Aliasing, Confinement, and Ownership in Object-Oriented Programming IWACO Workshop Report

Dave Clarke[1], Sophia Drossopoulou[2], Peter Müller[3], James Noble[4], and Tobias Wrigstad[5]

[1] Katholieke Universiteit Leuven, Belgium, Dave.Clarke@cs.kuleuven.be
[2] Imperial College, London, UK, sd@doc.ic.ac.uk
[3] ETH Zurich, Switzerland, peter.mueller@inf.ethz.ch
[4] Victoria University of Wellington, New Zealand, kjx@mcs.vuw.ac.uk
[5] Purdue University, USA, wrigstad@cs.purdue.edu

Abstract The power of objects lies in the flexibility of their interconnection structure. But this flexibility comes at a cost. Because an object can be modified via any alias, object-oriented programs are hard to understand, maintain, and analyze. Aliasing makes objects depend on their environment in unpredictable ways, breaking the encapsulation necessary for reliable software components, making it difficult to reason about and optimize programs, obscuring the flow of information between objects, and introducing security problems.
Aliasing is a fundamental difficulty, but we accept its presence. Instead we seek techniques for describing, reasoning about, restricting, analyzing, and preventing the connections between objects and/or the flow of information between them. Promising approaches to these problems are based on ownership, confinement, information flow, sharing control, escape analysis, argument independence, read-only references, effects systems, and access control mechanisms.

1 Introduction

The aim of the IWACO workshop was to address the question how to manage interconnected object structures in the presence of aliasing. In, particular the following issues were covered:

- models, type and other formal systems, programming language, separation logic, mechanisms, analysis and design techniques, patterns, tools and notations for expressing object ownership, aliasing, confinement, uniqueness, and/or information flow;
- optimization techniques, analysis algorithms, libraries, applications, tools, and novel approaches exploiting object ownership, aliasing, confinement, uniqueness, and/or information flow;
- empirical studies of programs or experience reports from programming systems designed with these issues in mind;
- novel applications of aliasing management techniques such as ownership types, ownership domains, confined types, region types, and uniqueness.

History. IWACO 2008 was the fourth ECOOP workshop focusing on aliasing. The previous workshops were IWACO 2007 [11], IWACO 2003 [10], and the Intercontinental Workshop on Aliasing in Object-Oriented Systems (IWAOOS) in 1999. The issues addressed in this workshop were first brought into focus with the Geneva Convention on the Treatment of Object Aliasing [19].

Program. The workshop provided a forum for two invited talks, seven presentations of submitted papers (including three position papers), four tool demos, and ample discussions. It was organized in four sessions, which we summarize in the following four sections.

2 Session 1: Invited Talk

The presentation of our first invited speaker, Jonathan Aldrich from Carnegie Mellon University, was entitled "Define, don't Confine". It identified three major challenges that need to be addressed in order to bring alias control into practice.

First, the community has to identify *applications* where the benefit of making program structure explicit has a significant and immediate benefit. Two promising candidates are concurrency and verification. For both applications, it will be necessary to make the annotations lightweight (possibly through inference) and to improve the expressiveness in order to cover common program styles and idioms.

Second, the community has to increase the *adoptability* of alias control by reducing the annotation burden through inference and by providing support for existing languages and programs.

Third, the community has to increase the *applicability* of alias control to be able to handle more programs. Aldrich's position here is that the community has focused too much on *restricting* aliasing rather than *documenting* the aliasing in programs and using this information for reasoning. He showed various examples that support his position.

To the great satisfaction of the IWACO crowd, Aldrich's final slide was entitled "The Future of Alias Control is Bright"—provided that the three challenges will be addressed successfully.

3 Session 2: Ownership

Two of the open challenges for type systems expressing ownership and related properties are (1) determining *what* the type system should express and *how* it should express it, and (2) type inference. The four talks in the second session addressed one or both of these issues.

Expressiveness A lot of research has gone into determining *how* to best express information such as ownership and immutability. As suggested in Aldrich's keynote, there needs to be shift more towards addressing *what* needs to be expressed, as directly extending existing type systems tends to intermingle the policy with the mechanism of the type system. Nonetheless, work on improving the expressiveness of individual type systems still produces useful technical machinery.

