Evaluating Soundness of a Gradual Verifier with Property Based Testing

By Jan-Paul Ramos-Dávila
Department of Computer Science, College of Computing and Information Science

Abstract

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 C0, supports programs that manipulate recursive, mutable data structures on the heap and minimizes dynamic checks with statically available information. The design of Gradual C0 has been formally proven sound; however, this guarantee does not hold for its implementation.

In this paper, we introduce a lightweight approach to testing the soundness of Gradual C0'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 C0 and a fully dynamic verifier for C0, 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 C0's soundness caught a number of significant implementation bugs from Gradual C0'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.

Introduction

Motivation

Historically, we are familiar with a family of formal verification techniques as providing rigorous approaches to ensure the correctness of computer programs, in particular static verification. These techniques require the programmer to fully specify a program's predicted behavior to give formal assurance that the implementation will abide by what the programmer has in mind: a specification. Unfortunately, this approach does not support incrementality, as it requires the previously mentioned full specification to support quality code coverage. Further, static verification tools cannot provide verification feedback on any partial specifications written on the way to complete static specification. This means that if a user has a program and they only want to verify a particular method, they have to pedantically specify the entire method's annotations for the static verifier to receive with full coverage. This includes completely specifying loop invariants, preconditions, postconditions, etc., none of which can be at the discretion of the verifier, as it does not have enough information to make inferences. On the other hand, dynamic verification techniques focus at asserting specific test cases based on inputs given by a user at run-time. This technique avoids having to specify all information in a program, albeit it experiences massive overhead in regards to run-time costs that limit its practicality. In addition, dynamic verification techniques do not provide full code coverage and are meant to work in tandem with formal verification techniques.

Bader, Aldrich, and Tanter (2018), introduces the idea of *gradual verification*, which soundly

This work is licensed under CC BY 4.0.
To view a copy of this license, visit http://creativecommons.org/licenses/by/4.0/

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 & Taha, 2006; Siek & Taha, 2007), 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 (trees, graphs, lists, etc.). Wise et al. proved such a gradual verification system sound, and showed its adherence to the *gradual guarantee* property – relaxing specifications does not introduce new static or dynamic verification errors, if it is sound at full static specifications, then it is sound at the relaxation.

DiVincenzo et al. (2022) introduces the design and implementation of *Gradual C0*, the first gradual verifier for recursive heap data structures inspired by Wise et al.'s foundational theory. It is an extension of the pedagogical *C0* language used by Carnegie Mellon University to teach the fundamentals of dynamic verification with a subset of C that supports dynamic specifications, albeit it lacks memory allocation functions well known in C, e.g. *malloc*. Gradual *C0* is built to minimize dynamic checking with statically available information. DiVincenzo et al. (2022) also shows in a performance study that Gradual *C0* reduces run-time overhead by 50-90% on average compared to dynamic verification. This shows that *Gradual C0* is—conservatively—the most optimal alternative to dynamic verifiers, as the programmer can choose to statically specify trivial control flow in their program and remove significant overhead that would otherwise have been computed at runtime. With this relaxation in specifications, the user can also focus on other parts of the program if they verify a more complex aspect.

Introduction

Framework

Separation logic, introduced by Reynolds (2002), supports static verification of the programs that use the heap structure, a particular part in memory with an ordering property which establishes a maximum, greater than or equal to hierarchy, or a minimum, lesser than or equal to, hierarchy. Dealing with the verification of heaps is difficult because we have to make a distinction in permissions given to particular parts of different memory chunks rather than objects in the same memory chunk. Users can make a distinction about the heap's memory chunks with the points-to-predicate operator "$\mapsto$" and the separating conjunction operator "$*$". The arrow operator asserts both ownership of a heap location and its value, e.g. $x.f \mapsto 2$ states that the location *x.f*—accessing the value of field *f* in structure *x*—is uniquely owned and contains the value 2. Further, the separating conjunction works similarly to the logical and, but it's extended to ensure that two chunks of memory—sub-heaps—are distinct in memory. For example, $x.f \mapsto 2 * y.f \mapsto 2$ states that the heap locations *x.f* and *y.f* are distinct (i.e. $x \neq y$), are each owned, and each contain the value 2. This is useful for verification techniques at runtime as it will allow the tool to make a distinction between sub-heaps that would otherwise be indistinguishable because they share the same field value.

