

A Dynamic Analysis for Revealing Object Ownership and Sharing

Derek Rayside, Lucy Mendel, and Daniel Jackson
MIT Computer Science and Artificial Intelligence Laboratory
{drayside, lmendel, dnj}@mit.edu

ABSTRACT

We present a dynamic analysis for inferring object ownership and sharing, defined in terms of the *write control graph*. We render the results in an interactive hierarchical matrix visualizer.

The purpose of the analysis and visualization is to reveal object ownership and sharing relations in the program, to facilitate program understanding and modification tasks.

Categories and Subject Descriptors

D.2.2 [Software Engineering]: Design Tools and Techniques—*object-oriented design methods, modules and interfaces*

D.2.3 [Software Engineering]: Coding Tools and Techniques—*object-oriented programming*

D.2.5 [Software Engineering]: Testing and Debugging—*debugging aids, diagnostics*

General Terms

design, documentation, languages

Keywords

dynamic analysis, ownership inference, design extraction, software visualization

1 INTRODUCTION

Sharing mutable data (via aliasing) is a powerful programming technique. For example, the model-view-controller design pattern [14] captures the essential structure of many graphical user interfaces: many controllers and views share one model object.

To facilitate sharing, object-oriented programming languages permit the programmer to selectively break encapsulation boundaries. Visibility keywords such as `private` suggest that some data should be encapsulated, but do not prevent public methods from returning aliases to that (supposedly) internal data.

However, sharing data makes programs harder to understand and reason about, because, unlike encapsulated data, shared data cannot be reasoned about in a modular fashion.

Encapsulating mutable data facilitates modular reasoning about object invariants. For example, consider a linked list implementation with a sentinel at the head and the invariant that the `next` field of the elements forms a cycle. If we know that elements are only manipulated by the list that owns them, then we need only examine the code of `LinkedList` and `LinkedListElement` in order to verify the invariant. The more data is encapsulated, the easier it is to reason about the program.

Previous work has developed various type-theoretic notions of object ownership to enable the programmer to specify and enforce encapsulation [2, 4, 7, 12, 18, 19, 27, 28]. We borrow the notion of ownership from this work, but consider it from an analytical rather than a type-theoretic perspective. Instead of having the programmer specify the data that is encapsulated (i.e., not shared), our tool shows the programmer which data is shared (i.e., not encapsulated). The primary contributions of this work are:

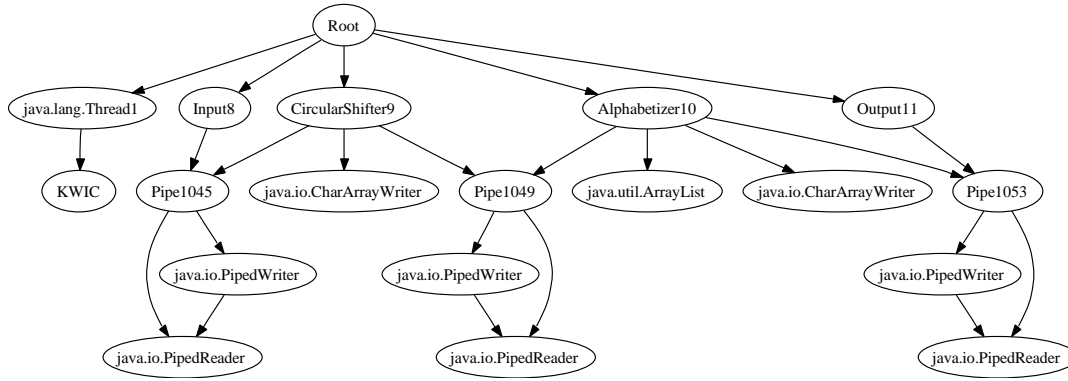
- **An analytic approach to characterizing sharing and encapsulation.** Most ownership type systems have the programmer annotate the objects that are encapsulated. Our system, on the other hand, visualizes the objects that are shared. This approach has four key benefits:
 - It focuses the programmer’s effort on what should be the exceptional case (sharing), rather than what should be the normal case (encapsulation).
 - It characterizes the sharing that exists.
 - It encourages the programmer to reduce unnecessary sharing, eliminate erroneous sharing, and document essential sharing.
 - It requires less up-front effort from the programmer, because the program does not need to be annotated with ownership type declarations.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WODA’06, May 23, 2006, Shanghai, China.

