Recent Techniques Review on Heap Verification with Class Objects

René Haberland

***Abstract*—This review paper is supposed to provide an overview of recent approaches and techniques in specifying and verifying dynamic memory with class objects. It may be used to show absence of memory leaks, valid-only memory accesses, and corresponding memory shape validation.**

Introduction

First, the problems involved in reasoning dynamic memory are formulated coming from recent case studies and previously suggested research propositions. Second, an overview of existing tools for a specification based verification of dynamic memory is provided. A brief comparison of specification-free tools and approaches follows. Third, existing models to reason about dynamic memory are introduced, and particularly for the memory separating model the benefits, current limitations and related issues are provided. In the fourth section theories on objects are reviewed in more detail, which are essential to today's favourable object-oriented programming paradigm. Section five provides an overview on applications of the verification of dynamic memory within the compiler domain, particularly the alias analysis phase and the garbage collection of unused locations. The last section summons up related techniques with a slightly different approach.

I. Problem Actuality

[27] is the *classic* paper on Hoare calculus. It's main contribution is to claim a mathematically profound way, a precise way, to describe the states before and after a program statement is fulfilled. The idea behind it is to formalise the state, s.t. it is possible to infer and check a given program satisfies a given pre- and post-condition, this is called the *Hoare-triple*. Hoare remarks because of the formal description simplicity may become a problem, and in conclusion this affects abstraction more generally. Hoare worked out proving may become too complex, at least too complex for an untrained person. He points out some particular programming features may be more difficult to formalise and to prove than others, like labels, arbitrary jumps and name parameters, for example.

[58] refines [27] by defining theoretical boundaries of computability. [58] postulates the difficulty of a Hoare-system is merely about the specification language rather than the rules of the Hoare-system. Wand referes to Cook's term of incompleteness, meaning a program by definition is incomplete whenever the program does not terminate and/or due to the expressibility of the assertion language. A Hoare-logic in second or even higher level is not decidable in terms of first-order predicate logic.

[11] reviews Hoare's proposition [27] which can be said was widely accepted and driven by the computer-aided proof community, and proofs parameter passing and other imperative features are sound and complete. It considers particularly the passing of procedures as functions and procedures modifying incoming parameters.

Despite its age [20] can still be considered as actually not resolved research problems, for instance the *aliasing problem*, which is the case whenever two locations point to the same value memory cell in dynamic memory. Morever, the Hoare calculus is found inappropriate as it has been suggested initially for (i) the automated generation of loop invariants for an arbitrary incoming program (ii) procedures which are passed as arguments (iii) changing the scope mode of variables (e.g. static, global, thread-local variables, etc.) and (iv) inner procedures. Clarke suggests for sake of completeness to forbid *location sharing*, which will be dropped here. [20] shows it is possible to prove correctness of co-routines when recursion is guaranteed to be absent.

The report [41] categorizes bugs in open source and commercial environments from within a period of over a decade. The outcome is that alias problem is by far one of the most important and challenging problems still present as it can be re-discovered again and again even in newer publications, like [42], [26] - so, the core problem did not change over time. The bug rate was found to be 23% in commercial environments under UNIX, and about 7% in GNU utilities. The most often and expensive bug class was "*invalid pointer and array access*". The authors point out that sad enough often memory-related issues were not detected at all or only pop up when porting a software product to a different platform.

[26] discusses the alias problem not just as a correctness issue, but also as a performance drawback. Hind points out some inefficient algorithms may work faster just because they base on aliases than alias-free algorithms. [26], [36] and me, too, heavily suspect a Alias Analysis which does not take into consideration the flow-graph of a program will have almost no effect at all on an applications overall performance. Hind works mainly with heuristics only and requests a research that focuses only on very local memory regions.

[36] shows aliasing problems are NP-hard which are due to the pointer reference levels, these are (1) determine whether two locations *must* or *may* alias and (2) if two locations alias within a procedure or beyond a procedure's boundaries (intra vs. inter-procedural alias analysis). The challange of getting

an efficient result lies in Interprocedural Alias Analysis. From [36] it can be concluded that Alias Analysis might be stronger when being enriched with SSA [21].

[17] discusses the inaccuracy of comparing certain proof tools with certain memory issues w.r.t. future proof tools. Bessey states (1) code updates complicate specifications and output predictions, and therefore should be avoided, (2) ordinary errors should be found ordinarily, and (3) more analysis does not necessarily cause better quality.

