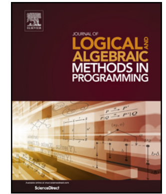
Journal of Logical and Algebraic Methods in Programming 85 (2016) 98–120

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp



Language definitions as rewrite theories

Vlad Rusu [a], Dorel Lucanu [b,*], Traian-Florin Şerbănuţă [c], Andrei Arusoaie [b,a], Andrei Ştefănescu [d], Grigore Roşu [d,b]

[a] Inria Lille Nord Europe, France
[b] "Alexandru Ioan Cuza" University of Iaşi, Romania
[c] University of Bucharest, Romania
[d] University of Illinois at Urbana Champaign, USA

ARTICLE INFO

Article history:
Received 6 October 2014
Received in revised form 31 August 2015
Accepted 1 September 2015
Available online 7 September 2015

Keywords:
Operational semantics
Rewrite theories
Symbolic execution
K Framework
Maude

ABSTRACT

𝕂 is a formal framework for defining operational semantics of programming languages. The 𝕂-Maude compiler translates 𝕂 language definitions to Maude rewrite theories. The compiler enables program execution by using the Maude rewrite engine with the compiled definitions, and program analysis by using various Maude analysis tools. 𝕂 supports symbolic execution in Maude by means of an automatic transformation of language definitions. The transformed definition is called the *symbolic extension* of the original definition. In this paper we investigate the theoretical relationship between 𝕂 language definitions and their Maude translations, between symbolic extensions of 𝕂 definitions and their Maude translations, and how the relationship between 𝕂 definitions and their symbolic extensions is reflected on their respective representations in Maude. In particular, the results show how analysis performed with Maude tools can be formally lifted up to the original language definitions.

© 2015 Elsevier Inc. All rights reserved.

1. Introduction

𝕂 [1] is a formal framework for defining operational semantics of programming languages. The version of 𝕂 that we are using in this paper[1] includes options that have Maude [2] as a backend: the 𝕂 compiler translates a 𝕂 definition into a Maude module, and then, the 𝕂 runner uses Maude to execute or analyse programs in the defined language.

The Maude backend of 𝕂 has been extended with symbolic execution support [3]. Briefly, a 𝕂 language definition is automatically transformed into a *symbolic language definition*. The concrete execution of a program using the symbolic definition is the symbolic execution of the same program using the original language definition. The transformation consists of two steps: (1) incorporating *path conditions* in program configurations, and (2) changing the semantics rules to match on *symbolic configurations* and to automatically update the path conditions. A symbolic execution path is called *feasible* if its path conditions are satisfiable. Two results relating concrete and symbolic program executions are proved in [3]: *coverage*, saying that for each concrete execution there is a feasible symbolic execution along the same program path; and *precision*, saying that for each feasible symbolic execution there is a concrete execution along the same program path. If both coverage and precision hold we say that we have a *symbolic extension* relation between a language and a symbolic language.

* Corresponding author.
E-mail address: dlucanu@info.uaic.ro (D. Lucanu).

[1] 𝕂 version 3.4 is available in the online interface https://fmse.info.uaic.ro/tools/K-3.4/. A virtual machine running 𝕂 3.4 can be downloaded from http://www.kframework.org/imgs/releases/kvm-3.4.zip. The above links are also accessible from the main page of 𝕂 http://www.kframework.org/.

http://dx.doi.org/10.1016/j.jlamp.2015.09.001
2352-2208/© 2015 Elsevier Inc. All rights reserved.

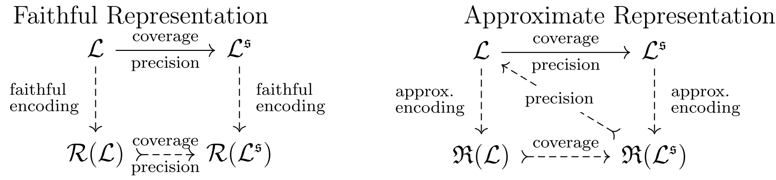


Fig. 1. Faithful vs. approximate representations.

In this paper we propose two ways of representing $\mathbb{K}$ language definitions in Maude: a *faithful* representation and an *approximate* one. We then study the relationship between $\mathbb{K}$ language definitions (including symbolic ones, obtained by the above-described transformation) and their representations in Maude. We also show how the relationship between a language $\mathcal{L}$ and its symbolic extension $\mathcal{L}^{\mathfrak{s}}$ is reflected on their respective representations in Maude. These results ensure that (symbolic) analysis performed with Maude tools on the (faithful and approximate) Maude representations of languages can be lifted up to the original language definitions. The various results that we have obtained are graphically depicted in the diagrams in Fig. 1, where the arrows have the following meaning:

$\xrightarrow{p}$ transformations preserving the property p,

$\rightarrowtail^{p}$ relations preserving the property p.

