

Efficiently Verifiable Escape Analysis

Matthew Q. Beers, Christian H. Stork, and Michael Franz

School of Information and Computer Science
University of California, Irvine
Irvine, CA 92717-3425, United States
{mbeers,cstork,franz}@uci.edu

Abstract. Escape analysis facilitates better optimization of programs in object-oriented programming languages such as Java, significantly reducing memory management and synchronization overhead. Unfortunately, existing escape analysis algorithms are often too expensive to be applicable in just-in-time compilation contexts. We propose to perform the analysis ahead of time and ship its results as code annotations. We present an interprocedural, flow insensitive, static escape analysis that is less precise than traditional escape analyses, but whose result can be transported and verified efficiently. Unlike any other escape analysis that we know of, our method optionally provides for dynamic class loading, which is necessary for full Java compatibility. Benchmarks indicate that, when compared to Whaley and Rinard’s elaborate escape analysis, our simple analysis can pinpoint 81% of all captured allocation sites (69% when dynamic loading is supported), with negligible space overhead for the transport of annotations and negligible time overhead for the verification.

1 Introduction

Just-in-time compilation systems for mobile code do not always use the best available optimization algorithms. Many of the analyses and optimizations that are commonplace in off-line compilers are simply too time-consuming to perform while an interactive user is waiting for program execution to commence. As a result, most just-in-time compilers are skewed towards compilation speed rather than code quality.

Annotation-guided optimization systems [1–6] try to resolve this conflict between compilation speed and code quality. In these systems, analyses are performed off-line and appended to the mobile code as program annotations. This reduces the just-in-time compilation overhead at the code consumer and enables optimizations that would otherwise be too time consuming to perform on-line.

Escape analysis [7–9], a technique that identifies objects that can be allocated on the stack as opposed to on the heap, is a good candidate for annotation-guided optimizations. Escape analysis can also reveal when objects are accessible only to a single thread. This information can then be used to eliminate unnecessary synchronization overhead.

An elaborate escape analysis is time consuming and requires lots of memory for the internal graph representation of each method in a program [7–9].

Benchmarks indicate, however, that its use can result in substantial performance gains, even in case of simpler linear-time analyses [10]. Ideally, we would annotate programs with escape analysis information that can be transported with the program and exploited by an annotation-aware just-in-time compilation system at the target site.

However, there are two primary drawbacks to the use of such annotations for escape analysis: first, they introduce transfer overhead (for the extra annotation information); and second—and more seriously—their use is *unsafe*. That is, if someone accidentally or maliciously changed the escape-analysis result recorded for an allocation site from “heap allocatable” to “stack allocatable”, then the memory safety of the whole target system is potentially in jeopardy.

Hence, one needs to *verify* such annotations, similar to the way that Java bytecode is verified. Verification of traditional escape analysis annotations, however, would essentially be as demanding as performing the original analysis in the first place, negating the original objective of reducing the workload on the code consumer. We are not aware of any prior work on *safe* annotations of escape analysis that would be applicable to Java, i.e., annotations that could actually be verified at the target. All published annotation-based solutions in this domain [2–4] are unsafe.

In this paper, we introduce and evaluate an escape analysis that

- can be performed by the code producer in linear time,
- has a very small space overhead for the annotations,
- can be verified with little time overhead by the code consumer, and
- optionally supports dynamic class loading without the need for deoptimization.

Outline. The rest of the paper is organized as follows. Section 2 introduces our escape analysis. Section 3 explains how to annotate programs and how to efficiently verify these annotations. Section 4 presents our experimental results. Section 5 lists related work. A concluding section summarizes our contributions.

2 Escape Analysis

Escape analysis identifies *captured objects*, i.e., objects with lifetimes that do not exceed that of the method in which they are created. Identification of captured objects enables several optimizations. Most importantly, captured objects can be allocated on the stack avoiding the overhead of heap allocation and garbage collection. Furthermore, all synchronization of captured objects can be eliminated since only a single thread can ever access a captured object. Both optimizations have been shown to improve program performance [7–9] noticeably. Capturedness also enables additional minor optimizations, for example, dead store removal and object inlining, i.e., replacing objects by local variables that represent their fields [10–12].

Table 1. Conceptual source code transformations

	Before	After
Instance Method Declaration	<code>class C {... C_{ret} m(C_{p1} p1,...){ ...} ...}</code>	<code>class C {... C_{ret} m(C this, C_{p1} p1,...){ ...} ...}</code>
Instance Method Call	<code>v₀.m(v₁,...)</code>	<code>m(v₀,v₁,...)</code>
Constructor Declaration	<code>class C {... C(C_{p1} p1,...){ ...} ...}</code>	<code>class C {... void init_C(C this, C_{p1} p1,...){ ...} ...}</code>
Constructor Call	<code>v₀ = new C(v₁,...v_n)</code>	<code>v₀ = new C; init_C(v₀,v₁,...v_n)</code>

Commonly, escape analysis is achieved by constructing a variant of a *points-to-graph* that models object lifetimes and object aliasing. Based on this model, the analysis indicates which objects are captured by the method in which they are created. Whaley and Rinard’s escape analysis [7] follows this approach.

We propose a different escape analysis technique, one that considers *variables*¹ that are bound to objects instead of the objects proper. Variables are classified as *local variables* or *formal variables* (also called *parameters*). Intuitively, we consider a variable *captured* if it is never returned from its defining method, is passed only to captured parameters of called methods, and is never assigned to an escaping variable. If an object during its lifetime is only ever referenced by captured variables, then the object is captured in the traditional sense. So, given an allocation site

```
v = new C();
```

the allocated *object* *o* is captured if the *variable* *v* is captured. This holds because by our definition “no object can escape through *v*” and *v* is the initial reference to *o*; therefore *o* cannot escape. This does not imply that objects that are pointed to by the fields of *o* are also captured. Indeed, we conservatively assume that all field references *v.f* escape.

