Gradual Verification of Fractional Permissions

Craig Liu
Purdue University
West Lafayette, USA
liu3477@purdue.edu

Abstract

Static verification assures program correctness but requires heavy user annotation. Gradual verification alleviates this burden by allowing users to write partial, imprecise specifications that are checked statically where possible and dynamically when necessary. The first gradual verifier, Gradual C0, reasons about shared heap memory through a permission logic. This paper describes an extension to Gradual C0 supporting fractional permissions in its permission logic.

1 Introduction

Static verification provides strong guarantees of program correctness. Unfortunately, it requires users to write numerous, highly detailed specifications, rendering it difficult to use in practice.

In response, Bader et al. [2] proposed *gradual verification*, which supports the incremental specification and verification of simplistic programs. Gradual verification supports partial, imprecise specifications signified with the `?` operator that can be fully unknown (`?`) or joined with a static part (`? && a > 5`). During verification, these specifications may be augmented with unknown information that is needed for static verification to succeed and does not contradict what is known statically. Dynamic checks for this information are added to the program to ensure soundness. Further work on gradual verification has extended it to work on recursive heap data structures [5, 11].

In particular, DiVincenzo et al. [5] designed and implemented the first and currently only working gradual verifier, Gradual C0. Gradual C0 is built on top of the Viper [6] verification infrastructure, which uses symbolic execution for reasoning. Viper and Gradual C0 support reasoning about heap locations through the *implicit dynamic frames* (IDF) logic [10], a variant of separation logic [9]. IDF uses *accessibility predicates* (e.g. `acc(x->f)`) to denote permission to access object x's field f. A heap location, such as `x->f` cannot be accessed in code or in specifications without corresponding permission, otherwise the verifier will produce an error. Consequently, accessibility predicates can be used to specify the shape of the heap. Another important aspect of IDF is the *separating conjunction* (`&&` in Viper) [8], which is used to assert that two heap locations are disjoint. When its two arguments are not accessibility predicates, then it behaves as the normal logical conjunction. With accessibility predicates, an assertion `acc(x->f) && acc(y->f)` means that `x->f` and `y->f` are different heap locations, or disjoint.

Currently, Gradual C0 only supports accessibility predicates that give full write permission to a heap location (e.g. `acc(x->f, 1/1)`). Therefore, Gradual C0 lacks the ability to verify more fine-grained access, such as read and write behavior to a heap location. This is especially limiting when verifying concurrent programs, where multiple threads may read the same location. Fortunately, *fractional permissions* [4, 6] provide a mechanism by which read-only behavior may also be specified and verified. Fractional permissions (e.g. `acc(x->f, 1/2)`) have an associated fraction between zero and one. A heap location can be read if the fraction is non-zero, and writing to it is allowed if the fraction is one. Then, fractions may have arithmetic operations performed on them to express the division or combination of access rights. For instance, the separating conjunction now performs addition of permissions. Specifying `acc(x->f, 1/2) && acc(x->f, 1/2)` gives `x->f` permission 1, and so the ability to write to it. In addition, two heap locations are now disjoint if the sum of their permissions is greater than `1`. This paper presents the design of an extension of Gradual C0 that supports fractional permissions, which is inspired by Viper's implementation of fractional permissions.

2 Approach

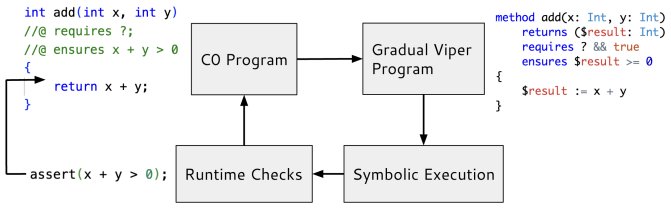

Figure 1. Gradual C0 Verification Pipeline

Gradual C0 verifies programs written in C0 [1], a subset of C designed for educational use. Its pipeline is shown in Figure 1 along with a small example for context. First, it translates C0 programs into its intermediate representation, the Gradual Viper language, which is an extension of the Viper intermediate language that supports `?` in specifications. Then the Gradual Viper backend statically verifies the program using symbolic execution, producing the necessary run-time checks. Gradual C0 embeds these checks within the C0 program for dynamic verification. In the example, the C0 function `add` is first translated into a Gradual Viper method. Although the precondition of `add` is not strong enough to