Copyright 2004 ACM 1-58113-820-2/04/0007 ... \$5.00.

Figure 1 Write control graph for KWIC pipe-and-filter style



- **A lightweight ownership inference algorithm.** Most previous work on ownership inference (e.g., [1, 3, 8, 16, 25, 35]) has focused on inferring annotations that will typecheck in some type system. We just infer the ownership structure, without inferring local annotations, which is both faster and easier. Recent concurrent work by Mitchell [24] also follows this approach.

2 KWIC PIPE-AND-FILTER EXAMPLE

We illustrate our inference and visualization with a pipe-and-filter style implementation of Parnas’s *Key Words In Context* (KWIC) program [30]. A KWIC program reads a text file, produces circular shifts of the lines, alphabetizes these shifts, and prints out the results. This pipe-and-filter style implementation of KWIC is used by Scerbakov [34] in his graduate course on software architecture. The students in the course study four different KWIC implementations: flowchart, object-oriented, event-driven, and pipe-and-filter. (The first two styles are the ones Parnas compares in [30].)

The pipe-and-filter implementation is an interesting illustration of our analysis because each of the filters is a separate thread. (The filters are Input, CircularShifter, Alphabetizer, and Output.) Multi-threaded programs such as this are easier to analyze dynamically than statically. This particular KWIC also requires an analysis that can distinguish different objects of the same type. Our previous static analysis [32] did not handle either of these features well.

Write control graph. Figure 1 shows the *write control graph* for KWIC-PF produced by our analysis. Each node in the graph represents an object in an execution of the program; the edges are write control edges; numbers on node names are automatically assigned identifiers. Nodes with no successors have been elided for presentation; write control graphs often have many nodes with no successors.

Ownership matrix. Figure 2 shows the ownership matrix for KWIC-PF. The hierarchy down the side and across the top is the immediate dominator tree of the write control graph pictured in figure 1. For example, Root is the immediate dominator of Thread1 and Input and Pipe1045 and others; Thread1 is the immediate dominator of KWIC (the program entry point).

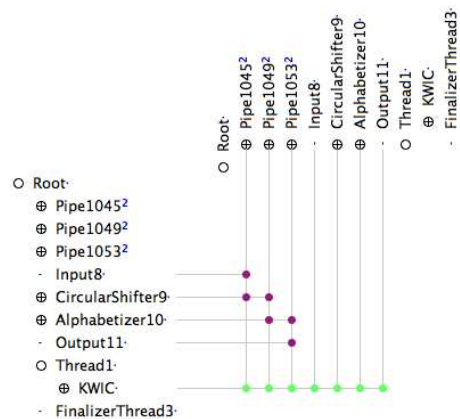
In our visualizer, like other hierarchical matrix visualizers (e.g., [33, 36]), a dot represents some sort of *use* (in Parnas’s terminology [31]). In our visualizer, a dot represents write control edge or a read control edge or a creation edge: it’s essentially an adjacency matrix.

In many visualizers a use is represented as an arrow between two boxes (e.g., [20, 26]). Although hierarchical matrices are at first less intuitive than box-and-arrow diagrams, recent work suggests that they may scale better [33, 36].

We elide uses/edges that align with the ownership hierarchy: it is expected that an owner interacts with the objects it owns. The dots in figure 2 are the uses that cross the ownership hierarchy. This focuses the programmer’s attention on unexpected cases.

In figure 2 we see that Pipe1045 connects Input to the CircularShifter, which is in turn connected to the Alphabetizer via Pipe1049, and finally Pipe1053 leads to the Output. All of the magenta dots in this ‘staircase’ shape indicate a read+write. The row of seven green dots from KWIC indicates that KWIC constructs all of the other objects.

