Fast Escape Analysis and Stack Allocation for Object-Based Programs

David Gay[1]* and Bjarne Steensgaard[2]

[1] EECS Department, University of California, Berkeley
dgay@cs.berkeley.edu
[2] Microsoft Research
rusa@microsoft.com

Abstract. A fast and scalable interprocedural escape analysis algorithm is presented. The analysis computes a description of a subset of created objects whose lifetime is bounded by the lifetime of a runtime stack frame. The analysis results can be used for many purposes, including stack allocation of objects, thread synchronization elimination, dead-store removal, code motion, and iterator reduction. A method to use the analysis results for transforming a program to allocate some objects on the runtime stack is also presented. For non-trivial programs, typically 10%-20% of all allocated objects are placed on the runtime stack after the transformation.

1 Introduction

Several program optimizations require that objects be accessible from only one thread and/or have a bounded lifetime. For example, in order for an object to be allocated in a stack frame of a thread's runtime stack, the lifetime of the object has to be bounded by the lifetime of the stack frame. Escape analyses compute bounds of where references to newly created objects may occur.

This paper presents a fast interprocedural escape analysis algorithm for whole-program analysis of object-oriented programs written in Java™-like programming languages. The complexity of the algorithm is linear in the size of the program plus the size of the static call graph. The algorithm is simple and the analysis results are good enough to be useful in practice. The algorithm also demonstrates that a limited form of polymorphic analysis is possible in linear time.

There are many potential uses for the analysis results. For example, if a Java object is known to be accessible only to the thread creating the object, then most synchronization operations on the object can be eliminated and dead stores to fields of the object are easier to identify. Java's memory consistency model dictates that all memory reads and writes are fully performed at synchronization points, but if an object is known only to a single thread then it is not detectable

* The work was performed while the author was working at Microsoft Research.

A. Watt (Ed.): CC/ETAPS 2000, LNCS 1781, pp. 82–93, 2000.

if reads and writes to the fields of the object are moved around synchronization points.

The usefulness of the analysis results are demonstrated in this paper by a stack allocation transformation. A subset of the objects whose lifetime is bounded by the lifetime of a runtime stack frame are allocated in the stack frame instead of on the heap. If a method creates and returns an object then the object may instead be pre-allocated in the caller's stack frame (a common scenario for iterator objects in Java). Some objects allocated in a stack frame can be replaced by local variables for the object fields (another common scenario for iterator objects). Called methods of the reduced object can either be inlined to use the field variables, or the field variables passed as arguments to specialized methods.

In Section 2 the escape analysis algorithm is presented. The value of the analysis results is demonstrated by using the analysis results for a stack allocation transformation described in Section 3 and for an object reduction transformation described in Section 4. The efficacy of the escape analysis results is discussed in Section 5, Section 6 discusses related work, and Section 7 concludes.

2 Escape Analysis

The objective of the analysis is to keep track of objects created during the execution of a method. The objects may be created directly in the method or in methods called by the method. An object is considered to have *escaped* from the scope of a method if a reference to the object is returned from the method, or if a reference to the object is assigned to a field of an object.

The above stated rules can be almost directly encoded as constraints on elements of a simple type system. Solving the constraint system can be done in time and space linear in the number of constraints.

The constraints can be derived directly from the subject program. The program is assumed to be in Static Single Assignment (SSA) form [CFR+91]. In SSA form, each local variable is assigned exactly once. When two or more different definitions of a variable can reach a control-flow merge point, all the definitions are renamed and the value of the variable at the merge point is "calculated" by a special *ϕ-function* which takes all the new variables as arguments. The constraint derivation will be presented for the representative set of statements listed in Fig. 1. The use of SSA makes the dataflow explicit without needing to consider the control-flow statements, so only the `return` and `throw` control statements are interesting. The `return` statement is annotated with the method in which it occurs. The `new` statement creates an object of the specified class but does not initialize the object apart from filling the memory block with zeros as required by the JVM [LY99].