Arrays are handled like objects and array elements are treated as object fields. Therefore, a one-dimensional array can be captured, but its elements escape, and multi-dimensional arrays can only be captured in their first dimension since they are modelled as nested one-dimensional arrays in Java. An array allocation site

```
v = new C[];
```

allocates a captured array—just like for regular objects—if *v* is captured according to our analysis.

¹ Also often called *references*.

For the rest of our analysis it is beneficial to consider slightly transformed source code rather than the original Java source code. The transformation is illustrated in Table 1. First, we make the otherwise implicit declaration of the `this` parameter explicit in order to treat all parameters uniformly. See the transformations for instance method declarations and constructor declarations. Instance method invocations are changed correspondingly. Second, assignments of freshly constructed objects to variables are transformed by splitting the creation of new objects into two consecutive statements akin to the treatment of constructor calls in Java bytecode. The first statement returns a reference to the allocated and zeroed object; the second statement calls the corresponding initializer with the otherwise implicit `this` reference as first argument. This allows us to treat initializer calls formally as regular method calls.²

2.1 Run-Time Type Constraints

Generally speaking, our analysis consists of two passes. Each pass builds and then solves a constraint system, which is a set of inequalities simply referred to as *constraints*. The first pass generates constraints for the run-time type $rtt(v)$ of variables v . The second pass uses the results of the first pass to generate constraints for the escape predicates $esc(v)$; the second pass is the subject of the next section.

The value of the run-time type property $rtt(v)$ is either a class C , uninitialized (\perp), or initialized but unknown (\top). These elements form a flat lattice with partial order \leq and \top being the least upper bound of any two distinct elements. The meaning of $rtt(v) = \top$ is “ v ’s run-time type is its declared type or any subtype thereof”.

The run-time type property must obey certain constraints that are generated according to the rules given in Table 2. Assigning a new object of class C to a variable v lifts $rtt(v)$ to at least C . Therefore, if our analysis encounters another assignment of a new instance of class B to v , then $rtt(v)$ becomes \top unless $B = C$.

Our analysis makes no assumptions about what is passed into a method and, therefore, the run-time type of formal parameters is unknown. We assume conservatively that the run-time type of fields, array elements, and method results is also unknown.

The run-time type constraint corresponding to an assignment $v_0 = v_1$ enforces that “ v_0 has at least v_1 ’s run-time type”. Note that assigning the `null` reference to a variable does not generate any constraint. This means, for example, if v is exclusively assigned the `null` reference then $rtt(v)$ keeps its default value \perp and every use of v could be replaced by `null`.

Each constraint variable $rtt(v)$ in the constraint system that our analysis builds is initialized with the trivial constraint $rtt(v) \geq \perp$. The remaining con-

² If a constructor body does not begin with an explicit constructor invocation we consider the implicit super class constructor call as part of the constructor body unless we are at the root of the class hierarchy.

Table 2. Representative statements with their corresponding constraints where v , v_i stand for local variables or formal parameters, s for static fields, f for instant fields, m for methods, C for classes, and C_i for arbitrary types.

Run-Time Type Constraints	
$v = \text{new } C$	$r_{tt}(v) \geq C$
$C_0 \ m(C_1 \ p_1, \dots) \{ \dots \}$	$\forall \text{ parameters } p_i: \ r_{tt}(p_i) = \top$
$v = s$	$r_{tt}(v) = \top$
$v_0 = v_1.f$	$r_{tt}(v_0) = \top$
$v_0 = v_1[\dots]$	$r_{tt}(v_0) = \top$
$v_0 = v_1$	$r_{tt}(v_0) \geq r_{tt}(v_1)$
$v_0 = m(v_1, v_2, \dots v_n)$	$r_{tt}(v_0) = \top$
Escape Constraints	
return v	$esc(v) = \top$
throw v	$esc(v) = \top$
$s = v$	$esc(v) = \top$
$v_0.f = v_1$	$esc(v_1) = \top$
$v_0[\dots] = v_1$	$esc(v_1) = \top$
$v_0 = v_1$	$esc(v_0) \Rightarrow esc(v_1)$
$v_0 = m(v_1, v_2, \dots v_n)$	$esc(v_0) = \top \ \wedge$ $\forall \text{ parameters } p_i^{m'} \text{ of methods } m' \text{ invocable as } m:$ $esc(p_i^{m'}) \Rightarrow esc(v_i)$

straints are produced by traversing the code and generating the constraints corresponding to the statements. The run-time type constraints system then consists of inequalities of simple terms: constraint variables $r_{tt}(v)$ and constants C_i , \top , and \perp . Note that $r_{tt}(v) = \top$ is equivalent to $r_{tt}(v) \geq \top$.

We are trying to find the solution to the constraint system that has the most specific information, which means the minimal solution to the constraint system. We know that a unique minimal solution exists and that we can find it in linear time since the constraint system is a definite and simple constraint set as defined in [13]. In order to find the minimal solution, we employ a standard worklist algorithm to lift the values according the given constraints. Since the values of the constraint variables change only monotonically and since the lattice height is constant (three in this case) the number of property value changes is linear in the number of constraints. Using pending lists to efficiently implement the processing of dependencies, the overall complexity of the analysis is linear in the number of constraints.

