Software Verification Technology and Tools

MAP-i - 2017/18 course edition

Summary

This document describes a proposal of a PhD level course intended as a Curricular Unit in Technology for the MAP-i Doctoral Programme in Informatics. The proposal is offered jointly by the Informatics Department of the University of Minho (DI-UM), the Electronics, Telecommunications, and Informatics Department of the University of Aveiro (DETI-UA), and the Computer Science Department of the University of Porto (DCC-FCUP).

Proponents: Jorge Sousa Pinto (DI-UM), Miguel Oliveira e Silva (DETI-UA), Pedro Vasconcelos (DCC-FCUP), Sandra Alves (DCC-FCUP).

Context and Motivation

The verification of software is increasingly important in software engineering, particularly in the context of critical applications, which is regulated by norms that advise or enforce the use of formal or semi-formal validation techniques. The importance of software verification in the safety-critical domain cannot be overstated. It suffices to cite the recommendations contained in industry norms such as ISO/IEC-15408 (known as Common Criteria, for security-sensitive applications), CENELEC EN 50128 (in the railway domain), or DO-178C (in the aerospace domain), which recommend or even enforce the adoption of such principles in the development of critical applications.

The importance of using formal verification techniques is not however restricted to critical systems, since even in non-critical scenarios the quality of software is an increasingly important feature for companies. Design by Contract^{TM}, a practical application of formal verification to general software construction, provides a powerful modular approach that may strongly affect some of the more relevant software quality factors, especially correctness and robustness. On the other hand, the growing complexity of modern software makes it difficult to produce error-free code, and there has been an enormous research effort in developing methods and tools that can be integrated into the development environments used by programmers, to help in this goal. Such tools provide rigorous guarantees of quality, are highly automated, and capable of scaling to cope with the enormous complexity of software systems.

The current industrial relevance of software verification is testified by the investments of companies like Microsoft and Facebook in this area. The former company, through its research branch, is responsible for the development of major tools, from satisfiability solvers to deductive verifiers and software model checkers (many of these tools can be tried online at http://riseforfun.com). A Facebook team on the other hand has published a paper entitled

Moving Fast with Software Verification, describing the team's "... experience in integrating a verification tool based on static analysis into the software development cycle at Facebook".

In the last years program verification has reached a stage of maturity, and a number of tools are now available allowing users to prove properties of programs written in real-world languages. The timeliness of this topic is also testified by the establishment of initiatives like The Verified Software Initiative[1], a cooperative international project directed at the scientific challenges of large-scale software verification, or by a significant number of software verification competitions, like VerifyThis[2], SV-COMP[3], and VSCOMP[4].

A first category of tools that will be studied is that of satisfiability solvers, in particular Satisfiability Modulo Theory (SMT) solvers. These tools have a wide range of applications (software and hardware verification, static program analysis, test case generation, scheduling, etc.) and have gained enormous popularity over the last years. They can either be used as standalone tools or as proof engines, as part of verification frameworks.

The vast majority of tools we will look at will target the verification of software. A key concept will be the notion of Behavioral Interface Specification Language (BISL), which is used to express the intended behavior a program, in terms of its safety properties, its functional properties, or even resource consumption. BISLs are useful for a number of reasons: they allow for a very accurate form of documentation, and can be used for the generation of test cases as well as in debugging.

In this course BISLs will be used for the purpose of software verification. The idea is that the behavior of a program is specified by writing annotations (using a given BISL) directly as comments in the code. Roughly speaking, an annotation is a code-aware version of the requirements. A typical verification tool will read an annotated program and check that it conforms to the behavior specified by the annotations. As an example the annotations accompanying a program function or method may indicate the conditions that should be met before it is executed, as well as describe the logical state of the program after its execution. Such function/method annotations, consisting of preconditions, postconditions, and frame conditions, are usually called the function's contract. The rationalisation of the code annotation methodology, coupled with its integration within the software engineering discipline, gave rise to a software development paradigm based on this notion of contract, as pioneered in the Eiffel programming language, which implements the notion of runtime or dynamic verification of code with respect to its contracts.