A *fresh method* returns an object created during the execution of the method. A *fresh variable* is a variable whose defining statement either creates an object directly (via `new`) or indirectly (via a call of a fresh method). For each fresh variable, the analysis must determine if the value assigned may escape in any

Domains:	Effect Statements:	Control Statements:
$m \in M$ = methods	$s = v$	`if` v
$f \in F$ = fields	$v = s$	`return`$_m$ v
$s \in S$ = static fields	$v_0.f = v_1$	`goto`
$c,\ c_0,\ c_1, \ldots \in C$ = classes	$v_0 = v_1.f$	`throw` v
$v,\ v_0,\ v_1,\ \ldots \in V$ = local variables	$v_0 = v_1$	
	$v_0 = \phi(v_1, \ldots, v_n)$	
	$v_0 = v_1.m(v_2, \ldots, v_n)$	
	$v =$ `new` c	

Fig. 1. representative intermediate language statements

way. For each variable the analysis determines whether the values assigned to the variable may be returned; this serves the dual purpose of keeping track of values that escape a method by being returned from it, and allows polymorphic tracking of values through methods that may return an object passed as an argument to the method.

The analysis computes two boolean properties for each local variable, v, of reference type. The property *escaped*(v) is true if the variable holds references that may escape due to assignment statements or a `throw` statement. The property *returned*(v) is true if the variable holds references that escape by being returned from the method in which v is defined.

Properties are introduced to identify variables that contain freshly allocated objects and methods returning freshly allocated objects. The set of Java reference types (classes) is augmented with a $\bot$ and a $\top$ element to form a flat lattice, τ, with partial order $\leq$. The proper elements of τ are *discrete*; the least upper bound of two distinct proper elements is $\top$. The τ property *vfresh*(v) is a Java reference type if v is a fresh variable (as defined above) assigned a freshly allocated object of exactly that type, and is either $\top$ or $\bot$ otherwise. Intuitively, the $\bot$ value means "unknown" and the *top* value means "definitely not fresh". *vfresh* is $\top$ for all formal parameter variables. The τ property *mfresh*(m) is a Java reference type if the method m is a fresh method (as defined above) returning a freshly allocated object of exactly that type, and is either $\top$ or $\bot$ otherwise.

Each statement of a program may impose constraints on these properties. The constraints imposed by the interesting representative statements are shown in Fig. 2. No attempts are made to track references through assignments to fields, so any reference assigned to a field is assumed to possibly escape from the method in which the assignment occurs. In the rule for method invocation, if a reference passed as an argument to a method may be returned from the method then the *escaped* and *returned* properties are propagated as if there was an assignment from the actual parameter to the left-hand-side variable. The function *methods-invoked* returns a set of methods that may be invoked at the call site as indicated by the given call graph. The function *formal-var* returns the formal parameter variable indicated by the given method and parameter index.

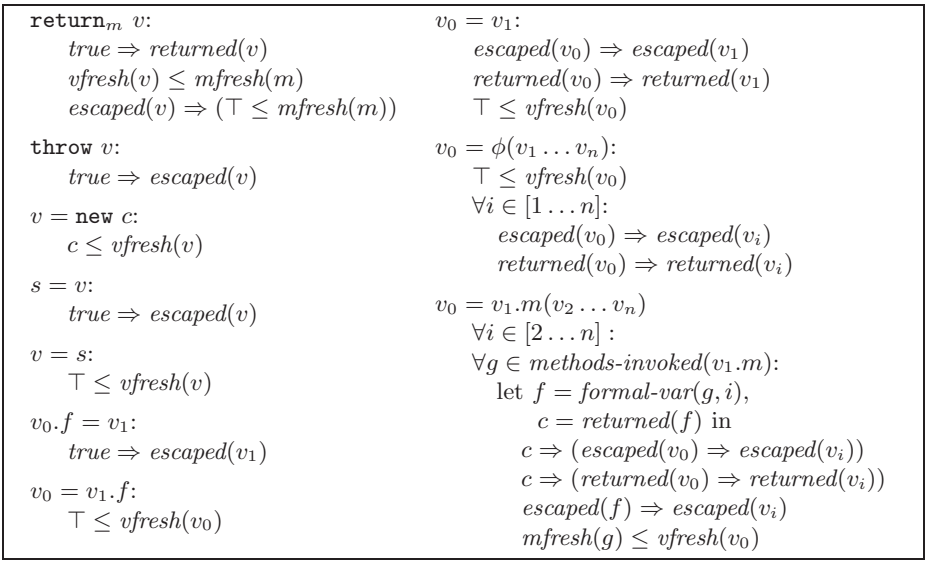

Fig. 2. constraints for *escaped*,*returned*, *vfresh*, and *mfresh* implied by the syntactic forms of statements.

The number of constraints generated from the program is linear in the size of the program plus the size of the call graph measured as the total number of corresponding actual/formal parameter pairs at all call sites.

