Alternate Annotation Checkers Using Fractional Permissions

Chao Sun
University of Wisconsin Milwaukee
csun@uwm.edu

Abstract

Although existing annotation checker based on fractional permissions is powerful, it causes great space and runtime overhead. To address this issue, we propose to use a multi-layered approach for checking annotations. In addition to the heavyweight permission checker, we use two lightweight checkers: a conservative checker for those obviously correct cases, and a liberal checker for those obviously wrong cases. The type system for the conservative checker is more high-level, albeit less precise. To prove its soundness, we piggy-pack its proof to that of fractional permission, which is already proven sound. We also plan to implement both checkers on Fluid, an analysis framework for Java programs, and use various benchmarks to compare the performance of both approach.

Categories and Subject Descriptors D.1.5 [*Software*]: PROGRAMMING TECHNIQUES—Object-oriented programming

General Terms languages

Keywords permissions,twelf,fluid

1. Purpose

One common safety issue in imperative programming languages is *aliasing*, which happens when a data location in memory is accessed through different symbolic names in program. With the presence of aliasing, it is difficult to reason about program behavior, because write on a memory cell through one variable may affect the read by some another variable. The write action could happen at a totally unrelated point in the program, which makes program's behavior extremely hard to reason about.

Aliasing can have serious affect on *information hiding* and *encapsulation*, which are essential elements in object-oriented programming. For instance, a private member of an object may have aliases outside the scope that refers to it, thus violating the purpose. Modern softwares are often required to offer *implementation transparency*, which means the ability to change internal implementation without affecting the rest of the system.

To resolve this issue, many researchers have suggested using annotations. Unlike program types, which are "hardcoded" in the language, and concern more about low-level semantics, annotations are more about the high-level program behaviors. They usually will not change the run-time behavior, and therefore can serve two purposes: implementors can attach their *design intent* to the program, for better understanding, and maintainers can use them to extract more semantic information about the program, for better analysis of its behavior. In general, annotations enable a component supplier to offer *contract* in which certain *demands* are described, and clients are supposed to follow these requirements, to guarantee result of execution meets the expectation.

Another reason of using annotation is its flexibility. Rather than design a new language and add all the desired features, people can deploy annotations as an *optional type system*. Existing languages can be improved in this way without their essential elements altered, as well as the compiler and run-time system.

Several annotations have been brought into literature to address the aliasing problem. Among them, most widely discussed are uniqueness and ownership types.

The notion of uniqueness comes from linear types [18], and [15] is one of the earliest work which brought linearity into object-oriented languages. To prevent aliasing on unique variable, *destructive read* is used. That is, once a unique variable is read, it is immediately set to `null`. In this way, information is transferred via *moving*, instead of *copying*, therefore guarantees every unique pointer is the only copy to the object it refer to. Although destructive read preserves the uniqueness, it also causes many inconveniences in practical usage - not only that the `null` pointer has unpleasant side-effects, but language semantics also needs to be changed to accommodate the uniqueness.

Alias burying [3] resolves the above issue by delaying the nullification as long as no alias is *read*. When one alias is read, all other aliases are set to `undefined`, and cannot be used anymore. The advantage is that a separate static program analysis can be used to replace the need of modifying the compiler. Since aliasing is a global effect, to enable intra-procedure analysis, a `borrowed` annotation is introduced in this approach to grant temporary access to a unique variable.

Another branch is ownership types [8]. This approach, instead of *forbidding* alias, attempts to *confine* [2] aliases in a certain scope. Alias confinement is especially important when coping with large-scale software system, since it allows one to reason about one module at a time, independent of the others. In the ownership approach every object is inside the scope of another one (called the owner object), and all the objects on heap form a tree structure. In fact, the idea of restricting the scope of alias is similar to adoption [9], in which the adopter of a linear value can be seen as its owner. This idea is further made explicit in an object-oriented language setting [4], where ownership is established by *nesting* a linear value into a special data group [14].

One important consequence of the above is, for an object, it guarantees every access path from *root* to it has its owner as one of the node [7]. Therefore, the approach is also called "owners-as-dominators".

In contrast, another approach is "owners-as-modifiers". The difference is the latter allows an object be referenced by any other object, as long as the latter doesn't *modify* it.

SPLASH'11 Companion, October 22–27, 2011, Portland, Oregon, USA.
ACM 978-1-4503-0940-0/11/10.

Ownership Domain [1] is a further attempt on finding a balance between safety and expressiveness. Unlike previous ownership techniques, which impose a fixed structure on objects, Ownership Domain separates aliasing policy from ownership mechanism. In specific, a class can declare multiple *domains* to represent different encapsulation levels, and *links* can be established between domains to grant one access to the other.

Unlike previous works, which tend to introduce new annotations, give them with meanings, and propose a set of rules for checking them, fractional permission [5] provides a foundation for various annotations to ground. Based on separation logic, it provides a logic to reason about the semantic meaning of most annotations. Under this infrastructure, various annotations, such as `unique`, `nonnull`, data group, ownership and effects, can be expressed.

