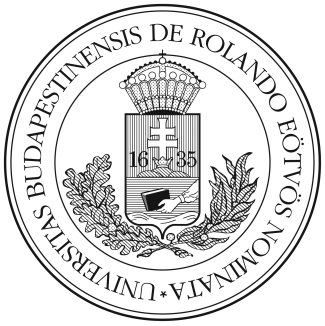
Verification and Application of Program Transformations

Theses of the doctoral dissertation

Dániel Horpácsi

Supervisor: Zoltán Horváth



Eötvös Loránd University, Doctoral School of Informatics
Foundations and Methodologies of Informatics

Head of School: Erzsébet Csuhaj-Varjú, DSc.
Head of Program: Zoltán Horváth, PhD.

August 2018

Introduction

Program transformations serve as essential, fundamental elements of software development. As long as we express descriptions, algorithms, and solutions to various problems with programs, transformations will be part of the development process.

There is a distinguished class of program transformations: refactorings change the program in a way that improves quality but does not affect meaning. The importance of refactoring was recognized by both industry and academia a long time ago. A number of specifications and tool implementations have been created for various programming languages, with the aim of assisting and promoting semi-automatic refactoring. Such tools proved themselves to be extremely useful and they can significantly increase the productivity of programmers. Just think of a simple modification like renaming a function, but in a million-line software; then imagine doing it by hand, or letting a dedicated software do the job instead. In industrial scale, extensive refactoring is impractical and dangerous without tool support.

As the dissertation points out, refactoring tools are of ever increasing interest, but at present they are not trustworthy enough, and they have a rarely discussed potential to implement a number of different kinds of semantics-aware program transformations.

Program analysis and transformation in Erlang

Refactoring systems implement complex static analysis and transformation to support precise side condition checks and advanced program manipulation. In 2006, ELTE started to develop a refactoring system for Erlang, a functional programming language, in cooperation with Ericsson Hungary. I joined the project one year later, and since then I contributed to almost all components of the system. The refactoring tool was designed to support fast and incremental static semantic analysis, as well as complex program transformations on Erlang programs.

From the very beginning, the focus was on building an industrial-scale solution. Correctness of the algorithms was supported by a huge number of manually written test cases, but formal verification seemed to be out of reach. It raised my attention to investigate possibilities in static and advanced dynamic verification of the correctness of the analysis and transformation system. My primary motivation was to gain a deep understanding of static analysis and program transformations, and to increase the trustworthiness of the automatic transformations as much as possible, but I also dealt with special applications of transformations in the refactoring system.

Scope

Generally speaking, my dissertation is dealing with solving complex problems by carefully identifying and precisely defining problem-specific abstractions. This involves understanding and establishing syntactic and semantic relationship between the original problem and the solution. The first two theses literally introduce new languages (with new abstractions) that bring improvements on how effectively and reliable we can define solutions to complex problems. The third thesis is also language-oriented, yet it is more about identifying and exploiting capabilities of well-known language processors.

From the value proposition point of view, my work improves on the reliability, the trustworthiness of refactoring transformation definitions, and in addition, I showcase a special application of the capabilities of a refactoring system by using it for implementing language extensions. The approach I use along with the voice of the presentation is highly determined by the fact that my entire doctoral research was focussed around static analysis and transformation of software source code.

Summary of contributions

I contributed to the design of the analysis and transformation framework [10, 3] of our Erlang refactoring tool, as well as to the design and implementation of various concrete analysis [9, 13, 4] and transformation [12] steps. I also took part in a project that used the system for automatic parallelisation of Erlang programs [2, 1]. These results provide the background for the contributions of the dissertation.

