Evaluating Soundness of a Gradual Verifier with Property Based Testing

JAN-PAUL RAMOS-DÁVILA, Cornell University, USA
jvr34@cornell.edu

Gradual verification supports partial specifications by soundly applying static checking where possible and dynamic checking when necessary. This approach supports incrementality and provides a formal guarantee of verifiability. The first gradual verifier, Gradual C_0, supports programs that manipulate recursive, mutable data structures on the heap and minimizes dynamic checks with statically available information. The design of Gradual C_0 has been formally proven sound; however, this guarantee does not hold for its implementation.

In this paper, we introduce a lightweight approach to testing soundness of Gradual C_0's implementation. This approach uses Property Based Testing to empirically evaluate soundness by establishing a truthiness property of equivalence. Our approach verifies a test suite of incorrectly written programs and specifications with both Gradual C_0 and a fully dynamic verifier for C_0, and then asserts an equivalence between the results of the two verifiers using the dynamic verifier as ground truth. Any inconsistency between the results, indicates a problem in Gradual C0's implementation. We also show in this paper, as a proof of concept, that this lightweight approach to testing Gradual C_0's soundness caught a number of significant implementation bugs from Gradual C_0's issue tracker in GitHub. A number of these bugs were only previously caught by human inspections of internal output of the tool. An automated generator for the test suite is our next research step to increase the rigor of our evaluation and catch new bugs never found before.

CCS Concepts: • **Theory of computation** → **Logic and verification**; **Separation logic**;

Additional Key Words and Phrases: gradual verification, program correctness, property based testing

1 INTRODUCTION

Static verification is used to ensure the correctness of programs. Unfortunately, this approach to verification requires users to specify programs completely and in great detail to support inductive proofs of correctness. Further, static verification tools cannot provide verification feedback on any partial specifications written on the way to complete static specification. Similarly, dynamic verification experiences run-time costs that limit its practicality. [Bader et al. 2018] introduces the idea of *gradual verification*, which soundly combines both static and dynamic verification techniques to support the incremental specification and verification of programs. Inspired by *gradual typing* [Garcia et al. 2016; Siek and Taha 2007, 2006], with gradual verification the programmer gains control over the trade-offs between static and dynamic checking by way of partial specifications, allowing the behavior of unspecified components to be verified at run-time. [Wise et al. 2020] introduces the theory of gradual verification for programs that manipulate recursive heap data structures (like trees, graphs, and lists). [Wise et al. 2020] proved such a gradual verification system sound, and show its adherence to the *gradual guarantee* property—which states that relaxing specifications does not introduce new static or dynamic verification errors. Finally, [DiVincenzo et al. 2022] introduces the design and implementation of *Gradual* C_0, the first gradual verifier for recursive heap data structures inspired by [Wise et al. 2020]'s foundational theory. They built Gradual C_0 to minimize dynamic checking with statically available information and showed in a performance study that Gradual C_0 reduces run-time overhead by 50-90% on average compared to dynamic verification.

Gradual C_0's design has been proven sound and will catch all violations of a specification. However, there are no techniques available to ensure the implementation of Gradual C_0 is correct. While empirically evaluating Gradual C_0's performance in [DiVincenzo et al. 2022], Gradual C_0 was used to verify thousands of partial specifications that are correct and approximate the gradual

guarantee. A number of bugs were caught and fixed by hand, in which Gradual C_0's design was implemented incorrectly. To complement the aforementioned evaluation that only looks at correct specifications and programs, we introduce a *property based testing* (PBT) pipeline that empirically evaluates the correctness of Gradual C_0's implementation through incorrect programs and specifications. It has been shown [Claessen and Hughes 2000] that capturing the truthiness of a property's results with lightweight methods provides good coverage for finding implementation bugs. In Gradual C_0, the truthiness for all programs consists of a pair of outputs: dynamic and gradual verification output message given by Gradual C_0. Failed equivalence between this pair of outputs informs us of bugs in Gradual C_0's implementation that do not break the gradual guarantee and would not have been caught otherwise.

2 APPROACH

We implement a three-stage pipeline framework that sequentially gradually verifies a program, stores the output message, either a success or a failure message, followed by pure dynamic verification of the same program, and compares its output to the previously stored gradual output. The three stages are composed of a *reference model language* with Gradual C_0's specifications, an *input generator* which is a test suite of examples that are not supposed to verify correctly which we randomly permute to test on, and a *checker* which compares the output from Gradual C_0 and Dynamic C_0 [1]. The checker establishes Dynamic C_0 as the ground truth, expecting either an error or a pass from Gradual C_0 if Dynamic C_0's output is a pass, but they should never differ if Dynamic C_0's output is an error.

The input generator is made up of a dozen programs that come from Gradual C_0's benchmark test suite. The methods from each test are changed to have incorrect specifications and implementations that do not obey each other. The tests in the input generator also have to maintain certain ways of stating specifications. To prevent a trivial failure in the Gradual C_0, programs must avoid specifying preconditions and *fold/unfolds* that won't be met while running. These folds control the availability of predicate information, which is considered an iso-recursive interpretation of predicates. Because static verifiers rely on iso-recursive reasoning, the static verification step in Gradual C_0 will trivially fail with the presence of unmet predicate information.