2.2 Escape Constraints

Our analysis specifies the boolean predicate $esc(v)$ for local variables or parameters v . We write $esc(v) = \top$ if the escape predicate is true and $esc(v) = \perp$ if it is false, thereby interpreting the boolean values as a two-point lattice. If $esc(v)$

Table 3. Determining the constraints of escape predicates $esc(v_i)$ for arguments v_i of method calls $m(v_1, v_2, \dots, v_n)$ based on invocable declarations of m . If m is non-static, then the dynamic dispatch is based on v_1 's run-time type. If $rtt(v_1) = C$, then we refer to m 's only invocable method declaration as m_C , otherwise ($rtt(v_1) = \top$) we refer to the potentially multiple invocable methods as $m_{\leq D}$ where D stands for v_1 's declared (compile-time) type (We have not yet implemented the special treatment of package protected methods within sealed packages in the open world case)

final, private, or static methods m	\forall parameters p_i of the only invocable method m' : $esc(p_i) \Rightarrow esc(v_i)$	
$rtt(v_1) = C$	\forall parameters p_i of method m_C : $esc(p_i) \Rightarrow esc(v_i)$	
package protected methods m within a sealed package P	\forall parameters $p_i^{m_{\leq D}}$ of methods $m_{\leq D}$ within package P : $esc(p_i^{m_{\leq D}}) \Rightarrow esc(v_i)$	
all other methods m	Open World \forall variables v_i : $esc(v_i) = \top$	Closed World \forall parameters $p_i^{m_{\leq D}}$ of methods $m_{\leq D}$ invocable as m within the whole program: $esc(p_i^{m_{\leq D}}) \Rightarrow esc(v_i)$

is false we say that v is *captured*. Note the difference from most other escape analyses, which model capturedness of objects, not variables. The meaning of $esc(v) = \top$ is roughly “an object could escape through v ”. Therefore, if an object o is only referenced by captured variables, then o is captured. This does not mean that a captured variable always references a captured object! For example, it is perfectly in accordance with our analysis to assign an escaping variable to a captured variable.

The escape constraints for the relevant source code statements are also given in Table 2. The first five escape constraints define our notion of “directly escaping”. References escape if they are returned, thrown, or assigned to static variables, fields, or array elements. We exclude the assignment of a captured variable to an escaped one since such an assignment by definition might cause the variable to escape.

We assume that all method results escape and we ensure that passing a variable v_i as an argument to method m lets this variable escape if the corresponding parameter p_i escapes. For a language with dynamic method dispatch such as Java, the former necessitates knowledge of the type hierarchy to determine all method declarations m' that could be invoked when calling m . We chose a relatively conservative approximation of what we mean by “invokable”. Table 3 summarizes our notion of invocable methods and the accompanying constraints.

Whichever condition of the left column applies first causes the generation of the escape constraints to the right. If m is a static, private, or final method then there is exactly one invocable implementation; otherwise the invocable methods depend on the run-time type property $rtt(v_1)$ of the target reference. $rtt(v_1)$ is either a specific class C , in which case only C 's declaration of m is invocable, or $rtt(v_1)$ is unknown, i.e., \top , in which case all implementations of m for v_1 's declared type or subtypes thereof are invocable. Note that $rtt(v_1)$ cannot be uninitialized due to Java's definite assignment rules.

This is the only place in our analysis where supporting dynamic class loading makes a difference. In a *closed world* without dynamic loading, a whole-program analysis can inspect all subclasses of v_1 's declared type. In an *open world* in which additional classes can be added dynamically at any time, this is not possible. We have to assume the worst case, so all arguments of m escape.

There is one exception to this worst case scenario. Java's sealed packages ensure that package protected methods, i.e., methods with default access, can be overridden only by the package's classes that are part of their containing JAR file. In the context of our analysis, this results in a closed world assumption for package protected methods within sealed packages. We have not yet taken advantage of this additional knowledge since the expected gain is rather small — only 4% of our benchmarks' methods are declared package protected and similarly approximately 4% of all method invocation sites are package protected.

Note that the run-time type information of the previous pass is needed to determine the implications that depend on the set of invocable methods; this is the sole purpose of the run-time type property in our framework. The second pass also requires knowledge of the class hierarchy, more specifically, of the call graph to determine the invocable methods in cases where $rtt(v_1) = \top$.

Given that $esc(v_0) \Rightarrow esc(v_1)$ is equivalent to $esc(v_0) \leq esc(v_1)$, the escape constraint system can be solved just as the previous constraint system. In the open world scenario, the number of generated constraints is linear in the size N of the program.³ In the closed world scenario, the number of constraints is linear in the size N of the program plus the size G of the call graph measured as the total number of corresponding argument/parameter pairs at all call sites.

2.3 Example

We demonstrate the generation of constraints by means of the code excerpt in Figure 1. The example consists of two Java methods, which are part of an enclosing class `MyClass` (not shown) and which implement a rather typical collection traversal in public method `transformMap` using Java's iterator interface. `transformMap` has a private helper method `transformVal`, which is contrived to demonstrate our analysis. `transformVal` instantiates a `Transformer t` and it invokes its `do` method to transform the `val` object. A `token` object is handed through `transformMap` and through `transformVal` to `t`'s `do` method. Our anal-

³ This ignores the sealed package option mentioned earlier.

```

public Map transformMap(Map m, Object token)
{
    Iterator iter;
    Object key;
    Object val;

    iter = m.keySet().iterator();
    while (iter.hasNext())
    {
        key = iter.next();
        val = m.get(key);
        m.put(key, transformVal(val, token));
    }
    return m;
}

private Object transformVal(Object val, Object token)
{
    Transformer t = new Transformer();
    return t.do(val, token);
}

```

Fig. 1. Example Java source code excerpt

ysis will show that `token` and `t` will not escape unless the `do` method lets them escape.

Figure 2 shows the example code after the described code transformation is performed. This is the code used for generating the constraints. Table 4 lists the constraints generated by our two-pass analysis for the open world scenario. The annotations in Figure 2 are the minimal solution to the constructed constraint system. For this solution we assumed that the `Transformer`'s constructor captures its self reference and that its `do` method does not let its arguments escape.

3 Annotations and Verification

In order to transport the analysis results, we annotate local and formal variables v at their declaration site with their $esc(v)$ predicate and their run-time type property $rvt(v)$. Figure 2 shows the annotations resulting from the solved constraint system of our running example. Note again that it is not necessary to annotate the captured allocation sites proper (line 26 of Figure 2).