This paradigm has nowadays become very popular, in fact almost every widespread programming language benefits from a contracts layer. Let us cite for instance the programming language SPEC#, which can be seen as a superset of C#. Like Eiffel, both languages natively support the paradigm, but they additionally support the static verification of contracts. In the context of the Java programming language, different annotation systems exist that are based on the JML annotation language, such as Esc/Java, KeY and Krakatoa. C is also a popular choice in the safety critical industry, and the contract-based approach fits well the need for the static assurance of safety properties. A complete analysis and validation platform for C that provides a contracts layer is the Frama-C toolset based on the ACSL annotation language, which in turn was inspired by JML.

[1] https://sites.google.com/site/verifiedsoftwareinitiative/
[2] http://www.verifythis.org/
[3] http://sv-comp.sosy-lab.org/
[4] https://sites.google.com/a/vscomp.org/main/

Deductive program verification tools are in general powered by a Verification Conditions Generator (VCGen), a tool that produces verification conditions that are then passed to a satisfiability (SAT or SMT) solver to be discharged. A current trend is to use a generic VCGen for a small intermediate programming language. Instead of developing specific VCGens for each target programming language, a tool can be obtained by translating programs into the intermediate language, and encoding properties of the target language in the logic of the generic VCGen. The two foremost generic VCGens are Why3 (developed at LRI, Paris) and Boogie (a Microsoft research tool that is used in the development of numerous other tools). In addition to being used as a generic VCGen, Why3 can also be seen as a tool for the verification of algorithms independently of any given development language.

A second approach for software verification is to exploit recent developments in strongly typed functional programming languages to ensure the intended behavior of programs. A complementary approach is the use of type-based approaches for automating dynamic testing, in particular the use of property-based testing. These "lightweight formal methods" approaches have had industrial applications in recent years, in areas such as telecommunications [6], finance [27, 26], web-applications [32], embedded systems [19], amongst others.

In this context we propose to present the fundamental principles of typed functional programming languages, as well as show implementations using open-source software, such as the state-of-the-art GHC compiler for Haskell and the QuickCheck library for random property based testing for Haskell programs.

Related Courses

The course is at a similar level and covers overlapping material with the following advanced modules taught at leading academic institutions, as well as advanced summer schools, namely:

- *"Bug Catching: Automated Program Verification and Testing"* at Carnegie Mellon University, by Edmund Clarke, Sagar Chaki, Arie Gurfinkel.
- *"Formal Methods in Software Development"* at Michigan State University, by Laura Dillon.
- *"Proof of Programs"* at Université Paris Diderot, by Claude Marché, Arthur Charguéraud.
- *"Software Verification"* at ETH Zurich, by Bertrand Meyer, Carlo Furia, Sebastian Nanz.
- *"Software Verification"* at Oxford University, by Vijay D'Silva, Daniel Kroening.
- *"Advanced Functional Programming"* at Chalmers University of Technology, by Alejandro Russo.

Objectives

The objective of this course is to expose students to a range of technologies and tools for source code verification. The course will equip young researchers, in the areas of computer science or software engineering, with a basic verification toolbox that can be used in their projects and throughout their research careers, whenever a piece of software requires high assurance guarantees. A few cutting-edge research topics in relevant areas will also be discussed.

Learning Outcomes

Upon successful completion of this course, students should be able to:

- Use SMT solvers for logical modeling and solving.
- Apply Design by Contract in practical software development, using different real-world programming languages.
- Use the Why3 tool and its language for specifying and verifying algorithms, and as a background tool for constructing program verifiers.
- Verify safety and functional properties of C programs using the Frama-C tool.
- Use algebraic data types in a functional language to express and enforce desired invariants of data.
- Express correctness properties of functional algorithms using QuickCheck. Define generators and shrinking strategies for custom data types. Use the QuickCheck functionalities to collect statistic information on test distribution.

Prerequisites

A basic understanding of logic and programming language concepts will be assumed.

Course Contents

The course will focus on the techniques and tools used for (semi-)automated software verification. It is organized as follows:

Satisfiability Solving Tools

Satisfiability solvers have been applied with success in several domains of computer science and are the subject of very active research. SMT solvers have a wide range of applications and have gained enormous popularity over the last years. They can be either used as standalone tools or as proof engines, as part of verification frameworks.