Most of the constraints are boolean implications. The minimal solution to these constraints may be found by initializing all properties to be false and updating the properties as constraints are added. The properties values change monotonically. A true property will always stay true. A property that is false at one point may become true as more constraints are added. When adding a constraint where the trigger is a property whose value is false, a pending list for that property value may be used to ensure the constraint is processed again, should the property value become true. Similarly, pending lists can be used to reprocess $\leq$ constraints when the left-hand element changes value; the constraints are based on a lattice of height 3, so each constraint is processed at most 3 times. It follows that a minimal solution to the set of constraints can be found in time and space linear in the number of constraints.

For method calls, the implication constraints conditioned on *returned*(f) calls makes the analysis *non-sticky*, meaning that one invocation of a method does not affect the analysis results for another invocation of the same method (except for recursive method invocations). This kind of "polymorphism" is rarely seen for algorithms with linear time complexity.

The number of constraints can be limited to the size of program if the call graph is computed by an analysis like Rapid Type Analysis [BS96], or in general if the co-domain of *methods-invoked* is a power-set that can be partially ordered

by the subset operator. This can be achieved by summarizing the properties for all methods in each set in the co-domain of *methods-invoked*. The summary for each set is computed by adding constraints from the summary of the largest other set it contains that is in the co-domain of *methods-invoked* and from the individual methods not in the contained set. The rule for adding constraints for method invocations should of course be modified to use the summary properties for the set of methods possibly invoked.

Constraints for parts of a program can be computed independently and subsequently linked together to obtain a solution valid for the entire program. It follows that a partial solution for library code can be precomputed and thus doesn't need to be recomputed for every program. If partial solutions for *all* parts of a program are linked together, the analysis results will be the same as if the analysis was performed on the entire program as a whole.

A conservative result can also be obtained in the absence of parts of a program. If unknown methods may be called, a conservative approximation to their effects is to assume that all formal parameters escape and that each formal parameter may be returned as a result of the method.

3 Stack Allocation

The efficacy of the escape analysis results is demonstrated by using the analysis results for stack allocation of objects.

To keep it simple, the stack allocation transformation allocates objects in the stack frame of a method. No separate stack of objects is used. To avoid the use of a frame pointer, the stack frame size must be computed at compile time. No attempt is made to stack allocate arrays since the length of the arrays may not always be known at compile time[1]. Objects created in a loop can only be stack allocated if objects from different loop iterations have non-overlapping lifetimes, so the used memory area can be reused in subsequent iterations.

Some methods create objects and return a reference to the newly created object. The object cannot be stack allocated in the stack frame of the method creating the object. Instead, memory for the object may be reserved in the stack frame of a calling method, and a pointer to the reserved memory area may be passed to a specialized version of the object-creating method. Initialization of the object is done in the specialized method.

A new effect statement is added to create an object on the stack:

$$v = \texttt{newStack}\ c.$$

It is left to the compiler backend to reserve memory in the stack frame and to translate the `newStack` operator into a computation of the address of the reserved memory.

[1] It is possible to stack allocate arrays whose length is known at compile time, but the details will not be presented in this paper.

An extra property, $loop(v)$, is introduced to identify when stack allocation is impossible due to overlapping lifetimes. $loop(v)$ is a boolean property that is true if the local variable v is modified in a loop and objects referenced in different iterations of the loop have overlapping lifetimes. The property is only interesting for variables containing references that do not otherwise escape.

Given a method in SSA form, objects created by a given `new` statement in a given method execution can only have mutually overlapping lifetimes if an object may escape or if a reference to an object is stored in a local variable used as an argument to a ϕ expression at a loop header.

Figure 3 shows the constraints on the *loop* property imposed by the interesting representative statements. For the purposes of exposition, it is conservatively assumed that all ϕ expressions occur at loop headers.

$v_0 = v_1$:	$v_0 = v_1.m(v_2 \ldots v_n)$
$\quad loop(v_0) \Rightarrow loop(v_1)$	$\quad \forall i \in [2 \ldots n]$:
$v_0 = \phi(v_1 \ldots v_n)$:	$\quad \forall g \in methods\text{-}invoked(v_1.m)$:
$\quad \forall i \in [1 \ldots n]$:	$\quad\quad$ let $f = formal\text{-}var(g, i)$,
$\quad\quad true \Rightarrow loop(v_i)$	$\quad\quad\quad c = returned(f)$ in
	$\quad\quad\quad c \Rightarrow (loop(v_0) \Rightarrow loop(v_i))$

Fig. 3. constraints for *loop* implied by the syntactic form of statements.