Verifying the annotations only involves generating and verifying the constraints according to Table 2 while traversing the code. (As expected, all annotations in our example satisfy the constraints of Table 4.) Of course, the run-time system needs to provide access to invocable methods for the check of escape constraints of method arguments. Note that verification can happen incrementally

```

1 public Map transformMap(
2     MyClass this,          // Annotation: esc(this) = ⊥, rtt(this) = ⊤
3     Map m,                // Annotation: esc(m) = ⊤, rtt(m) = ⊤
4     MyToken token)       // Annotation: esc(token) = ⊥, rtt(token) = ⊤
5 {
6     Iterator iter;        // Annotation: esc(iter) = ⊤, rtt(iter) = ⊤
7     Object key;           // Annotation: esc(key) = ⊤, rtt(key) = ⊤
8     Object val;           // Annotation: esc(val) = ⊤, rtt(val) = ⊤
9
10    iter = iterator(keySet(m));
11    while (hasNext(iter))
12    {
13        key = next(iter);
14        val = get(m, key);
15        put(m, key, transformVal(this, val, token));
16    }
17    return m;
18 }
19
20 private Object transformVal(
21     MyClass this,          // Annotation: esc(this) = ⊥, rtt(this) = ⊤
22     Object val,            // Annotation: esc(val) = ⊥, rtt(val) = ⊤
23     MyToken token)       // Annotation: esc(token) = ⊥, rtt(token) = ⊤
24 {
25     Transformer t;        // Annotation: esc(t) = ⊥, rtt(t) = Transformer
26     t = new Transformer();
27     initTransformer(t);
28     return do(t, val, token);
29 }

```

Fig. 2. Example with annotations for local and formal variables that result from solving the constraints of Table 4

and on-demand, in no way interfering with the workings of a JIT compiler. If, during verification, one of the constraints fails to be satisfied by the provided annotations, then this implies that the program or the annotations have been tampered with after the analysis was performed.

Verification only ensures only that the annotations are a valid solution of the constraint system. The code consumer will not notice if the received annotations are suboptimal, or even just the trivial solution where everything escapes.

In both the open and the closed world scenarios we analyze the appropriate library methods that are called by the subject program and we require that their parameter annotations satisfy the constraints of the subject program at verification time. In practice, this means that the authors of libraries have to commit to certain escape analysis annotations in order to maintain binary compatibility with respect to the escape analysis. Of course, the run-time system can always

Table 4. Constraints generated from sample program. Variables are indexed to indicate their method, where index M stands for `transformMap` and V for `transformVal`. Duplicate constraints are listed only for the first line in which they appear

Source Line	Run-time Type Constraint	Escape Constraint
2	$r_{tt}(\mathbf{this}_M) = \top$	
3	$r_{tt}(m_M) = \top$	
4	$r_{tt}(\mathbf{token}_M) = \top$	
10	$r_{tt}(iter_M) = \top$	$esc(iter_M) = \top$
10		$esc(m_M) = \top$
13	$r_{tt}(key_M) = \top$	$esc(key_M) = \top$
14	$r_{tt}(val_M) = \top$	$esc(val_M) = \top$
15		$esc(\mathbf{this}_V) \Rightarrow esc(\mathbf{this}_M)$
15		$esc(val_V) \Rightarrow esc(val_M)$
15		$esc(token_V) \Rightarrow esc(token_M)$
21	$r_{tt}(\mathbf{this}_V) = \top$	
22	$r_{tt}(val_V) = \top$	
23	$r_{tt}(token_V) = \top$	
26	$r_{tt}(t_V) \geq \mathbf{Transformer}$	
27		$esc(\mathbf{this}_{\mathbf{Transformer}}) \Rightarrow esc(\mathbf{this}_V)$
28		$esc(\mathbf{this}_{Do}) \Rightarrow esc(t_V)$
28		$esc(val_{Do}) \Rightarrow esc(val_V)$
28		$esc(token_{Do}) \Rightarrow esc(token_V)$

revert to the “everything escapes” worst-case assumption if it encounters an unmet assumption; but due to cascading effects in our analysis this can result in potentially disruptive deoptimization of the system.

Even though our implementation augments canonicalized abstract syntax trees, the annotations should be easily adaptable to Java classfiles. This would provide a low-impact addition of escape analysis optimizations for existing JVMs.

4 Evaluation

To evaluate our framework, we first compared the accuracy of our escape analysis to that of the most precise known escape analysis, namely that of Whaley and Richard [7]. To do this, we compared the number of static allocation sites generated by both techniques. This comparison also discusses allocations within loops and measuring static versus dynamic allocations. Next we estimate the overhead incurred by adding our annotations to standard Java classfiles. We conclude the experimental section by examining the speed of verification.

We based our evaluation on a series of standard benchmarks. We chose sections 2 and 3 of JavaGrande [14] and a subset of SPECjvm98 [15] as listed in Table 5. These are the benchmarks for which source code was readily available or for which we were able to derive the source code from the provided class files. Section 1 of the JavaGrande benchmarks was not analyzed because it evaluates the performance of specific virtual machine features and is therefore not a representative benchmark.