Figure 2 Ownership matrix for KWIC pipe-and-filter style



Sharing. Objects with multiple incoming edges in the write control graph are shared. The graph in figure 1 has two kinds of shared nodes: those in ‘triangle DAGs’ (the PipedReaders at the bottom of the diagram) and those in ‘diamond DAGs’ (Pipe1045, Pipe1049, and Pipe1053).

We measure the degree that an object is shared by the *shortest distance to its immediate dominator*. Objects that have more interesting sharing are further from their immediate dominator. For example, Pipe1045, Pipe1049, and Pipe1053, all have a distance of two from their immediate dominator (Root). This two is indicated with a superscript next to the node name in the visualizer (figure 2).

An object that is not shared will have a distance of one to its immediate dominator: e.g., the `ArrayList` owned by the `Alphabetizer`. Objects that are shared in a triangle DAG configuration, such as the `PipedReader` objects in figure 1, will also have a distance of one to their immediate dominator (i.e., owner), even though they are shared. However, such sharing is less interesting because the owner (e.g., `Pipe1045`) has direct knowledge of the owned object (e.g., `PipedWriter`).

3 OBJECT OWNERSHIP

The intuitive notion of object ownership is that object x owns subobject y if y is part of the abstract state of x . This notion has a fairly obvious interpretation in the realm of abstract data types: e.g., a list owns its links, and probably does not own the objects stored in those links.

However, ownership is less clear for objects that are not part of an ADT, and for tricky cases like iterators, that require privileged access to ADT internals. An ownership inference for design visualization still needs to try to determine a reasonable owner for these objects: it's not useful to say that these objects are shared globally, because then the programmer just sees a mass of globally shared objects.

Our challenge is to devise a definition of ownership that selects a reasonable owner for every object. We define object ownership through the concept of *control*.

[Defn 1] Ownership. Object x owns subobject y if x is an immediate dominator of y in the program's *write control graph*.

The basic idea is: when a field write happens, record some state of the program execution; after the execution, combine all of these observations and compute a hierarchy. The state that we are going to record is the list of objects in *control* at the time of the field write, and we are going to combine these lists to form a *write control graph*.

[Defn 2] Control. Every frame on the call stack has a corresponding object that controls it (usually the receiver). Objects that are being constructed (i.e., they are the receiver of a constructor on the call stack), are distinguished.

We can write down the list of these objects like so: $\langle \text{root}, o_1, o_2, \dots, \hat{o}_c, \dots, o_t \rangle$, where o_t is the object in control of the top of the call stack, \hat{o}_c is an object being constructed, and `root` is a distinguished object that precedes all others.

Static methods do not have a receiver. When a static method is invoked, the object that was previously in control is still in control. In the case of the program entry point `main`, the object representing the current thread is in control (the distinguished `root` object precedes this thread object).

When a field is written to, we record the list of objects in control. If no objects are currently being constructed, we record the entire list $\langle \text{root}, \dots, o_t \rangle$. If there is currently at least one object \hat{o}_c in control, we record $\langle \hat{o}_c, \dots, o_t \rangle$ such that `'...'` contains no other object under construction.

The write control graph is formed by combining all of the recorded control chains.

The definitions of dominator and immediate dominator are traditional:

[Defn 3] Dominator. In a directed graph with a distinguished root node, node x dominates node y if every path from the root to y passes through x . Each node has a set of dominators that includes itself.

[Defn 4] Immediate Dominator. Node x is the immediate dominator of node y if the dominator set of y comprises exactly the dominator set of x plus y itself: i.e., $\text{dom}(y) = \text{dom}(x) \cup \{y\}$. Every node (in a directed graph with a distinguished root element) has exactly one immediate dominator, and so the immediate dominator relation forms a tree.