However, while fractional permission provides a powerful tool for checking annotations, the downside is its complexity. Currently, the implementation [17] has both high runtime overhead and space overhead, and it may take a considerable amount of time to analyze a reasonable large program. This hinders its use in practice. Therefore, it is sensible for us to ask: can we implement a lightweight version of the fractional permission, which can check *most* of the annotated programs, while being much more efficient?

2. Goals

The goal of our research is not to invent some new annotations and provide meanings to them, but to make the existing annotation checking more *efficient*.

Currently, the implementation of fractional permission is on Fluid [12], which is an framework for program analysis. Due to the heavyweight nature of permission analysis, the implementation is rather complicated. For instance, to model base permission, the implementation needs lattices for both location and fraction, and has two separate maps for them. To model Java evaluation at a low-level, the transfer function needs to simulate stack operations, and therefore a stack lattice in which elements are of some other base lattices is used. One side-effect for this is that all the primitive types in Java, such as `int`, need to have a lattice representation too. Besides, along the control flow, the analysis also needs to collect various facts, like the equality (inequality) between object locations, as well as nesting situations.

Because of the large amount of information that needs to be tracked, and the operations on them (especially the join operation upon control flow merge), the analysis has high runtime and space overhead. This makes it impractical to use on reasonable large-size program.

To make annotation checking run faster, instead of applying the heavyweight checker on the input program directly, a better strategy is to use a more "conservative" checker first. The conservative checker should run much faster, albeit less precise. Instead of encoding fractional permission directly, it uses much higher-level types. The fractional permission type system, instead, serves as a foundation for the new type system to be built on. This approach gives us two benefits: first, we can have better understanding of semantics of annotations, using fractional permission, and therefore derive better type system to check them; second, we can build the soundness proof of the type system directly on that of fractional permission.

Besides the conservative checker, we also intend to use a "liberal" checker, which identifies those *obviously wrong* cases. Inside an annotated program, some methods may contain errors that are easy to detect. After failing the conservative checker, these methods are passed to permission checker, which makes the overhead even worse. Instead, by applying the liberal checker on these methods, we can reject these methods very quickly, without utilizing the

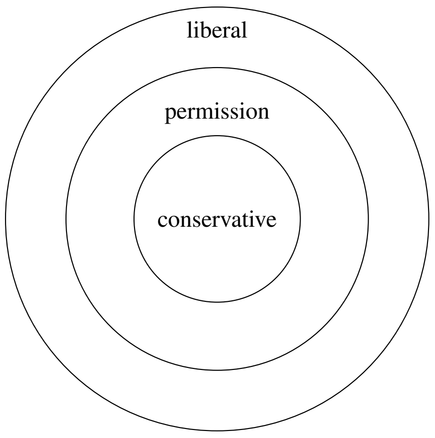

Figure 1. Multi-layered annotation checkers

```
class Node {
  @InRegion("Instance")
  @Unique Node next;
}

class List {
  @Unique Node head;

  @RegionEffect("writes␣head")
  void prepend(@Unique Node n) {
    n.next = head;
    head = n;
  }
}
```

Figure 2. Sample annotated code in Fluid

heavyweight checker. Some obviously wrong cases include storing a `borrowed`-annotated method parameter into a `unique` field, or not returning the effects passed in to the method.

A type system can not be trusted without soundness property. For the conservative checker, we intend to use machine-checked proof, based on fractional permission system [6]. The proof is written using Twelf [16], which is a implementation of LF logical framework [13] that is especially useful for proving properties of programming languages and logics. Specifically, for the conservative type system, we piggy-pack its proof on the permission system. That is, for each term in the language, we show that if it can be checked under the conservative system, then it can also be checked under the permission system, and since the latter is already proved sound, this shows the former is sound too. With this approach, we avoid the need to prove progress and preservation, and the semantics of the language is also separated from the type system.

3. Technical Approach

Our approach will be in two directions. First, we will formalize the type system and use machine-checked proof for its correctness. Second, based on the type system we will implement a prototype annotation checker, and compare the performance with the single type system approach.

The related annotations that are currently supported in Fluid include `unique`, `shared`, `borrowed`, and effects. Although we do not support ownership parameters currently in either formalization and implementation, we believe it should be a straightforward extension to the above.

Formalization Compared to the permission type system, the conservative type system is designed to be mostly flow-insensitive. Although this will make several kinds of type errors difficult or impossible to detect, it guarantees the overhead of the checker to be low. However, for effect analysis, we need to track the source of local variables and record effects on them. This requires Binding Context Analysis (BCA) [11], which is flow-sensitive. However, BCA is independent of the main type checking process, and is much faster than the permission analysis. We also intend to use various optimizations to speed up this process.