The reasons of dynamic memory erros may be (a) invalid heap memory access (b) access to uninitialised memory (c) out-of-memory exceptions/performance slow-down caused by too high memory consumption or (d) general memory leaks. The consequences may become totally unpredictable behaviour in worst case and harmless program malfunction or shutdown in a most optimistic case, for more details please refer to [41].

II. Existing Tools

Smallfoot [15] is the first verifier based on Separation Logic. It is a minimalistic academic implementation only which works experimental. It supports a minimal set of built-in predicates for heap data-structures. SpaceInvader Abductor [19] is Smallfoot's predecessor and supports at least to some extent abductive reasoning. jStar [22] is Smallfoot's object-oriented extension approach, which does support class invariants and restricted predicates to specify the heap in Java. jStar maps internally all program statements into JIMPLE, a GCC-like intermediate representation of the program control-flow-graph against which the verification conditions are validated. Among all mentioned verifiers jStar seems to be the most applicable tool for industry purposes for the amount of language feature reason. Verifast [31] allows user-defined predicates for a C-like input program, however it requires interaction from the user whilst performing a proof in a separate user-interface. Hurlin's tool [30] is very similar to [22], [15], but its main contribution is for multi-threaded applications.

Cyclone [24] is a heap verifier based on the Region-Calculus approach. SATlrE [49] is a Prolog-based framework based on the Shape Analysis approach from [46] and [54] with an improved re-calculation circuiting of the overall dependency graph. YNot [45] is an OCaML-based SMT-solver.

Alternative approaches and tools include (1) KeY/VDM++ [14], [60] which support object-orientation for industrial use and integrates with UML, but does no proof on heap memory, (2) dynamic memory verifying, with tools such as valgrind, ElectricFence [3], [1], [34] (3) compiler embedded static analysis, like SAFECode [2], (4) Program-to-XML approach in which the given program is analysed in a XML data-structure [12].

III. Memory Model

This section provides an overview of recent heap models and characteristics. It discusses the shape-based heap approach, the heap-separating approach and alternatives.

Shape Analysis

The goal of Shape Analysis [54] and [46] is to investigate how the overall heap memory shape changes over time for a particular program statement. In contrast to separating the heap into heaplets, Shape Analysis describes the entire heap graph. Changes in the graph are described by *transfer functions* that map stack, heap and an abstract location set into itself. There might be a basic asset of transfer functions needed in order to describe all kind of heap transformations, such as null, pointer or field assignment and heap allocation. [54] represents the classic paper on Shape Analysis and defines three states for the aliasing problem: alias, does not alias or may alias. [35] and [50] investigate in further detail why must-alias and may-alias remain undecidable problems in general. One problem [54] and [46] have in common is the location vertex naming in graphs, because there is only a compound graph to be described. In contrast to [54] [46] presents a faster, newer and improved attempt since [54] uses a reduced subset of shapes associated with a single program instruction. Both, [54] and [46] do not allow by default pointers of pointers, object pointers, dynamic array sizes, class field/method support, nor do they support memory sharing of data structures as in unions in the C language. [49] stresses that [54] and [46] may fuzzy gained aliases and so may cause unsoundness, e.g. if common subexpressions in an if-statement may alias but another branch must alias, the result could produce a must-alias, however, it must be a may-alias occassionally. A second remark is a particular transformation may become no more invertible if only a location points to null.

[49] presents a concise introduction and comparison of [54] and [46]. [49] considers [46] as more accurate approximation of aliases and proposes an own tail optimisation for shape graphs with the same value and same alias information — which is in analogy to common subexpression elimination, for a complexity of $O(\binom{m+1}{2})$ and a speed-up of roughly 90%. Pavlu suggests to simulate the more comprehensive inter-procedural alias analysis by a simpler intra-procedural alias analysis by renaming outgoing variables. He claims that context-insensitive analyses should not be considered in the future since it seems they are less (comp. [Hind99]). Pavlu raises the question of further performance improvement if summary nodes are separated from non-sharing summary nodes.

[47] presents a visualization toolkit for shape analysed heap in order to detect, for instance, rare or alternatively global heap invariants, such certain access path. It makes uses of abstract shape graphs which is a sub-graph folding and may be browsed within the toolkit, however, it does not resolve the principal problem underneath that all heap nodes need to be specified all the time. For abstraction reasons it is, for instance, hard to navigate between two heap cells. Moreover, aliases in visualised graphs become a non-trivial interpretation issue.

