Implementation of an End-to-End Gradual Verification System

Hemant Gouni*
gouni008@umn.edu
University of Minnesota, Twin Cities
United States

Conrad Zimmerman*
conrad_zimmerman@brown.edu
Brown University
United States

Abstract

Static verification is used to ensure the correctness of programs. While useful in critical applications, the high overhead associated with writing specifications limits its general applicability. Similarly, the run-time costs introduced by dynamic verification limit its practicality. Gradual verification validates partially specified code statically where possible and dynamically where necessary. As a result, software developers gain granular control over the trade-offs between static and dynamic verification. This paper contains an end-to-end presentation of gradual verification in action, with a focus on applying it to C_0 (a safe subset of C) and implementing the required dynamic verification.

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

Keywords: gradual verification, program correctness, implicit dynamic frames

ACM Reference Format:
Hemant Gouni and Conrad Zimmerman. 2021. Implementation of an End-to-End Gradual Verification System. In *Companion Proceedings of the 2021 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '21), October 17–22, 2021, Chicago, IL, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3484271.3484980

1 Motivation

Programs fully annotated with static specifications provide unmatched correctness guarantees. However, static verification tools often require complete and thorough specifications, introducing a prohibitive overhead for modern software. Dynamic verification tools largely eliminate this burden at the cost of performance, and are limited in their guarantees.

Gradual verification creates a smooth spectrum of program specification between static and dynamic verification. Specifically, it enables static specifications to be developed incrementally, allowing the behavior of unspecified components to be verified dynamically. However, previous work on gradual verification [11] has not implemented the dynamic portion of the system, nor support for a language capable of using it.

2 Introduction

While static verification requires extensive specifications to prove programs correct, gradual verification allows non-contradictory strengthening of incomplete specifications to complete proofs. In order to preserve soundness when this occurs, a gradual verifier emits executable checks which ensure a program behaves according to its specifications at run time.

Our work on gradual verification follows from previous work on verification in the context of gradual typing. Bader et al. [3] developed a verification system for simple arithmetic specifications by building on the Abstracting Gradual Typing [5] technique. Wise et al. [9] advanced this system to support reasoning about memory and complex data structures. More recently, Zhang and Gorenburg [11] extended the Viper verification toolchain [6] to support static discharging of imprecise specifications. We have further extended their work to emit the necessary dynamic checks, ensuring soundness in the presence of imprecision. Additionally, we have implemented a C_0 [1] frontend for Gradual Viper which extends the C_0 compilation pipeline to support gradual verification. The next section describes the generation of dynamic checks and the novel technical challenges that we overcame to implement end-to-end support for gradual verification.

The C_0 language is a safe subset of C tailored for teaching and academic use. Its small surface area simplifies the implementation of a new verification system. Additionally, the familiarity of C_0 users with its existing support for (dynamically verified) specifications allow us to more readily compare the efficacy of gradual verification to that of other methods in the future.

---

[*] Both authors contributed equally to this research.

SPLASH Companion '21, October 17–22, 2021, Chicago, IL, USA

ACM ISBN 978-1-4503-9088-0/21/10…$15.00
https://doi.org/10.1145/3484271.3484980

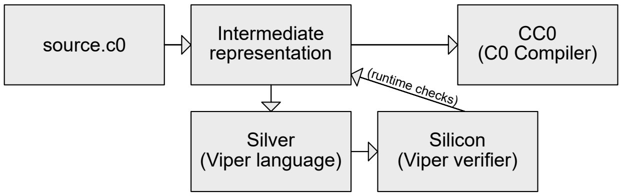

Figure 1. Gradual C_0 program verification pipeline

3 Approach

The pipeline for our gradual verification toolchain is summarized in Figure 1. An example C_0 program implementing logic for a bank account is shown in Figure 2. 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.

```
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 2. Use of gradual verification in a C_0 program

The C_0 program is converted to an intermediate representation (IR), that targets both C_0 source output and Viper's intermediate language, Silver. Translation to Silver has been previously implemented for Go [10], Python [4], and Rust [2], among others. For gradual verification, however, we need to both convert the semantics of the C_0 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 C_0 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 [7], 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 C_0 IR.

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

Figure 3. An example branch-dependent run-time check

Figure 3 shows a simple dynamic check. The `withdraw` call in Figure 2 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. [9] extended gradual verification to support heap-allocated data structures using implicit dynamic frames (IDF) [8]. 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.

4 Conclusion

In the process of extending existing verification tools to implement a full gradual verification system, we encountered additional challenges such as the addition of run-time checks and dynamic verification of side-effects using IDF. This implementation has limited applicability due to the restrictions of C_0 , but lays the groundwork for future applications in more widely used languages. Moreover, it represents the first toolchain that allows the use and benefits of gradual verification to be evaluated in practice.

5 Acknowledgements

This research was supported by the National Science Foundation under Grant No. CCF-1901033.

References

[1] Rob Arnold. 2010. *C0, an imperative programming language for novice computer scientists.* Ph.D. Dissertation. Master's thesis, Department of Computer Science, Carnegie Mellon University.

[2] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J Summers. 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

[3] J. Bader, J. Aldrich, and É. Tanter. 2018. Gradual Program Verification. In *VMCAI*. https://doi.org/10.1007/978-3-319-73721-8_2

[4] Marco Eilers and Peter Müller. 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

[5] 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

[6] Peter Müller, Malte Schwerhoff, and Alexander J Summers. 2016. Viper: A verification infrastructure for permission-based reasoning. In *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 41–62. https://doi.org/10.1007/978-3-662-49122-5_2

[7] Malte H Schwerhoff. 2016. *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution.* Ph.D. Dissertation. ETH Zurich. https://doi.org/10.3929/ethz-a-010835519

[8] Jan Smans, Bart Jacobs, and Frank Piessens. 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

[9] Jenna Wise, Johannes Bader, Cameron Wong, Jonathan Aldrich, Éric Tanter, and Joshua Sunshine. 2020. Gradual Verification of Recursive Heap Data Structures.. In *OOPSLA*. https://doi.org/10.1145/3428296

[10] Felix A Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João C Pereira, and Peter Müller. 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

[11] Mona Zhang and Jacob Gorenburg. 2020. Design and implementation of a gradual verifier. In *Companion Proceedings of the 2020 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. 31–33. https://doi.org/10.1145/3426430.3428137