Table 5. Description of benchmarks

JavaGrande Benchmarks		SPECjvm Benchmarks	
crypt	IDEA encryption	jess	Java Expert Shell System
heapsort	Integer sorting	raytrace	3-dimensional ray tracer
fft	Fast Fourier Transform	db	Database benchmark
lufact	LU Factorization	javac	Java compiler
sor	Successive over-relaxation	jack	Java parser generator
sparse	Sparse matrix multiplication		
series	Fourier coefficient analysis		
euler	Computational fluid dynamics		
moldyn	Molecular dynamics simulation		
montecarlo	Monte Carlo simulation		
raytracer	3-dimensional ray tracer		
search	Alpha-beta pruned search		

We perform a rather Java-specific transformation on string concatenations in order to attain more comparable results with the Java bytecode-based implementation of Whaley and Rinard. The standard way of dealing with string concatenation in Java bytecode is to generate a series of `append` method calls to an anonymously allocated `StringBuffer` object. Given the statement

```
String s = s1 + s2;
```

the concatenation operation creates a new `String` object `s` that is the string concatenation of `s1` and `s2`. Primarily for efficiency purposes, the Java compiler converts the above statement into the following:

```
String s = new StringBuffer(s1).append(s2).toString();
```

The latter statement necessitates the allocation of a `StringBuffer`, creating additional allocation sites that are not present in our canonicalized abstract syntax tree. However, we need a common basis for a precise comparison of our analysis to Whaley and Rinard. Therefore, we perform the same transformation and, knowing that the `StringBuffer` does not escape, mark the newly allocated object as captured.

4.1 Efficacy

Our method is by design less accurate than more traditional escape analyses at denoting captured objects. This loss of precision is acceptable because we gain verifiability. Likewise, by introducing an open world assumption, we lose accuracy, but gain the ability to dynamically load classes without invalidating our escape analysis annotations. We base our comparison with Whaley and Rinard on static coverage of allocation sites because this seems to be the best basis for a meaningful comparison. (We will discuss the omission of dynamic numbers at the end of this section.) As explained in Section 2, we consider an allocation site as captured if the new object is assigned to a captured variables. Each benchmark was analyzed under both open and closed world assumptions.

Table 6. Comparison of captured allocation sites

Benchmark	Whaley & Rinard		Our Analysis			Rel. Capt.	
	Sites	Closed W.	Sites	Closed W.	Open W.	C.W.	O.W.
crypt	11	4 (36%)	11	3 (27%)	3 (27%)	75 %	75 %
fft	8	5 (63%)	8	5 (63%)	5 (63%)	100%	100%
heapsort	5	3 (60%)	5	2 (40%)	2 (40%)	67 %	67 %
lufact	8	3 (38%)	8	3 (38%)	3 (38%)	100%	100%
series	10	8 (80%)	10	8 (80%)	8 (80%)	100%	100%
sor	5	2 (40%)	5	2 (40%)	2 (40%)	100%	100%
sparsematmult	8	2 (25%)	8	2 (25%)	2 (25%)	100%	100%
euler	39	11 (28%)	39	10 (26%)	10 (26%)	91 %	91 %
moldyn	7	2 (29%)	7	2 (29%)	2 (29%)	100%	100%
montecarlo	41	22 (54%)	41	18 (44%)	13 (32%)	82 %	59 %
raytracer	46	9 (20%)	46	6 (13%)	5 (11%)	67 %	56 %
search	18	8 (44%)	19	8 (42%)	8 (42%)	95 %	95 %
raytrace	129	55 (43%)	129	29 (22%)	13 (10%)	53 %	24 %
jess	433	164 (38%)	436	155 (36%)	141 (32%)	94 %	85 %
db	41	30 (73%)	41	28 (68%)	21 (51%)	93 %	70 %
jack	209	123 (59%)	214	114 (53%)	105 (49%)	91 %	83 %
javac	760	200 (26%)	750	145 (19%)	121 (16%)	73 %	61 %
Total	1777	651 (36%)	1777	540 (30%)	464 (26%)	81 %	69 %

The results of Whaley and Rinard’s analysis were obtained by inserting a custom counting pass into the `PointerAnalysis` package of the FLEX research compiler [16]. This custom counting pass utilized the points-to-graph to determine the escapedness for each analyzed allocation site. The FLEX compiler infrastructure begins its analysis with the `main` method of a Java classfile, and only analyzes methods that are reachable from there. We have attempted to limit our analysis to only these methods to present a more accurate comparison. Semantic differences between source and bytecode cause a small variation in the total number of allocation sites between their results and ours.

Table 6 shows the number of allocation sites that allocate captured objects relative to the total number of allocation sites. On average, Whaley and Rinard mark 36% of the static allocation sites as captured.

As expected, their comprehensive analysis marks a higher percentage of static allocation sites captured. Our escape analysis fares quite well, however, marking on average 30% allocation sites captured. Perhaps more interesting is how well the algorithm performed in the open world case, covering an average of 26% of the static allocation sites.

The final columns of Table 6 show the coverage our algorithm achieves when compared with Whaley and Rinard. For each benchmark, the percentage is the number of allocation sites that our analysis marks as captured compared against the number of sites their analysis marks as captured. On average, we cover 81% of the static allocation sites in the closed world, and 69% in the open world. This means that even with dynamic class loading, we can still find and

potentially optimize about two thirds of the allocation sites that Whaley and Rinard’s analysis finds—and transmit this information to a just-in-time compiler in a verifiable manner.

Whaley and Rinard’s algorithm explicitly tracks objects through potentially many method calls, which contributes to its higher costs. However, even compared against our analysis under the open world assumption, these higher costs do not seem to materialize in a proportional advantage.

Allocations Inside Loops Object allocations inside loops warrant extra attention since they are potentially executed many more times than allocations outside of loops. To better understand how our analysis performs with respect to loops, we compare the relative number of captured allocation sites inside and outside of loops. (We consider only whether an allocation site is located inside a loop or not, that is, we ignore the nesting level of loops.)

The results in Tables 7 and 8 show the percentage of captured allocation sites located inside versus outside of loops under the closed and open world assumptions, respectively. We found that the percentage of allocation sites captured inside of loops is markedly higher than those captured outside of loops. We believe that this is partly due to our choice of benchmarks. However, it still appears as if objects are normally more short-lived within loops and that our analysis benefits from this. This indicates that the dynamic allocations of captured objects should be higher than our overall averages indicate.

Stack-Allocatability We refrained from trying to obtain dynamic allocation numbers for two reasons. First, we lack the appropriate infrastructure for such experiments. Second, adapting an existing VM would have required us to modify its allocation strategy in a way that allows a meaningful comparison with currently published dynamic allocation measurements. This is not trivial because there are additional factors to consider for stack-allocatability in specific VMs.