Discussion. This inference has the following qualities:

- *Referring to an object does not make an ownership claim.* Simply storing an object in a field (or method parameter or local variable), does not make an ownership claim, so an ADT does not necessarily make an ownership claim on the objects stored in it.
- *Calling a method that causes mutation does make an ownership claim.* Consider a method `LinkedList.add()` that doesn't write to any fields of the `LinkedList`: it just writes to the `next` and `data` fields of `Link`. Suppose object `client` calls `add()`: we will record the control chain $\langle \dots, \text{client}, \text{list}, \text{link} \rangle$, which indicates that `client` has an ownership claim on the `list` object, and the `list` object has a claim on the `link` object.

It is likely that there is some `link` object in the middle of the list that is never directly stored into one of the `list`'s fields, but only into the fields of other links. Our analysis still determines that such a `link` is owned by the `list`, despite never being directly referenced by the `list`.

Similarly, if the `client` just creates the `list` as a temporary object and never stores it into a field, we still capture that `client` owns the `list`; even temporary objects need owners.

- *Creating an object does not make an ownership claim.* Consider the control chain $\langle \text{root}, \dots, o_k, \hat{o}_c, \dots, o_t \rangle$ when a field of o_t is written. o_k does not make an ownership claim to \hat{o}_c because we only record the suffix-chain $\langle \hat{o}_c, \dots, o_t \rangle$. This approach allows us to capture creational design patterns such as factory method [14] in an intuitive way. For example, iterators [14] are often created by a factory method of a collection. Without the special treatment of constructors, the factory would own the constructed object, rather than the client who requested it.

4 PRAGMATICS

Other information recorded for visualization. The visualizer displays more than the ownership hierarchy: it also displays the read, write, and creation edges that cross the hierarchy. We record control chains for reads similar to the manner for writes. For creation we only record the pair $\langle o_k, \hat{o}_c \rangle$, where \hat{o}_c is the object created and o_k is the object in control of that creation.

Corner cases. It is possible, but unlikely, for an object to have no incoming write edges: for example, an immutable object that is never in control at the time of a field write. For such objects we make an ownership claim from their readers or constructing object. In the case there are no readers and the construction of the object was not observed (as happens for some JDK objects), then `root` owns the object.

Application objects vs System objects. A JVM typically creates some objects that are part of the system, rather than the application. For example, there is a system hash table that holds thread objects that are waiting to be called as part of the VM shutdown sequence. How do we distinguish this hash table from one used in an address book application? Our analysis considers that application objects are those instantiated by code in application packages or by other application objects. The first criterion captures objects created by static methods, for example `main`. The second criterion captures, for example, JDK objects created by the application. We only record control chains when an application object is read from or written to.

Exceptions. Exceptions can cause the call stack to pop back an arbitrary number of frames. Our trace collector keeps a stack of the objects that are the receivers of the methods on the call stack. When the call stack pops back, we need to pop back our receiver stack as well. To do this, we add a new local variable to every method, and populate it with a unique integer every time the method is executed. Whenever the program being analyzed calls the trace collector, it also passes the ID of the current stack frame, which the collector then checks against its internal records. If an exception has occurred, the trace collector's receiver stack may be mis-aligned with the actual call stack; by knowing the ID of the current call stack frame, the receiver can pop it's receiver stack back until it is properly aligned with the VM's call stack.

Sometimes exceptions are caught in uninstrumented code. For example, `java.lang.ClassLoader` often creates `java.net.URLClassLoader$1` objects which routinely throw `ClassNotFoundException`. Since we instrument `java.net.URLClassLoader$1` but not `java.lang.ClassLoader`, we see the exception being thrown, but not caught. In this case, we compare the call stack of the exception with the call stack of the program when it next enters an instrumented method, and align our trace collector's stack to their common prefix.

Arrays. We consider arrays as objects with a single field (i.e., we coalesce array indices).