Yu David Liu presented *Pedigree Types*, joint work with Scott Smith. Pedigree types aim to obtain the benefits of owner parameterization, as in in Ownership Types [13], with the simple syntactic convenience of Universe Types [15].

Parameters are never explicitly stated on classes, and instead type inference is used to fill in types and class parameters omitted by the programmer. Ownership is described by adapting metaphors from human genealogy, which could aid programmers in understanding and expressing the ownership structure in their programs. Owners take the general form $\mathbf{parent}^a.\mathbf{child}^b$, describing a traversal up the ownership tree, and then down again, under the constraints that $a \geq 0$ and $b \in \{0, 1\}$. Existing owners can be described in a natural manner $\mathbf{rep} = \mathbf{child}$, $\mathbf{self} = \mathbf{parent}^0.\mathbf{child}^0$, $\mathbf{sibling} = \mathbf{peer} = \mathbf{owner} = \mathbf{parent}.\mathbf{child}$, along with new ones, such as **parent**, $\mathbf{grandparent} = \mathbf{parent}^2$, and $\mathbf{uncle} = \mathbf{parent}^2.\mathbf{child}^1$. In principle, the types can also express owners not available and expressible in the other systems, such as whenever the child's index is greater than 1 an encapsulation violation occurs. Classes are (implicitly) parameterized by such indices, allowing classes to be used in different places with different pedigrees within an ownership tree. A natural notion of pedigree subsumption also exists, permitting the relationship $\mathbf{self} \leq \mathbf{sibling}$, $\mathbf{parent} \leq \mathbf{uncle} = \mathbf{parent}^2.\mathbf{child}^1$, and, more generally, $\mathbf{parent}^a.\mathbf{child}^b \leq \mathbf{parent}^{a+1}.\mathbf{child}^{b+1}$. The type system presented in the paper is sound and decidable. It can express deep ownership and has a natural runtime representation. Various extensions to the system are also described in the paper, including *opting-out*.

Alex Potanin presented the position paper *Towards Unifying Immutability and Ownership*, describing joint work with Paley Li, James Noble, and Lindsay Groves. The paper called for the unification of immutability and ownership in order to improve the expressiveness of each notion. The goal is not to merely put the two notions together in the same language, but to really unify them. The paper emphasized the need to avoid *observational exposure* [6], which requires that mutations to an object are not observed by other objects—this is essentially the difference between immutability and read-only. The paper presented three possible ways of unifying the two mechanisms, leveraging the Generic Ownership approach [29]. The three approaches were:

- direct combination of generic ownership and generic immutability—have separate parameter spaces representing ownership information and immutability information;
- generic immutability and ownership—combine the 'type' hierarchies representing ownership and immutability into one, thereby requiring only a single parameter space to represent them; and
- generic access rights for immutability and ownership—define, more or less, a language of access rights, along the lines of *Capabilities for Sharing* [7].

Some debate arose suggesting that a fourth possibility, namely, annotating the owners with write/read-only/immutable access modes, as in Joe$_3$ [26], though this needed external uniqueness [12] to work.

Nick Cameron presented the position paper *Variant Ownership with Existential Types*, joint work with Sophia Drossopoulou. The presentation described various advantages and possibilities offered by existential types. The type system, called Jo∃, outlined ideas from Java's wildcards in the context of Generic Ownership, to gain expressiveness yet remain compatible with Java. Java's wildcards soften the subtyping relation by allowing variance in a generic type's parameters. The main research issue with this work is whether adding existential types opens the door too far. That is, if existential types are used to forget ownership information, will the constraints imposed by ownership and the benefits thereby gained be lost? It was conjectured that type (ownership) bounds could be used to retain the information required to enforce such constraints, though further research is required to determine whether this is the case. An additional open question is that of decidability of the type system.

Type Inference. Developing type inference systems for type systems expressing ownership is crucial for their adoption, as they are required to add annotations to library code and to reduce the volume of annotations in programmers' code. Two papers described various aspects of the inference process.

Yu David Liu's work reduces type inference to the problem of finding suitable parent and child indices; thus, type inference can be expressed as a constraint problem over integers. With previous attempts at type inference in parameterized ownership type systems, the number of parameters can grow in an unbounded manner. This was handled by unifying different possible parameters whenever a recursive occurrence of a class was encountered.