To illustrate this point, Gay and Steensgaard [10] used their escape analysis results to implement stack allocation of captured objects in the Marmot VM [17], which prefers a constant frame size. Therefore they introduced an extra predicate, which depends on their use of SSA, to indicate overlapping life ranges of loop allocations. Whaley and Rinard described an implementation of stack allocation based on the Jalapeño compiler [18]. Objects were only stack allocated if the allocation site is executed at most once per invocation of the method. This was to prevent a statically unbounded number of objects from being created on the stack. Both approaches were unsuitable for us since we would have had to provide additional verifiable annotations to guarantee stack-allocatability within a fixed-size stack frame. Neither of these approaches attempted to allocate arrays on the stack since their size is also potentially dynamic.

The most simplistic way to stack allocate all captured objects would be to allow the stack to grow during a method’s lifetime. However, this leads to an unbounded frame size and necessitates an additional frame pointer, which is uncommon among current VMs. Alternatively, one could consider implementing

Table 7. Distribution of captured allocation sites within loops under the closed world assumption

Benchmark	Allocations in loop (AIL)	Alloc. outside loop (AOL)	Captured AIL	Captured AOL	CAIL /AIL	CAOL /AOL
crypt	3 (27%)	8 (73%)	3	0	100%	0%
fft	0 (0%)	8 (100%)	0	5	—	63%
heapsort	2 (40%)	3 (60%)	2	0	100%	0%
lufact	0 (0%)	8 (100%)	0	3	—	38%
series	3 (30%)	7 (70%)	3	5	100%	71%
sor	0 (0%)	5 (100%)	0	2	—	40%
sparsematmult	0 (0%)	8 (100%)	0	2	—	25%
euler	7 (18%)	32 (82%)	0	10	0%	31%
moldyn	2 (29%)	5 (71%)	0	2	0%	40%
montecarlo	6 (15%)	35 (85%)	5	13	83%	37%
raytracer	2 (4%)	44 (96%)	0	6	0%	14%
search	3 (16%)	16 (84%)	3	5	100%	31%
raytrace	14 (11%)	115 (89%)	1	28	7%	24%
jess	71 (16%)	365 (84%)	27	128	38%	35%
db	8 (20%)	33 (80%)	5	23	63%	70%
jack	56 (26%)	158 (74%)	44	70	79%	44%
javac	132 (18%)	618 (82%)	26	119	20%	19%
Total	309 (17%)	1468 (83%)	119	421	39%	29%

Table 8. Distribution of captured allocation sites within loops under the open world assumption

Benchmark	Allocations in loop (AIL)	Alloc. outside loop (AOL)	Captured AIL	Captured AOL	CAIL /AIL	CAOL /AOL
crypt	3 (27%)	8 (73%)	3	0	100%	0%
fft	0 (0%)	8 (100%)	0	5	—	63%
heapsort	2 (40%)	3 (60%)	2	0	100%	0%
lufact	0 (0%)	8 (100%)	0	3	—	38%
series	3 (30%)	7 (70%)	3	5	100%	71%
sor	0 (0%)	5 (100%)	0	2	—	40%
sparsematmult	0 (0%)	8 (100%)	0	2	—	25%
euler	7 (18%)	32 (82%)	0	10	0%	31%
moldyn	2 (29%)	5 (71%)	0	2	0%	40%
montecarlo	6 (15%)	35 (85%)	4	9	67%	26%
raytracer	2 (4%)	44 (96%)	0	5	0%	11%
search	3 (16%)	16 (84%)	3	5	100%	31%
raytrace	14 (11%)	115 (89%)	0	13	0%	11%
jess	71 (16%)	365 (84%)	24	117	34%	32%
db	8 (20%)	33 (80%)	5	16	63%	48%
jack	56 (26%)	158 (74%)	44	61	79%	39%
javac	132 (18%)	618 (82%)	15	106	11%	17%
Total	309 (17%)	1468 (83%)	103	361	33%	25%

Table 9. Comparison of captured object allocation sites that could be stack allocated given static stack frames versus dynamic stack frames in both the closed and open world assumptions

Benchmark	Alloc. Sites	Closed World		Open World	
		Static Frame	Dyn. Frame	Static Frame	Dyn. Frame
crypt	11	0 (0%)	+ 3 (27%)	0 (0%)	+ 3 (27%)
fft	8	3 (37%)	+ 2 (25%)	3 (37%)	+ 2 (25%)
heapsort	5	0 (0%)	+ 2 (40%)	0 (0%)	+ 2 (40%)
lufact	8	2 (25%)	+ 1 (12%)	2 (25%)	+ 1 (12%)
series	10	0 (0%)	+ 8 (80%)	0 (0%)	+ 8 (80%)
sor	5	1 (20%)	+ 1 (20%)	1 (20%)	+ 1 (20%)
sparsematmult	8	1 (12%)	+ 1 (12%)	1 (12%)	+ 1 (12%)
euler	39	9 (23%)	+ 1 (2%)	9 (23%)	+ 1 (2%)
moldyn	7	1 (14%)	+ 1 (14%)	1 (14%)	+ 1 (14%)
montecarlo	41	12 (29%)	+ 6 (14%)	8 (19%)	+ 5 (12%)
raytracer	46	4 (8%)	+ 2 (4%)	3 (6%)	+ 2 (4%)
search	19	0 (0%)	+ 8 (42%)	0 (0%)	+ 8 (42%)
raytrace	129	18 (13%)	+ 11 (8%)	4 (3%)	+ 9 (6%)
jess	436	123 (28%)	+ 32 (7%)	115 (26%)	+ 26 (5%)
db	41	19 (46%)	+ 9 (21%)	16 (39%)	+ 5 (12%)
jack	214	68 (31%)	+ 46 (21%)	59 (27%)	+ 46 (21%)
javac	750	112 (14%)	+ 33 (4%)	102 (13%)	+ 19 (2%)
Total	1777	373 (20%)	+167 (9%)	324 (18%)	+140 (7%)

dynamic regions, which are heap-allocated stacks for each method. In either case, these simpler solutions do not produce dynamic allocation results comparable to published work since they stack-allocate more objects.