Instrumentation. We use a 'Java agent' to dynamically instrument bytecode as it is loaded into the VM. We use the ASM bytecode instrumentation framework. We instrument all classes that the VM loads, except our own, the ASM classes, the GNU Trove collections classes (used by our trace collector), and those in `java.lang` and `java.io`. We observe that Apple's JVM runs the instrumenter in a separate thread, which can lead to deadlocks between the instrumenter and the class loader if one isn't careful (this is why we don't instrument `java.io`, for example).

The VM needs to load a few hundred classes before it is ready to load our Java agent. We redefine these pre-loaded classes with instrumented versions once the VM is initialized by calling `java.lang.Instrument.redefineClasses()`.

Caching. The trace file contains much redundant information. For example, if a method m reads (the same value) from a field f multiple times, each read would be recorded in the trace. The information recorded also includes the list of controllers at the time of the read. To reduce the amount of redundant information in the trace file, for each object we keep a hash of the control chain the last time that object was written to or read from. We only write control chains to the trace file whose hash differs from the hash in the cache. This technique reduces the size of the trace file by about 50%.

5 RELATED WORK

5.1 Ownership

Research in object ownership has been motivated by at least four issues:

Encapsulation: If one knows that there are no external aliases to a subobject, then one can reason locally about the correctness of the owning object, because we know that no other object will be surreptitiously mutating its state [9, 27, 28]. Boyapati et al. [9] express the encapsulation objective as follows:

D1. An object x *depends* on subobject y if x calls methods of y and furthermore these calls expose mutable behaviour of y in a way that affects the invariants of x .

D2. An object should own all of the subobjects it depends on.

Concurrency: Data races can be prevented if each object is either locked, immutable, thread-local, or has a unique pointer; these properties can be enforced with an ownership type system [10]. Deadlock can be prevented by enforcing a lock acquisition order with an ownership type system [8].

Memory management: An entire region of memory can be de-allocated simultaneously if an ownership type system can establish that there are no incoming pointers [11]. In languages with explicit de-allocation (such as C++), if ownership can be connected between the allocator and the de-allocator, then that object is not leaked, nor is it double-deleted [16]. Ownership can also be used to detect memory leaks [24].

Visualization [17, 19, 32] and program understanding [3]. Understanding aliasing and sharing through ownership can assist with program maintenance and debugging tasks. Aldrich et al. [3] and Lam and Rinard [19] require the programmer to annotate the program, and then make soundness claims about the information reported to the programmer. Hill et al. [17], like the present work, infers ownership for an un-annotated program from a dynamic analysis: this approach requires less up-front work by the programmer, but does not give the same guarantees.

5.2 Ownership inference

Most previous work on ownership inference has been static analyses designed to produce sound annotations for a static ownership type system (e.g., [3, 8, 16, 25]). Besides the obvious difference of being static, these prior works are less focused on communicating the inferred ownership and sharing information to the programmer. Aldrich et al. [3] and Boyapati et al. [8] use intra-procedural analysis to reduce the number of ownership type annotations the programmer must write; the programmer does not see the inferred information, and the inference requires the programmer to write some initial annotations. Moelius and Souter [25] extend [3] with an inference based on escape-analysis; they concede that the copious inferred program annotations are difficult for a person to read. Heine and Lam [16] use ownership to detect potential memory leaks in non-annotated C/C++ programs; the programmer gets a report about potential leaks, but does not see the ownership information that was used to produce that information.

Hill et al. [17] present a dynamic analysis that generalizes over an entire execution for visualization, similar to ours.

Wren [35] develops a formal system for inferring ownership type annotations for a fairly restricted subset of Java (for example, every variable must be `final`). He proposes, but does not implement, a combination of static and dynamic analysis.

Agarwal and Stoller [1] use a combination of dynamic and static analysis to infer types for Parameterized Race Free Java (PRFJ) [10]. The dynamic analysis observes types for fields and method parameters, and a static analysis infers types for local variables.

Mitchell [24] implements a dynamic ownership inference for detecting memory leaks. This analysis examines a single heap snapshot, rather than the entire program execution: consequently, the runtime overhead is lower, and the approach scales to larger programs (tens of millions of live objects).