Ana Milanova's presentation of *Static Inference of Universe Types* described an algorithm for inferring Universe Types for Java programs. This work extends her past research on inferring Ownership Types for Java programs [23]. The algorithm was based on a points-to analysis and performed the following steps: construct static object graph; compute dominance boundary of each object; assign types to object graph edges; and assign types to fields and variables. The approach aims to produce a deep tree, but whenever a write upwards in the tree occurs, it forces a shallower ownership structure. The main challenge was that there were many possible type assignments, and no precise notion of principal assignment. Promising preliminary results were given.

An interesting open question is whether the two approaches can be combined. That is, can the constraint-based approach be applied to Universe Types?

4 Session 3: Concurrency and Ownership Demos

Concurrency. Nicholas Matsakis presented joint work with Thomas Gross. They describe a flow sensitive type and effects system that requires methods to declare the partitions of the heap that are read or written, resp. Effect agreements can be used to limit the conditions in which a method can be called. With this system, multi-threaded programs follow safe conventions that guarantee the program is free of data races.

A partition is a compile-time abstraction that identifies a distinct set of locations (object-field pairs) in the heap. Partitions are similar to data groups [21], but have scope, which can be exploited to achieve, for example, thread-local state. Methods are annotated with five kinds of effects that they have on partitions: read, write, atomic read, atomic write, and intersection. Atomic read/write indicates that the partition was accessed from within a block guaranteed to execute atomically. An intersection effect records that two partitions were made to intersect: this needs to be made explicit and trackable by analysis because when two partitions intersect, data that is added to one must also be considered added to the other.

In addition to being annotated with effects, a method may also be annotated with a contract, known as an effect agreement, that constrains what can happen before the method is called or after it returns. Effect agreements are always in the negative, they describe what must not have occurred prior to invocation (pre agreements), and what must not happen after the method has returned (par agreements). A pre agreement is generally used to require that certain partitions have not been intersected and are thus known to be disjoint. A par agreement is used to indicate that the method has started a new thread and that certain effects should not happen in parallel with that thread's execution.

The discussion of this presentation focused on the relation with Dave Cunningham's work (to be presented at FTfJP the next day) as well as on some possible variants of the proposed annotations, as e.g., in Java wildcards.

John Boyland argued that Java's volatile fields are difficult to reason about in a strictly linear fashion as found in concurrent separation logic [8] or fractional permissions [5]. In these approaches, accesses to volatile fields can be modeled using atomic blocks and auxiliary state, which Boyland finds unsatisfying because such a description is rather low-level. Instead, volatile fields are more easily handled by using non-linear concepts such as immutability and ownership [9], where they can be treated as loop holes, that is, accesses to volatile fields are not checked by the system. Boyland argued that a combination of linear and nonlinear reasoning is highly desirable. He encourages research in how to formalize ownership as a nonlinear subsystem in a mostly linear logic.

Ownership Demos. In the demo section of this session, Alex Potanin demonstrated the type checkers for Ownership Generic Java (OGJ) [29] and Immutable Generic Java (IGJ) [31]. Both systems build on Java generics to check the additional properties. Peter Müller presented some of the ETH tools for Universe Types, namely the type checkers for Generic Universe Types (GUT) [14] and Universe Types with Transfer (UTT) [24]. Both Universe checkers are implemented in the compiler for the Java Modeling Language (JML) [20].

5 Session 4: Verification

The final session began with a second invited talk from Dino Distefano, describing his jStar system, based on a paper (with Matthew Parkinson) that he will present at OOPSLA later this year [17]. jStar's key contribution is that it is based on separation logic [28], rather than ownership, applying techniques from earlier separation logic based checkers [2,16] to object-oriented programs.

Being based on separation logic from the outset gives jStar a number of immediate advantages over ownership-based approaches such as JML [20] and Spec♯ [22]. First, jStar does not impose any restrictions on the topology or use of pointers in object-oriented programs: no "owners as dominators", "owners as modifiers", or "owners cover invariants" discipline is required in program design. Second, because of this lack of an ownership discipline, programmers do not have to annotate their programs to describe how particular classes use that disciple. Third, the only annotations (method pre- and post- conditions) that are required are much briefer than in ownership based systems, because a single annotation language (separation logic) covers both the propositional content of assertions and the framing required to deal with heap storage and delineate potential aliasing. Fourth, using only stand-alone predicates and eschewing class invariants means that many of the complexities of whether and when a class is in a "valid state" can be replaced by instantaneously asserting particular predicates [27]. On the other hand, of course, these advantages come at cost: principally, that programmers must write assertions in separation logics, rather than traditional computation logics.