We will begin with a brief review of propositional and predicate logic. We will then introduce some basic concepts of computational logic and present an overview of modern SAT and SMT solving technologies. Finally, we will focus on SMT solvers, presenting background logics and theories, pragmatic features, and encoding problems.

Design by Contract and Runtime Verification

Design by Contract™ (DbC) is a practical software methodology focusing on the construction of correct and robust software. It was proposed and developed in 1986 within the Eiffel programming language, and is specially well adapted to modular object-oriented programming mechanisms. Several languages, libraries and tools exist which support it in different programming languages, ranging from the simplest (but utterly incomplete) support of an assert instruction, to the full support spectrum provided by Eiffel. It can be used to specify, test, and document software modules, and it provides a simple, yet quite powerful, disciplined exception mechanism. Although sometimes

static verification tools are applicable, in practice its use is frequently based on runtime verification of assertions. Dynamic contract verification – though not as complete as static verification – can always be applied in general software development, and provides a strong engineering process aiming at software reliability. Also, though programmers have much to gain from a strong mathematical and computer science formal education, programmers without such skills can still use DbC in their software development process with clear benefits in the resulting program quality.

In this course we will give both a theoretical and a practical presentation of DbC, particularly in the context of object-oriented programming. The topics covered are: DbC as a general programming methodology; method and class contracts; contracts, inheritance and subtyping; disciplined exception mechanism; fault-tolerance; contracts and concurrency. Throughout the whole lectures on DbC, a practical engineering approach will be privileged by exemplifying its use in different general purpose programming languages such as Eiffel, Java, Python and C.

Software Verification with Why3

Why3 is a popular software verification platform. It provides a rich language for specification and programming, called WhyML, and relies on external theorem provers, both automated and interactive, to discharge verification conditions (it can be easily extended with support for new theorem provers). Why3 comes with a standard library of logical theories (integer and real arithmetic, Boolean operations, sets and maps, etc.) and basic programming data structures (arrays, queues, hash tables, etc.). A user can write WhyML programs directly and get correct-by-construction OCaml programs through an automated extraction mechanism.

Why3 can also be used as a software library, through an OCaml API, which allows WhyML to be employed as an intermediate language for the verification of programs written in other programming languages. It has been used in particular as background tool of verifiers for the C, Java, and SPARK/Ada programming languages. In fact, in this course we will study two different tools that use the Why3 verification platform in different ways: the Frama-C suite of tools, dedicated to source-code analysis of C software; and the EasyCrypt framework for verifying the security of cryptographic constructions in the computational model.

Software Verification with Frama-C

In recent years it has become viable to use formal verification techniques in practice for real-life programs. Although this is particularly important for code running in safety-critical contexts, some experience with a static verifier may prove to be very useful for any software engineer: not only does it help in achieving higher assurance degrees in verified applications, but it may also help in the algorithmic understanding of the code, and contribute to an improved development method overall.

Frama-C is an extensible and collaborative platform, based on a plug-in architecture, dedicated to the analysis of the source code of software written in C. The WP plug-in is based on the use of weakest precondition computations in order to generate verification conditions, and interaction with a background SMT solver to discharge these conditions. It allows one to prove that C functions satisfy their specifications, expressed as ANSI-C Specification Language (ACSL) contracts. These proofs are modular: the specifications

of the called functions are used to establish proofs without looking at their code, using instead the information provided by the contracts.

We will study the ANSI-C Specification Language and use it to write contracts specifying the behavior of C programs. We will then see how the Frama-C tool, and in particular its WP plugin, can be used to statically check the safety and correctness of a given C program.

Verification of Functional Programs

Advances in functional programming languages, such as Haskell, OCaml, F# and Scala, increasingly allow correctness properties to be captured using advanced type systems. This allows correctness of programs to be guaranteed by construction instead of relying on a separate verification mechanism.

We will present the foundations for typed functional programming: starting with the lambda-calculus, the simple types system, the Hindley-Milner type system and "ad-hoc" polymorphism using type classes.

We will move on to explore extensions such as phantom types, GADTs and higher-ranked types, in a real functional language implementation (the GHC Haskell compiler). In this setting, types can also delimit computational effects (e.g. IO, state, exceptions) using classes such as functors, applicatives and monads.

To complement the type-based static approach, we will also introduce run-time testing based on properties using the QuickCheck library. This consists of defining suitable properties and also data generators for tests. We will also present the notion of automatic shrinking of counter-examples. Finally we will discuss the use of methods for collecting statistic distribution of test cases, as a way to gain confidence on the quality of testing.

Syllabus

1. Satisfiability Solving Tools
   (a) Propositional and predicate logic (review)
   (b) Basic concepts of computational logic
   (c) Overview of modern SAT and SMT solving technologies
   (d) SMT solvers: logics and theories, pragmatic features, and encoding problems
2. Design by Contract and Runtime Verification
   (a) DbC as a general programming methodology
   (b) Method and class contracts
   (c) Contracts, inheritance and subtyping
   (d) Disciplined exception mechanism
   (e) Fault-tolerance contracts and concurrency
3. Software Verification with Why3
   (a) Basic aspects of deductive verification with Why3: ghost variables, functions, maps, vectors, data structures, lemma functions, and exceptions

(b) Aliasing and programs with pointers

(c) The craft of deductive verification

4. Software Verification with Frama-C

   (a) Specifying the behavior of C programs using the ANSI-C Specification Language (ACSL)

   (b) Introduction to the Frama-C program analysis tool

   (c) Statically checking the safety and correctness of C programs using the WP plugin

5. Verification of Functional Programs

   (a) Syntax and operational semantics of a minimal functional language based on the lambda-calculus

   (b) Type systems for functional programming languages

   (c) Type classes for computational effects

   (d) Property-based testing using QuickCheck

Teaching Methods and Student Assessment

Tutorial module, supported with demos and experimental lab work.

Assessment based on written or practical assignments proposed at the end of selected course units. This may include a talk given on a suggested paper, or a demo of some related tool.

Textbooks and Reading Material

On Satisfiability solving tools: [7, 14]
On Design by Contract and Runtime Verification: [25, 9]
On Verification of Functional Programs [28, 11, 12]
On Software Verification with Why3: [20, 8]
On Software Verification with Frama-C: [2, 10]

Lecturing Team

The proponent team consists of members of the Informatics Department of the University of Minho, the Electronics, Telecommunications, and Informatics Department of the University of Aveiro, and the Computer Science Department of the University of Porto.

The team of instructors is actively involved in research in the field software verification. A short biography of the instructors is provided below:

Jorge Sousa Pinto is a senior member of the Association for Computing Machinery, Associate Professor at the Department of Informatics of the University of Minho and a researcher at HASLab/INESC TEC. He obtained his habilitation degree from the University of Minho in 2015, and his degree of Docteur de L'Ecole Polytechnique (Paris) in 2001. In the past he has worked on linear logic and functional programming; more recently his work focused on deductive program verification and model checking of software. He has published around 45 papers in international conferences and journals, and is one of the authors of the textbook "Rigorous Software Development: an Introduction to Program Verification". He is a member of the Scientific Committee of the MAP-i programme. In the area of the present course proposal, he was from 2012 to 2015 the principal investigator of the FCT-funded AVIACC project, on the verification of concurrent software. *Selected relevant publications:* [24, 23, 2, 21, 13, 1, 18]

Miguel Oliveira e Silva is an Assistant Professor at University of Aveiro Electronics Telecommunications and Informatics Department (DETI). He received his BSc and Master in Electronics and Telecommunications Engineering in 1990 and 1994, and his PhD in Informatics Engineering in 2007, all from the University of Aveiro. Miguel's MSc and PhD thesis were both strongly related to Design by Contract^{TM} (DbC). The former on a signal processing library and programming language, and the latter on object-oriented concurrent programming language mechanisms. He was the primary responsible for introducing DbC ten years ago in DETI's first year programming courses at University of Aveiro, and since 2009 that he has been offering a last year course on DbC related Object-Oriented Concurrent Programming. He also lectured a MSc last year optional course on DbC in 2010. Currently, his main research activities are in the area of concurrent object-oriented languages, object-oriented and design by contract^{TM} programming, dynamic dispatch language mechanisms, and Multimodal Human-Machine Interaction. *Selected relevant publications:* [17, 16, 15]

Pedro Vasconcelos is an assistant professor at the Department of Computer Science of the University of Porto, and a member of the Laboratório de Inteligência Artificial e Ciência de Computadores (LIACC) of the University of Porto. He obtained his PhD degree in 2008, from the University of St Andrews, UK. His research focuses on type systems for predicting resource usage of functional programs. More relevant to the proposed course is his work on type- and effect-systems for stack and heap usage in the Hume research language, and the use of type-based amortisation for cost analysis of lazy functional languages. *Selected relevant publications:* [22, 33, 29, 31]

Sandra Alves is an assistant professor at the Department of Computer Science of the University of Porto, and a member of the Center for Research in Advanced Computing Systems (CRACS) of INESC TEC. She received the PhD degree in computer science in 2007 from the University of Porto. She has worked for several years in the area of lambda-calculus and type theory, with a focus on the impact of linearity in the definition of efficient models for programming languages. Of particular relevance to the present course proposal is her work on linear type systems, more specifically, the impact of linear constraints in the computational power of languages based on the lambda calculus. *Selected relevant publications:* [5, 4, 30, 3]

References

[1] José Bacelar Almeida, Manuel Barbosa, Jean-Christophe Filliâtre, Jorge Sousa Pinto, and Bárbara Vieira. CAOVerif: An open-source deductive verification platform for cryptographic software implementations. *Sci. Comput. Program.*, 91:216–233, 2014.

[2] José Bacelar Almeida, Maria João Frade, Jorge Sousa Pinto, and Simão Melo de Sousa. *Rigorous Software Development - An Introduction to Program Verification*. Undergraduate Topics in Computer Science. Springer, 2011.

[3] Sandra Alves, Maribel Fernández, Mário Florido, and Ian Mackie. Iterator Types. In H Seidl, editor, *Foundations of Software Science and Computational Structures, Proceedings*, volume 4423 of *Lecture Notes in Computer Science*, pages 17–31. Springer-Verlag Berlin, 2007.

[4] Sandra Alves, Maribel Fernández, Mário Florido, and Ian Mackie. Godel's System T Revisited. *Theoretical Computer Science*, 411(11-13):1484–1500, 2010.

[5] Sandra Alves, Maribel Fernández, Mário Florido, and Ian Mackie. Linearity and Recursion in a Typed Lambda-Calculus. In Peter Schneider-kamp and Michael Hanus, editors, *Ppdp 11 - Proceedings Of The 2011 Symposium On Principles And Practices Of Declarative Programming*, PPDP, pages 173–182. Assoc Computing Machinery, 2011.

[6] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. Testing telecoms software with quviq quickcheck. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, ERLANG '06, pages 2–10, New York, NY, USA, 2006. ACM.

[7] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[8] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Let's verify this with why3. *STTT*, 17(6):709–727, 2015.

[9] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *STTT*, 7(3):212–232, 2005.

[10] Jochen Burghardt, Jens Gerlach, and Timon Lapawczyk. ACSL by example: Towards a verified C standard library. Technical report, Fraunhofer FOKUS, 2015.

[11] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM.

[12] Koen Claessen and John Hughes. Testing monadic code with quickcheck. *SIGPLAN Not.*, 37(12):47–59, December 2002.

[13] Daniela Carneiro da Cruz, Maria João Frade, and Jorge Sousa Pinto. Verification conditions for single-assignment programs. In Sascha Ossowski and Paola Lecca, editors, *Proceedings of the ACM Symposium on Applied Computing, SAC 2012, Riva, Trento, Italy, March 26-30, 2012*, pages 1264–1270. ACM, 2012.

[14] Leonardo Mendonça de Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, 2011.

[15] Miguel Oliveira e Silva. Concurrent object-oriented programming: The MP-Eiffel approach. *Journal of Object Technology*, 3(4):97–124, 2004.

[16] Miguel Oliveira e Silva. Automatic realizations of statically safe intra-object synchronization schemes in MP-Eiffel. In *Symposium on Concurrency, Real-Time, and Distribution in Eiffel-Like Languages, York, United Kingdom*, pages 91–118, July 2006.

[17] Miguel Oliveira e Silva and Pedro G. Francisco. Contract-Java: Design by contract in Java with safe error handling. In Maria João Varanda Pereira, José Paulo Leal, and Alberto Simões, editors, *3rd Symposium on Languages, Applications and Technologies, SLATE 2014, June 19-20, 2014 - Bragança, Portugal*, volume 38 of *OASICS*, pages 111–126. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014.

[18] R. A. B. e Silva, L. A. Burgareli N. N. Arai, J. M. P. de Oliveira, and J. S. Pinto. Formal verification with Frama-C: a case study in the space software domain. *IEEE Transactions on Reliability*, To appear.

[19] Trevor Elliott, Lee Pike, Simon Winwood, Pat Hickey, James Bielman, Jamey Sharp, Eric Seidel, and John Launchbury. Guilt free ivory. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, Haskell '15, pages 189–200, New York, NY, USA, 2015. ACM.

[20] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013.

[21] Maria João Frade and Jorge Sousa Pinto. Verification conditions for source-level imperative programs. *Computer Science Review*, 5(3):252–277, 2011.

[22] Steffen Jost, Pedro Vasconcelos, Mário Florido, and Kevin Hammond. Type-based cost analysis for lazy functional languages. *Journal of Automated Reasoning*, pages 1–34, 2017.

[23] Cláudio Belo Lourenço, Maria João Frade, and Jorge Sousa Pinto. A bounded model checker for SPARK programs. In Franck Cassez and Jean-François Raskin, editors, *Automated Technology for Verification and Analysis - 12th International Symposium, ATVA 2014, Sydney, NSW, Australia, November 3-7, 2014, Proceedings*, volume 8837 of *Lecture Notes in Computer Science*, pages 24–30. Springer, 2014.

[24] Cláudio Belo Lourenço, Maria João Frade, and Jorge Sousa Pinto. Formalizing single-assignment program verification: An adaptation-complete approach. In Peter Thiemann, editor, *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9632 of *Lecture Notes in Computer Science*, pages 41–67. Springer, 2016.

[25] Bertrand Meyer. *Object-Oriented Software Construction, 2nd Edition.* Prentice-Hall, 1997.

[26] Yaron Minsky. Ocaml for the masses. *Commun. ACM*, 54(11):53–58, 2011.

[27] Simon Peyton Jones, Jean-Marc Eber, and Julian Seward. Composing contracts: An adventure in financial engineering (functional pearl). *SIGPLAN Not.*, 35(9):280–292, September 2000.

[28] Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, 1st edition, 2002.

[29] Vitor Rodrigues, Benny Akesson, Mário Florido, Simão Melo de Sousa, João Pedro Pedroso, and Pedro B. Vasconcelos. Certifying execution time in multicores. *Sci. Comput. Program.*, 111:505–534, 2015.

[30] Mário Florido Sandra Alves, Maribel Fernández and Ian Mackie. Linearity and Iterator Types for Gödel's System T. *Higher-Order and Symbolic Computation*, 23(1):1–27, 2010.

[31] Hugo R. Simões, Pedro B. Vasconcelos, Mário Florido, Steffen Jost, and Kevin Hammond. Automatic amortised analysis of dynamic memory allocation for lazy functional programs. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 165–176. ACM, 2012.

[32] Michael Snoyman. *Developing Web Applications with Haskell and Yesod.* O'Reilly Media, Inc., 2012.

[33] Pedro B. Vasconcelos, Steffen Jost, Mário Florido, and Kevin Hammond. Type-based allocation analysis for co-recursion in lazy functional languages. In *Proceedings of the 24th European Symposium on Programming, ESOP*, volume 9032 of *Lecture Notes in Computer Science*, pages 787–811. Springer, 2015.