[19] is a mixture of points-to internal and a shape graph-based approach external behaviour. It identifies heaps, e.g. in loops, and abducts the next matching conclusion by comparing

with the first least different heap conclusions coming from applicable verification rule conclusions. The authors apply the longest precondition first heuristics. A restricted unification takes place while abducting.

Alternative Approaches

[57] tracks a functional approach to allocate regions on a stack, which are a set of locations. In total this approach is very ML-language specific (also see [24]). While transforming heap variables into regions the live-range analysis requires further remarks. It is able to distinguish between must and may alias cases. This approach does not allow lists or any other inductively recursive data objects nor procedures as return values for functions, and it insists the return type is known at compilation type.

[56] proposes *pointer rotations* which are known to be safe under the following conditions: first, the heap content does not change. Second, all elements still exist after a rotation and third, the number of locations is invariant. One benefit of rotations is their application may, if composed out of "*safe*" rotations only, be guaranteed garbage collection free. It may be used efficiently for list processings and safe copy operations. However, it has several drawbacks: Even if it does not require explicit heap specifications, it remains hard to adjust because small argument modification for the rotations may have very difficult to predict behavior, especially, when arguments may alias. Standard rotations are too rigid and may be too superficial for real scenarios, so argument modifications and rotation compositions may be needed.

Meyer [39] considers beside program soundness efficient garbage collection as the biggest heap challenge and advocates a move from heap objects to stack regions similar as can be found in [57]. He defines object simplicity and abstraction as most challenging object-orientation issues that need to be taken into consideration. The paper fully discusses objects, however, recursive class-objects will require further investigation. The soundness of assignments is proven sound by example, which does not hold in general for procedures with self-updating code. For sake of abstraction, Meyer proposes helper variables in order to keep a heap specification small.

Separation Logic

[52], [16], [53] provide an overview of *Separation Logic* which is *substructural* [51], which eliminates constants, like boolean values, and formalises rules on structural placeholder objects instead. In a Separation Logic structural rules [51] are rules for *Thinning*, *Contraction* and *Exchange*, and entities are heap memory cells. The comma in rules is replaced by the '$\star$'-operator which separates two distinct non-interleaving heaps unless specified further. Heaps are inductively defined. The '$\mapsto$'-operator denotes a mapping between a location on the left and the value from any valid (object) domain on the right. The Frame Rule states that if a subroutine call does not affect certain parts of the heap, namely the frame denoted by F, then in the antecedent it is sufficient to prove the Hoare-triple without frame F. [15] is a reference implementation with numerous simplications over the origin.

[16] introduces a separating heap calculus with unrestricted pointer arithmetics, dynamic arrays and for recursive procedures. It refers to a fix asset of heap predicates in order to allow reasoning over recursively defined data structures. Berdine et al. raise the question if typing is still essential when it comes to verify the heap against a provided specification only. The undecidability of unrestricted pointer arithmetics, even for a simple memory offset, is causing a serious problem for most recent garbage collectors.

Bornat [18] proposes conceptually a very similar notation for $\star$ which becomes $\oplus$ and is called *spatial separation*, but refers to the same idea behind it. He refers to first-order predicates to denote heaps which rapidly may loose abstraction. His main contribution is that objects can be treated as arrays, so each component becomes a individual pointer with a prefixed object name for each location.

Hurlin's main contribution [30] is the context of a heap separating model multiple objects may be accessed via a new accessor pattern in multiple thread in order to improve productivity. Since heaps are recursively structured Hurlin defines heaps as being a product of multiple heap-factors. This is a proposition to characterise heaps as numbers as a product of only prime numbers. Heap specifications are allowed to be arbitrary by using an anonymous heap '_'.

Parkinson [48] presents an object-oriented extension of the classic heap separating approach [52] with a reference implementation for Java. Encapsulation and inheritance are modelled through *ownership transfer* and *Abstract Predicate Families*. Bornat's permission model [18] is used since continuity holds on the frame rule. His predicates are introduced to encode the entire heap and stack state, but it does not allow arbitrary nested predicates, for instance. Predicates with same names but a different arity are allowed, the definition of his predicates does not match with Java and can be characterised as descriptive, so it does not know of types and intents to evaluate its arguments symbolically. Parkinson points out his predicates form a partially ordered set over its dependencies. Predicates may be added or removed as long there is no naming clash nor a missing predicate name. He proposes super calls, static fields, introspection, inner classes and quantification over predicates for further research.

