Compile-Time Execution Structure of Object-Oriented Programs with Practical Ownership Domain Annotations

Marwan Abi-Antoun
Carnegie Mellon University
marwan.abi-antoun@cs.cmu.edu

Abstract

Ownership domain annotations express and enforce design intent related to object encapsulation and communication directly in real object-oriented code.

First, this work will make the ownership domains type system more expressive. Second, ownership domain annotations enable obtaining, at compile time, the execution structure of an annotated program. The execution structure is sound, hierarchical and scales to large programs. It also conveys more design intent that existing compile-time approaches that do not rely on ownership annotations. Finally, tools will infer these annotations semi-automatically at compile time, once a developer provides the design intent.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Patterns

General Terms Design, Documentation, Experimentation

Keywords ownership types, ownership domains, execution structure, runtime structure, dynamic structure

1. Problem Description

To correctly modify an object-oriented program, a developer often needs to understand both the *code structure* (static hierarchies of classes) and the *execution structure* (dynamic networks of communicating objects). Several tools can generate the code structure of a program at compile-time.

Naturally, one way to obtain the execution structure is to use dynamic analyses [12, 7]. But these analyses suffer from several problems. First, runtime heap information does not convey design intent. Second, a dynamic analysis may not be repeatable, i.e., changing the inputs or exercising different use cases might produce different results. Finally, some analyses carry a significant runtime overhead [7].

OOPSLA'07, October 21–25, 2007, Montréal, Québec, Canada.
ACM 978-1-59593-865-7/07/0010.

Obtaining a sound execution structure of an object-oriented program at compile-time is more challenging due to aliasing, precision and scalability issues. The execution structure is *sound* if it does not fail to reveal relationships that may actually exist at runtime, up to a minimal set of assumptions regarding reflective code, external libraries, etc. Existing compile-time approaches use sound but heavyweight and thus unscalable analyses [11] or use unsound analyses [9]. All produce non-hierarchical object graphs.

2. Research Statement

Hypothesis #1: Flexible ownership domain annotations can express and enforce design intent related to encapsulation and communication in object-oriented programs.
Ownership domains [3] divide objects into conceptual groups called *domains*. Each object is in a single ownership domain and each object can in turn declare one or more public or private domains to hold its internal objects, thus supporting hierarchy. An ownership domain can convey design intent and represent an architectural tier, e.g., the `Model` tier in the Model-View-Controller (MVC) design pattern.

Preliminary Work. We annotated two Java programs [2] consisting of 15,000 lines of code each. While applying these annotations, we identified limitations in the expressiveness of ownership domains [2]. This work will modify the type system to address these limitations.

Expected Contribution #1: We plan to make ownership domain annotations more expressive and more flexible. The modified set of annotations will be implemented and evaluated using case studies on non-trivial programs.

Hypothesis #2: Ownership domain annotations enable a sound approximation of the system's execution structure at compile-time, the Ownership Object Graph, that is hierarchical and conveys design intent.
By grouping objects into clusters called *domains*, ownership domain annotations enable an abstraction of the structure of an application that is coarser than an object, and promote both high-level understanding and detail using hierarchy and clustering [1]. The ownership domains type system also guarantees that two object instances in two different domains cannot be aliased, so the analysis can distinguish be-

tween instances of the same class in different domains that would be merged in a class diagram. This is more precise than aliasing-unaware analyses [9] and more scalable than heavyweight alias analyses [11]. Since the annotations are specified by a developer, they can convey more design intent than arbitrary aliasing information obtained using a static analysis that does not rely on annotations [13]. Finally, unlike approaches that require annotations just to obtain a visualization [10], ownership annotations enforce useful object encapsulation properties [5, 3, 6].

The novel contribution here is obtaining an execution structure based on ownership domain annotations. Ownership annotations give important information about the program's runtime object structures, at compile time. Indeed, previous dynamic analyses have used ownership to organize a program's execution structure. But unlike dynamic analyses which show the execution structure for a given program run, a sound compile-time execution structure shows ownership relations that are invariant over all program runs.

