Incremental Verification, Gradually

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

1 Scientific Contribution

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. Incrementality is a key enabler of that impact: it allows the analysis tool or its users to select the portion of the code to be verified, making sure that analysis time—and specification time, where relevant—is spent most effectively. For example, we might want to verify just the changes to the codebase [4], a feature readily supported by Facebook's Infer tool. Alternatively, we might want to verify properties of one component without specifying and verifying the related properties of surrounding components. Later on in the development process, we can incrementally extend verification to other components, one at a time.

This second form of incrementality—adding one component at a time to the body of verified code—remains an open challenge for many forms of static verification. The precondition of a function f might be specified or inferred, but if the caller's precondition is not specified and cannot be inferred successfully, then the tool may not be able to determine whether f's precondition has been established. A developer can address that problem by adding a specification to the caller, but this takes valuable time that might be best invested elsewhere—and besides, the caller may itself have a caller, etc., forcing specification of the whole program instead of incrementally.

We have recently proposed *gradual verification* [1] to address this problem by smoothly combining static and dynamic verification. When specifications are provided or can be inferred, a component can be verified statically. Otherwise, dynamic checks are inserted at the boundary between components that are specified and verified on the one hand, and components that are not on the other. Gradual verification is inspired by gradual typing—a technology that in various forms is having major impact in industry, including via Facebook's Hack—and we adapt several key concepts and properties of gradual typing research to the verification setting:

1. In addition to providing specifications for only some components, it is possible to give *partial* (or *imprecise*) specifications anywhere in the code. For example, the precondition "$? \; * \; x < 10$" means "This function requires that $x < 10$ and it also has other, currently unspecified, requirements." Even if "$x < 10$" alone is not enough to verify a function, if there is some condition that can be locally plugged in for "?" that is sufficient for the function to be verified, the static analyzer will not give an error, but will insert appropriate run-time checks. Dually, if $x < 10$ is definitely violated by a client, the analyzer can report a static error.
2. Gradual verification provides a *smooth* continuum between static and dynamic verification, captured by a property known as the *gradual guarantees* [7]: relaxing the precision of a legal specification is still a legal specification, and will not result in additional run-time or compile-time errors. This ensures that programmers can move smoothly from no specifications to full specifications, including all points in between, and will get static/dynamic errors only when the specifications and code they write is somehow inconsistent, not merely because some specifications are missing or incomplete.
3. Run-time verification checks only the part of a condition that cannot be verified statically, potentially saving substantial run-time overhead compared to always checking the full condition dynamically. Our approach builds on the Abstracting Gradual Typing approach [3], and is also reminiscent of abduction as used in Infer [2].

We believe these properties are key to achieving incremental specification and verification in practice. Property #1 allows programmers to specify only the properties they care about and get useful feedback from the tool even with imprecise specifications; more properties can be added incrementally. Property #2 ensures that incrementally adding (or removing) specifications will not cause warnings that are spurious in the sense that the added specifications are correct but incomplete; the requirement that developers add specifications until a complete proof is formed is one of the most problematic aspects of many specification-based verification tools. Gradual verification only statically enforces the *plausibility* of programs to be well behaved. Property #3 is important because run-time checks can be quite computationally expensive, so any approach to minimize their cost will help us get more dynamic verification out of a fixed computational cost budget.

2 Proposed Work

Our initial paper on gradual verification [1] develops the notions above for a simple Hoare logic; important research needs to be done to fulfill the promise of the idea. We propose to extend our theory, including the gradual guarantee, to more realistic logics, starting with implicit dynamic frames [8]—an area where we have done some preliminary work already—and then moving to a separation logic with recursive predicates, focusing in particular on the logic used internally within Infer to see if it can be integrated into the gradual verification research agenda. On a technical level, we propose to handle recursive predicates isorecursively, so that when (un)folding a predicate we reason about (and potentially track at run time) what state is owned by that (un)folding's part of the predicate, and likewise handle any imprecision in that unfolding of the formula. These advanced logics are challenging as the dynamic analysis must track the portions of the heap owned by different parts of a formula, including subformulas that are framed away during a function call. Note that we are mainly interested in checking first-order assertions, such as those found in Hoare logic pre/post-conditions; recent results in gradual typing suggest that reasonable performance is achievable in this situation [6].

We will investigate whether gradual verification helps developers write good specifications and find bugs faster than state-of-the-art, non-gradual tools. We hypothesize that the ability to provide partial specifications backed by runtime checks will be useful in a variety of quality-related tasks. Gradual verification should also be helpful for other tools, e.g. automated testing and fixing, due to the identification of the static-dynamic checking boundary conditions.

3 Routes to Deployment

Although there is prior work hybridizing static and dynamic verification, our gradual verification approach is the first to offer the key properties listed in Section 1. However, the limitations of that initial work prevent it from being immediately deployed in industrial-strength verification tools. Our goal in this proposal is to address these limitations, first in a simple theoretical and prototype implementation setting in the coming year; within a year from now, the technology will be advanced enough that we will be able to experimentally add it Infer. We will focus on different analyses, starting with non-null checking.

While an obvious application of gradual verification is to add incrementality to tools that rely on programmer-written specifications, such as Dafny, we believe that applying gradual verification to automated tools such as Infer could has even greater dividends, as these tools are more widely used in industry. For example, Infer may be able to come up with a precondition for a function, but may not be able to either prove or disprove that a given caller establishes the precondition. In such situations, today's tools must choose between giving the programmer a possibly-spurious warning or ignoring a potential error. Our gradual verification theory would help Infer decide when to give warnings—e.g. it includes an explicit representation of where a specified or inferred assertion is imprecise and can use that information to detect definite inconsistencies that may not be obvious without this information. It can also allow Infer to insert run-time assertions that can be checked in any test cases that exercise the code, making those test cases more likely to find defects. At the same time, our approach can help minimize the computational cost of those run-time checks. Gradual verification ideas could be used to enhance Infer's modeling capability, so that partial models can be given (with the partiality explicit, rather than implicit, again aiding in checking). Finally, gradual verification can provide a way to help Infer users put more explicit specifications in code when desired, making the tool more powerful while still retaining the incrementality properties that Infer already provides. Co-PI Tanter has a forthcoming paper on gradual liquid type inference [9], which shows the benefits of the gradual approach in a setting with inference and logical information, hence closely related to that of Infer.

4 Research Team

PI Aldrich has expertise in applied verification for resource logics, as well as the design of usable type systems and analysis tools. **Co-PI Tanter** brings expertise in gradual typing, particularly for advanced typing disciplines such as gradual refinement types [5] and gradual liquid inference [9]. They worked with **Johannes Bader** to develop the first iteration of gradual verification, which formed Bader's master's thesis. Bader has been admitted to CMU's Ph.D. program, where he will be advised by Aldrich and Tanter, but he deferred for a year, during which he is working at Facebook. **Co-PI Sunshine** is an expert on evaluating and improving the usability of programming languages, type systems, and related tools. We hope that close coordination with Johannes as well as Facebook researchers in related areas, such as Peter O'Hearn, will facilitate collaboration with Facebook and eventual transition of the research ideas into practical use. We will involve at least one more PhD student in the project (**Jenna Wise**, CMU).

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. 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.

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

[7] 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.

[8] 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.

[9] 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.