Composing Code Rewriting Directives
Laure Gonnord & Sébastien Mosser
16 octobre 2017

General Context

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 $\rho_1$ and $\rho_2$, 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 $\rho_1$ and $\rho_2$ interfere (e.g., the former produces elements that will be rewritten by the latter), applying $\rho_1$ then $\rho_2$ does not yield the same program than applying $\rho_2$ and then $\rho_1$. We proposed in previous work [MBFD12] a commutative operator that complements the classical function composition operator (where $\rho_2 \circ \rho_1(p) \neq \rho_1 \circ \rho_2(p)$) with a parallel semantics. Using this operator (denoted as $\parallel$), 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.

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 these 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), …

\(^1\) https://blog.regehr.org/archives/1170
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 supervised mostly by Sebastien Mosser in Sophia Antipolis and remotely by Laure Gonnord in Lyon.

**Références**