The first two theses deal with dynamic and static verification of refactoring transformations. The dynamic verification approach is based on property-based testing of refactoring with randomly generated programs. My main contribution is the method of synthesising data generators for L-attribute grammars [5], and the attribute grammar I composed for a well-formed subset of Erlang. I also contributed to the application of the generator for property-based testing of analysis and transformation components [14]. A few years later I shifted my attention towards static, formal verification of refactoring transformations. With the help of Judit Kőszegi, I designed a refactoring specification language [7] that allows for defining executable transformations that are automatically verifiable for semantics-preservation. For the demonstration of the applicability of the approach, we composed a complex case study refactoring [8]. The third thesis is about a novel approach to using refactoring frameworks for extending languages [6], which has been applied to implement embedding workflow description operators in Erlang [11].

Property-based testing of refactoring systems

"Discovering the unexpected is more important than confirming the known."
(George E. P. Box)

The first thesis defines a method to dynamic verification of refactoring correctness. Although testing, unlike formal proofs, cannot ensure the absence of errors, it is seen as the most important and widespread verification technique for large-scale software. In its simplest form, testing is based on the inspection of output values (and side effects) on given input values, but there is an obvious limitation: the input domain of the computation is typically infinite, which makes it practically impossible to enumerate all the possible input values.

In case-based testing, a number of test cases are composed manually, which are expected to cover all the important (and representative) input-output pairs, and execute as many paths in the program as possible. The result of this process is a so-called test suite, which can be used for unit testing and regression testing, and can be regarded as a partial input-output specification of the program. In the case of refactoring transformations, such a test collection would contain a number of source code pairs showing the result of a particular transformation on a piece of code. There is an apparent connection between the original and the transformed program: they have to be semantically equivalent.

Property-based testing (PBT) is a generalisation of the traditional case-based testing: rather than enumerating a huge number of input-output pairs, we specify the expected behaviour of the program in a precondition-postcondition fashion. Namely, concrete input values are generalized to pre-conditions on the input domain (data set), and concrete output values are replaced by post-conditions on the resulting data. The system performs the testing by randomly generating the elements of the input domain by consecutive calls to the input data generator. This testing technique can be very effective, but for complex programs operating on complex data, both specifying the data generator and establishing the formal property may be pretty challenging.

In refactoring systems, both the input and the output is program source code. When specifying the correctness of a refactoring transformation, the precondition would ensure that the input is a compilable source code, whilst the postcondition tells if the resulting source code is semantically equivalent to the original one. If we can make sure that the data generator for the input domain only generates compilable (or well-formed) programs, no further precondition is necessary, but creating a data generator for the language of well-formed Erlang programs is not straightforward.

Writing a data generator for producing random Erlang programs by hand would be a tedious job, and would likely result in an incomprehensible imperative program containing a large amount of boilerplate. Since languages are practically defined with formal grammars, I believe that the problem of program generation is best solved by employing a formal grammar to synthesise a data generator for syntax trees of well-formed programs. From a declarative, readable description, an imperative generator is created automatically. With this, we solve the problem in two halves: defining the language of interest with a formal grammar, and creating a method that turns the grammar into a data generator.

Thesis 1. *I have developed a method for transforming stochastic L-attributed grammars into QuickCheck data generators. I have composed an L-attributed grammar for a sub-language of Erlang and used the previous method to synthesise a data generator for well-formed programs. By using this generator, analysis and transformation implementations have been verified on randomly generated, semantically valid programs.*

This result enables property-based testing of refactoring engines with randomly generated source code. The generic approach I designed is based on L-attributed grammars, which are given a meaning by transformation to the language of QuickCheck generators. This conversion turns an inherently declarative, easily comprehensible definition into an imperative implementation. As a result, sentences of any language defined with an L-Attribute grammar can be sampled, and shrunk in property-based test properties. In my particular case study, the Erlang programming language was formalised with an attribute grammar to enable random generation of well-formed Erlang programs, which comply with the extended static semantics of the language.

Related publications. The original idea of using property based testing for verifying refactoring correctness, as well as the attribute grammar language and the method of translating attribute grammars to data generators is presented in [5]. The paper also discusses bi-simulation based program equivalence checking, which uses the grammar-based generator. Another application of the Erlang program generator is presented in [14], where it is used in testing static analysis against its specification.