A few years later, Smans (2009) presented implicit dynamic frames (IDF) as an alternative to separation logic, which asserts ownership of a heap location and its value separately. Moreover, ownership is ensured through the use of accessibility predicates, e.g. *acc(x.f)*. Then, $acc(x.f.) * x.f == 2$ states that *x.f* is uniquely owned and contains the value 2. Parkinson and Bierman (2009) and Smans et al. (2009) extended separation logic and IDF respectively to support *recursive abstract predicates* and thus recursive heap data structures—trees, lists, graphs, etc. Abstract predicates can be thought of as pure boolean functions. For example, we

consider the following predicate, which specifies that a list is acyclic:

predicate acyclic(Node root) = root == null ? true : acc(root.val) ∗ acc(root.next) ∗ acyclic(root.next).

Acyclic recursively generates accessibility predicates for each node in a list, and joins the predicates with the separating conjunction. Thus, *acyclic(l)* simple denotes that all heap locations in list *l* are distinct—*l* is *acyclic*—providing recursive behavior.

Partial—*imprecise*—specifications contain incomplete static information, which are marked with a question mark "?", e.g. *? ∗ x.f == 2* where *x.f == 2* is the static part. Empty specifications are completely imprecise, e.g. ? or ? ∗ *true*. Then, during static verification, imprecise specifications are statically strengthened (in non-contradictory ways) to support proof goals. Wherever strengthening occurs, dynamic assertions are inserted to preserve soundness and complete verification. Bader et al.'s (2018) approach of Gradual Verification smoothly supports the spectrum between static and dynamic verification via the previously mentioned gradual guarantee, *conservative extension*, and pay-as-you-go properties. A gradual verifier is a conservative extension of a static verifier if the two verifiers coincide on fully-precise programs. It also exhibits a pay-as-you-go cost model when users are rewarded with increased static correctness guarantees and decreased dynamic checking, as specifications are refined. Gradual C0 exhibits a static verifier that supports implicit dynamic frames and abstract predicates. The static verifier is extended with *gradual formulas* that support the *?* operator. The entire static verifier is then *lifted*—all imprecise states are *optimistically* processed for later dynamic checking. Wise et al. (2020) proved that their system is sound, is a conservative extension of their static verifier, and adheres to the gradual guarantee. However, we currently have no lightweight techniques to verify the soundness of the implementation of Gradual C0, even if we have reassurance the theory is correct. We propose a *lightweight* technique to incrementally verify the soundness of Gradual C0's implementation and adhere to not just the gradual guarantee, but also a soundness property in regards to Gradual C0 emitting correct dynamic checks.

Introduction

Gradual Verifier Architecture

The following example program shows an overview of the Gradual C0 pipeline:

```
1.    void withdrawFee(Account* account)
2.     /*@ requires acc(account→balance) &&
3.          account → balance >= 5; @*/
4.     /*@ ensures acc(account→balance) &&
5.          account → balance >=0; @*/
6.    {
7.          account → balance -= 5;
8.    }
9.    void monthEnd(Account* account)
10.   /*@ requires ? &&
11.         acc(account→balance); @*/
12.   /*@ ensures ? &&
13.         acc(account→balance) &&
14.         account → balance >= 0; @*/
15.   {
16.         if (account → balance <= 100)
17.               withdrawFee(account);
18.   }
```

This program emulates a machine to withdraw money from a bank account. We want to make sure that the bank account has enough money to withdraw from, never go negative. Line 2 implements how side-effects are reasoned about, using *Implicit Dynamic Frames* (IDF). Access to memory locations is specified using `acc(object-field)`. In line 7, program states are represented by the verifier as formulas in a resource logic. Static information at the end of `withdrawFee` includes the account balance, account balance being greater or equal than zero, and the account balance equaling the old account balance minus five. In line 10 we have our first notation exclusive of Gradual Verification, `?`, which allows the verifier to assume anything necessary to complete proofs. The assumption made in order to satisfy the precondition of the method is that the account balance is greater than five.

Finally, in line 17, we have that wherever specifications are strengthened by the verifier, dynamic checks are inserted into the compiled program to ensure proper behavior at runtime, therefore here the verifier asserts that the account balance is greater than five.