The other advance embodied in jStar is the use of abductive reasoning for abstract reasoning, particularly, it seems, regarding heap topologies. jStar includes a set of inference rules that embody abstraction functions, taking lower level heap states (e.g., sequences of cons nodes) up to more abstract data structure (e.g., a list spine and its contents). This means that—compared with other systems, in some sense jStar performs something similar to "ownership inference" as well as program proving based on the inferred heap properties. This inference is not general purpose (as inference must for an ownership type system) but context specific: different sets of abstraction rules are required for significantly different implementations of each abstraction. For example, one rule set it seems can handle all kinds of singly-linked lists, but a doubly-linked list, or an array-list would require a different set of rules. The ability to customize jStar to handle different abstractions is clearly very powerful, and enables jStar to verify programs without any annotations other than pre and post conditions. These abstraction rules may turn out to be brittle in practice, or to need customizing to suit each system being verified—more experimentation is clearly needed here, but the demonstrated system was very promising!

At least for small examples, however, jStar provides a convincing argument for the benefits of this approach. jStar provides "full automatic" verification of a range of programs, even when incorporating examples that ownership-based systems find very difficult to model, such as the Observer pattern, structure sharing, and ownership transfer. As demonstrated, the performance of jStar doing full verification did not seem much slower than a Java compiler running on the same examples: raising the question of why bother with complex "intermediate" systems such as ownership (or even language-level types) if a program prover can verify programs without these annotations? On the other hand, it seems as if ownership systems can avoid the need for these inference rules, because programs' abstractions are already structured via ownership.

The second presentation in this session, by Christian Haack and Clément Hurlin, also used separation logic. Christian and Clément presented a series of specifications for Iterators of various different kinds. This separation logic uses a form of linear implication to represent state transitions (as well as heap separation) and includes Boyland-style fractional permissions [18]. These features enable the system easily to encompass typestate-style modeling (e.g., an iterator is ready for reading; has been accessed; or is at the end of the traversal) and to distinguish between read-write, read-only, and immutable accesses to objects.

The key contribution of this work seems to be that the Iterator specifications are parameterized. The final declaration of an Iterator interface is:

```
interface Iterator
 /*@<perm p, boolean isdeep, Collection<isdeep> iteratee>@*/
```

with three parameters `p`, `isdeep`, and `iteratee`. (In this system, specifications are given in extended comment syntax.) The `iteratee` parameter is the most straightforward to explain; it is the collection to be iterated over. The `p` parameter is a fractional permission (thus `perm`) controlling access to the collection: set to 1, the Iterator has exclusive (and thus read-write) access to the collection, set to less than one, the Iterator has shared (read-only) access to the collection. Finally the third parameter `isdeep` captures an ownership relation between the container and its elements: a "deep" collection owns its elements while a "shallow" collection does not. (Note that distinction is similar to that between "full" and "flexible" alias protection [25], although the alias protection schemes controlled references, while the these separation-based schemes control only the permission to read or write references.) Then, a single specification for an iterator protocol can be configured in a number of different ways: an iterator over a collection of immutable elements; as a set of concurrent read-only iterators; or a shallow iterator over a mutable collection.

This presentation again demonstrated the utility of separation logic for describing complex and flexible structures, especially where structure sharing is involved, and the parameterization mechanism clearly makes specifications more concise, especially where families of related specifications are concerned. A particularly interesting feature of this work was the "`isdeep`" ownership parameter: it is not clear whether this is an accident of the particular specification examples chosen, or illustrates some more essential role for ownership even in systems where the underlying representation is separation logic.

The workshop was bookended with a tool demo by Jonathan Aldrich, who also opened the workshop. He demonstrated the Plural tool, work carried out with his student Kevin Bierhoff [4]. Plural is a practical typestate checker for object-oriented programs, a successor to Rob DeLine and Manuel Fähndrich's Fugue [30] in that both systems model abstractions of objects' state and check that methods are only called on objects in permissible states. The key difference between Plural and Fugue is that Plural's analyses are based on permissions (similar to Boyland's capabilities for sharing [7] and fractional permissions [5]). In contrast to Fugue, where an object could only change typestate while it was unique, Plural's permissions mean that typestate analysis is feasible in the presence of aliasing [3,1].

