Composing Code Rewriting Directives

Laure Gonnord & Sébastien Mosser

12 octobre 2020

General Context

CAPESA Project context This internship takes place in the context of the CAPESA joined team between CASH (Lyon) and the ACE group (Montréal). In this project we study the characteristics of code evolution *at small granularity*, by focusing on *small-steps* changes. Contrarily to existing approaches, we want to address several evolution mechanisms, but restrict ourselves to changes at a small level of granularity : small code changes in a program. The internship we propose in an instance of one of the challenges we want to address in the project.

Infering structural information from code rewrites Rewriting code tools (e.g. Spoon [PMP+15]) that perform source-to-source transformations of a given program, are used everywhere, from code optimisation to automatic repairing, anti-pattern solving. However, all these tools face the same kinds of problems : how to deal with conflicting writes ?

The general problem is the following : consider two rewriting rules ρ_1 and ρ_2, both to be applied to the very same program p. Each rule is defined as a function that takes as input a program, and produce another one according to its semantics. If ρ_1 and ρ_2 interfere (*e.g.*, the former produces elements that will be rewritten by the latter), applying ρ_1 then ρ_2 does not yield the same program than applying ρ_2 and then ρ_1. We proposed in previous work [MBFD12] a commutative operator that complements the classical function composition operator (where $\rho_2 \bullet \rho_1(p) \neq \rho_1 \bullet \rho_2(p)$) with a parallel semantics. Using this operator (denoted as $||$), applying both rules always yields the same result, *i.e.* the expected program or an error if the rules interfere. We propose here to enhance this work by analyzing what a conflict is from a code rewriting point of view, and how it can be anticipated and/or automatically solved.

There are many instances of this problem : for instance, in the context of a recent collaboration with the *Université du Québec à Montréal*, has been proposed a set of energetic rewriting rules that permits to rewrite over-consuming android statements into less consuming ones. We can also cite the tool Alive [1] [LMNR15] that performs peephole optimizations inside the LLVM compiler. Graph transformations has also investigated this problem by working on conflicting graph transformations identification and automated scheduling [MTR05, SVL15]. The TOM language is dedicated to code rewriting [BBK+07], and the Coccinelle approach address a similar problem with flow-based program matching [BDH+09]. We propose to explore this problem from an innovative point of view, considering techniques designed by the compilation and formal method community to complement the existing software engineering approaches.

1. https://blog.regehr.org/archives/1170

UQAM – Montreal & LIP – UMR CNRS / ENS Lyon / UCB Lyon 1 / INRIA
Contact E-mail: Laure.Gonnord@ens-lyon.fr and mosser.sebastien@uqam.ca

Internship Objective

In this internship, we propose to formalize the notion of code rewriting in the specific context of program refactoring [Fow99]. The idea is to *(i)* implement program rewriting rules to support refactoring directives, *(ii)* formally analyze theses rule definitions according to different methods and *(iii)* define an empirical benchmark that measure the accuracy of the conflict detection mechanisms when applied to real-life programs. We might find inspiration in paper coming from communities such as package systems, code rewriting, sat-solving, and for the "optimisation" problem from logic (unsat/sat core), operational research (with the good encoding and a reasonable objective function), ...

The expected result is a prototype that demonstrates main refactoring rules applied simultaneously to reference programs, and to confront theoretical results with empirical benchmarks.

Expected skills

— A taste for object-oriented programming ;
— The ability to identify trade-off among multiple solutions ;
— Good communication skills and will to share ideas and discuss results.

This work will be cosupervised mostly by Laure Gonnord and Sebastien Mosser. The internship place can be discussed.

Références

[BBK+07] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom : Piggybacking rewriting on java. In *Conference on Rewriting Techniques and Applications - RTA'07*, volume 4533 of *LNCS*, pages 36–47, Paris/France, France, June 2007. Springer-Verlag.

[BDH+09] Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. A foundation for flow-based program matching : Using temporal logic and model checking. *SIGPLAN Not.*, 44(1) :114–126, January 2009.

[Fow99] Martin Fowler. *Refactoring : Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[LMNR15] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 22–32, New York, NY, USA, 2015. ACM.

[MBFD12] Sébastien Mosser, Mireille Blay-Fornarino, and Laurence Duchien. *A Commutative Model Composition Operator to Support Software Adaptation*, pages 4–19. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[MTR05] Tom Mens, Gabriele Taentzer, and Olga Runge. Detecting structural refactoring conflicts using critical pair analysis. *Electronic Notes in Theoretical Computer Science*, 127(3) :113 – 128, 2005. Proceedings of the Workshop on Software Evolution through Transformations : Model-based vs. Implementation-level Solutions (SETra 2004).

[PMP+15] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon : A library for implementing analyses and transformations of java source code. *Software : Practice and Experience*, 46 :1155–1179, 2015.

[SVL15] Eugene Syriani, Hans Vangheluwe, and Brian LaShomb. T-core : a framework for custom-built model transformation engines. *Software & Systems Modeling*, 14(3) :1215–1243, Jul 2015.