Gradual C0 addresses new technical challenges in gradual verification: Gradual C0's symbolic execution algorithm is responsible for statically verifying programs with imprecise specifications and producing *minimally sufficient* run-time checks for soundness. Achieving these goals with symbolic execution is nuanced. In particular, Gradual C0 tracks the branch conditions created by program statements and specifications to produce run-time checks for corresponding execution paths. At run time, branch conditions are assigned to variables at the branch point that introduced them, which are then used to coordinate the successive checks as required. Further, Gradual C0 creates run-time checks by translating symbolic expressions into specifications—reversing the symbolic execution process by DiVincenzo et al. (2022). The run-time checks produced by Gradual C0 contain branch conditions, simple logical expressions, accessibility predicates, separating conjunctions, and predicates. Each of these constructs are specially translated into source code that can be executed at run-time for dynamic verification. Logical expressions are turned into assertions. Accessibility predicates and separating conjunctions are checked by tracking and updating a set of owned heap locations. Finally, predicates are translated into recursive boolean functions. This is where a lot of soundness bugs in Gradual C0 originate, by not correctly tracking the set of owned heap locations and losing information at run-time.

For example, the verifier experienced a bug in regards to the nature of managing heap permissions demonstrates that an arbitrary method runs when it should not. We assume that a method `assign` assigns the value at an address `x` to be 1. This method is very simple, therefore a user might decide to define an imprecise predicate, `imprecise() = ?`, such that the precondition for the method is left up to dynamic verification. The postcondition just ensures true, and we *unfold* (expanding the abstract predicate to give permission into its body) the imprecise postcondition right before the assignment. If we want to then call the method by *folding* (repacking body) the predicate `imprecise()`, allocating memory for `x`, and declaring `x` to be `0`, when we call the method and assert the postcondition that `x = 0`, the program should error. We already defined in the method for the address value to be 1, therefore asserting the return value to be 0 because we manually allocated the address to 0 manually should not change anything. However, the presence of imprecision allowed for the program to successfully verify in Gradual C0. If we denote impreciseness to represent all heap conditions, the verifier assumed that the permissions represented by iso-recursive predicates won't change, but if they are imprecise, their equi-recursive unrolling includes permission to the entire heap. Their permissions will change even after they are folded. In essence, folding and unfolding a predicate that did nothing, which was set as the precondition for a benign assignment method, displayed unsoundness with the implementation of Gradual C0. Catching an issue such as this one requires heavy lifting when analyzing the formalization of Gradual Verification and provides several roadblocks to maintain a formal methods approach to verify Gradual C0, namely the uncertainty on whether to change the actual semantics of Gradual Verification or utilize the front-end to hack a solution in Gradual C0.

Gradual C0 has two major subsystems: 1) the gradual verification pipeline and 2) the C0 pipeline. The gradual verification pipeline is responsible for statically verifying C0 programs and producing run-time checks for soundness. First, a C0 program is translated into a Gradual Viper (an extension of the Verification Infrastructure for Permission-based Reasoning language) program by Gradual C0's frontend module, GVC0. Next, the Gradual Viper

module uses a symbolic execution approach that handles imprecise formulas to statically verify the Gradual Viper program (Viper comprises a novel intermediate verification language). Wherever imprecise formulas are strengthened in support of proofs, Gradual Viper creates run-time checks in its language to ensure soundness. Finally, GVC0 takes those run-time checks and produces a C0 program from them, in addition to the original C0 program. The C0 pipeline takes this C0 program and feeds it to the C0 compiler, CC0, which executes the program.

An example C0 program implementing logic for a bank account is shown in Figure 3. The `monthEnd` method uses the `withdraw` method to remove 5 units from the account when its balance is less than or equal to 100. Gradual specifications partially define the behavior of both `monthEnd` and `withdraw`. For example, the account balance must be a positive value for a call to `withdraw` to be valid. The postcondition of withdraw is unspecified as indicated by `?`. A `?` in the specifications indicates imprecision, allowing the verifier to optimistically assume information, such as access to the balance field, where necessary.

The C0 program is converted to an intermediate representation (IR), that targets both C0 source output and Viper's intermediate language, Silver. For gradual verification, we need to both convert the semantics of the C0 program into Silver and insert verifier-provided dynamic checks into the program before compilation. Intermediate values (such as complex expressions in a method call's arguments) may need to be verified at run-time, and previous values may need to be examined to determine if a check is necessary at run-time. To meet these requirements, the C0 program's IR is transformed to remove re-assignments, similar to single-static-assignment (SSA) transformations.

