Modular Language Specifications for Program Analysis

Mark Hills

Centrum Wiskunde & Informatica, Amsterdam, The Netherlands

Using language definition frameworks, a program analysis can be defined as a non-standard program evaluation semantics over an appropriate domain of abstract values. An advantage of using executable frameworks, such as rewriting logic semantics [15,16] or K [18], is that the analysis itself can be performed by encoding programs as algebraic terms and evaluating them, using the defined semantics, in an execution engine such as Maude [5]. An example of this approach is the unit safety analysis developed first for BC [3] (a small calculator language) and then for a small subset of C [17]. In this analysis, the abstract values were units of measurement [1] (e.g., meters, seconds, lumens). The analysis semantics modeled the operation of language constructs over units, with execution of programs in the semantics used to detect errors in cases where units were used incorrectly, e.g., when values of two different units were added or compared.

However, in this work, rules specific to the analysis were tangled with analysis-agnostic rules, making it challenging to reuse parts of the existing semantics in a new analysis. Improving this separation of analysis-generic and analysis-specific specifications, and providing better techniques and constructs for modularity, motivated our work both on K and on policy frameworks [7], modular program analysis frameworks designed to allow sharing between analysis policies (the semantics and annotation language used by an analysis) and between frameworks for different languages. Our first instantiation of this was the C Policy Framework [8], or CPF. The core of CPF includes an analysis-generic frontend, allowing annotations to be added (in comments) as function contracts or within function bodies, and an abstract, modular C semantics, designed with a focus on reuse. To define a specific policy, this core is extended with an analysis-specific definition of abstract values, a specific annotation language, and equational definitions for a number of policy-generic hooks, representing points in the semantics that differ between analysis policies. A second framework was then defined for SILF [10], a simple, paradigmatic imperative language, which provides a simpler, more modular environment for experimenting with new analysis policies.

In our work on policy frameworks we developed multiple policies for both C and SILF, including a sample memory policy for detecting cases where null pointers in C programs are dereferenced [7], a type checker for SILF [10], and unit safety policies for both SILF [10] and C [8]. An example program fragment using the C Units policy is shown in Figure 1, showing the preconditions and postcondition for the radiationLength function—the policy, UNITS, is given explicitly, allowing annotations for multiple policies to be included in the same program file. Figure 2 shows the actual rules used to check unit arithmetic for addition and multiplication; Rule 2 is an error case, since we require that the
double radiationLength(Element * material) {
    double A = material->atomicWeight;
    double Z = material->atomicNumber;
    double L = log( 184.15 / pow(Z, 1.0/3.0) );
    double Lp = log( 1194.0 / pow(Z, 2.0/3.0) );
    return ( 4.0 * alpha * re * re) * ( NA / A ) * ( Z * Z * L + Z * Lp );
}

Fig. 1. Electron Energy Example, in C.

These policies illustrate that reuse works in practice: the memory and unit safety policies in C, and the type and unit safety policies in SILF, both reuse large parts of the policy frameworks for C and SILF, respectively, while the unit safety policies for C and SILF share the same definition of the units domain. The unit safety policy for C also showed that this approach to program analysis can work on practical programs, providing a similar annotation burden, false positive rate, and error detection rate as Osprey [11], a purpose-built tool for checking dimension safety of C programs, while also providing reasonable performance.

Unfortunately, even though the frameworks for C and SILF were structured modularly [14,2], they were sometimes not modular to extend in practice. To actually implement a new policy, the implementer must possess detailed knowledge of both Maude and of the entire provided core semantics. This includes knowing which hooks must be defined to provide the policy-specific semantics and which modules provide base variants of functionality that can be directly reused or extended. To address this, we are investigating two complementary techniques, described below, which are currently working in a prototype implementation. First, we are using reflective techniques to allow policy frameworks to be annotated, allowing information about extension points to be automatically extracted. Second, we are defining DSLs for specifying parts of analysis policies. Currently, we are focusing on analysis domains and policy rules, with plans to also investigate a DSL for modeling memory operations—an issue in languages such as C, where different policies may need more- or less-precise representations of heap-allocated values. The end goal is to move all policy-specific definitions into higher-level, modular, domain-specific specifications, allowing new policies to be created without requiring an in-depth knowledge of either rewriting logic semantics or the K framework.
Reflective Extraction of Extension Point Information: Framework designers can indicate extension points in the framework using annotations. Using a combination of standard rewriting and Maude’s reflective capabilities, these annotations are then used to automatically extract information on extension points, which can then be used by interactive tools during the definition of an analysis policy. For instance, extracting information on policy hooks generates a relation from defining modules to the signature of each defined hook. This allows a separation between framework designers, who need in-depth knowledge of the entire framework, and policy designers, who can then focus just on those parts of the framework that are extended (or just used) in policies.

DSLs for Policy Definition: Using Rascal, we are designing domain-specific specification languages targeted at specific parts of analysis policies. Figure 3 shows how specifications defined in these languages are used. Given a framework definition, extension point information is first extracted. This is used to generate template information for defining policy rules, and is being extended to provide information for other DSLs as well. Programs in the two DSLs are then used to generate the analysis policy specification. From the DSL for domain values, a Maude specification of the analysis domain is generated. This specification includes rules for pretty-printing, creating, and filtering domain values (the latter needed for cases where annotations for multiple policies are present in the same program). An example specification in this DSL including custom pretty-printing rules is shown in Figure 4. From the DSL for policy rules, the analysis-specific behavior for
each hook is defined, making use of both the policy values defined in the policy
values language and of predefined reporting operators for errors and warnings
(potentially with source locations), to allow for more detailed error messages).
The generated policy can then be used like existing, manually built policies, with
programs evaluated in the policy semantics to discover potential errors.

References
1. NIST Website, International System of Units (SI).
3. F. Chen, G. Roşu, and R. P. Venkatesan. Rule-Based Analysis of Dimensional
4. M. Clavel. Reflection in Rewriting Logic: Metalogical Foundations and Metapro-
5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L.
   Talcott, editors. All About Maude - A High-Performance Logical Framework, How
to Specify, Program and Verify Systems in Rewriting Logic, volume 4350 of LNCS.
   UIUCDCS-R-2008-2931, Department of Computer Science, University of Illinois at
   Urbana-Champaign, 2008.
   of Units of Measurement in C. In Proc. of RULE’08, volume 290 of ENTCS, pages
9. M. Hills, P. Klint, and J. Vinju. RLSRunner: Linking Rascal with K for Program
10. M. Hills and G. Roşu. A Rewriting Logic Semantics Approach To Modular Program
    Analysis. In Proc. of RTA ’10, volume 6 of Leibniz International Proc. in Informatics,
11. L. Jiang and Z. Su. Osprey: A Practical Type System for Validating Dimensional
12. P. Klint, T. van der Storm, and J. Vinju. RASCAL: A Domain Specific Language
14. J. Meseguer and C. Braga. Modular Rewriting Semantics of Programming Lan-
    2004.
15. J. Meseguer and G. Rosu. The rewriting logic semantics project. Theoretical