Table 9 illustrates the difference between how many objects could be allocated on a static frame size stack versus a dynamic frame size stack. The static frame size numbers are determined by finding all allocation sites that are not contained within a loop and are not array allocations. The dynamic frame size numbers are given as increments over the static numbers and reflect those allocation sites that are within loops, or allocate an array. These results are given for the open and closed world scenarios. In both scenarios it seems that almost one third of the captured allocation sites requires dynamic stack frames in order to be exploitable.

4.2 Annotation Size

The size of the annotations required to encode the escape and run-time type annotations for local variables and formal parameters varies widely depending on which framework is used for the encoding.

A naive implementation based on the annotation framework provided by the Java class file format [19] needs six bytes per method for the `attribute_info` data structure. For each annotated variable, we need to encode the predicate $esc(v)$, which requires one bit (if we are willing to go through the associated

Table 10. Approximate size of escape annotations (in bytes) relative to the size of the original and uncompressed class file

Benchmark	Original Classfile Size	1 Bit + 4 Bytes Encoding	2 Bits Encoding
crypt	5018	338 (7%)	106 (2%)
fft	5164	404 (8%)	104 (2%)
heapsort	2892	202 (7%)	86 (3%)
lufact	6031	684 (11%)	144 (2%)
series	3238	222 (7%)	87 (3%)
sor	2873	216 (8%)	70 (2%)
sparsematmult	3298	200 (6%)	69 (2%)
euler	22447	906 (4%)	186 (1%)
moldyn	10586	462 (4%)	136 (1%)
montecarlo	35793	1600 (4%)	629 (2%)
raytracer	18334	1208 (7%)	379 (2%)
search	10872	538 (5%)	163 (1%)
raytrace	57000	3290 (6%)	1111 (2%)
jess	396393	10194 (3%)	3140 (1%)
db	12087	838 (7%)	227 (2%)
jack	130889	5640 (4%)	1961 (1%)
javac	561462	25652 (5%)	7791 (1%)
Total	1284377	52594 (4%)	16389 (1%)

decoding process), and the property $rtt(v)$, which can either be encoded as a four byte long reference into the class file’s constant table or as one bit, based on the following remarks.

Assuming we allow for some simple code transformations, the property $rtt(v)$ can actually be turned into a boolean predicate $rtt_{bool}(v)$ with an approximate meaning of “ v ’s run-time type is equal to its declared type D ”. The meaning of $rtt(v) = \perp$ is that v is not the target of a non-`null` assignment in the program and all of v ’s occurrences in the program could be replaced by `null`. This transformation allows us therefore to ignore the case of $rtt(v) = \perp$. Now, if $rtt(v) = C \neq D$ then C has to be a subclass of D and all assignments to v are—by definition of our constraints—of run-time type C . Therefore the program is still valid after changing v ’s declared type to C . Given the previous and the latter transformations, $rtt(v)$ is either the declared type D or the unspecified type \top , which are mapped to $rtt_{bool}(v)$ being true or false, respectively. We have not implemented this optimization yet, nevertheless we will consider it for our estimate of annotation overhead.

Table 10 shows the approximate overhead of including annotations in a regular Java classfile. (We are not accounting for the fact that class files are normally compressed as part of a JAR file prior to distribution.) As expected, the numbers indicate that the size increase is relatively insignificant.

Table 11. Time (in milliseconds) to perform the analysis and verification in the closed world case

Benchmark	Analysis Time	Verification Time	No-op Time
crypt	41.37	3.27	0.44
euler	73.28	21.14	3.48
fft	46.57	4.60	0.43
heapsort	39.23	2.08	0.24
lufact	39.60	3.88	0.96
moldyn	49.79	4.40	0.64
montecarlo	111.79	26.10	3.29
raytracer	78.29	12.68	0.81
search	48.11	5.62	0.71
series	39.47	2.40	0.27
sor	40.89	1.90	0.26
sparsematmult	42.14	2.17	0.25
jess	369.28	63.88	9.34
raytrace	141.29	34.68	4.05
db	79.91	12.93	1.20
javac	550.62	137.06	11.23
jack	187.42	55.36	5.39
Total	1979.04	394.15	43.00

4.3 Verification

Verification is the most time critical operation, since it is performed by many code receivers under potentially high time pressure, whereas the analysis is only performed by one code producer under arguably lower time pressure. To demonstrate efficiency of verification, we constructed a verifier that performs all the necessary steps to validate the correctness of the annotations. Table 11 compares the time needed to perform the closed world escape analysis versus time needed to verify the annotations.⁴ Since we argue that verification can easily be integrated into another pass over the code we also provide the timings for a no-op pass that only traverses the program without performing any operations. Therefore, we could look instead at the difference between the verification and no-op times.

On average the verification time is only one fourth of the analysis time. We expect the verification time to drop further after the verification is integrated into the run-time environment. Furthermore, the verifier is simplistic, and could be improved. This comparison does not take into account that, in contrast to the analysis, the verification can be performed in a just-in-time fashion for each method.

⁴ All measurements were performed on a P4/1.8GHz/512MB PC running Java 2 (Blackdown 1.4.1.01 without special flags) on Linux Kernel 2.4.22.

5 Related Work

Lifetime analysis as dealt with in this paper was first described by Ruggieri and Murtagh [20]. The term *escape analysis* was coined by Park and Goldberg [21] in the context of functional languages. Their research spawned work on algorithms which represent the escaping objects by integers [22, 23]. In contrast, the most precise escape analyses for Java use augmented points-to-graphs to model a program’s behavior [8, 7].