5.3 Visualization

Software visualizers may be roughly divided on four criteria: hierarchical vs flat, matrix vs graph (box-and-arrow), interactive vs non-interactive, and syntactic vs semantic analysis. Interactive hierarchical visualizers tend to scale to larger programs. The work presented here is an interactive hierarchical matrix based on a semantic analysis.

Previous hierarchical matrix visualizers [33, 36] have been interactive and based on syntactic analyses. De Pauw et al. [13] present some non-interactive flat matrix visualizations based on a semantic analysis.

Rigi [26] is an example of an early interactive hierarchical box-and-arrow visualizer. SHriMP [20] extends the interactive nature of Rigi with a zooming user interface. Both Rigi and SHriMP usually visualize the results of syntactic analyses.

Lackwit [29] is a flat, non-interactive box-and-arrow visualization of an alias analysis. Lam and Rinard [19] present a similar visualization of ownership type annotations. Their type system guarantees that the diagrams are a conservative approximation of the possible interactions in the program.

Mitchell [24] presents a flat, non-interactive box-and-arrow visualization of a dynamic ownership analysis. His visualizer scales to large programs through extensive graph transformation and summarization.

Hill et al. [17] present an interactive hierarchical browser that is an augmented tree, rather than a matrix or box-and-arrow diagram. Their visualizer is based on a dynamic analysis similar to ours, that also generalizes over an entire execution.

Our previous work [32] presented a static analysis with the same objectives as this dynamic analysis. The static analysis was based on RTA [5, 6], which is context-insensitive. However, since our concept of ownership and our visualizer are context-sensitive, this analysis tended to produce unacceptable over-approximations for most non-trivial programs. The present dynamic analysis produces a precise under-approximation, which is more useful to the programmer than the sound over-approximation of our previous static analysis. Furthermore, the present analysis produces better results for design patterns such as factory and iterator.

6 SUMMARY AND FUTURE WORK

We have presented an object ownership inference technique based on the concept of *object control*. We implemented a dynamic analysis to capture *write control chains*, and an offline analyzer to merge these chains into a *write control graph*, and from that graph abstract an *ownership tree* using the immediate dominator relation. The results of the analysis are rendered in an interactive hierarchical matrix visualizer so the programmer can inspect the ownership and sharing relations in the program.

This technique is focused on producing reasonable design representations of real programs that may include idioms such as iterators that violate strict encapsulation, or that may have design defects such as representation exposure. It may not be feasible or tractable to put programs like these through an ownership type system, since they do not consistently have the properties the type system is trying to enforce. Our technique attempts to visualize where aliasing may lead to encapsulation violations in existing un-annotated programs.

Future work includes:

- *Empirical validation* that ownership inference helps real programmers solve real programming tasks.
- *Static analysis*. An object control graph for a program could be produced by a static analysis; probably something along the lines of object-sensitive points-to analysis [22, 23]. Contrasting the conservative results from this static approach with the precise results from the present dynamic approach might be interesting.
- *Extending DSM partitioning algorithms*. One of the main difficulties in software visualization is automatic layout. One of the contributions of Sangal et al. [33] is to apply Design Structure Matrix (DSM) partitioning/sorting algorithms to software visualization. However, these algorithms consider many orderings to be equivalent that are, for the programmer, quite different. We have started to work on extending these algorithms, which is why figure 2 looks so orderly.
- *Memory leak detection*. The temporal information associated with the control chains recorded by our dynamic analysis may be useful for memory leak detection.
- *Lightweight annotations*. A dynamic analysis such as this one could be used to check lightweight and partial annotations: for example, ‘objects of this class are never shared’, or ‘objects from this instantiation site are shared’. Such annotations could assist a programmer complete maintenance tasks on unfamiliar code.

ACKNOWLEDGMENTS

Thanks to Michael Ernst and the students of 6.883 for comments on earlier versions of this work. Thanks to Jeff Perkins and Adam Kiezun for helpful discussions on dynamic analysis and instrumentation. The stack frame numbering idea is due to Jeff Perkins.