Plural is implemented as an Eclipse plugin, and can calculate permissions across all references in entire programs via a flow-sensitive analysis. Then, methods specifications in terms of permissions and typestates can be checked against the actual behavior of objects' client code. For example, a file `close` method has specification such as:

```
class File { ...
 @Full{requires = "open", ensures = "close"}
 public void close{};
}
```

The key contribution of this work is that its access permissions combine both aliasing and typestate information. The annotation on the `close()` method states both that a "`@Full`" permission is required—this reference may read and write, other references may read (OIRW$\bar{\text{W}}$ [7])—and that the method must be called in the "`open`" typestate and changes the object to the "`closed`" typestate. (Note that Java annotations are used to encode specifications, rather than extended comments.)

6 Future

It appears that the community working on aliasing and ownership has reached critical mass, if the number of submissions, participants, and presentations are any indication. Consequently, we plan to repeat the workshop in conjunction with ECOOP 2009.

A Participants

IWACO gathered 28 participants from 8 different countries.

Suad Alagic	University of Southern Maine (USA)
Jonathan Aldrich	Carnegie Mellon University (USA)
Anindya Banerjee	Kansas State University (USA)
Frédéric Besson	IRISA/INRIA (France)
John Boyland	University of Wisconsin-Milwaukee (USA)
Nicholas Cameron	Imperial College (UK)
Dave Clarke	Katholieke Universiteit Leuven (Belgium)
David Cunningham	Imperial College (UK)
Dino Distefano	University of Cambridge (UK)
Sophia Drossopoulou	Imperial College (UK)
Patrick Eugster	Purdue University (USA)
Adrian Fiech	Memorial University (Canada)
Christian Haack	Radboud University Nijmegen (The Netherlands)
Clément Hurlin	INRIA (France)
Yu David Liu	The Johns Hopkins University (USA)
Nicholas Matsakis	ETH Zürich (Switzerland)
Ana Milanova	Rensselaer Polytechnic Institute (USA)
Peter Müller	Microsoft Research (USA)
James Noble	Victoria University of Wellington (New Zealand)
Johan Östlund	Purdue University (USA)
Alex Potanin	Victoria University of Wellington (New Zealand)
Jan Smans	Katholieke Universiteit Leuven (Belgium)
Rok Strnisa	University of Cambridge (UK)
Alex Summers	Imperial College (UK)
Tiphaine Turpin	IRISA/INRIA (France)
Jan Vitek	Purdue University (USA)
Stefan Wehr	University of Freiburg (Germany)
Tobias Wrigstad	Purdue University (USA)

B Program Committee

Kevin Bierhoff	Carnegie Mellon University (USA)
John Boyland	University of Wisconsin-Milwaukee (USA)
Werner Dietl	ETH Zurich (Switzerland)
Manuel Fähndrich	Microsoft Research Redmond (USA)
Jeff Foster	University of Maryland, College Park (USA)
Peter Müller (chair)	Microsoft Research Redmond (USA)
David Naumann	Stevens Institute of Technology (USA)
Matthew Parkinson	University of Cambridge (UK)
Arnd Poetzsch-Heffter	University of Kaiserslautern (Germany)
Mooly Sagiv	Tel-Aviv University (Isreal)
Tobias Wrigstad	Purdue University (USA)

C Organizers

Dave Clarke	Katholieke Universiteit Leuven (Belgium)
Sophia Drossopoulou	Imperial College (UK)
James Noble	Victoria University of Wellington (New Zealand)
Tobias Wrigstad	Purdue University (USA)

References

1. N. Beckman, K. Bierhoff, and J. Aldrich. Verifying correct usage of atomic blocks and typestate. In G. Kiczales, editor, *Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, ACM SIGPLAN Notices. ACM Press, 2008. To appear.
2. J. Berdine, C. Calcagno, and P.W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Formal Methods for Components and Objecs (FMCO)*, volume 3780 of *LNCS*, pages 115–137. Springer, 2005.
3. K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, ACM SIGPLAN Notices. ACM Press, 2007.
4. K. Bierhoff and J. Aldrich. PLURAL: Checking protocol compliance under aliasing. In *Demonstration in ICSE Companion*, pages 971–972, 2008.
5. J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis (SAS)*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.
6. J. Boyland. Why we should not add readonly to java (yet). *Journal of Object Technology*, 5(5):5–29, 2006.
7. J. Boyland, J. Noble, and W. Retert. Capabilities for Sharing: A Generalization of Uniqueness and Read-Only. In J. Lindskov Knudsen, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 2072 of *LNCS*. Springer, 2001.
8. S. Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3):227–270, 2007.
9. D. Clarke. *Object Ownership and Containment.* PhD thesis, University of New South Wales, 2001.