Verifiable and executable specification of refactoring

> *"A language that doesn't affect the way you think about programming is not worth knowing." (Alan J. Perlis)*

Refactorings can be defined in various formats at various abstraction levels. Some widely referred refactoring catalogues use informal, English explanations supported by some simple examples, while academy keeps looking for mathematically precise definition methods that are amenable to static verification. Typically, refactorings are expressed in high-level, general purpose programming languages, which are impractical to verify because they are too complex to have tractable formal semantics definitions (i.e. definitions which can be reasoned about easily). Yet, sometimes more correctness guarantees are of demand than that of dynamic verification can provide.

Program transformations and refactorings are understood as algorithms taking source code and returning source code, but for practical reasons, this process is usually divided into three phases: analysis, transformation and synthesis. Syntactic and semantic analysis turns source code into a complex program model, the model is altered by transformations, and finally, synthesis converts the model back to source code. If we treat analysis and synthesis as trusted, the transformation logic to be verified is an algorithm on the model level. Still, even on the model level, manually verifying tens of transformations would be pretty challenging.

The question of proven correct refactoring systems is still open even in the research community. There have been papers and dissertations claiming that they present formally verified transformations, but either the proofs are partly informal, or the solution is too theoretical and is not applicable in practice. The difficulty of verification depends on the abstraction level on which the transformation is defined. Low-level implementations are hard to be verified, because they contain too many details, while high-level specifications do not define the transformation fine-grained enough.

We must focus on expressing a restricted set of transformations on a level of abstraction that is in between the corner cases mentioned above. We need to design a language that only allows for defining a restricted set of graph transformations, those that define behaviour-preserving modifications. The refactoring specification has to be low-level enough to be interpretable as an algorithm or function that maps program models to program models, while at the same time it has to be high-level enough to provide readability and verifiability with a reasonable amount of effort. Last but not least, it should not be too restricted, because it has to be able to express a fair amount of real-world, useful refactoring transformations.

To tackle the above challenge, I propose a novel formalism that allows for specifying refactoring transformations in an executable and automatically verifiable way. This means that the specifications in this formalism precisely define graph transformations that implement refactorings, while at the same time, they can be mapped to formulas that express their correctness property and can be automatically verified in most cases.

The proposed method is based on an enhanced variant of conditional and strategic term rewriting on semantic program graphs. Complex, extensive context-sensitive transformations are made automatically verifiable via semantics-driven refactoring schemes. Schemes are pre-defined and pre-verified refactoring skeletons, which are parametrised with conditional rewrite rules. Schemes ensure completeness and consistency of the multiple changes in the program by defining the proper traversal and control strategy for the rewriting.

Thesis 2. *I have developed scheme-based term rewriting with semantic strategies and semantic conditions executed on semantic program graphs, and showed that this novel approach provides an effective system for defining correct-by-construction refactoring. Erlang refactoring definitions specified in this method via decomposition to scheme instances can be automatically verified and interpreted in an Erlang static analysis and transformation tool. I have specified several complex refactorings with this method to demonstrate its applicability.*

I believe that with this work I establish the basics of refactoring oriented programming, which may have influence on how semantics-preserving transformations are specified formally. With the complex case studies specified in the language, the applicability of the method has already been demonstrated. Even though the presentation of the results is partly Erlang-specific, the methodology can be adapted to other languages.

Related publications. The first results in creating a strategy-based term rewriting system with semantic conditions for semantic graphs is illustrated in [2]; this work motivated the careful design of the refactoring language. The idea of introducing semantic predicates in rewritings and using refactoring schemes for extensive transformations is presented in [7]. A more complete overview of the refactoring programming paradigm and a detailed case study of refactoring specification is detailed in [8].

Extending languages via program transformations

> *"If someone claims to have the perfect programming language, he is either a fool or a salesman or both." (Bjarne Stroustrup)*

No programming language is perfect, but we opt for one or the other based on their unique features that make them the best choice for solving our particular problem. For instance, applications written in Erlang are famous for their robustness, fault-tolerance and scalability, which may be definite and understandable reasons to choose Erlang to implement a wide variety of systems. On the other hand, many programmers are dissatisfied with the syntax of the language, the lack of static checks and the limits of extendability.