This research was supported, in part, by grants 0086154 (‘Design Conformant Software’) and 6895566 (‘Safety Mechanisms for Medical Software’) from the ITR program of the National Science Foundation.

REFERENCES

- [1] RAHUL AGARWAL AND SCOTT D. STOLLER. Type inference for parameterized race-free Java. In GIORGIO LEVI AND BERNHARD STEFFEN, editors, *Proceedings of the 5th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, volume 2937 of *Lecture Notes in Computer Science*, pages 149–160, Venice, Italy, January 2004. Springer-Verlag.
- [2] JONATHAN ALDRICH. *Using Types to Enforce Architectural Structure*. PhD thesis, University of Washington, August 2003.
- [3] JONATHAN ALDRICH, VALENTIN KOSTADINOV, AND CRAIG CHAMBERS. Alias annotations for program understanding. In Matsuoka [21], pages 311–330.
- [4] PAULO SERGIO ALMEIDA. Balloon types: Controlling sharing of state in data types. In MEHMET AKSIT AND SATOSHI MATSUOKA, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, Jyväskylä, Finland, June 1997. Springer-Verlag. ISBN 3-540-63089-9.
- [5] DAVID F. BACON. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages*. PhD thesis, University of California, Berkeley, December 1997. UCB/CSD-98-1017.
- [6] DAVID F. BACON AND PETER F. SWEENEY. Fast static analysis of C++ virtual function calls. In JAMES COPLIEN, editor, *Proceedings of the 11th ACM/SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 324 – 341, San Jose, CA, October 1996.
- [7] CHANDRASEKHAR BOYAPATI. *SafeJava: A Unified Type System for Safe Programming*. PhD thesis, MIT Electrical Engineering and Computer Science, 2004.
- [8] CHANDRASEKHAR BOYAPATI, ROBERT LEE, AND MARTIN RINARD. Ownership types for safe programming: Preventing data races and deadlocks. In Matsuoka [21].
- [9] CHANDRASEKHAR BOYAPATI, BARBARA LISKOV, AND LIUBA SHRIRA. Ownership types for object encapsulation. In GREG MORRISSETT, editor, *Conference Record of the 30th ACM Symposium on the Principles of Programming Languages (POPL)*, New Orleans, Louisiana, January 2003.
- [10] CHANDRASEKHAR BOYAPATI AND MARTIN RINARD. A parameterized type system for race-free java programs. In JOHN VLISSIDES, editor, *Proceedings of the 16th ACM/SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Tampa, Florida, October 2001.
- [11] CHANDRASEKHAR BOYAPATI, ALEXANDRU SALCIANU, WILLIAM BEEBEE, AND MARTIN RINARD. Ownership types for safe region-based memory management in real-time java. In Gupta [15].
- [12] DAVID G. CLARKE, JOHN M. POTTER, AND JAMES NOBLE. Ownership types for flexible alias protection. In CRAIG CHAMBERS, editor, *Proceedings of the 13th ACM/SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Vancouver, British Columbia, Canada, October 1998.
- [13] WIM DE PAUW, RICHARD HELM, DOUG KIMELMAN, AND JOHN VLISSIDES. Visualizing the behaviour of object-oriented systems. In ANDREAS PAEPCKE, editor, *Proceedings of the 8th ACM/SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Washington, D.C., September 1993.
- [14] ERICH GAMMA, RICHARD HELM, RALPH JOHNSON, AND JOHN VLISSIDES. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [15] RAJIV GUPTA, editor. *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, June 2003.
- [16] DAVID L. HEINE AND MONICA S. LAM. A Practical Flow-Sensitive and Context-Sensitive C and C++ Memory Leak Detector. In Gupta [15].
- [17] TRENT HILL, JAMES NOBLE, AND JOHN POTTER. Scalable visualizations of object-oriented systems with ownership trees. *Journal of Visual Languages and Computing*, 13:319–339, 2002.
- [18] JOHN HOGG. Islands: Aliasing protection in object-oriented languages. In ANDREAS PAEPCKE, editor, *Proceedings of the 6th ACM/SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 271–285, Phoenix, AZ, October 1991.
- [19] PATRICK LAM AND MARTIN RINARD. A Type System and Analysis for the Automatic Extraction and Enforcement of Design Information. In LUCA CARDELLI, editor, *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP)*, volume 2743 of *Lecture Notes in Computer Science*, pages 275–302, Darmstadt, Germany, July 2003. Springer-Verlag. ISBN 3-540-40531-3.
- [20] MARGARET-ANNE STOREY AND HAUSI A. MÜLLER AND KENNY WONG. Manipulating and Documenting Software Structures. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, pages 275–284, Opio (Nice), France, October 1995.
- [21] SATOSHI MATSUOKA, editor. *Proceedings of the 17th ACM/SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Seattle, WA, October 2002.
- [22] ANA MILANOVA, ATANAS ROUNTEV, AND BARBARA G. RYDER. Parameterized object sensitivity for points-to and side-effect analyses for Java. In PHYLLIS FRANKL, editor, *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 1–11, Rome, Italy, July 2002.
- [23] ———. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, January 2005.
- [24] NICK MITCHELL. The runtime structure of object ownership. In DAVE THOMAS, editor, *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP)*, Nantes, France, July 2006.
- [25] SAMUEL E. MOELIUS, III AND AMIE L. SOUTER. An object ownership inference algorithm and its applications. In MARCO T. MORAZAN, editor, *Mid-Atlantic Student Workshop on Programming Languages and Systems (MASPLAS)*, Seton Hall University, April 2004.