Following this transformation, the IR is translated into Silver, which is further translated into a logical formula representation used by Silicon Schwerhoff (2016), the verification engine for Viper. During optimistic static verification, the verifier generates run-time checks wherever an optimistic assumption takes place. Where possible, checks are avoided using static information. Further, some checks are only required for specific execution paths through the program; path information is attached to these checks. All checks are emitted to the frontend, which translates and injects them into the C0 IR.

Figure 4 shows a simple dynamic check. The `withdraw` call in Figure 3 elicits this check before the termination of `monthEnd` in order to ensure a valid account balance, but only for the path denoted by the conditional branch. Wise et al. (2020) extended gradual verification to support heap-allocated data structures using implicit dynamic frames (IDF) (Smans et al., 2009). In addition, Viper uses IDF in its implementation of static verification. IDF imposes constraints on the accessibility of fields in heap-allocated data structures. Since gradual verification may require dynamic verification of specifications, gradual verification using IDF must verify field accessibility at run time. To implement this, an additional argument is added to each method. This argument is used to specify the fields accessible by the method. When calling a fully specified method, the caller passes only the permissions specified in the callee's preconditions. However, for gradually specified methods, all of the caller's permissions are passed. A dynamic check for field access asserts that this set contains a tuple of the field and its parent `struct` reference. This allows the side-effects of fully specified methods to be known during static verification even if they call gradually specified methods where side-effects are not specified.

Approach

Lightweight Verification

While empirically evaluating Gradual C0's performance in DiVincenzo et al. (2022), Gradual *C0* 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 C0'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 C0's implementation through incorrect programs and specifications. It has been shown that capturing the *truthiness* of a property's results with lightweight methods provides good coverage for finding implementation bugs (Claessen & Hughes, 2000). In Gradual C0, the truthiness for all programs consists of a pair of outputs: dynamic and gradual verification output message given by Gradual C0. Failed equivalence between this pair of outputs informs us of bugs in Gradual C0's implementation that do not break the gradual guarantee and would not have been caught otherwise.

Unlike classical tools for property based testing, we are not generating input for programs, rather generating programs themselves to input into the pipeline. 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*—Gradual C0's specification language—an *input generator*—test suite of examples that are not supposed to verify correctly which we randomly permute to test on—and a *checker*—compares the output from Gradual C0 and Dynamic C0 (Figure 1). The checker establishes Dynamic C0 as the ground truth, expecting either an error or a pass from Gradual C0 if Dynamic C0's output is a pass, but they should never differ if Dynamic C0's output is an error.

We choose Dynamic C0 as the ground truth because Gradual C0 already has an empirical reassurance of static soundness in the verifier thanks to our benchmarking system, which remedies the issue with Dynamic C0 being complete but not sound: we don't guarantee that a program satisfies a specification if it passes. We evaluate the previously mentioned 50-90% efficiency of the tool by emulating Takikawa et al., (2016)'s performance lattice method, a method to measure run-time cost of gradual typing by testing various configurations of the typed and untyped code, but with relaxed specifications instead. Our benchmark is made up of four fully statically verified algorithms–namely BST (Binary Search Tree), AVL (Adelson-Velsky and Landis BST), Composite, Linked list insertion. If the benchmarking is a success, then we have a pseudo-empirical reassurance of our static verifier because we have passed a fully-precise program. However, dynamic checks while relaxing specifications could be unsound. We therefore use Dynamic C0 as the ground truth, by comparing how the pure dynamic verifier asserts dynamic checks with no static information (it should always emit them correctly, although the program will be very slow to verify), against Gradual C0 which is asserting dynamic checks given optimistic static information.

The input generator is made up of a dozen methods that come from Gradual C0'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 of the static verifier in Gradual C0, programs must avoid specifying preconditions and *fold/unfolds* (explicit statements to control the availability of predicate information) that won't be met while running. These folds control the availability of predicate information. Verification tools cannot automatically deal with recursive information in specifications. If a recursive function is referenced in a precondition, for example, the programmer must explicitly fold/unfold the recursive information, similar to specifying loop invariants. These folds/unfolds are considered

an iso-recursive interpretation of predicates. Because static verifiers rely on iso-recursive reasoning, the static verification step in Gradual C0 will trivially fail with the presence of unmet predicate information.