For each statement of the form

$$v = \texttt{new}\ c$$

any object created is only used within the method and the object has non-overlapping lifetimes with other objects created by the statement if $escaped(v)$, $returned(v)$, and $loop(v)$ all are false. In that case, the `new` operator may be replaced by the `newStack` operator to allocate the object on the runtime stack.

Methods that return a freshly created object may be specialized to instead take a freshly created object as an extra argument. The freshly created object may be created on the stack at those call sites where the object originally returned does not escape. At the remaining call sites, the unspecialized method may be called or the specialized method may be called with a freshly created object on the heap.

Initialization of objects must be carefully considered. The JVM semantics dictate that any freshly allocated block of memory be filled with zeros. A block of memory allocated in a runtime stack frame can be filled with zeros at allocation time. If a pointer to such a memory block is passed as an extra argument to a specialized method, then the memory block is only known to be filled with zeros for one use. To ensure that the memory block is only used once, a depth-first traversal of each method m is performed, ensuring that for each return variable v the following holds:

- The definition of v does not occur in a loop in m, and
- All paths in m from the definition of v terminate with a statement: `return` v.

If either condition is not satisfied, the following constraint is added to indicate that a specialized method should not be created:

$$\top \leq \mathit{mfresh}(m).$$

For each method m, for which $\mathit{mfresh}(m)$ is a proper element, a specialized version, m', of the method is created with an extra formal parameter, *site*. For each variable v, in m, for which $\mathit{vfresh}(v)$ and $\mathit{returned}(v)$ both are true ($\mathit{loop}(v)$ and $\mathit{escaped}(v)$ are both false), the body of m is specialized as follows:

- If the definition of v is a statement of the form $v =$ `new` c, eliminate the statement.
- If the definition of v is a statement of the form $v = h(v_1, \ldots, v_n)$, specialize the statement to be $h'(v_1, \ldots, v_n, \mathit{site})$, where h' is the specialized version of h.
- Substitute all uses of v with uses of *site*.

The language fragment shown in Fig. 1 does not include methods without return values, but the specialized methods do not need to return a value and may be specialized accordingly.

Each statement in the program of the form

$$v_0 = v_1.m(v_2, \ldots, v_n),$$

for which $\mathit{vfresh}(v_0)$ is a proper element c and the properties $\mathit{returned}(v_0)$, $\mathit{escaped}(v_0)$, or $\mathit{loop}(v_0)$ all are false, is modified to invoke a specialized method

$$v_1.m'(v_2, \ldots, v_n, v_0).$$

The objects returned from the called method in the original program may be allocated in the stack frame of the calling method, so a statement

$$v_0 = \texttt{newStack}\ c$$

is inserted before the method invocation.

4 Object Reduction

Some objects may be replaced by local variables representing the fields of the objects. A requirement for doing so is that the conditions for stack allocation as described in the previous section are met. In addition, all the uses of the objects must be inlined to use the local variables rather than the fields.[2] This object

[2] In some cases the field values may be passed as arguments to methods instead of the object reference, but that experiment was not attempted for this paper.

reduction is often possible for objects of type `java.util.Enumeration`, and has the advantage of removing a large fraction of the overhead due to using iterator objects.

Inlining may of course cause code growth. Object reduction is a trade-off between code growth and object elimination. Object reduction is usually advantageous for iterator objects.

5 Empirical Results

The stack allocation and object reduction transformations have been implemented in the Marmot compiler [FKR+99]. The efficacy of the transformations is evaluated on the set of benchmarks described in Table 1. For these benchmarks, the stack allocation transformation typically places 10–20% (and 73% in one case) of all allocated objects on the stack as shown in Table 2.

Performance improvements due to the transformations will depend greatly on quality of the underlying compiler and the details of the machine (especially the memory subsystem) the program is being executed on. Performance improvements in general come from less work for the memory allocator and garbage collector, increased data locality, and the optimizations enabled by doing object elimination. Improvements in running time were typically 5–10% for non-trivial programs compiled with Marmot [FKR+99], an optimizing Java to native code (x86) compiler.[3] The analyses and transformations for stack allocation and object reduction increase compile time by approximately 1%.

The generated code and the garbage collection systems are high quality. The Marmot backend does not currently try to reuse memory in the stack frame for stack allocated objects with non-overlapping lifetimes. Doing so would likely yield further performance improvements.