IV. Object Calculi

[7], [25] can be considered the standard references on Object Theory. The semantics of objects are axiomatised, a type system is introduced for objects which can be chosen for formal proofs. Objects are considered tradionally as class instances, subtyping and polymorphism are discussed upon several sketch proof rule systems.

In [8] objects are considered not as abstract data types but as pure data records [23]. Neither does [8] introduce recursive class definitions nor does it consider pointers nor objects with aliases. In this model object regions may not share memory.

Types (T) can be defined as either integers or compound class-typed. An object is defined as an instance of some atomic type Int or a class which consists of an arbitrary number of distinguished fields f_i and methods m_j. Fields are of kind T, where methods are of kind $T_j \rightarrow T_{j+1} \rightarrow \cdots \rightarrow T_k$. The check if a class-object belongs to a certain class or whether its class or subclass of another class can be defined as component-wise set inclusion-check of unique occurences of f_i and m_j . The state of memory cells and regions are expressed by temporal predicates referring explicitly to one state before and one state after statement execution using a single result register. Recursive specifications are generally not allowed in [6]. [38] tries to relax this restriction by the introduction of an algebraic ideal construction, however, the restriction induces further drawbacks making it impractical because of a loss of soundness and its restriction to object-based calculi only. The presented model is not separating heaps and occassionally programs may not be proven regarding a general incompleteness restriction being demonstrated.

[13] presents a language that supports purely stacked objects of the same region (cmp. [57]). The approach in [13] shifts all local objects to the stack. It does not allow by language definition dangling pointers. It also does not allow recursive predicates over objects, and by global invariant it actually means object-dependencies that do not mutate. Banerjee notes a rising abstraction problem and urges the need for a object-wide specification with different views.

V. Applications

In this sections the most important areas of applications are presented, namely alias analysis and garbage collection.

Alias Analysis

Weihl [59] provides a very decent overview on the topic despite its age. It defines alias analysis as an approximation process in which pointer locations share the same heap content with pointers being read or written. The challange is to find fast and simply all aliasing relations, because every program statement can cause a strong increase of possibilities. Weihl notes that if a procedure is called it may change the outgoing variables and therefore is more difficult to process than intra-procedural. He introduces levels in order to express how much more complexity comes for every further statement.

[43] contains a very concise summary of all current and previous aliasing papers from himself. It is an excellent reference to this topic, provides up-to-date materials and can be considered as state-of-the art technique. It categorizes alias analysis into flow sensitive and insensitive, must-alias and may-alias approaches, and inter-procedural or intra-procedural analysis. It is worth noting at this point GCC provides an active compiler-switches which can improve alias calculations by making explicit may-not alias relations which finally can improve code quality.

[28] is an extension of Muchnick's earlier alias technique. It introduces a coinciding bitvector for aliases which is a special case of the data-flow dependent approach presented in [33]. The way local definitions and uses are used is actually very related to dependency webs within the static single assignment [21], [5]. Naeem [44] proves concept this is indeed a hot research area with further opportunities for improvement, so he substitutes, for instance, a dynamic list of aliases by a hash-table, his approach can also be considered as an attempt to re-formulate alias analysis in terms of SSA.

[49] distinguishes two main approaches in aliasing, an *unification-based* approach [55] which finds more must-aliases than its competing approach – it is the *inclusion-based* approach [9] which is weaker but therefore much more efficient than [55].

Garbage Collection

Jones et al. [32] provide a very complete and concise overview on garbage collection and discuss recent state-of-the art techniques. In particular it broadly focuses on parallelisation, which however is not the main focus of this paper.

Meyer [40] claims to keep garbage collection on all the time. He proposes to turn heap-allocation into stack-allocation.

Appel's [10] main idea is to force garbage collection to do almost nothing, so the amortized costs are lower than managing objects in stack as suggested by [57], [40]. It is common sense that an *application binary interface* and a *von-Neuman architecture* still require stack-objects are stored to and removed from the stack, operations which rapidly establish a performance bottleneck and are certainly not for free in terms of throughput. The solution proposed is a binary division of the available heap which synchronise each other on demand whenever a modification was performed. Appel's approximation is whenever the total amount of allocated data is no more than seven times the amount of free memory, the *copying garbage collection* [32] becomes indeed free. This result is due to the fact explicit heap deallocation is more expensive than copying redundant parts at once. Appel notes that nowadays memory is not an essential restriction anymore, and for garbage collection is far cheaper than any special hardware equipment or sophisticated because intractable collection techniques.

