Optimization of a Gradual Verifier: Lazy evaluation of Iso-recursive Predicates as Equi-recursive at Runtime

JAN-PAUL RAMOS-DÁVILA, Cornell University, USA
jvr34@cornell.edu, Advised by Jonathan Aldrich, Undergraduate, 9119779

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. However, the current approach for evaluating these dynamic checks is naive during incremental specificity. Dynamic checks are re-asserted for all imprecise logic, even if these formulas might share the same trace path.

In this paper, we introduce an optimization pipeline for identifying and minimizing these common traces. We accomplish this by treating all *iso-recursive* predicates as *equi-recursive* at runtime. To not break the soundness of Gradual C_0, we do not make any changes to the semantic evaluation of the static verifier. Instead, we preserve the evaluation of iso-recursive predicates statically and carry this information into Gradual C_0's *Gradual Viper* intermediate representation. By unrolling all static predicates as if they were dynamic checks, we are able to correlate when the naive dynamic checks from the *optimistic* IR overlap. This allows us to perform a more sophisticated predicate equivalence matching using an SMT solver. We suggest that the lazy treatment of static specifications at the source level is more in line with the incremental philosophy of gradual verification, and better supports more complex dynamic checks. We show the effectiveness with our benchmarks of tree data structures. We also raise the question of synthesizing *intermediate predicates* to bridge partial predicate matching.

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

Additional Key Words and Phrases: gradual verification, program correctness, second-order predicates

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. [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. Gradual C_0 uses *folds/unfolds* to develop static specifications. Unfolding a predicate consumes the *predicate instance* and introduces its body into the analysis. If the body has impreciseness (?), the predicate is treated *optimistically*. Folding the predicate instance consumes its body in favor of the instance itself, in order to satisfy any user-written specification. This controlling of predicate availability is *iso-recursive*. We consequentially treat dynamic checks *equi-recursively*, which treats the predicate as their complete unfolding [Summers and Drossopoulou 2013].

We 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. However, these benchmarks do not take into account the average cases for which developers write incremental specifications, a study of mostly static systems with minimal dynamic checks. For example, we can take a specification with an imprecise pre-condition and precise post-condition, with precise loop invariants. Gradual C_0 will naively, but soundly, produce dynamic checks to satisfy the pre-condition because it will reason that these assertions cannot be determined at runtime, as the user has specified. However, the static verifier does reason about the specified loop invariants and post-conditions. The optimistic static verifier takes all the predicate slices for which it could not reason statically, and asserts them in the same IR wherever impreciseness (?) was introduced.

However, due to the verifier reasoning statically about the precise specifications, there comes an overlapping between the dynamic verifier's traces of execution and the already reasoned about predicate slices, as part of a larger static predicate. Because the static verifier treats the predicates *iso-recursively*, the unrolling of the body is fully not exposed and the dynamic verifier cannot make sense of an overlapping between these sliced conditions. This is a consequence of over-impreciseness from the user, which should be caught by the verifier.

2 EXAMPLE

Gradual C_0 uses a set of benchmarks in the domain of tree data structures (linked list, AVL tree, binary search tree, etc.). When we make the method in Figure [1] from the AVL benchmark imprecise[1], we encounter that at a certain trace in the IR, two dynamic asserts which could not be reasoned about statically are followed by two static function calls. When we reason manually about the execution of these static functions, the variables that the verifier's SMT solver declared overlap for the case `_1 = node->leftHeight` and `_ = node->right`. In addition, the dynamic assertions coincide with the *unrolled body* of the static functions (for `avlh := .. && root->leftHeight - root->rightHeight < 2 && ..`). This is particularly an issue for the verification of loop invariants, which will re-assert the entire predicate for `avlh` when the loop might just affect a single branch of the tree.

```
// Recursive function to insert a key in the subtree rooted
// with node and returns the new root of the subtree.
struct Node* insert(struct Node* node, int h, int key, struct Height *hp)
  //@ requires ?;
  //@ ensures ? && acc(hp->height) && avlh(\result, hp->height);
{
```

```
assert(_1 - node->leftHeight < 2); // Dynamic assertion
assert(node->leftHeight >= 0); // Dynamic assertion
avlh(node->right, _1, _ownedFields); // Static function call
avlh(_, node->leftHeight, _ownedFields); // Static function call
```

Fig. 1. Imprecise AVL method and resulting IR

3 APPROACH

We implement a (broadly) three-stage pipeline that preserves the semantic structure of static checks, while just changing when we want dynamic checks to occur. In Section 2, we showed an example for which the overlapping assertions are called in the same control flow. However, it is possible that "independent" control flow might have side effects that result in the same traces. To reason about these, we treat all assertions, no matter where in the IR they are, as a flattened set under the same execution branch. To reason about the body of the static function, we have to treat the