Table 1. Benchmarks

Name	LOC	Description
marmot	88K	Marmot compiling itself
jessmab	11K	Java Expert Shell System solving “Bananas and Monkeys” problem
jessword	11K	Java Expert Shell System solving the “Word game” problem
jlex	14K	JLex generating a lexer for sample.lex
javacup	8.8K	JavaCup generating a Java parser
parser	5.6K	The JavaCup generated parser parsing Grm.java
slice	1K	Viewer for 2D slices of 3D radiology data

[3] Performance numbers for a larger and partially overlapping set of benchmarks are presented in [FKR+99].

Table 2. Objects allocated on the stack

Name	% Code	Stack	Allocated		% Stack allocated		% Reduced	
	increase	increase	objs	bytes	objects	bytes	objects	bytes
marmot	8.5	24.4KB	98M	2289MB	20.6	13.9	11.1	7.1
jessmab	1.2	936 B	510K	19MB	19.1	10.5	0.3	0.2
jessword	1.2	1080 B	305K	11MB	9.7	5.3	0.1	0.1
jlex	2.5	328 B	41K	1.6MB	15.7	7.2	1.8	0.5
javacup	6.9	328 B	785K	21MB	12.6	8.8	1.8	1.3
parser	0.3	116 B	1201K	32MB	1.7	1.2	0.0	0.0
slice	3.3	220 B	463K	16MB	72.9	62.4	72.2	62.1

The data demonstrates that a significant fraction of all objects can be identified as allocatable on the runtime stack using a very simple analysis algorithm. The increase in stack size is relatively small.

6 Related Work

Escape analysis and stack allocation or compile-time garbage collection based on same have been performed on functional languages like SML or Lisp [Bla98, Deu97, Hug92, ISY88, Moh95, PG92], and recently for Java [CGS+99, Bla99, BH99, WR99]. The precision of the escape analyses considered for functional languages is much greater than the precision of the algorithm presented in this paper. For instance, some of them can find that a function's result does not contain elements from the spine of its second argument (a list). This extra precision appears to be necessary for effective stack allocation in these list-oriented languages. The only algorithm with a near-linear complexity is that of Alain Deutsch [Deu97] (the complexity is $n \log^2 n$ where n is the program's size). A study of the performance of this algorithm on some SML benchmarks was performed by Bruno Blanchet [Bla98]. On the only large program considered, this algorithm placed 25% of allocated bytes on the stack and gave a performance improvement of 3-4%. An extension of this work to Java [Bla99] gives an average speedup of 21%, but is also eliminating some synchronization operations.

Phase 1 of the algorithm of Bogda and Hölzle [BH99] is similar to the escape analysis of this paper, except that it is flow-insensitive. Also Bogda and Hölzle apply their analysis to synchronization elimination rather than stack allocation. Choi et al [CGS+99] and Whaley and Rinard [WR99] present escape analyses for Java based on building points-to style graphs. These analyses achieve greater precision, and allocate more objects on the stack, at the expense of a much more complex analysis. Direct use of alias and points-to analysis [CWZ90, Deu90, Hic93, JM81, SF96, VHU92] and other miscellaneous techniques [Bar77, BS93, JM90, JL89, Sch75] have also been considered for compile-time garbage collection. These analyses are generally expensive and their effectiveness is unclear.

Dolby and Chien's object inlining techniques [DC98] can also be applied to stack allocation, but the details are only alluded to.

The presented escape analysis assumes that any reference assigned to a field escapes. It seems likely that incorporating a points-to analysis algorithm (like [Ste96]) could help the escape analysis to allow stack allocation of objects referenced in fields of other stack allocated objects.

A general discussion of the kinds of constraint problems that can be solved in linear time appears in [RM98].

7 Conclusion

The simple escape analysis algorithm presented has demonstrated its usefulness by using the results as the basis for a simple stack allocation system that allocates Java objects on the call stack instead of on the heap. This system is effective, allocating 10–20% of all objects on the stack. When used in Marmot, the transformations described yield speedups of 5–10% on a collection of non-trivial benchmark programs.