prove the postcondition `x + y > 0`, `?` can optimistically justify it, and so a run-time check `assert(x + y > 0)` is inserted into the original C0 program before the return statement.

Gradual Viper's symbolic executor tracks program information using a symbolic state composed of a variable store, a set of path conditions maintaining constraints down an execution path, and a heap. To support gradual verification, the state also has a boolean denoting whether the state is imprecise, a set of run-time checks collected, and an optimistic heap. The optimistic heap contains permissions that are valid due to optimism in the program. The heap maintains the invariant that the permission to a location can never exceed one, but the optimistic heap does not.

We use Figure 2 to demonstrate how Gradual C0 should work with fractional permissions, focusing on interesting technical challenges and solutions we implemented. The function `addBalance` adds the two bank account balances, assigns the resulting sum to the first balance, and returns it.

```
1  int addBalance(Account *a, Account* b)
2  //@ requires ? && acc(a->balance, 1/2);
3  /*@ ensures acc(a->balance, 1/2) &&
4      \result == a->balance; @*/
5  {
6      int firstBal = a->balance;
7      int secondBal = b->balance;
8      a->balance = firstBal + secondBal;
9      return a->balance;
10 }
```

Figure 2. Gradual Verification Example

Since the precondition explicitly gives permission to read `a->balance` with `acc(a->balance, 1/2)`, the assignment to `firstBal` on line 6 succeeds. Next, the assignment to `secondBal` on line 7 would fail statically since there doesn't exist permission to read `b->balance`. However, since the precondition is imprecise and having permission to read `b->balance` does not contradict known static information, an assertion that `b->balance`'s permission is positive is added to the set of run-time checks and verification continues. Additionally, the symbolic state is now imprecise due to the imprecise precondition, and so optimistic verification is now permitted.

Permission to access `b->balance` is also added to the optimistic heap so that future run-time checks may be avoided in the program. However, we cannot add it with any concrete permission, since that would break the *dynamic gradual guarantee* [2], which states that if a program takes a step at run-time, then a less precise program will also take the same step. Informally, this means that replacing specifications with `?` should never result in an earlier run-time failure.

To illustrate why, consider adding `b->balance` to the optimistic heap with some arbitrary positive permission, such as `1/2`. To preserve soundness, the run-time check for permission to `b->balance` would need to be for `1/2` permission rather than just some positive permission. As a result, this check would fail at run-time if `b->balance`'s permission is a fraction less than `1/2`, such as `1/4`. But now consider a more precise version of the method that specified `acc(b->balance, 1/4)` instead of `?` in the precondition. Reading `b->balance`'s would succeed statically. In addition, such a program would be able to read `b->balance` at run-time even if it only had permission `1/4` dynamically. In this case, the more precise version would execute further at run-time than the original version, breaking the dynamic gradual guarantee. This is not a problem without fractional permissions since heap locations would only either have full write permission or none, and so it would not be possible for a run-time check for a heap location to fail when it has some permission.

To address this issue, we adopt an idea similar to *counting permissions* [3]. We add `b->balance` to the optimistic heap with permission ε, which represents a non-zero permission less than one. Crucially, if a heap location has ε permission to it, then it possesses less permission than any positive permission. In this way, we avoid breaking the gradual guarantee.

