An Annotation System for Specifying Aliasing Invariants on Object Fields

Aurélien Coet
aurelien.coet@unige.ch
Computer Science Department
University of Geneva
Geneva, Switzerland

ABSTRACT

Aliasing is an essential concept in programming languages, used to represent self-referential structures and share data between components. Unfortunately, it is also a common source of software bugs that are often hard to find and fix. In response, a plethora of methods have been proposed to tame aliasing. They usually rely on uniqueness and/or immutability to establish strong safety guarantees, but are often too restrictive to write common idioms, as they generally enforce a single-writer policy. This paper suggests to relax this constraint by focusing on the specific parts of an object representation for which aliasing should be controlled, otherwise allowing unrestricted mutations of its fields.

CCS CONCEPTS

• **Theory of computation** → **Invariants**; **Program specifications**; **Program verification**; • **Software and its engineering** → **Constraints**; **Formal software verification**; *Compilers*.

KEYWORDS

Aliasing, invariants, contract-based programming

ACM Reference Format:
Aurélien Coet. 2020. An Annotation System for Specifying Aliasing Invariants on Object Fields. In *Companion Proceedings of the 4th International Conference on the Art, Science, and Engineering of Programming (<Programming'20> Companion), March 23–26, 2020, Porto, Portugal.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3397537.3398480

1 INTRODUCTION

Aliasing refers to the situation where a computer program uses two different names, a.k.a. *references*, to designate the same value in memory. Though useful, the construct is a recurrent source of pernicious bugs in programs. Numerous approaches have therefore been studied to alleviate issues associated with it [1, 9].

Rust[1] and *Pony*[2] are notorious examples of languages that successfully implement elaborate type systems [2, 5] to ensure memory safety guarantees. They are however known for having steep learning curves and to be too restrictive in their treatment of aliasing [6]. Rust, for example, enforces the single writer constraint to implement *uniqueness* [3]. Although this provides memory safety, it also makes implementing common structures like mutable graphs challenging. While the language allows the *unsafe* manipulation of references, this feature effectively transfers the burden of ensuring memory safety from the compiler to the programmer, hence weakening the guarantees that can be made about a program's correctness.

This work introduces a more permissive annotation system to express uniqueness and immutability properties [8] on aliases. Scope-based invariants can be defined on object fields, making them temporarily immutable. These invariants can then be checked either dynamically or statically to guarantee various functional properties. The approach is defined as an opt-in feature for existing languages, which can make its adoption more gradual and easier than alternatives.

2 MOTIVATION

```
1  func rmEven(nums: inout [Int], i: Int) {
2    if nums[i] % 2 == 0 {
3      nums.remove(at: i)
4    }
5  }
6
7  var numbers = Array(0 ... 10)
8
9  for idx in 0 ..< numbers.count {
10   numbers[idx] *= numbers[idx]
11   rmEven(nums: &numbers, i: idx)
12 }
```

Listing 1: Loop invariant violation (Swift)

Consider Listing 1. The function `rmEven` takes a reference to an array of integers and an index as inputs, and removes the value at the given index if it is even. The program instantiates a mutable array `numbers`, creates a range from 0 to its size, and iterates over its elements, calling `rmEven` after each one of them has been squared.

[1] https://www.rust-lang.org/
[2] https://www.ponylang.io/

<Programming'20> Companion, March 23–26, 2020, Porto, Portugal

ACM ISBN 978-1-4503-7507-8/20/03…$15.00
https://doi.org/10.1145/3397537.3398480

The execution fails during the 7^{th} iteration of the loop, because the mutation of `numbers` through its alias in `rmEven` (line 11) causes `0 ..< numbers.count` to no longer match the indices of the array. In other words, the error is due to the violation of a loop invariant.

3 PROPOSED APPROACH

A large body of work exists on the use of annotations to express invariants for the specification of functional properties in programs. Seminal work by Meyer introduced the concept of *design by contract* (DbC) [7] and later led to *behavioural interface specifications* [4]. This paper borrows concepts from these areas and extends them to the specification and verification of aliasing properties.

