Ph. D. Proposal for 2016

Certified refactoring for C programs

<table>
<thead>
<tr>
<th>Ph. D. Thesis Director</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME, Surname          : DOUENCE, Rémi</td>
</tr>
<tr>
<td>Research Team         : Ascola</td>
</tr>
<tr>
<td>Laboratory            : LINA (UMR 6241)</td>
</tr>
<tr>
<td>Affiliation           : Mines Nantes</td>
</tr>
<tr>
<td>E-mail                : Remi.Douence (at) mines-nantes.fr</td>
</tr>
<tr>
<td>Advising Rate         : 50 %</td>
</tr>
<tr>
<td>Current Ph. D. Students : 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ph. D. adviser</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME, Surname  : COHEN, Julien</td>
</tr>
<tr>
<td>Laboratory     : Université de Nantes (Polytech)</td>
</tr>
<tr>
<td>Advising Rate  : 50 %</td>
</tr>
</tbody>
</table>
Certified refactoring for C programs

Abstract. Refactoring tools are not correct in every possible cases, and programmers cannot trust them. We are interested in building certified refactoring tools and we focus on handling C programs, which are used in industrial developments. We are particularly interested in refactoring operations provided by popular tools, such as Eclipse. We will also consider a complex program transformation based on a sequence of atomic refactoring operations, such as a software architecture transformation or a desobfuscation tool. Finally, we will study how the tactic language of Coq can be used to support the composition of refactoring operations.

Keywords. Refactoring, C language, Coq proof assistant, CompCert C, program transformations, remodularization.
Introduction

Context

In software engineering, series of evolutions downgrade the quality of code [Leh96, HL05]. Indeed, each modification gets harder to implement.

This eventually requires to fix or change the structure of the program, without changing its behavior, in order to ease future evolutions [ADD+10, LML+15]. Such architecture modifications are integrated in Agile development processes.

Unfortunately, tools that change the structure of programs without modifying their behavior (i.e., refactoring tools) are generally not correct [Soa12]. Indeed, refactoring tools may change the behavior of the program and so their use require systematic and extensive testing in order to detect the newly introduced bugs. This is the reason why users do not trust these tools and tend to prefer manual modifications [SSS15, BS15].

Problems and opportunities

This thesis addresses the correctness of refactoring tools. Since the grammar of C is rich, numerous cases must be taken into account in the design of such tools, which makes the task difficult. Moreover, refactoring operations should preserve as much as possible the layout of the source code as well as its comments and its preprocessing directives (macros, pragma, etc.).

In addition to a difficult design, the implementation or a refactoring tool is critical. So a theoretical study of a transformation is not sufficient to ensure the quality of the final tool.

For this reason, a correct-by-construction approach (e.g., [SEdM08]) should work well here. Up to now, this could not be applied to industrial strength programming languages, which lack of formal semantics (existing formalizations are restricted to a subset of the languages, for instance Featherweight Java [IPW01] is a functional subset of Java). A few works are devoted to prove the correctness of refactoring tools, but for micro-langages only (such as [ST08]).

Thanks to the CompCert C project [Ler15, BL09], there is now a fully formalized semantics for C (including MISRA C used by car industry). This formalization is supported by the proof assistant Coq that provides powerful mechanisms for automated reasoning and certified code generation. This provides a good basis for building certified refactoring tools (see [Coh15]).
Work

Goals

The goal of this thesis is to provide certified refactoring tools for C. Both atomic refactoring operation and composite ones must be addressed. The work should be validated by a case study coming from an industrial partner.

At the completion of this thesis, the doctor will have high skills in formalization and automated proof of programs. Code certification is a hot topic both in the research community and in industry. The doctor will also be an expert of the C language (semantics and program transformations).