Next, the method assigns to `a->balance` the sum of the two balances on line 8. Writing to a heap location checks that it has the requisite permission, removes it from the heap, and then adds it again with its new value. Although `a->balance`'s permission is only `1/2`, writing to it is justified by the imprecise state and because `a->balance` having write permission does not contradict known information. Unfortunately, doing this also requires removing all heap locations that are not statically proven disjoint from `a->balance`, which means removing `b->balance`. This is because `b` and `a` may alias, so removal of both is necessary since they both could have been written to. A run-time check for `acc(a->balance, 1/1)` is generated, and `a->balance` is added to the heap with permission one and its new value. Finally, the method returns `a->balance` on line 9, and the postcondition verifies successfully since `a->balance`'s permission is greater than or equal to `1/2`, and the return value is equal to `a->balance`.

Although fractional permissions are most useful for concurrent programs, this example also demonstrates a minor benefit for sequential programs — easing specification of aliasing. Since `addBalance` only reads from the balances of `a` and `b` one could replace `?` with `acc(b->balance, 1/2))` in the precondition to statically verify the program. In that case, since the sum total of permissions to both balances is less than 1, `a` and `b` could alias. But without fractional permissions, specifying aliasing would require a precondition similar to `a == b ? acc(a->balance) : (acc(a->balance) && acc(b->balance))`, as the separating conjunction would require that its two arguments be disjoint. In addition, fractional permissions also benefit symbolic execution performance by avoiding the branching present in this latter specification.

3 Implementation

We have implemented fractional permissions in Gradual Viper, extending its syntax, type-checking, and symbolic execution algorithm to reflect the behavior of the example in the previous section. The symbolic execution algorithm is based on four main functions: `eval`, `produce`, `consume`, and `exec`. They respectively evaluate expressions, add permissions and constraints to the state, check constraints and remove permissions from the state, and execute statements. An SMT solver is used to discharge proof obligations.

First, we provide some preliminary definitions. A symbolic state σ is a 6-tuple ($\texttt{isImprecise}$, $h_?$, h, γ, π, $\mathcal{R}$). As described in the previous section, it consists of a boolean flag `isImprecise`, an optimistic heap $h_?$, a heap h, a variable environment γ, a set of path conditions π, and a set of run-time checks $\mathcal{R}$. The symbolic heaps are multi-sets of heap chunks. In particular, a field chunk $id(r;\ \delta,\ p)$ represents the expression $r.id$; a field name id and a receiver r mapping to symbolic value δ with permission p. The function $\texttt{check}(\pi,\ t)$ queries the SMT solver to see if the given path condition π implies the constraint t.

Figure 3 is an example of one modified Gradual Viper function: evaluation of field access for $e.f$. We choose to show `eval` due to its importance. It corresponds to reading a field of a struct pointer `e->f` in C0 (or `b->balance` from Figure 2). In addition, `eval` is also used extensively by the other 3 symbolic execution functions. The `eval` function takes in as argument a symbolic state σ and an expression to be evaluated. It will output a new symbolic state as well as a symbolic value that is the result of the evaluation. The `eval` rule also has a third parameter Q since all symbolic execution functions are defined in continuation-passing style. Q is a continuation which represents the remaining symbolic execution. The motivation for this is to simplify the branching and joining of execution paths during symbolic execution.

First, `eval` evaluates the receiver e to a symbolic value t, generating a new symbolic state σ_2 in the process. Then, it checks if there exists a field chunk in σ_2's heap with receiver equal to t and permission greater than 0. If so, it will "return" the symbolic value δ of the chunk stored in the heap by passing it to the continuation Q along with σ_2. If there is no such chunk in the heap, it will then perform the same check, but now in the optimistic heap of σ_2.