Approach

Inputting to the Input Generator

Any contradictory output regarding the success of Gradual C0 and Dynamic C0's output will result in a reduction of the code to find which method is resulting in the error, informing us where the bug could be in Gradual C0's implementation. For example, the lightweight technique known as *QuickCheck* attempts to write assertions about logical properties that a function should fulfill, and attempts to generate a test case that falsifies such assertions, our reduction follows the same pattern. Once such a test case is found, QC tries to reduce it to a minimal failing subset by removing or simplifying input data that are unneeded to make the test fail. Ideally, however, a fully *fuzzy* tool would auto generate random input to empirically test soundness. Our current tool does not have a way to generate Gradual C0 programs that contradict in specification and implementation. Instead, we take an approach closer to *mutation testing*, in which we slightly modify individual methods in our three of our four benchmarking algorithms. We then have a lower level of property-based testing in which we generate inputs for these methods in association to whatever structure was changed, if we encounter an arithmetic change we have integer literals. These slight modifications search for basic operators in the core logic, e.g. greater than, addition, loop termination, and replaces them with the dual operation, e.g. lesser than, subtraction, different loop termination value, respectively.

We can strengthen the exhaustiveness by including examples which have been caught by hand in the past, as shown in Figure 2. This example begins with an append method with no permissions, and in an imprecise state. When we encounter the while loop on line 10, we correctly emit a check for asserting that we have access to `n->next`. The first iteration of the loop is fine, because we have checked that we have access to `n->next`. Before we begin the next iteration, though, we should check again that we have access to `n->next` (since we might have lost it on the current iteration). If we don't have access, the program shouldn't be able to evaluate the loop condition, and should crash. Gradual C0 did not crash at the time of the bug being found, because it was unsound. This was due to an issue on permissions given to the optimistic heap at a certain time in compilation. Our tool was able to make a distinction between Gradual C0's output and Dynamic C0's resulting error and immediately identify the bug.

Tool Analysis

To test the correctness of our tool, we retrospectively find bugs through Gradual C0's issue tracker on GitHub, and run the tool on our test suite. In addition to previously mentioned 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 with the formalization of Gradual Verification from Wise et al. (2020) regarding the internal output of Gradual C0 after the behavior had been formalized—*Footprint Splitting*. In this issue, Gradual C0 was not removing information from the *optimistic heap*, framed by an imprecise specification, when it 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 C0'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 C0's error.

We can further understand the usefulness of our tool by running our test suite on a point of Gradual C0's GitHub commit history. Table 1 shows us which issues were caught by our tool. Our entire test suite ran at a rollback of Gradual C0 to the dates listed under the *Commit found* column and we specify which example in the suite found the bug if any. The custom loop example on April 1st is shown in Figure 2. This analysis helps to prove the efficacy of PBT in Gradual C0, capturing most bugs. 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.

Conclusion

Gradual Verification is a powerful tool that supports the relaxation of static specification without losing any information to create unsoundness issues. It leverages dynamic and static techniques and has been proven to have soundness guarantees. However, there are various bugs in the implementation of Gradual C0, the first gradual verification tool, that cannot currently be addressed with formal methods, due to a lack in consistency when bridging the formal semantics of Gradual Verification and non-theoretical hacks to solve implementation issues in Gradual C0. To combat this uncertainty, we develop a lightweight tool relying on Property Based Testing for finding implementation bugs in Gradual C0 that do not follow Gradual Verification's formal design, instead opting for front-end based solutions. However, there are still challenges that must be addressed to exhaust this lightweight method for a rigorous evaluation of new bugs with arbitrary programs. Currently, the test suite is implemented by hand by iterating through the benchmark examples which all pass the gradual guarantee. The new bugs that have been found are associated with this set of (exhaustive) examples and arbitrary programs might expose bugs which aren't guaranteed to be caught with this tool. We know that testing is enhanced by the specifications that are written at the boundaries – they might even help with generating tests. Conversely, the dynamic verification part of our work is only useful if test cases cover the places where the assertions are.

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 C0.