10. D. Clarke, S. Drossopoulou, and J. Noble. Aliasing, confinement, and ownership in object-oriented programming. In *Object-Oriented Technology. ECOOP Workshop Reader*, volume 3013 of *LNCS*, pages 197–207. Springer, 2004.
11. D. Clarke, S. Drossopoulou, J. Noble, and T. Wrigstad. Aliasing, confinement, and ownership in object-oriented programming. In M. Cebulla, editor, *Object-Oriented Technology. ECOOP Workshop Reader*, volume 4906 of *LNCS*, pages 40–49. Springer, 2007.
12. D. Clarke and T. Wrigstad. External uniqueness is unique enough. In L. Cardelli, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 2743 of *LNCS*, pages 176–200. Springer, 2003.
13. D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, volume 33(10) of *ACM SIGPLAN Notices*, pages 48–64. ACM Press, 1998.
14. W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In E. Ernst, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 4609 of *LNCS*, pages 28–53. Springer, 2007.
15. W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, 2005.
16. D. Distefano, P.W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3920 of *LNCS*, pages 238–252. Springer, 2006.
17. D. Distefano and M. J. Parkinson. jStar: Towards practical verification for Java. In G. Kiczales, editor, *Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, ACM SIGPLAN Notices. ACM Press, 2008. To appear.
18. C. Haack and C. Hurlin. Separation logic contracts for a Java-like language with fork/join. In *Algebraic Methodology and Software Technology (AMAST)*, volume 5140 of *LNCS*, pages 199–215. Springer, 2008.
19. J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. The Geneva Convention on the treatment of object aliasing. *OOPS Messenger*, 3(2):11–16, 1992.
20. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, and Daniel M. Zimmerman. JML reference manual. Department of Computer Science, Iowa State University. Available from www.jmlspecs.org, 2008.
21. K. R. M. Leino. Data groups: Specifying the modification of extended state. In *Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, volume 33(10) of *ACM SIGPLAN Notices*, pages 144–153, 1998.
22. K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *LNCS*, pages 491–516. Springer, 2004.
23. Y. Liu and A. Milanova. Ownership and immutability inference for uml-based object access control. In *International Conference on Software Engineering (ICSE)*, pages 323–332. IEEE Computer Society, 2007.
24. P. Müller and A. Rudich. Ownership transfer in Universe Types. In *Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, pages 461–478. ACM Press, 2007.
25. J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In E. Jul, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 1445 of *LNCS*, pages 158–185. Springer, 1998.
26. J. Östlund, T. Wrigstad, D. Clarke, and B. Åkerblom. Ownership, uniqueness, and immutability. In R. Paige and B. Meyer, editors, *TOOLS Europe*, volume 11 of *Lecture Notes in Business Information Processing*, pages 178–197. Springer, 2008.

27. M. J. Parkinson. Class invariants: the end of the road. Presented at IWACO, 2007.
28. M. J. Parkinson and G. Bierman. Separation logic, abstraction, and inheritance. In *Principles of Programming Languages (POPL)*, pages 75–86. ACM Press, 2005.
29. A. Potanin, J. Noble, D. Clarke, and R. Biddle. Generic ownership for generic java. In W. Cook, editor, *Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, volume 41(10) of *ACM SIGPLAN Notices*, pages 311–324. ACM Press, 2006.
30. DeLine R and M. Fähndrich. Typestates for objects. In *European Conference on Object-Oriented Programming (ECOOP)*, LNCS, pages 465–490. Springer, 2004.
31. Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kieżun, and M. D. Ernst. Object and reference immutability using java generics. In *European software engineering conference and foundations of software engineering (ESEC-FSE)*, pages 75–84. ACM Press, 2007.