In case of such well-established languages, language extensions are incorporated slowly, with a high attention on preserving stability; in exchange, usually they provide a fairly easy way to add ad-hoc extensions to the language via compiler plugins manipulating the syntax tree or the intermediate representation. Erlang, along with its compiler, was not designed with flexible extensibility in mind: although the compiler supports compile-time syntax tree transformations via the so-called 'parse transformations', the use of these is extremely limited.

If one wants to add features, or just complex syntactic sugars to Erlang, the only viable option is creating a new compiler or a pre-compiler for the extended language (not counting the possibility of forking the official one). Creating such a system requires implementing the entire compiler architecture, from syntactic analysis to semantic analysis, modelling, transformation and synthesis. This can be achieved with a standalone implementation or in a dedicated language workbench, but it takes a lot of effort for sure.

In one of our projects dealing with programming cyber-physical systems, we needed to embed a work-flow language in Erlang, but we could not fit the task operations and the remote execution into the language. Neither the syntax nor the semantics was proper. We found, however, that the language features we needed for the work-flow embedding could be realised as transformations to the original language by translation. Rather than putting our hand on the official compiler, we decided to employ the Erlang analysis and refactoring framework, which implements all those components that the compiler does.

We investigated if we can fully reuse the refactoring engine to implement language extensions via some kind of translation, and what pragmatics such a methodology has. The proposed method allows for adding extensions to the language without modifying a single line of the official compiler. Our approach in some sense is similar to hygienic macros, but on semantic program graphs, and implemented by a pre-compiler built upon a refactoring tool.

Thesis 3. *I have developed the first implementation of custom operators and code migration in Erlang by employing an Erlang refactoring system as a precompiler for the extended version of the language. Added language features are given a translation semantics to pure Erlang by means of refactoring-like program transformations.*

The solution demonstrates the applicability of a refactoring framework for easy prototyping of language extensions via employing translation semantics. Since the refactoring system implements very similar functionality and components that of a compiler, it is easily turned into a pre-compiler for an extended language. Building on this idea, we designed translation for two new language elements in Erlang: user-defined operator symbols and portable function closures. The former requires an expression re-parsing technique, while the latter is realised by defining a new closure semantics for Erlang anonymous functions. Worth mentioning that I gave the first implementation of code migration in Erlang with my method, which is may not be the fastest implementation, but it is lightweight and very effective in terms of dependencies it can handle.

Related publications. The general idea of exploiting the static analyisis and refactoring system for language feature prototyping is explained in [6], along with two case studies: user-defined operators and portable functions. The paper explains the transformations on different representation levels in the refactoring system, and discusses how the pre-compilation is realised to give translational semantics to language extensions. The follow-up paper [11] demonstrates how features realised with this method can be incorporated when implementing work-flow systems in Erlang.

Bibliography

[1] I. Bozó, V. Fördős, Z. Horváth, M. Tóth, D. Horpácsi, T. Kozsik, J. Kőszegi, A. Barwell, C. Brown, and K. Hammond. Discovering parallel pattern candidates in Erlang. In *Proceedings of the Thirteenth ACM SIGPLAN Workshop on Erlang*, pages 13–23, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3038-1. doi: http://doi.acm.org/10.1145/2633448.2633453.

[2] I. Bozó, V. Fördős, D. Horpácsi, Z. Horváth, T. Kozsik, J. Kőszegi, and M. Tóth. Refactorings to enable parallelization. In J. Hage and J. McCarthy, editors, *Trends in Functional Programming*, pages 104–121, Cham, 2015. Springer International Publishing. ISBN 978-3-319-14675-1.