If there exists no such field chunk in either of the heaps, then `eval` checks if σ_2 is imprecise. If it not imprecise, then the symbolic execution will fail. But if it is imprecise, then this means that it may be possible for the field access to be optimistically justified by some imprecise specification. In this case, it first checks that evaluating the field access does not contradict any known information. To do this, it suffices to check the receiver t must not be `null`. If so, a run-time check is created for $\textbf{acc}(e_t.f,\ \varepsilon)$, which can be interpreted as checking if $e.f$ has some positive permission. The function

```
eval(σ1, e.f, Q) =
  eval(σ1, e, (λ σ2, t .
  if (∃ f(r; δ, p) ∈ σ2.h .
        check(σ2.π, r = t ∧ 0 < p)) then
    Q(σ2, δ)
  else if (∃ f(r; δ, p) ∈ σ2.h? .
        check(σ2.π, r = t ∧ 0 < p)) then
    Q(σ2, δ)
  else if (σ2.isImprecise) then
    res, _ := assert(σ2.isImprecise, σ2.π, t ≠ null)
    et := translate(σ2, t)
    R' := addcheck(σ2.R, e.f, acc(et.f, ε))
    δ := fresh
    π' := pc-add(σ2.π, {t ≠ null})
    (res ∧
    Q(σ2{h? := σ2.h? ∪ f(t; δ, ε), π := π', R := R'}, δ))
  else failure())
```

Figure 3. Gradual Viper's Symbolic Evaluation of Field Access Extended to Support Fractional Permissions

will then pass to Q a new symbolic state where with this new run-time check, as well as $e.f$ added to the optimistic heap mapped to a fresh symbolic value δ.

We also plan to implement fractional permissions in the Gradual C0 frontend (e.g. C0 programs and their translation to Gradual Viper) after we prove the soundness of the current design. We believe that modifying the frontend will be relatively straightforward and sketch a high-level overview of how we intend to run-time check fractional permissions. Currently, Gradual C0 checks heap locations dynamically by maintaining a shadow heap with valid heap locations. We will modify the shadow heap to also keep track of the permission amounts to each heap location. To check separation of heap locations, Gradual C0 adds heap locations one at a time to a new shadow heap until the same heap location is detected. With fractional permissions, we can add heap locations one at a time to a new shadow heap until the permission to a location exceeds one.

4 Steps to Concurrency

Static verification of certain concurrent programs is now possible with our extension for fractional permissions, as Gradual Viper is able to statically model concurrent programs using its already existing specification features plus fractional permissions. In particular, we are able to statically verify programs with fork-join concurrency. Generally, fork-join concurrency can be reified as follows:

```
tk = fork m(x)
...
r = join tk
```

The fork statement will execute function `m` applied to argument `x` in parallel, returning a token `tk` which can later be used to join `m`, assigning to `r` the return value of `m(x)` and resuming sequential execution. Modeling fork-join concurrency can be done by tweaking verification of method calls. Normally, symbolic execution of a method call in Gradual Viper will first `consume` its precondition, which checks that the precondition is upheld and removes any heap chunks that the precondition mentions from the symbolic heaps. Then, it will `produce` its postcondition, which adds the constraints from the postcondition to the path conditions and adds any heap chunks that the postcondition specifies into the symbolic heap. With a forked method call, its precondition still needs to be consumed when it is forked, but we must delay producing its postcondition until it is finally joined.

To model this, we use *recursive abstract predicates* [7], originally meant for specifying recursive heap data structures. For example, specifying the acyclicity of a linked list can be done with a recursive predicate:

```
predicate acyclic(Node *node) =
    node == NULL ? true :
    acc(node->data) && acc(node->next)
    && acyclic(node->next);
```

Importantly, predicates must be manually unrolled by `unfold` and `fold` statements due to the difficulty of fully unrolling them statically. Each `unfold` statement replaces a predicate with a copy of its body by consuming the predicate and producing the body. The `fold` statement does the inverse by consuming the predicate body and producing the predicate. Then, delaying producing the postcondition of a function can be done by replacing the precondition of the function with a predicate that contains the body of the original precondition. Forking the method then becomes a fold on the predicate to ensure the precondition is consumed. Then, joining the resulting token is replaced with calling the method, which will `consume` the predicate and `produce` the postcondition. Figure 4 shows this translation in pseudo-code.

