Incremental Verification, Gradually

Jonathan Aldrich, Éric Tanter, and Joshua Sunshine

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 inpact: 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 that is 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, we can incrementally extend this 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 the developer may prefer to invest elsewhere—and besides, the caller may itself have a caller, and specifying the whole program may be infeasible.

We have proposed *gradual verification* [1] to address this problem by 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 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* 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 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.
2. The *gradual guarantee* ensures that any relaxation of a complete specification is still a legal specification, and will not result in a run-time or compile-time warning. 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; 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. Property #3 is important because run-time checks can be quite computationally expensive, so anything we can do to minimize their cost will help us get more dynamic verification out of a fixed computational cost budget.

2 Proposed Work

Although we have published an initial paper on gradual verification, important research needs to be done

to fulfill the promise of the idea. Our prior work developed gradual verification for a simple Hoare logic. We propose to extend our theory (including the gradual guarantee) and our implementation to a separation logic with recursive predicates, similar to today's advanced static verifiers. We propose to handle 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 (i.e. ?'s) in that unfolding of the formula. While we previously worked out an initial theory for optimizing run-time verification checks, we propose to extend our run-time checking approach to handle recursive predicates, then measure overhead in a real implementation and work on ways to further optimize it. There has been a lot of concern about the run-time overhead of gradual typing, with some declaring that the overhead is so high that sound gradual typing is "dead"" [5]. We are mainly interested in checking first-order assertions, as typically found in Hoare logic pre- and post-conditions, so we believe gradual verification can be implemented without the higher-order wrappers that cause performance problems in the gradual typing setting.

3 Routes to Deployment

Although there is prior work hybridizing static and dynamic verification, we published the first exploration of a gradual verification system with the 3 key properties above only this year, and the limitations in that work prevent it from being immediately deployed in industrial-strength verification tools. Our goal is to work on those limitations in a simple theoretical and prototype implementation setting in the coming year, supported by a Facebook gift, so that by a year from now, the technology will be advanced enough that it could be experimentally added to tools such as infer in the following year.

While the obvious application of gradual verification is to add incrementality to tools such as Dafny that are based on programmer-written specifications, we believe that applying gradual verification to more automated tools such as Infer could have 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 partial 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 to 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 could provide a way for Infer users to put more explicit specifications in code when desired, making the tool more powerful while still retaining the incrementality properties that Infer already provides.

Acknowledgments and Potential Student Researchers. Many ideas in this proposal came out of Johannes Bader's master's thesis, advised by Aldrich and Tanter; additional ideas came from recent research with Jenna Wise, advised by Aldrich and Sunshine. Jenna is a CMU student who will work on the project should it be funded. Johannes Bader was admitted to CMU but deferred for a year, in which he is working at Facebook; afterward, he may work on the project as well. 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.

References

[1] J. Bader, J. Aldrich, and Éric Tanter. Gradual program verification. In *VMCAI*, 2018.

[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 E. Tanter. Abstracting gradual typing. In *Principles of Programming Languages*, 2016.

[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] A. Takikawa, D. Feltey, B. Greenman, M. S. New, J. Vitek, and M. Felleisen. Is sound gradual typing dead? In *Principles of Programming Languages*, 2016.