[3] I. Bozó, D. Horpácsi, Z. Horváth, R. Kitlei, J. Kőszegi, M. Tejfel, and M. Tóth. RefactorErl – Source code analysis and refactoring in Erlang. In *Proceedings of the Symposium on Programming Languages and Software Tools*, SPLST'11, pages 138–148, Tallinn, Estonia, 2011.

[4] I. Bozó, M. Tóth, M. Tejfel, D. Horpácsi, R. Kitlei, J. Kőszegi, and Z. Horváth. Using impact analysis based knowledge for validating refactoring steps. *Studia Universitatis Babes-Bolyai Informatica Journal*, 56(3):57–64, 2011.

[5] D. Drienyovszky, D. Horpácsi, and S. Thompson. Quickchecking refactoring tools. In *Proceedings of the 9th ACM SIGPLAN workshop on Erlang*, Erlang '10, pages 75–80, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0253-1. doi: http://doi.acm.org/10.1145/1863509.1863521. URL http://doi.acm.org/10.1145/1863509.1863521.

[6] D. Horpácsi. Extending Erlang by Utilising RefactorErl. In *Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang*, Erlang '13, pages 63–72, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2385-7. doi: 10.1145/2505305.2505314. URL http://doi.acm.org/10.1145/2505305.2505314.

[7] D. Horpácsi, J. Kőszegi, and S. Thompson. Towards Trustworthy Refactoring in Erlang. In G. Hamilton, A. Lisitsa, and A. P. Nemytykh, editors, Proceedings of the Fourth International Workshop on *Verification and Program Transformation,* Eindhoven, The Netherlands, 2nd April 2016, volume 216 of *Electronic Proceedings in Theoretical Computer Science*, pages 83–103. Open Publishing Association, 2016. doi: 10.4204/EPTCS.216.5.

[8] D. Horpácsi, J. Kőszegi, and Z. Horváth. Trustworthy Refactoring via Decomposition and Schemes: A Complex Case Study. In A. Lisitsa, A. P. Nemytykh, and M. Proietti, editors, Proceedings Fifth International Workshop on *Verification and Program Transformation,* Uppsala, Sweden, 29th April 2017, volume 253 of *Electronic Proceedings in Theoretical Computer Science*, pages 92–108. Open Publishing Association, 2017. doi: 10.4204/EPTCS.253.8.

[9] D. Horpácsi and J. Kőszegi. Static analysis of function calls in erlang: Refining the static function call graph with dynamic call information by using data-flow analysis. *e-Informatica Software Engineering Journal*, 7:65–76, 2013. doi: 10.5277/e-Inf130107.

[10] R. Kitlei, L. Lövei, M. Tóth, Z. Horváth, T. Kozsik, R. Király, I. Bozó, C. Hoch, and D. Horpácsi. Automated syntax manipulation in RefactorErl. In *Proceedings of 14th International Erlang/OTP User Conference*, 2008. URL http://www.erlang.se/euc/08/.

[11] T. Kozsik, A. Lőrincz, D. Juhász, L. Domoszlai, D. Horpácsi, M. Tóth, and Z. Horváth. Workflow Description in Cyber-Physical Systems. *STUD UNIV BABES-BOLYAI SER INFO*, LVIII(2):20–30, 2013. ISSN 2065-9601.

[12] L. Lövei, C. Hoch, H. Köllö, T. Nagy, A. Nagyné Víg, D. Horpácsi, R. Kitlei, and R. Király. Refactoring module structure. In *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG*, ERLANG '08, pages 83–89, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-065-4. doi: http://doi.acm.org/10.1145/1411273.1411285. URL http://doi.acm.org/10.1145/1411273.1411285.

[13] G. Oláh, D. Horpácsi, T. Kozsik, and M. Tóth. Type inference in Core Erlang to support test data generation. *STUD UNIV BABES-BOLYAI SER INFO*, LIX(1):201–215, 2014. ISSN 2065-9601.

[14] M. Tejfel, M. Tóth, I. Bozó, D. Horpácsi, and Z. Horváth. Improving quality of software analyser and transformer tools using specification based testing. *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae*, 37:355–368, 2012.