Preliminary Work. We have formally defined the Ownership Object Graph of an annotated program [1]. We implemented a tool and evaluated it on two real 15,000-line Java programs that we previously annotated. In both cases, the Ownership Object Graph fit on one page and illustrated the design intent, e.g., JHotDraw's MVC design [1], better than existing flat object graphs. The Ownership Object Graph also gave us insights into possible design problems.

Expected Contribution #2: The Ownership Object Graph would be most useful if it were sound. Otherwise, the technique would not be an improvement over existing unsound heuristic approaches that do not require annotations. To show that the Ownership Object Graph is a faithful representation of the runtime object graph, we will produce a formal proof of soundness of the Ownership Object Graph by defining the invariants imposed on the data structures we generate and their well-formedness rules.

Hypothesis #3: Once a developer manually adds a small number of annotations, a tool can infer most of the remaining annotations semi-automatically.
The Ownership Object Graph requires ownership domain annotations, but adding these annotations manually to a large code base is a significant burden. In our experience, simple defaults can only produce between 30% and 40% of the annotations [2]. We plan to extend the earlier work on compile-time annotation inference by Aldrich et al. [4] and improve its precision and usability. In our approach, a developer indicates the design intent by providing a small number of annotations manually. Scalable algorithms and tools then infer the remaining annotations automatically.

Expected Contribution #3: We will develop a semi-automated interactive inference tool to help a developer add annotations to a code base without running the program. The tool will be evaluated by taking the programs that were previously annotated manually, removing the annotations and then using the tool to infer the annotations. We chose this methodology since applying ownership domain annotations often promotes decoupling the code, for example by programming to an interface instead of a class [2].

Hypothesis #4: The Ownership Object Graph is useful for reasoning about important runtime properties.
The execution structure is typically needed to reason about a system's runtime performance, distribution or security characteristics. Reasoning about the execution structure may also be simpler than reasoning about the program directly. We will demonstrate more concretely the benefits of the Ownership Object Graph using one of these applications.

3. Conclusion

Ownership domain annotations are worth adding to a program because they: a) express and enforce the design intent directly in code; b) enable a sound execution structure at compile time; and c) can be inferred semi-automatically. The execution structure complements the code structure and is useful for reasoning about important runtime properties.

References

[1] M. Abi-Antoun and J. Aldrich. Compile-Time Views of Execution Structure Based on Ownership. In *IWACO*, 2007.

[2] M. Abi-Antoun and J. Aldrich. Ownership Domains in the Real World. In *IWACO*, 2007.

[3] J. Aldrich and C. Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *ECOOP*, 2004.

[4] J. Aldrich, V. Kostadinov, and C. Chambers. Alias Annotations for Program Understanding. In *OOPSLA*, 2002.

[5] D. G. Clarke, J. M. Potter, and J. Noble. Ownership Types for Flexible Alias Protection. In *OOPSLA*, 1998.

[6] W. Dietl and P. Müller. Universes: Lightweight Ownership for JML. *J. Object Technology*, 4(8), 2005.

[7] C. Flanagan and S. N. Freund. Dynamic Architecture Extraction. In *Workshop on Formal Approaches to Testing and Runtime Verification*, 2006.

[8] JHotDraw. `http://www.jhotdraw.org/`, 1996.

[9] D. Jackson and A. Waingold. Lightweight Extraction of Object Models from Bytecode. *IEEE Transactions on Software Engineering*, 27(2):156–169, 2001.

[10] P. Lam and M. Rinard. A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information. In *ECOOP*, 2003.

[11] R. W. O'Callahan. *Generalized Aliasing as a Basis for Program Analysis Tools*. PhD thesis, Carnegie Mellon University, 2001.

[12] D. Rayside, L. Mendel, and D. Jackson. A Dynamic Analysis for Revealing Object Ownership and Sharing. In *Workshop on Dynamic Analysis*, pages 57–64, 2006.

[13] D. Rayside, L. Mendel, R. Seater, and D. Jackson. An Analysis and Visualization for Revealing Object Sharing. In *Eclipse Technology eXchange (ETX)*, 2005.