Language Design for Meta-Programming in the Software Composition Domain

Paul Klint, Jurgen Vinju, Tijs van der Storm

Centrum voor Wiskunde en Informatica (CWI)
and
Informatics Institute, University of Amsterdam

Abstract. How would a language look like that is specially designed for solving meta-programming problems in the software composition domain? We present requirements for and design of Rascal, a new language for solving meta-programming problems that fit the Extract-Analyze-SYnthesize (EASY) paradigm.

1 Introduction

The question how to compose software is as old as computer science itself. For each approach one has to face several realities when applying it in practice:

- How to incorporate existing software in the composition framework?
- How to deal with the plethora of languages and intermediate formats that systems use to implement functionality and to exchange data?

Composition techniques can be placed on a linear scale. On the left-hand side of the scale are static, invasive, techniques as proposed by Assmann [1] that view the component as a white box and use meta-programming to adapt components to the composition framework. Invasive composition tools are static meta-programs that analyse and modify specific components before compilation time such that they are connected and can be integrated into a new software system. On the right-hand side are techniques that can dynamically combine existing components at the binary level without any software modification. We have, for instance, applied this with success in the ToolBus architecture [2]. The idea is that a process script defines the legal tool cooperations and that the binary program remains a black box and is encapsulated in a wrapper to let it satisfy the requirements of the ToolBus framework. Many composition techniques can be placed somewhere in the middle of this scale.

Composition of heterogeneous components that use different languages and intermediate formats requires an approach called *grammarware engineering* that we have introduced in [3]. It should be part of any meta-programming framework. The process of component adaptation can be seen as an instance of the Extract-Analyze-SYnthesize (EASY) paradigm and is shown in Figure 1. We start with the original components and extract information from them and record this

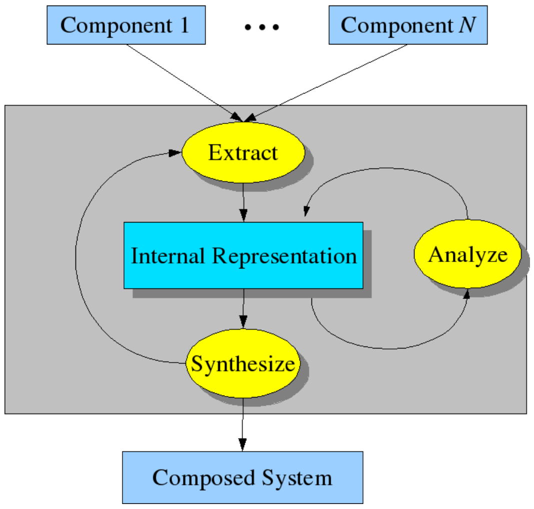

Fig. 1. The Extract-Analyze-SYnthesize Paradigm applied to Software Composition

in an internal representation. The latter is analyzed repeatedly until sufficient information is available to synthesize the composed system. It may even happen that requirements for synthesis lead to the necessity to extract more information from the original components. Many meta-programming systems take the syntax tree as internal representation and use tree transformation to achieve their goal.

The general theme of this paper is to explore the requirements for a meta-programming language to express invasive software composition of heterogeneous software components. We will now first explore requirements (Section 2) and then we describe the progress of the design of a new language called Rascal that satisfies these requirements (Section 3).

2 Requirements

The challenge of composing heterogeneous software components can be solved if we have the following functionality at our disposal:

– Definition of grammars for arbitrary languages and data formats and parsing of source code or data described by these grammars resulting in syntax trees.
– Semantic analysis of syntax trees.
– Transformation of syntax trees and code generation using results of analysis.