In Listing 1, one way to make sure that `idx` never refers to an index outside of bounds is to define the size of `numbers` as a *loop invariant*. This can be done with an explicit annotation delineating a scope in which the `numbers.count` property must remain *immutable* (Listing 2). *Only* operations that modify the locked field are forbidden in the invariant's scope, and mutating accesses to `numbers[idx]` hence remain possible in the loop.

```
1  ...
2  @invariant(numbers.count) {
3    for i in 0 ..< numbers.count {
4      numbers[idx] *= numbers[idx] // Still legal
5      rmEven(nums: &numbers, i: idx) // Illegal
6    }
7  }
```

Listing 2: Invariant annotation (Swift)

The use of invariants differs from existing approaches in two ways. Firstly, immutability is only defined for a limited scope, instead of the whole program or lifetime of some alias. Secondly, and most importantly, only the *path* to the `count` field of `numbers` is protected by the invariant, rather than the entire object.

Checking for mutating operations on a path is difficult, in particular when function calls are involved (since the called code must be analysed). In the presence of higher-order functions, determining what implementation is called even becomes statically undecidable. Two complementary methods of verifying aliasing invariants are therefore proposed. The first checks for path mutations at runtime, effectively bypassing the issue of higher-order. In the second, static analysis is made possible by annotations on functions' signatures indicating what fields they modify, in a way reminiscent of *contracts* in DbC and behavioural interface specifications. Listing 3 illustrates how the `rmEven` function is annotated to reflect that it mutates the `count` field of its input array, effectively making it illegal in the invariant's scope of Listing 2.

While annotations on functions help solve the issue of statically checking invariants, annotating large programs may quickly turn into an intractable task. A potential way of alleviating programmers from such a burden could therefore be to automatically infer annotations wherever possible, similarly to what is proposed for types in most modern, statically typed languages.

```
1  @mutates(nums.count)
2  func rmEven(nums: inout [Int], i: Int){
3    if nums[i] % 2 == 0 {
4      nums.remove(at: i)
5    }
6  }
```

Listing 3: Mutation annotation on a function (Swift)

4 CONCLUSION

This paper introduces a new way to check invariants on object fields, both dynamically and statically, with the help of annotations on functions. The proposed approach is currently being formalised in an operational semantics, and a proof-of-concept being implemented for the Swift programming language.

REFERENCES

[1] David G Clarke, John M Potter, and James Noble. 1998. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 48–64.

[2] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny Capabilities for Safe, Fast Actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE! 2015)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2824815.2824816

[3] Philipp Haller and Martin Odersky. 2010. Capabilities for Uniqueness and Borrowing. In *ECOOP 2010 – Object-Oriented Programming*. Vol. 6183. Springer Berlin Heidelberg, Berlin, Heidelberg, 354–378. https://doi.org/10.1007/978-3-642-14107-2_17

[4] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew Parkinson. 2012. Behavioral interface specification languages. *Comput. Surveys* 44, 3 (June 2012), 1–58. https://doi.org/10.1145/2187671.2187678

[5] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2017. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.* 2, POPL, Article 66 (Dec. 2017), 34 pages. https://doi.org/10.1145/3158154

[6] Amit Levy, Michael P. Andersen, Bradford Campbell, David Culler, Prabal Dutta, Branden Ghena, Philip Levis, and Pat Pannuto. 2015. Ownership is theft: experiences building an embedded OS in rust. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems - PLOS '15*. ACM Press, Monterey, California, 21–26. https://doi.org/10.1145/2818302.2818306

[7] B. Meyer. 1992. Applying 'design by contract'. *Computer* 25, 10 (Oct. 1992), 40–51. https://doi.org/10.1109/2.161279

[8] Alex Potanin, Johan Östlund, Yoav Zibin, and Michael D Ernst. 2013. Immutability. In *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. Springer, 233–269.

[9] Ayesha Sadiq, Yuan Fang Li, and Sea Ling. 2019. A survey on the use of access permission-based specifications for program verification. *Journal of Systems and Software* 159 (21 10 2019). https://doi.org/10.1016/j.jss.2019.110450