Despite its age today, [37] is still a valid approximation if caches or any other memory regions are available that have faster access capabilities. Larson considers the *compacting garbage collection* and defines a boundary function to the problem tractability depending on variable R, which denotes the region size to be freed, A the total amount of live data, H the amount of fast memory. He formulates two freeing strategies by defining equations based on allocation/freeing-frequency for a fixed period of time. Strategy 1 states: Maximize R, whenever $A \ll H$ does not hold. Strategy 2 states: Set $R = H$, if $A \ll H$ holds.

Apart from [10] garbage collection has one more principal addressing restriction. When xor-linked heap data-structures are used they can not be recognised by a classic collector without any extra cost, because its address becomes relative to a previous object address and as side-effect, for instance in a doubly xor-linked list, predecessor and successor may

be calculated with the same shift key instead of two separate pointers.

[4] is a technical report on Sun's *generational garbage collection* which stages allocated objects depending on its use frequency. The performance makes on average a fair optimum based on exhaustive statistics evaluation.

[29] gives an updated overview of technical boundaries on SSD-disks which have a very similar internal organisation as heaps. SSDs differ from hard disks because they may have bad blocks that will cause severe performance penalties due to additional write cascades. For a greedy collection tactic the slow-down gets unacceptable high (45%) for a utilisation rate of 0.5. Write operations take in general roughly 10 times longer than reads. The worst collection tactic will always write to a new page which simultaniously generates many problems at once. As in HDDs SSD's write performance significantly improves on sporadic long sequences are written. SSDs require more consideration today because these flash-memory are close to SDRAM, and the issue mentioned by Appel [10] becomes urgent.

REFERENCES

[1] Electricfence, http://perens.com/freesoftware.

[2] Safecode within llvm project. http://llvm.org.

[3] The valgrind project, http://www.valgrind.org.

[4] Memory management in the java hotspot virtual machine. Technical report, Sun Microsystems Inc., 2006.

[5] Static single assignment book, latest available at http://ssabook.gforge.inria.fr/latest/book.pdf. electronically, July 2014.

[6] Martin Abadi. Baby modula-3 and a theory of object. Technical report, Systems Research Center, Digital Equipment Corporation, 1993.

[7] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996. ISBN 0387947752.

[8] Martin Abadi and K. Rustan M. Leino. A logic of object-oriented programs. In *TAPSOFT '97: Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 682–696. Springer-Verlag, 1997.

[9] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[10] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Inf. Process. Lett.*, 25(4):275–279, 1987.

[11] Krzysztof R. Apt. Ten years of hoare's logic: A survey - part I. *ACM Trans. Program. Lang. Syst.*, 3(4):431–483, 1981.

[12] Greg J. Badros. Javaml: A markup language for java source code. In *Proceedings of the 9th international World Wide Web conference on Computer networks : the international journal of computer and telecommunications networking*, pages 13–15, 2000.

[13] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Regional logic for local reasoning about global invariants. In *In European Conference on Object Oriented Programming (ECOOP*, 2008.

[14] Michael Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[15] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, pages 115–137, 2005.

[16] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *APLAS*, pages 52–68, 2005.

[17] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, 2010.

[18] Richard Bornat. Proving pointer programs in hoare logic. In *MPC*, pages 102–126, 2000.

[19] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 36:289–300, 2009.

[20] Edmund Melson Clarke, Jr. Programming language constructs for which it is impossible to obtain good hoare axiom systems. *J. ACM*, 26(1):129–147, 1979.

[21] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[22] Dino Distefano and Matthew J. Parkinson. jstar: towards practical verification for Java. In *OOPSLA*, pages 213–226, 2008.

[23] Hartmut Ehrig and Barry K. Rosen. The mathematics of record handling. *SIAM J. Comput.*, 9(3):441–469, 1980.

[24] Dan Grossman, J. Gregory Morrisett, Trevor Jim, Michael W. Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *PLDI*, pages 282–293, 2002.