A major part of our research is the machine-checked proof for the soundness of type system. Not only will this give us strong confidence on the correctness of the system, but also it will force us to address all the possible cases which one may not notice in a natural language proof.

As a pilot study, the author has first proved a simple non-null type system [10] based on fractional permissions. The system itself is a very simple extension of a standard type system, with the following additions:

- in a class, every field is either annotated as `nonnull` or `maybenull`. Also, every field is implicitly `shared`.
- the effects for methods is implicitly `writes shared`, which, combined with the first, grant them privilege to write every field.
- the constructor is especially restricted; the body of it must be a sequence of assignments to the fields, followed by returning `this`.

The type system is built on a Java-like kernel language for reasoning about concurrent imperative language, which, alone with its semantics, is already defined in Twelf. Both the existing proof for permission type system and the conservative type system are based on the assumption of single-threaded programs, although a proof for multi-thread programs is ongoing.

The proof is done by (roughly) first transforming all the relevant environments (class map, method map, etc) to the corresponding structures under the permission system. Then, we prove that for every program in the kernel language, if it's well-typed under consistent environments, then after converting these environments, the same program can also be checked under the system of fractional permission.

The machine-checked proof for the non-null system consists of 15 files and 769 KBytes. Although it seems large, over 500K of the code is automatically generated, and contains many unused theorems. The definition of types also occupies a reasonable part of the code.

Implementation Our implementation of the type system is based on the Fluid project, which is an analysis framework implemented as plugin for Eclipse IDE. In the previous work, the fractional permission type system has been implemented [17] by William Retert on it as control-flow analysis, and some case studies have also been done both on some simple code fragments and JEdit, a reasonably large sample of annotated Java code.

Because our goal is to improve the speed of annotation checking based on the current permission analysis, we need to collect various statistics to see the result using different measures. Currently, the main issue is lacking of sample annotated code. Although JEdit has a reasonably large size, the annotations on it are mainly used for thread and lock analyses (only six fields are annotated as `unique`). In future, we need more sample code for benchmarking. The result of this not only can provide us a general idea of how much improvement can be achieved with the multi-layered approach, but also can give us feedbacks for the cases that the lightweight checkers should handle.

References

[1] Jonathan Aldrich and Craig Chambers. Ownership domains: Separating aliasing policy from mechanism. In Martin Odersky, editor, *ECOOP'04 — Object-Oriented Programming, 18th European Conference*, volume 3086 of *Lecture Notes in Computer Science*, pages 1–25, Berlin, Heidelberg, New York, 2004. Springer.

[2] Boris Bokowski and Jan Vitek. Confined types. In *OOPSLA'99 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, volume 34, pages 82–96, New York, October 1999. ACM Press.

[3] John Boyland. Alias burying: Unique variables without destructive reads. *Software Practice and Experience*, 31(6):533–553, May 2001.

[4] John Boyland and William Retert. Connecting effects and uniqueness with adoption. In *Conference Record of POPL 2005: the 32nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 283–295, New York, 2005. ACM Press.

[5] John Boyland, William Retert, and Yang Zhao. Comprehending annotations on object-oriented programs using fractional permissions. In Matthew Parkinson, editor, *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO)*, New York, 2009. ACM Press. To appear.

[6] John Tang Boyland. Semantics of fractional permissions with nesting. *ACM Trans. Program. Lang. Syst.*, 32:22:1–22:33, August 2010.

[7] David Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, Sydney, Australia, 2001.

[8] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA'98 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, volume 33, pages 48–64, New York, October 1998. ACM Press.

[9] Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation*, volume 37, pages 13–24, New York, May 2002. ACM Press.

[10] Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In *OOPSLA'03 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, volume 38, pages 302–312, New York, November 2003. ACM Press.

[11] Aaron Greenhouse. *A Programmer-Oriented Approach to Safe Concurrency*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, 2003.

[12] Aaron Greenhouse, T. J. Halloran, and William L. Scherlis. Using Eclipse to demonstrate positive static assurance of Java program concurrency design intent. In *Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, pages 99–103, October 2003.

[13] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.

[14] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In *Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation*, volume 37, pages 246–257, New York, May 2002. ACM Press.

[15] Naftaly Minsky. Towards alias-free pointers. In Pierre Cointe, editor, *ECOOP'96 — Object-Oriented Programming, 10th European Conference*, volume 1098 of *Lecture Notes in Computer Science*, pages 189–209, Berlin, Heidelberg, New York, July 1996. Springer.

[16] Frank Pfenning and Carsten Schürmann. Twelf user's guide, version 1.4. Available at `http://www.cs.cm.edu/~twelf`, 2002.

[17] William S. Retert. *Implementing Permission Analysis*. PhD thesis, University of Wisconsin–Milwaukee, Department of EE & CS, 2009.

[18] Philip Wadler. Linear types can change the world! In M. Broy and C. B. Jones, editors, *Programming Concepts and Methods*. Elsevier, North-Holland, 1990.