- [26] HAUSI A. MÜLLER AND K. KLASHINSKY. Rigi — A System for Programming-in-the-large. In *Proceedings of the 10th ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 80–86, Raffles City, Singapore, April 1988.
- [27] PETER MÜLLER. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, FernUniversität Hagen, 2001.
- [28] JAMES NOBLE, JAN VITEK, AND JOHN POTTER. Flexible alias protection. In ERIC JUL, editor, *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP)*, volume 1445 of *Lecture Notes in Computer Science*, Brussels, Belgium, July 1998. Springer-Verlag. ISBN 3-540-64737-6.
- [29] ROBERT O’CALLAHAN AND DANIEL JACKSON. Lackwit: A program understanding tool based on type inference. In *Proceedings of the 19th ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 338–348, Boston, MA, May 1997.
- [30] DAVID LORGE PARNAS. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.
- [31] ———. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, SE-5(2), March 1979.
- [32] DEREK RAYSIDE, LUCY MENDEL, ROBERT SEATER, AND DANIEL JACKSON. An analysis and visualization for revealing object sharing. In MARGARET-ANNE STORY AND LI-TE CHENG, editors, *Eclipse Technology Exchange (ETX)*, San Diego, CA, October 2005.
- [33] NEERAJ SANGAL, EV JORDAN, VINEET SINHA, AND DANIEL JACKSON. Using Dependency Models to Manage Complex Software Architecture. In RICHARD P. GABRIEL, editor, *Proceedings of the 20th ACM/SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, San Diego, CA, October 2005. ISBN 1-59593-031-0.
- [34] NICK SCERBAKOV. Software Architecture Course LV 706.016 & LV 706.017. Graz University of Technology, IICM, 2002. <http://coronet.iicm.edu/sa>.
- [35] ALISDAIR WREN. Ownership type inference. Master’s thesis, Department of Computing, Imperial College, 2003. URL <http://www.cl.cam.ac.uk/users/aw345/writings/>.
- [36] JÜRGEN ZIEGLER, CHRISTOPH KUNZ, AND VEIT BOTSCH. Matrix browser: visualizing and exploring large networked information spaces. In *ACM Conference on Human Factors in Computing Systems (SIGCHI) extended abstracts*, pages 602–603, Minneapolis, MN, 2002. ISBN 1-58113-454-1.