The dashed arrows show the results proved in this paper.

In the faithful encoding, each semantics rule of the language definition $\mathcal{L}$ is translated into a rewrite rule of the rewrite theory $\mathcal{R}(\mathcal{L})$. Equations are only introduced in order to express equality in the data domain. The resulting rewrite theory is proved to be *executable* by Maude, and the transition system generated by the language definition is shown to be isomorphic to the one generated by the rewrite theory. This ensures that the encoding theories $\mathcal{R}(\mathcal{L})$ and $\mathcal{R}(\mathcal{L}^{\mathfrak{s}})$ also satisfy the coverage and precision properties relating $\mathcal{L}$ and $\mathcal{L}^{\mathfrak{s}}$. Thus, we can say that the rewrite theory $\mathcal{R}(\mathcal{L}^{\mathfrak{s}})$ is a *symbolic extension* of $\mathcal{R}(\mathcal{L})$ (in terms of rewrite theories). This means that the symbolic extension and faithful encoding operations commute, as shown by the commuting diagram in the left-hand side of Fig. 1.

As a consequence, both positive and negative results of reachability analysis obtained on rewrite theories (i.e., by using the Maude *search* command) also hold on the original language definitions. Moreover, all symbolic reachability analysis results obtained on the rewrite theory representation $\mathcal{R}(\mathcal{L}^{\mathfrak{s}})$ of a symbolic language $\mathcal{L}^{\mathfrak{s}}$ also hold on the rewrite theory representation $\mathcal{R}(\mathcal{L})$ of the language $\mathcal{L}$. The latter property is analogous to the results obtained in [4], where *rewriting modulo SMT* is shown to be related to (usual) rewriting in a *sound* and *complete* way.

For nontrivial language definitions the faithful encoding is not very practical, because it typically generates a huge state-space that is not amenable to reachability analysis. This is why we introduce approximate representations of language definitions as *two-layered rewrite theories*. These approximations are obtained by splitting the semantic rules of the language into two sets, called *layers*, such that the first layer forms a terminating rewrite system. The one-step rewriting in such a theory is obtained by computing an irreducible form w.r.t. rules from the first layer (according to a given strategy), and then applying a rule from the second layer.

In an (approximating) two-layered rewrite theory $\mathfrak{R}(\mathcal{L})$, only a subset of the executions of programs in the original language $\mathcal{L}$ are represented, i.e., $\mathfrak{R}(\mathcal{L})$ is an under-approximation of $\mathcal{L}$. The consequence is that only positive results of reachability analysis on the two-layered rewrite theories can be lifted up to the corresponding language definitions. The approximate encoding of a language by a two-layered rewrite theory can also be seen as the output of a *compiler* that solves some semantics choices left by the language definition at compile-time. For example, in C and C++, the order in which the operands of addition are evaluated is a compile-time choice. By turning the operand-evaluation rules into first-layer rules, and by letting Maude automatically execute these rules in various orders according to certain strategies, one can reproduce the various design compile-time choices for the evaluation of arguments. However, this comes at a price. Due to the side effects of some operators, there are C/C++ programs with nondeterministic behaviour. This feature cannot be exhibited with the operand-evaluation rules in the first layer; in order to exhibit the nondeterminism, the rules evaluating the operators must be in the second layer. For programs using operators without side effect, there is no reason to introduce their evaluation rules in the second layer because the result is always the same due the confluence of these rules.

The approximate representations are also useful during the design of the semantics of a language. If one wishes to test the behaviour of some semantical rule, then one can include only that rule in the second layer and use the $\mathbb{K}$ stepper to see the effect of the rule.

We note that approximating two-layered rewrite theories have some limitations: only the coverage property relating the language definition $\mathcal{L}$ to its symbolic version $\mathcal{L}^{\mathfrak{s}}$ also holds on their respective approximate-encoding theories; the precision property holds only in some restricted cases (presented in Theorem 6 later in the paper). Problematic for this are the conditional rules. The symbolic version must execute both branches, when the condition holds and when the condition does not hold. Therefore the rules corresponding to the two cases must be in the second layer, otherwise the first layer could become non-terminating due to iterative statements. This means that some rules which are in the first layer in $\mathfrak{R}(\mathcal{L})$ are in the second layer in $\mathfrak{R}(\mathcal{L}^{\mathfrak{s}})$. This could affect the order in which the rules are being executed. Recall that the precision

Download English Version:

https://daneshyari.com/en/article/432608

Download Persian Version:

https://daneshyari.com/article/432608

Daneshyari.com