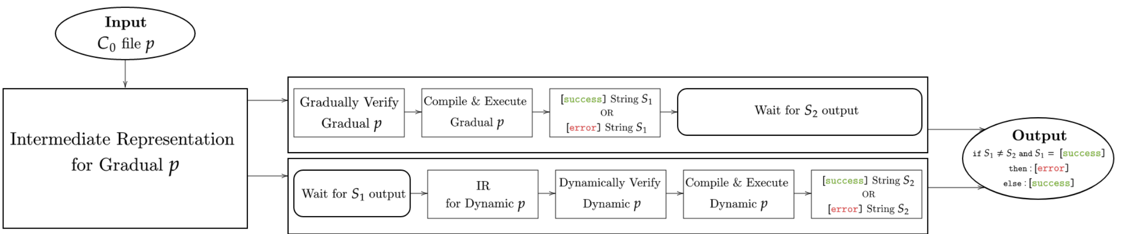

Fig. 1. *Checker* architecture

To test the correctness of our tool, we retrospectively find bugs through Gradual C_0's issue tracker on GitHub, and run the tool on our test suite. In addition to previously mentioned programs, In addition to these programs, our test suite is expanded to include tests that implement failing implementations from each issue in the issue tracker. A particularly interesting and significant issue was originally caught by hand inspection of the internal output of Gradual C_0 after the behavior had been formalized in [Wise et al. 2020] as *Footprint Splitting*. In this issue, Gradual C_0 was not removing information from the *optimistic heap*, framed by an imprecise specification, when it

Commit found	Bug and test that caught it	Property failure caught
August 16, 2022	Issue 38: AVL (Not implemented)	No
August 29, 2022	Issue 46: AVL (Not implemented)	No
May 13, 2022	Issue 27: BST	Yes
May 12, 2022	Issue 25: BST	Yes
August 15, 2022	Issue 44: AVL (Not implemented)	No
April 1, 2022	Issue 34: Custom loop example	Yes
March 9, 2022	Issue 24: List insertion	Yes

(a) Commits which address soundness issues

```
/*@
predicate list(struct Node *l) =
  ? && (l != NULL ? acc(l->value) && list(l->next) : true);
@*/
void append(Node *root, int value)
  //@requires ?;
  //@ensures ? && list(root);
{
  Node *n = root;
  while (n->next != NULL)
    //@loop_invariant ?;
    n = n->next;
  n->next = alloc(Node);
  n->next->value = value;
}
```

(b) Custom loop example referenced in Issue 34

Fig. 2. Tool analysis

should. Permissions would not be tracked inside precise methods that call imprecise methods or methods with internal precision. In our input generator, the set of examples that trigger this issue come from the binary search tree benchmark. This example fails the property because Gradual C_0's output would pass at the creation of the imprecise predicate, *treeRemove*, to delete a binary search tree. This predicate is called in the postcondition of the tree removal function, and recreating run-time permissions after the function is called is incorrect because it causes an accessibility predicate to be missing. Our tool catches this failing property and returns Dynamic C_0's error.

3 TOOL ANALYSIS

We can further understand the usefulness of our tool by running our test suite on a point of Gradual C_0's GitHub commit history. Figure [2].a shows us which issues were caught by our tool. Our entire test suite ran at a rollback of Gradual C_0 to the dates listed under the *Commit found* column and we specify which example in the suite found the bug if any.

This analysis helps to prove the efficacy of PBT in Gradual C_0, capturing most bugs[1]. The only issues that were not caught were due to a benchmark test that was not implemented in our test suite, the AVL benchmark. A more exhaustive test suite that implements this test could have identified all 7 soundness bugs.

4 CONCLUSION

We show that Property Based Testing is a practical tool for finding implementation bugs in Gradual C_0 that do not follow the formal design. However, there are still challenges that must be addressed to exhaust this lightweight method for a rigorous evaluation of bugs never found before. Currently, the test suite is implemented by hand by iterating through the benchmark examples which all pass the gradual guarantee.

This implementation has limited applicability due to the restrictive test suite. A promising approach to expand the domain of bugs caught by our tool relies on iterating through all examples in the benchmark test suite and breaking individual methods by generating random inputs that violate each method's specification. Nevertheless, we lay the groundwork for a consistent lightweight tool which is the first automated method for finding implementation bugs in Gradual C_0.

[1]Issues used for the analysis can be publicly accessed in gvc0 public repository and in silicon gv fork public repository.

REFERENCES

Johannes Bader, Jonathan Aldrich, and Éric Tanter. 2018. Gradual Program Verification. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 25–46.

Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *SIGPLAN Not.* 35, 9 (sep 2000), 268–279. https://doi.org/10.1145/357766.351266

Jenna DiVincenzo, Ian McCormack, Hemant Gouni, Jacob Gorenburg, Mona Zhang, Conrad Zimmerman, Joshua Sunshine, Éric Tanter, and Jonathan Aldrich. 2022. Gradual C0: Symbolic Execution for Efficient Gradual Verification. *arXiv preprint arXiv:2210.02428* (2022).

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting Gradual Typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. ACM, New York, NY, USA, 429–442. https://doi.org/10.1145/2837614.2837670

Jeremy Siek and Walid Taha. 2007. Gradual typing for objects. In *European Conference on Object-Oriented Programming*. Springer, 2–27.

Jeremy G Siek and Walid Taha. 2006. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, Vol. 6. 81–92.

Jenna Wise, Johannes Bader, Cameron Wong, Jonathan Aldrich, Éric Tanter, and Joshua Sunshine. 2020. Gradual verification of recursive heap data structures.