Work plan

Year 1: State of the art, implementation and proof of a first transformation.
— List operations which are formally studied in publications as well as the associated technics (for instance: [SDS+10, SK13]).
— Identify the most useful transformations.
— Identify more complex transformation relying on composition of simpler ones (for instance see [Ker04, CDA12, ACR13]).
— Study why these transformations are difficult to implement correctly (for instance interactions between the target syntactic construct and the other ones).
— Study formalizations of industrial world programming languages (CompCert C).
— Identify case studies for existing applications, such as software architecture or desobfuscation transformations.
— Build and prove a simple refactoring operation (in Coq).

Year 2: Description of a case study and its associated operations. Implementation of some of the corresponding transformations.
The case study will provide the basis for selecting the refactoring operations to implement and prove. We will distinguish atomic operations (to be fully treated) and composite operations (whose implementation and properties come “for free” [CA13]).
— Build and prove the selected operations.
— Factorize the proof process as a set of tactics.
— Study how (much) the tactic language of Coq can be used to define refactoring strategies to orchestrate transformations.

Year 3: Finalization of the operations and case studies. Consider the integration of the work in a development environment (Eclipse, Frama-C . . .). Further possible work:
— Study the complementarity of proof and test in certified systems (some parts of the tools are yet beyond reach of full formal treatment).
— Study the formalization of C preprocessors [Vit03, Gar05, Pad09, Spi10] and adapt the prototype to make it preserve preprocessor directives.
Candidates

Skills

— Mandatory skills: formal logics, C programming language, software engineering.
— Other skills: functional programming, programming language semantics, Hoare’s logic, induction, proof assistants.

Selected candidates

The position is open, there is currently no selected candidate.
Bibliography

and visitor implementations in Java. In Software Engineering and Advanced
Applications (SEAA), 2013 39th EUROMICRO Conference on, pages 25–32,
Sept 2013.

[ADD10] Nicolas Anquetil, Simon Denier, Stéphane Ducasse, Jannik Laval, Damien
Pollet, Roland Ducournau, Rodolphe Giroudeau, Marianne Huchard, Jean-
Claude König, and Abdelhak Djamel Seriai. Software (re)modularization:
Fight against the structure erosion and migration preparation, March 2010.

[BL09] Sandrine Blazy and Xavier Leroy. Mechanized semantics for the Clight subset

[BS15] J. Brant and F. Steimann. Refactoring tools are trustworthy enough and trust

[CA13] Julien Cohen and Akram Ajouli. Practical use of static composition of refac-
toring operations. In Proceedings of the 28th Annual ACM Symposium on Ap-

continuing modular maintenance. In Software Maintenance and Reengineering

https://hal.archives-ouvertes.fr/hal-01248121.

[Gar05] A. Garrido. Program refactoring in the presence of preprocessor directives.
PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA,
2005.

[HL05] Lorin Hochstein and Mikael Lindvall. Combating architectural degeneration:

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java:
a minimal core calculus for Java and GJ. ACM Trans. Program. Lang. Syst.,


CV of the thesis advisor

Rémi Douence defended his PhD thesis on functional languages compilation in 1996. His post doc at Carnegie-Mellon University in 1997 was devoted to static analysis of software architecture languages. In 1998, he was a expert engineer for Inria on partial evaluation of C programs. He is currently an assistant professor at Mines de Nantes in the Ascola Inria team. He is interested in programming languages for: functional programming, software architecture, reflexion, software components, aspect programming, constraint solvers, web service composition. He is interested in both expressivity and safety. He defended is HDR in 2015.

CV of the thesis co-advisor

Julien Cohen defended his Phd thesis on programming language in 2004 in Évry. Then, he was ATER at l'Univeristé de Paris-Sud – Orsay (LRI lab). He is assistant professor at Polytech Nantes since 2005 (LINA lab). He has advised a PhD thesis defended in 2013 on the application of refactoring tools to implement invertible software architecture transformations in Java and Haskell. In this context, he has developed and he maintains an IntelliJ plugin of Java refactoring operations. Recently, he has prototyped a refactoring tool for CompCert C.