[1]The example shown only has an imprecise pre and post-condition. However, the full example also exhibits imprecise assertions in the loop invariants, which can be found here. The comments in the figure are added for demonstration purposes.

function *equi-recursively*. This exposes the body at runtime, and we can now treat a full predicate as *sliced predicates*. These sliced predicates are what make up the bucket of predicates which will be candidates for overlapping and removal. Because we only care about the case for which runtime information is re-asserted, the normal static function calls will stay independent of the new *dynamic condition*, which guards the dynamic unrolled static functions.

Another assumption is an arbitrary unrolling. As *iso-recursive* functions are, non-surprisingly, recursive, we need a stopping point for unrolling the body. Because we are dealing with tree data structures, it is enough to assume that we can unroll the predicate once and not lose much performance from subsequent predicate calls. However, for more advanced algorithms, a heuristic for identifying a stopping point, such as *unit-stuttering* proposed by [Purandare et al. 2010] for runtime monitors. With these assumptions, the three-stage pipeline is as follows:

- ***Slice construction***: Predicates are gathered and unfolded to **1-depth** if they are recursive. This is done in the front-end of Gradual C_0's pipeline, at the source level, before any verification happens.
- ***Equivalence identification***: We keep track of the *path condition*[2] and identifies which conditions overlap and discard. This is using the back-end's implementation of the *Z3 SMT solver* as inherited by [Müller et al. 2016].
- ***Runtime assertions***: We insert the unfolded predicate into the verified code body, by the same mapping that keeps track of the imprecise specifications. The mappings for the traces from the SMT solver are reasoned about in *buckets*, which simply follow from our front-end metadata pre-processing (Slice construction). Finally, Figure [2] shows the final IR.

```
if (_ == node->right && _1 == node->leftHeight) { // New dynamic condition
    avlh(node->right->left, node->right->leftHeight) // Unrolling at 1-depth
    avlh(node->right->node->right, node->right->node->rightHeight)
    assert(node->leftHeight - node->rightHeight < 2) // Exposed body at runtime
    ...
    } else {
        avlh(node->right, _1, _ownedFields);
        avlh(_, node->leftHeight, _ownedFields);
    }
```

Fig. 2. Optimized IR

4 CONCLUSION

We show a technique for practically evaluating iso-recursive predicates as equi-recursive in Gradual C_0. Such an approach also motivates reasoning about future gradual semantics in contrast with the at-runtime implementation. However, there are is notable future work. Our implementation not only works for assertions $p(x); p(y)$, but also on assertions $p(x); q(x)$, as we're only looking at the unrolled body of such predicates. We can further abstract the cases for which predicate slices overlap by reasoning through implication. For example, a predicate of the form $p(x) := r(x) \wedge q'(x)$ where $q(x) \implies q'(x)$ should evaluate immediately at `assert` $q(x)$; `assert` $p(x)$, without having to evaluate $r(x)$.

Currently, this second-order predicate logic is difficult to reason about by making use of our current back-end implementation for SMT solving. Exploring work with an oracle-guided synthesis of possible candidates for targeting $q(x)$ [Polgreen et al. 2021] and determining their soundness using an SMT solver would introduce sound partial predicate evaluation. Nonetheless, we lay the groundwork for a sound approach to bridging the gap between static and dynamic analysis.

[2]The path condition is stored as part of the symbolic state, consisting of a boolean for impreciseness (how we identify the predicate is candidate), a symbolic heap, a symbolic store, and a collection. All this information is reasoned about in *Gradual Viper*'s back-end.

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.

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

P. Müller, M. Schwerhoff, and A. J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation (VMCAI) (LNCS, Vol. 9583)*, B. Jobstmann and K. R. M. Leino (Eds.). Springer-Verlag, 41–62.

Elizabeth Polgreen, Andrew Reynolds, and Sanjit A. Seshia. 2021. Satisfiability and Synthesis Modulo Oracles. https://doi.org/10.48550/ARXIV.2107.13477

Rahul Purandare, Matthew B. Dwyer, and Sebastian Elbaum. 2010. Monitor optimization via stutter-equivalent loop transformation. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. ACM. https://doi.org/10.1145/1869459.1869483

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.

Alexander J Summers and Sophia Drossopoulou. 2013. A formal semantics for isorecursive and equirecursive state abstractions. In *European Conference on Object-Oriented Programming*. Springer, 129–153.

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