Using this methodology with fractional permissions (specifying them in `m`'s pre- and postcondition) then allows us to statically verify programs using fork-join concurrency which simultaneously read the same heap location. By using this, we have written benchmarks in Gradual Viper such as a concurrent binary tree traversal and a parallel lambda-term substitution [3]. Note that this translation method results in a sequential program. This is fine for static checking, but gradual verification requires dynamic checking as well. So, while our fractional permission design supports the gradual verification of fractional permissions in sequential programs and the optimistic static verification of concurrent programs,

Listing 1. Original

```
int m(x)
    requires φ
    ensures ...
{...}

int main()
{
    tk = fork m(x)
    ...
    r = join tk
}
```

Listing 2. Translated

```
predicate m_pre(x) = φ

int m(x)
    requires m_pre(x)
    ensures ...
{...}

int main() {
    fold m_pre(x)
    ...
    r = m(x)
}
```

Figure 4. Modeling Fork-Join with Predicates

more work must be done to support dynamic checking (and thus the full gradual verification) of concurrent programs. This is left to future work.

5 Conclusion

This paper discusses an extension of Gradual C0 for fractional permissions, enabling it to distinguish between read and write behavior. This work is also an important step towards gradual verification of concurrent programs. As stated in section 3, we plan to formally prove the soundness of our design and its adherence to gradual properties [11, 12], and then change Gradual C0's frontend syntax, type-checking, and embedding algorithm for run-time checks to support fractional permissions.

6 Acknowledgements

I am very grateful to Dr. Jenna DiVincenzo for her guidance and support.

References

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

[2] Johannes Bader, Jonathan Aldrich, and Éric Tanter. 2018. Gradual Program Verification. In *International Conference on Verification, Model*

Checking, and Abstract Interpretation. Springer, 25–46. https://doi.org/10.1007/978-3-319-73721-8_2

[3] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. 2005. Permission accounting in separation logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 259–270. https://doi.org/10.1145/1047659.1040327

[4] John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis. SAS 2003. Lecture Notes in Computer Science, vol 2694*. Springer, Berlin, Heidelberg, 55–72. https://doi.org/10.1007/3-540-44898-5_4

[5] Jenna DiVincenzo, Ian McCormack, Conrad Zimmerman, Hemant Gouni, Jacob Gorenburg, Jan-Paul Ramos-Dávila, Mona Zhang, Joshua Sunshine, Éric Tanter, and Jonathan Aldrich. 2025. Gradual C0: Symbolic Execution for Gradual Verification. *ACM Trans. Program. Lang. Syst.* 46, 4, Article 14 (Jan. 2025), 57 pages. https://doi.org/10.1145/3704808

[6] Peter Müller, Malte H. 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] Matthew J. Parkinson and Gavin M. Bierman. 2005. Separation logic and abstraction. In *ACM-SIGACT Symposium on Principles of Programming Languages*. https://api.semanticscholar.org/CorpusID:6280787

[8] Matthew J. Parkinson and Alexander J. Summers. 2011. The Relationship between Separation Logic and Implicit Dynamic Frames. In *Programming Languages and Systems*, Gilles Barthe (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 439–458.

[9] J.C. Reynolds. 2002. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 55–74. https://doi.org/10.1109/LICS.2002.1029817

[10] 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/10.1145/2160910.2160911

[11] Jenna Wise, Johannes Bader, Cameron Wong, Jonathan Aldrich, Éric Tanter, and Joshua Sunshine. 2020. Gradual verification of recursive heap data structures. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–28. https://doi.org/10.1145/3428296

[12] Conrad Zimmerman, Jenna DiVincenzo, and Jonathan Aldrich. 2024. Sound Gradual Verification with Symbolic Execution. *Proc. ACM Program. Lang.* 8, POPL, Article 85 (jan 2024), 30 pages. https://doi.org/10.1145/3632927