This spans the universe from parser generators and attribute grammars to software query languages and term rewriting systems. Each of these approaches is good at solving a part of the problem, but how can we solve the whole problem? To integrate these techniques is a non-trivial software composition problem in its own right. Maslow's saying that "if all you have is a hammer, everything looks like a nail" [4] is also true in this domain. Attribute grammars are good for analysis but they are stretched beyond recognition when also used for transformation. Term rewriting is good for transformation but becomes artificial when used for software analysis. Relational calculus is good for querying source code facts but is unusable for transformation. What we need is an integrated view on the problem that uses the optimal technique for each subdomain. Here is our shortlist of essential techniques:

– Generalized LR or LL parser generation and parsing for syntax analysis.
– Relational calculus for semantic analysis.
– Term rewriting for transformation and code generation.

Remains the question how to integrate these techniques?

3 Rascal

The goals of Rascal are: (a) to remove the cognitive and computational overhead of integrating analysis and transformation tools, (b) to provide a safe and interactive environment for constructing and experimenting with large and complicated source code analyses and transformations such as needed for software composition, refactoring, code generation and renovation and (c) to be easily understandable by a large group of computer programming experts. We briefly review its main features.

Values, Datatypes and Types Rascal is a value-oriented language. This means that values are immutable and are always freshly constructed from existing parts and that sharing and aliasing problems are completely avoided. The language provides a rich set of datatypes. From Booleans, infinite precision integers and reals to strings and source code locations. From lists, (optionally labelled) tuples and sets to maps and relations. From untyped tree structures to fully typed abstract datatypes. Syntax trees that are the result of parsing source files are represented as abstract datatypes. There is a wealth of built-in operators and library functions available on the standard datatypes.

Pattern matching Pattern matching is *the* mechanism for case distinction in Rascal. We provide string matching based on regular expressions, list (associative) and set (associative, commutative, identity) matching, matching of abstract datatypes, and matching of concrete syntax patterns. All these forms of matching can be used in a single pattern. Patterns may contain variables that are bound when the match is successful. Patterns can also be used in an explicit match operator and can then be part of larger boolean expressions. Since a pattern match may have more than one solution, local backtracking over the alternatives of a match is provided.

Comprehensions, Generators, Enumerators and Tests Generators are used in comprehensions for lists, sets and maps and come in two flavours: tests and enumerators. The former test the validity of including a value in the result of a comprehension, the latter enumerate the values in a given (finite) domain, be it the elements in a list, the substrings of a string, or—and this is a novelty—all the nodes in a tree. Each value that is enumerated is first matched against a pattern before it is included in the results of the enumerator.

Visiting and Transforming Visiting the elements of a datastructure—in particular syntax trees—is one of the most common operations in our domain and we give it first class support by way of *visit expressions*. A visit expression consists of an expression that may yield an arbitrarily complex subject value and a number of cases. All the elements of the subject are visited and when one of the cases matches the statements associated with that case are executed. These cases may: (a) cause some side effect; (b) replace the current element; (c) fail and thereby undoing all side-effects due to the successful match itself and the execution of the statements sofar. The value of a visit expression is the original subject value with all replacements made as dictated by matching cases and thus forms the basis for software synthesis and code generation. The visit order can be explicitly defined by the programmer.

4 Concluding Remarks

Rascal is intended for the complete meta-programming domain and can therefore also be applied in various phases of the software composition or adaptation process. An initial version of the language has been designed and implemented[1] and experiments in various cases are in progress. We welcome feedback from the software composition community.

References

1. U. Assmann. *Invasive Software Composition.* Springer, 2003.
2. J.A. Bergstra and P. Klint. The discrete time ToolBus – a software coordination architecture. *Science of Computer Programming*, 31(2-3):205–229, July 1998.
3. P. Klint, R. Lämmel, and C. Verhoef. Toward an engineering discipline for grammarware. *ACM Trans. Softw. Eng. Methodol.*, 14(3):331–380, 2005.
4. A. H. Maslow. *The Psychology of Science: A Reconnaissance.* Harper & Row (1966) and Maurice Basset (2002), 1966.

[1] A command-line version and an Eclipse version are available from `http://www.meta-environment.org`