More Gradual Verification

Jonathan Aldrich (DBLP /GS), Éric Tanter (DBLP/GS), and Joshua Sunshine (DBLP /GS)

1 Introduction and Focus

Scientific advances from the past two decades such as separation logic have resulted in tools such as Infer that are having a profound impact on quality assurance at Facebook and other major companies. Despite promising advances such as incremental verification of codebase changes [4], many verification techniques remain all or nothing, making it difficult to justify investment in verification if the payoff does not come until an entire program is verified.

In recent Facebook-supported work, we have been developed *gradual verification* [1], an approach that borrows ideas from gradual typing in order to provide truly pay-as-you-go program verification. The key technical advance in gradual verification is providing explicit support for specifications that are partial, both in the sense of specifying only certain components within the program and in the sense of specifying only certain properties of those components. These partial specifications can be gradually extended to additional components of the program and to additional properties. Incremental specification effort is immediately rewarded with additional assurance—static assurance where specifications are complete enough to support it, and dynamic assurance otherwise. Errors are reported to the programmer only when the program is inconsistent with its specification—never because specifications are missing.

2 Prior Work

Gradual verification is unique in supporting specifications that are explicitly *partial* (or *imprecise*). For example, the precondition "? * $x < 10$" means "This function requires that $x < 10$ and it also has other, currently unspecified, requirements." This allows the verifier to give an error if $x < 10$ is not satisfied by the client; yet if the function body requires a stronger condition than $x < 10$, the verifier will avoid a spurious warning that other tools would give simply because the stronger condition has not yet been explicitly specified.

Our prior work applies the Abstracting Gradual Typing approach [3] to generate a gradual verification system in a principled way from a static verification system. We show that our approach enjoys the *gradual guarantees* [9], which formalizes the essence of graduality by stating that correct specifications that are added to a correct program never result in either static or dynamic errors. Our approach is also pay-as-you-go in terms of run-time costs, as dynamic checks are only added where static specifications are insufficient to verify an assertion.

In the past year of Facebook TAV-supported research, we have extended the core theory of gradual verification to support abstract predicates in the setting of implicit dynamic frames (IDF) [10], enabling us to modularly verify recursive heap data structures. We have also explored the application of the theory to lightweight static analysis by implementing a gradual null pointer analysis within Infer. Current static analysis approaches sometimes use ad-hoc techiques for providing actional error messages without bothering users with false positives; we are comparing the output of our tool to Infer's Eradicate analysis to see whether gradual verification theory can provide a more principled approach to this problem. These results will be submitted for publication in the near future.

3 Proposed Work and Techniques

Now that we have built solid theoretical foundations for gradual verification, we propose to evaluate its practical impact. We have already begun an implementation of a gradual verifier based on Hoare Logic, recursive predicates, and implicit dynamic frames. The first version of this verifier will target C_0, a version of C that is used in the 15-122 second-year computer science course at Carnegie Mellon University. C_0 is a good target because it already includes Hoare-style preconditions, postconditions, and loop invariants. These C_0 specifications are currently checked dynamically, but gradual verification presents an opportunity to smoothly integrate static checking as students are learning to program and to specify their programs. We plan to continue work on this verifier, bringing a working prototype to completion within the coming year.

At the same time, we will begin to explore the potential impact of gradual verification on education. A first step will be developing exercises and assignment extensions that could be used in integrating gradual verification into 15-122. We will then carry out pilot studies to determine whether students can successfully complete those exercises, and whether the exercises effectively expose them to verification concepts. Additionally, we will carry out usability studies with our prototype tool to make sure the tool design does not impose barriers to student learning, and we will refine the tool to mitigate any barriers we observe. This sequence of activities will prepare us to carry out a future larger-scale study to measure the impact of 15-122 on student learning, likely in the 2020-2021 school year.

We have been investigating gradual verification for the IDF resource logic, which raises a question: is there a general gradualization theory that would extend to other resource logics, including separation logic as well as ownership type systems? Resource logics are interesting because they capture ownership of the heap, which may be challenging to check dynamically. A general theory would enable us to predict what resource logics could be effectively gradualized, and eventually build resource logic gradualization into generic verification platforms such as Viper or Infer.

Along those lines, we plan to explore additional gradual analyses within Infer. We plan to focus on resource usage issues such as leaks, which are important industrially and may be amenable to the kind of resource logics (or related resource type systems) that we are using in our verifier. We believe we can uncover important principles for designing analyses that effectively find resource bugs without giving false positive warnings due to missing specifications.

4 Milestones and Outcomes

Six month milestones: Complete the back end of our gradual verification system. Report results from our gradual null pointer analysis and analyze how a gradual approach to reducing false positives compares to conventional approaches. Start the design of a gradual resource usage analysis. Prototype C_0 verification exercises. Begin to develop a theory of resource logic gradualization.

1-year milestones: Complete a first working gradual verifier for C_0. Implement the gradual resource usage analysis in Infer and evaluate it on open source code. Pilot C_0 verification exercises and address initial usability issues in the tool. Report initial results on a theory of resource logic gradualization.

References

[1] J. Bader, J. Aldrich, and É. Tanter. Gradual program verification. In I. Dillig and J. Palsberg, editors, *Proceedings of the 19th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2018)*, volume 10747 of *Lecture Notes in Computer Science*, pages 25–46, Los Angeles, CA, USA, Jan. 2018. Springer-Verlag.

[2] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6):26:1–26:66, Dec. 2011.

[3] R. Garcia, A. M. Clark, and É. Tanter. Abstracting gradual typing. In *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*, pages 429–442, St Petersburg, FL, USA, Jan. 2016. ACM Press.

[4] M. Harman and P. O'Hearn. From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In *Source Code Analysis and Manipulation*, 2018.

[5] N. R. Krishnaswami, J. Aldrich, L. Birkedal, K. Svendsen, and A. Buisse. Design patterns in separation logic. In *Types in Language Design and Implementation*, 2009.

[6] N. Lehmann and É. Tanter. Gradual refinement types. In *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2017)*, pages 775–788, Paris, France, Jan. 2017. ACM Press.

[7] F. Muehlboeck and R. Tate. Sound gradual typing is nominally alive and well, Oct. 2017.

[8] L. Nistor, J. Aldrich, S. Balzer, and H. Mehnert. Object propositions. In *Formal Methods*, 2014.

[9] J. G. Siek, M. M. Vitousek, M. Cimini, and J. T. Boyland. Refined criteria for gradual typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*, volume 32 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 274–293, Asilomar, California, USA, May 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[10] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames. *ACM Transactions on Programming Languages and Systems*, 34(1):2:1–2:58, Apr. 2012.

[11] N. Vazou, É. Tanter, and D. Van Horn. Gradual liquid type inference. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), Nov. 2018. To appear.