[25] Carl A. Gunter and John C. Mitchell (eds.). Theoretical aspects of object-oriented programming - types, semantics, and language design. MIT Press, 1994.

[26] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *ACM, PASTE'01*, June 2001.

[27] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[28] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, PLDI '89, pages 28–40, New York, NY, USA, 1989. ACM.

[29] X.-Y. Hu and R. Haas. The fundamental limit of flash random write performance: Understanding, analysis and performance modelling. Technical Report 99781, IBM Research Zuerich, March 2010.

[30] Clément Hurlin. *Specification and Verification of Multithreaded Object-Oriented Programs with Separation Logic*. PhD thesis, Université Nice - Sophia Antipolis, September 2009.

[31] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In *Proceedings of the Third International Conference on NASA Formal Methods*, NFM'11, pages 41–55, Berlin, Heidelberg, 2011. Springer-Verlag.

[32] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011.

[33] Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2009.

[34] Christian Kirsch. Zeigs mir - freie speichertools electricfence und valgrind (german). *Magazin fr professionelle Informationstechnik, iX*, 3:82–84, 2003.

[35] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1:323–337, 1992.

[36] William Landi and Barbara G. Ryder. Pointer-induced aliasing: A problem classification. In *POPL*, pages 93–103, 1991.

[37] Richard G. Larson. Minimizing garbage collection as a function of region size. *SIAM J. Comput.*, 6(4):663–668, 1977.

[38] K. Rustan M. Leino. Recursive object types in a logic of object-oriented programs. *Nordic J. of Computing*, 5(4):330–360, 1998. ISSN 1236-6064.

[39] Bertrand Meyer. Proving pointer program properties - part1: Context and overview, part2: The overall object structure. ETH Zrich, Journal of Object Technology, 2003.

[40] Bertrand Meyer. Proving pointer program properties - part2: The overall object structure. ETH Zrich, Journal of Object Technology, 2003.

[41] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. In *In Proceedings of the Workshop of Parallel and Distributed Debugging*, pages 1–22. Digital Equipment Corporation, 1990.

[42] Barton P. Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. Nichts dazugelernt – empirische studie zur zuverlssigkeit von unix-utilities. *Magazin fr professionelle Informationstechnik, iX*, 9:108–121, 1995.

[43] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., 2007. 856p.

[44] Nomair A. Naeem and Ondrej Lhoták. Efficient alias set analysis using ssa form. In *Proceedings of the 2009 international symposium on Memory management*, ISMM '09, pages 79–88, New York, NY, USA, 2009. ACM.

[45] Aleksandar Nanevski, Greg Morrisett, Avi Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Reasoning with the awkward squad. In *In ACM SIGPLAN International Conference on Functional Programming*, 2008.

[46] F Nielson, HR Nielson, and C Hankin. *Principles of Program Analysis*. Springer Verlag, 1999.

[47] Sascha A. Parduhn, Raimund Seidel, and Reinhard Wilhelm. Algorithm visualization using concrete and abstract shape graphs. In *Proceedings of the ACM 2008 Symposium on Software Visualization, Ammersee, Germany, September 16-17, 2008*, pages 33–36, 2008.

[48] Matthew Parkinson. *Local Reasoning for Java*. PhD thesis, Cambridge University, 2005.

[49] Viktor Pavlu. Shape-based alias analysis - extracting alias sets from shape graphs for comparison of shape analysis precision. Master's thesis, Vienna University of Technology, 2010.

[50] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, September 1994.

[51] Greg Restall. *Introduction to Substructural Logic*. Routledge, 2000. ISBN 041521534X.

[52] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

[53] John C. Reynolds. *An Introduction to Separation Logic*. Carnegie Mellon University, 2009.

[54] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002. ISSN 0164-0925.

[55] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[56] Norihisa Suzuki. Analysis of pointer "rotation". *Commun. ACM*, 25(5):330–335, 1982.

[57] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, 1997.

[58] Mitchell Wand. A new incompleteness result for hoare's system. In *STOC*, pages 87–91, 1976.

[59] William E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables and label variables. In Paul W. Abrahams, Richard J. Lipton, and Stephen R. Bourne, editors, *POPL*, pages 83–94. ACM Press, 1980.

[60] Georg Weienbacher. Ohne beweis — vdm++: Lightweight formal methods. *Magazin fr professionelle Informationstechnik, iX*, 3:157–161, 2001.