References

[Bar77] Jeffrey M. Barth. Shifting garbage collection overhead to compile time. *Communications of the ACM*, 20(7):513–518, July 1977. 90

[BH99] Jeff Bogda and Urs Hölzle. Removing Unnecessary Synchronization in Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, pages 35–46. ACM Press, October 1999. 90

[Bla99] Bruno Blanchet. Escape Analysis for Object-Oriented Languages: Application to Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, pages 20–34. ACM Press, October 1999. 90

[Bla98] Bruno Blanchet. Escape analysis: Correctness proof, implementation and experimental results. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 25–37, San Diego, California, January 98. 90

[BS93] E. Barendsen and S. Smetsers. Conventional and uniqueness typing in graph rewrite systems. In Rudrapatna K. Shyamasundar, editor, *Proceedings of Foundations of Software Technology and Theoretical Computer Science*, volume 761 of *LNCS*, pages 41–51, Bombay, India, December 1993. Springer-Verlag. 90

[BS96] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 31, 10 of *ACM SIGPLAN Notices*, pages 324–341, New York, October6–10 1996. ACM Press. 85

[CFR+91] 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 Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991. 83

[CGS+99] Jong-Deok Choi, M. Gupta, Mauricio Serrano, Vugranam C Shreedhar, and Sam Midkiff. Escape Analysis for Java. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, pages 1–19. ACM Press, October 1999. 90

[CWZ90] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. *ACM SIGPLAN Notices*, 25(6):296–310, June 1990. 90

[DC98] Julian Dolby and Andrew A. Chien. An Evaluation of Automatic Object Inline Allocation Techniques. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'98)*, pages 1–20. ACM Press, October 1998. 91

[Deu90] Alan Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In ACM-SIGPLAN ACM-SIGACT, editor, *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages (POPL '90)*, pages 157–168, San Francisco, CA, USA, January 1990. ACM Press. 90

[Deu97] Alain Deutsch. On the complexity of escape analysis. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 358–371, Paris, France, January 97. 90

[FKR+99] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: An optimizing compiler for Java. Technical Report MSR-TR-99-33, Microsoft Research, June 1999. Accepted for publication in Software — Practice & Experience. 89

[Hic93] James Hicks. Experiences with compiler-directed storage reclamation. In R. John M. Hughes, editor, *Record of the 1993 Conference on Functional Programming and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, Copenhagen, June 1993. Springer-Verlag. 90

[Hug92] Simon Hughes. Compile-time garbage collection for higher-order functional languages. *Journal of Logic and Computation*, 2(4):483–509, August 1992. 90

[ISY88] Katsuro Inoue, Hiroyuki Seki, and Hikaru Yagi. Analysis of functional programs to detect run-time garbage cells. *ACM Transactions on Programming Languages and Systems*, 10(4):555–578, October 1988. 90

[JL89] S. B. Jones and D. Le Métayer. Compile-time garbage collection by sharing analysis. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture '89, Imperial College, London*, pages 54–74, New York, NY, 1989. ACM. 90

[JM81] Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of Lisp-like structures. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 102–131. Englewood Cliffs, N.J.: Prentice-Hall, 1981. 90

[JM90] Thomas P. Jensen and Torben Mogensen. A backwards analysis for compile-time garbage collection. In Neil D. Jones, editor, *ESOP'90 3rd European Symposium on Programming, Copenhagen, Denmark, May 1990. (Lecture Notes in Computer Science, vol. 432)*, pages 227–239. Springer-Verlag, 1990. 90

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition edition, 1999. 83

[Moh95] Markus Mohnen. Efficient compile-time garbage collection for arbitrary data structures. Technical Report 95-08, RWTH Aachen, Department of Computer Science, 1995. 90

[PG92] Young Gil Park and Benjamin Goldberg. Escape analysis on lists. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI)*, pages 116–127, 1992. 90

[RM98] Jakob Rehof and Torben Æ. Mogensen. Tractable constraints in finite semilattices. *Science of Computer Programming*, 35(2-3):191–221, 1998. 91

[Sch75] J. T. Schwartz. Optimization of very high level languages – I. Value transmission and its corollaries. *Computer Languages*, 1(2):161–194, 1975. 90

[SF96] Manuel Serrano and Marc Feeley. Storage use analysis and its applications. In *Proceedings of the 1st International Conference on Functional Programming*, June 1996. 90

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, St. Petersburg, Florida, January 1996. 91

[VHU92] Jan Vitek, R. Nigel Horspool, and James Uhl. Compile-time analysis of object-oriented programs. In *Proceedings of the 4th Int. Conf. on Compiler Construction, CC'92*, Paderborn, Germany, 1992. Springer-Verlag. 90

[WR99] John Whaley and Martin Rinard. Compositional Pointer and Escape Analysis for Java Programs. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, pages 187–206. ACM Press, October 1999. 90