an equational theory $(\Sigma_{\mathcal{L}}, E_{\mathcal{L}})$, so that $\mathcal{L}$'s mathematical semantics is given by the initial algebra $T_{\Sigma_{\mathcal{L}}/E_{\mathcal{L}}}$, and its operational semantics is given by equational simplification with the (typically Church-Rosser) equations $E_{\mathcal{L}}$. The point of this generalization is that equational logic is well suited for specifying *deterministic* languages, but ill suited for concurrent language specification. In rewriting logic, deterministic features are described by *equations*, but concurrent ones are instead described by *rewrite rules* with a concurrent transition semantics.

It has also been understood from the early stages [22,27,19], that there is a natural semantic mapping of structural operational semantics (SOS) definitions [34] into rewriting logic. In essence, an SOS rule is mapped to a *conditional* rewrite rule [19,38,39,26,28]. Rewriting logic semantics combines the best features of algebraic semantics and SOS in a generalized framework that adds a crucial distinction between equations and rules (determinism vs. concurrency) missing in each of those two formalisms. Furthermore, the agreement between mathematical and operational semantics is extended to this general setting, taking the form of a completeness theorem for rewriting logic [22,6].

Rewriting logic's distinction between equations and rules is of more than academic interest. The point is that, since rewriting with rules R takes place *modulo* the equations E [22], only the rules R contribute to the size of a system's state space, which can be drastically smaller than if all axioms had been given as rules. This observation, combined with the fact that rewriting logic has several high-performance implementations [1,16,11] and associated formal verification tools [12,20], means that we can use rewriting logic language definitions to obtain practical interpreters and language analysis tools essentially *for free*. For example, in the JavaFAN formal analysis tool [14,15], the semantics of Java and the JVM are defined as rewrite theories in Maude, which are then used to perform formal analysis such as symbolic simulation, search, and LTL model checking of Java programs with a performance that compares favorably with that of other Java analysis tools.

Internal advances within rewriting logic have substantially increased its expressive power for programming language semantics purposes, leading to very succinct and expressive semantic rules, which can be executed in current language implementations. Relevant developments in this regard include: