Actionable Feedback during Software Development

Sebastian Erdweg
Delft University of Technology

My team works on programming tools that supply developers with actionable feedback during software development. Feedback is actionable if it is relevant to the programmer's task, if the programmer can rely on its correctness, and if it arrives in a timely manner. By providing actionable feedback, we protect developers against introducing performance bottlenecks, unsafe code, security vulnerabilities, or specification violations. Our feedback also influences development tools such as compiler optimizations and refactorings. We tackle this challenging research program in two focus areas: incremental computing and practical correctness proofs.

1. Incremental computing

My team develops building blocks for incremental algorithms that achieve high performance when reacting to a change in their input. Rather than repeating the entire computation over the changed input, an incremental algorithm only updates those parts of the previous result that are affected by the input change. This way, incremental algorithms provide asymptotical speedups in theory and we have observed multiple orders of magnitude speedups in practice.

Incremental algorithms are crucial for providing actionable feedback because the feedback needs to be updated after every code change the developer makes. Yet, existing algorithms, such as the Java type checker in Eclipse JDT, are one-off solutions that required years of engineering that cannot be reproduced. We develop building blocks for incremental computing and collect them in frameworks that execute regular algorithms incrementally.

In the **IncA project** [13–15], we study algorithms for incremental program analysis. The basic idea is to store the syntax tree of programs in a relational database and to run incremental Datalog queries over these relations. However, incremental solvers of Datalog are inherently limited in expressiveness. In particular, they lack support for lattices, which are ubiquitous in program analysis. We were the first to discover techniques for incrementally solving lattice-based Datalog queries and we have applied our techniques to achieve order-of-magnitudes speedups when analyzing C, Java, and Rust code.

In the **PIE project** (formerly pluto) [2, 8, 9], we develop incremental build systems. Incremental build systems are essential for fast, reproducible software builds and enable short feedback cycles when they capture dependencies precisely and selectively execute build tasks efficiently. A much-overlooked feature of build systems is the expressiveness of the scripting language, which directly influences the maintainability of build scripts. We develop new incremental build algorithms that allow build engineers to use a full-fledged programming language and where task dependencies can be discovered during building.

In the **CoCo project** [1, 10], we explore novel ways for co-contextual reasoning about code and how that can be used to achieve incrementality. Specifically, we are developing co-contextual type checkers for functional and object-oriented programming languages. A co-contextual type checker produces context requirements rather than reading context information as it traverses a syntax tree bottom-up. We have mostly focused on applying this technique to incremental type checking so far, yet applications to parallelization and streaming seem promising.

2. Practical correctness proofs

Correctness proofs ensure algorithmic results are correct. Conversely, incorrect feedback gives developers a false sense of security that is not actually warranted by their code. We develop theory and tools that simplify correctness proofs. Our long-term goal is to enable analysis developers to prove correctness in little time, without requiring extensive training. Provably correct analysis results will boost the confidence of programmers when reacting to analysis feedback.

In the **Sturdy project** [6, 7], we explore techniques for compositional correctness proofs. The key idea of compositional proofs is to decompose complex verification tasks into much simpler ones. Developers then only need to prove the simple tasks, from which overall algorithmic correctness follows by construction. Key to our approach is to capture the similarities between the specification and the implementation in a single shared program, parameterized over an arrow-based interface. We have instantiated our technique for program analyses and proved simple analyses sound with modest effort. To better understand and support practical scalability, we currently apply our framework to analyses of Java and JavaScript, as well as to code generators.

In the **Veritas project** [3–5], we explore techniques for automated verification of complex tasks. Specifically, we explore how existing off-the-shelve SMT solvers and first-order theorem provers can be applied to prove domain-specific verification problems. The key idea is to translate such problems into first-order logic in a way that existing provers can support. Due to the unpredictability of off-the-shelve solvers, this research is largely driven by empirical experiments that indicate how such translation can be successful.

In the **Soundx project** [11, 12], we develop techniques to guarantee the type safety of code generators. Code generators are hard to get right because they operate at the meta-level, where programs are data. This makes it is easy to generate code that does not type check, which in turn is hard to debug for users since the type errors refer to generated code. We develop automated techniques for ensuring that code generators can only produce code that is well-typed.

References

[1] S. Erdweg, O. Bračevac, E. Kuci, M. Krebs, and M. Mezini. A co-contextual formulation of type rules and its application to incremental type checking. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 880–897. ACM, 2015.

[2] S. Erdweg, M. Lichter, and M. Weiel. A sound and optimal incremental build system with dynamic dependencies. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 89–106. ACM, 2015.

[3] S. Grewe, S. Erdweg, P. Wittmann, and M. Mezini. Type systems for the masses: Deriving soundness proofs and efficient checkers. In *Proceedings of Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward)*, pages 137–150. ACM, 2015.

[4] S. Grewe, S. Erdweg, A. Pacak, and M. Mezini. An infrastructure for combining domain knowledge with automated theorem provers (system description). In *Proceedings of Conference on Principles and Practice of Declarative Programming (PPDP)*. ACM, 2018.

[5] S. Grewe, S. Erdweg, A. Pacak, M. Raulf, and M. Mezini. Exploration of language specifications by compilation to first-order logic (extended version). *Science of Computer Programming*, 155, 2018.

[6] S. Keidel and S. Erdweg. Toward abstract interpretation of program transformations. In *International Workshop on Meta-Programming Techniques and Reflection*, pages 1–5. ACM, 2017.

[7] S. Keidel, C. B. Poulsen, and S. Erdweg. Compositional soundness proofs of abstract interpreters. *Proceedings of the ACM on Programming Languages*, 2(ICFP), 2018.

[8] G. Konat, S. Erdweg, and E. Visser. Scalable incremental building with dynamic task dependencies. In *Proceedings of International Conference on Automated Software Engineering (ASE)*. ACM, 2018.

[9] G. Konat, M. J. Steindorfer, S. Erdweg, and E. Visser. PIE: A domain-specific language for interactive software development pipelines. *Art, Science, and Engineering of Programming*, 2(3), 2018.

[10] E. Kuci, S. Erdweg, O. Bračevac, A. Bejleri, and M. Mezini. A co-contextual type checker for Featherweight Java. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, 2017.

[11] F. Lorenzen and S. Erdweg. Modular and automated type-soundness verification for language extensions. In *Proceedings of International Conference on Functional Programming (ICFP)*, pages 331–342. ACM, 2013.

[12] F. Lorenzen and S. Erdweg. Sound type-dependent syntactic language extension. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pages 204–216. ACM, 2016.

[13] T. Szabó, S. Erdweg, and M. Völter. IncA: A DSL for the definition of incremental program analyses. In *Proceedings of International Conference on Automated Software Engineering (ASE)*. ACM, 2016.

[14] T. Szabó, M. Völter, and S. Erdweg. IncAL: A DSL for incremental program analysis with lattices. In *International Workshop on Incremental Computing (IC)*, 2017.

[15] T. Szabó, E. Kuci, M. Bijman, M. Mezini, and S. Erdweg. Incremental overload resolution in object-oriented programming languages. In *International Workshop on Formal Techniques for Java-like Programs*. ACM, 2018.