Figures/Tables

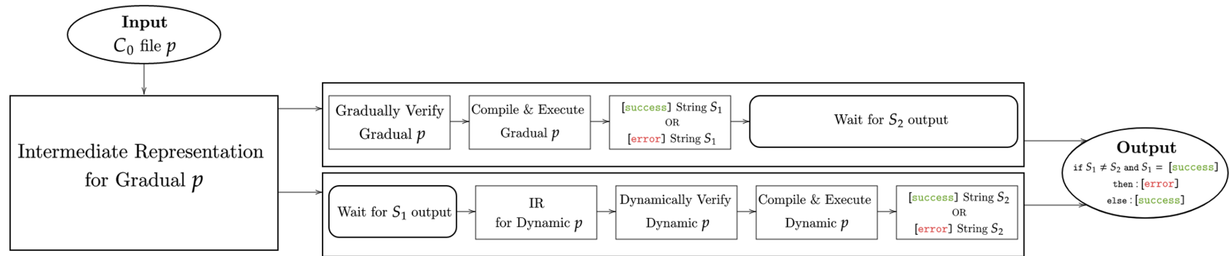

Figure 1. Checker Architecture

```
/*@
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;
}
```

Figure 2. Custom loop example for issue 34

```
void monthEnd(Account *account)
  /*@ requires ? && account->balance >= 0; @*/
  /*@ ensures ? && account->balance >= 0; @*/ {
  if (account->balance <= 100)
    withdraw(account, 5);
}

void withdraw(Account *account, int amount)
    /*@ requires acc(account->balance) &&
        account->balance >= 0; @*/
    /*@ ensures ?; @*/ {
    ...
}
```

Figure 3. Bank account example of a gradually verified program in C0

```
if (previous_account_balance <= 100)
   assert(account->balance >= 0);
```

Figure 4. Runtime check example in Gradual C0

Table 1. Bugs caught with the PBT tool

Commits found	Bugs 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: AVOL (Not implemented)	No
April 1, 2022	Issue 34: Custom loop example	Yes
March 9, 2022	Issue 24: List insertion	Yes

References

Astrauskas, V., Müller, P., Poli, F., & Summers, A. (2019). *Leveraging Rust types for modular specification and verification*. Proceedings of the ACM on Programming Languages 3, OOPSLA (2019), 1–30. https://doi.org/10.1145/3360573

Bader, J., Aldrich, J., & Tanter, É. (2018). *Gradual Program Verification*. In International Conference on Verification, Model Checking, and Abstract Interpretation. Springer, 25–46.

Claessen, K., & Hughes, J. (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

DiVincenzo, J., McCormack, I., Gouni, H., Gorenburg, J., Zhang, M., Zimmerman, C., Sunshine, J., Tanter, É., & Aldrich, J. (2022). *Gradual C0: Symbolic Execution for Efficient Gradual Verification*. arXiv preprint arXiv:2210.02428 (2022).

Eilers, M. & Müller, P. (2018). *Nagini: a static verifier for Python. In International Conference on Computer Aided Verification*. Springer, 596–603. https://doi.org/10.1007/978-3-319-96145-3_33

Garcia, R., Clark, A., & Tanter, É. (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

Parkinson, M., & Bierman, G. (2005). *Separation logic and abstraction*. In ACM SIGPLAN Notices, Vol. 40. ACM, 247–258.

Reynolds, J. (2002). *Separation logic: A logic for shared mutable data structures. In Logic in Computer Science*, 2002. Proceedings. 17th Annual IEEE Symposium on. IEEE, 55–74.

Schwerhoff, M. (2016). *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution*. Ph.D. Dissertation. ETH Zurich. https://doi.org/10.3929/ethz-a-010835519

Siek, J., & Taha, W. (2006). *Gradual typing for functional languages*. In Scheme and Functional Programming Workshop, Vol. 6. 81–92.

Siek, J., & Taha, W. (2007). *Gradual typing for objects*. In European Conference on Object-Oriented Programming. Springer, 2–27.

Smans, J., Jacobs, B., & Piessens, F. (2009). *Implicit dynamic frames: Combining dynamic frames and separation logic*. In European Conference on Object-Oriented Programming. Springer, 148–172. https://doi.org/10.1007/978-3-642-03013-0_8

Takikawa, A., Feltey, D., Greenman, B., New, M. S., Vitek, J., & Felleisen, M. (2016). Is sound gradual typing dead? ACM SIGPLAN Notices, 51(1), 456–468. https://doi.org/10.1145/2914770.2837630

Wise J., Bader J., Wong, C., Aldirch, J., Tanter, É., & Sunshine, J. (2020). *Gradual verification of recursive heap data structures*

Wolf, F., Arquint, L., Clochard, M., Oortwijn, W., Pereira, J., & Müller, P. (2021). *Gobra: Modular Specification and Verification of Go Programs*. In International Conference on Computer Aided Verification. Springer, 367–379. https://doi.org/10.1007/978-3-030-81685-8_17