With the exception of Gay and Steensgaard’s analysis [10], our (closed world) analysis is the only one we know of that can be performed in time $O(N + G)$ where N is program size and G is the size of the call graph. Overall, our analysis can be viewed as a simple non-SSA variant of Gay and Steensgaard’s algorithm. In comparison, our analysis has less precision. This is mostly due to the fact that they require SSA form and we do not. Their freshness analysis relies crucially on the single assignment property. It is, however, in spirit, close to our run-time type analysis. We employ the run-time type property solely for reducing the potentially invocable methods, whereas they use freshness of returned objects to allocate them on the stack. (In our previous example, their analysis would be able to identify the `iter` variable as captured.) In contrast to us, they suggest employing a whole program analysis such as Rapid Type Analysis to determine the run-time type of the variable on which the method dispatch is performed. While it seems that their analysis is as suitable for the safe transportation of efficiently verifiable annotations as ours, they apparently did not consider this. Furthermore, the use of Rapid Type Analysis hinders the process of just-in-time verification, because it requires the whole program to be available before the analysis can begin.

Our approach is also similar to the phase 1 analysis in Bogda and Hölzle [9] with the major difference that Bogda and Hölzle deal with alias sets instead of variables.

Hartmann et al. [11] enable safe object inlining by extending their SafeTSA code format to include escape analysis annotations. The core difference to our analysis is their assumption about what is known of the run-time environment. We assume knowledge of the libraries used by the code receiver insofar as we presuppose the escape analysis results for these libraries. (Allowing for dynamic class loading in the open world scenario is orthogonal to this assumption.) In contrast, Hartmann et al. assume no knowledge beyond the given program. Within the given program they use a more coarse-grain static analysis, but they compensate for this loss in precision by providing a third kind of type annotation, “may-escape”. In principle, they mark variables as “may-escape” which we mark as escaped in the open world and as captured in the closed world. Of course, to determine their final status these variables require a form of escape analysis at load time and classes loaded later might cause dynamic deoptimization of the whole program.

Hummel et al. [2] employ annotations for register allocation hints in order to improve compilation time and run-time performance. Verifiability is addressed by associating a virtual register with a type and integrating the verification step

with the bytecode verifier. Similarly, Jones et al. [3] describe an annotation for register allocation; however, verifiability of these annotations is not addressed. Finally, Krintz et al. [1] propose an annotation framework, and describe a series of annotations, that are focused on reducing the overhead of dynamic compilation, primarily through hints indicating “hot” methods. Verifiability is not a concern in this framework, because none of the annotations could lead to an incorrect program; at worst, the dynamic compiler will not efficiently optimize the program. Other annotation systems [5, 6] do not address verifiability at all.

The fact that our annotations are verifiable in the same sense that an explicit proof of certain properties, e.g., type safety, is checkable, lies at the heart of similarities with techniques that utilize more explicit proofs. Our verification process does not directly relate to proof-carrying code [24] or certifying compilation [25] since we employ neither verification conditions nor general theorem provers. Our approach is more directly related to credible compilation [26, 27] and translation validation [28] even though these approaches use techniques that are much more comprehensive and more heavyweight than ours. A credible compiler provides for each successfully compiled program a proof showing that the compilation preserved the semantics of the program. Rinard and Marinov [26, 27] employ a two stage process to prove the correctness of program transformations performed by a certifying compiler. The first stage proves the correctness of the analysis results and the second stage establishes the correctness of program transformations utilizing the result of the first stage. Our annotations are essentially a transmission of first stage analysis results. In our case the proof is so simple that we are not transmitting it, instead it is implied.

6 Conclusion

In this paper, we have presented and evaluated a verifiable escape analysis for object-oriented languages such as Java. We have introduced a simple and flow-insensitive escape analysis that uses an inexpensive intraprocedural type analysis to enhance the interprocedural escape analysis proper. For this analysis, we have introduced low-overhead annotations to communicate the analysis results to the code consumer in an efficiently verifiable manner. Analysis, annotations, and low-cost verification together enable a shift of analysis costs from code consumer to code producer. We have provided experimental evidence that our analysis has an acceptable efficacy compared to other state-of-the-art analyses and that transport and verification are indeed efficiently achievable.

We have presented an open world variant of our escape analysis that is not invalidated in the presence of dynamic class loading and that is, therefore, amenable for integration into standard-conforming Java virtual machines. To our knowledge, no other escape analysis provides this feature.

Acknowledgements. This work evolved out of earlier joint work with Chandra Krintz and Vivek Haldar [29]. We are thankful to Alexandru Sălcianu for his help with the FLEX compiler and to Urs Hölzle, the anonymous referees, Peter

Fröhlich, Cristian Petrescu-Prahova, Christian Probst, Jeffery von Ronne, and Vasanth Venkatachalam for their comments.

This effort is partially funded by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-99-1-0536, by the National Science Foundation under grants CCR-0205712 and CCR-0105710, and by the Office of Naval Research under grant N00014-01-1-0854.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and should not be interpreted as necessarily representing the official views, policies or endorsements, either expressed or implied, of Defense Advanced Research Projects Agency (DARPA), the National Science foundation (NSF), the Office of Naval Research (ONR), or any other agency of the U.S. Government.

References

1. Krintz, C., Calder, B.: Using annotations to reduce dynamic optimization time. In: Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation, Snowbird, Utah (2001) 156–167 *SIGPLAN Notices*, 36(5), May 2001.
2. Azevedo, A., Nicolau, A., Hummel, J.: Java annotation-aware just-in-time compilation system. In: ACM Java Grande Conference. (1999) 142–151
3. Jones, J., Kamin, S.: Annotating Java class files with virtual registers for performance. *Concurrency: Practice and Experience* **12** (2000) 389–406
4. Pominville, P., Qian, F., Vallee-Rai, R., Hendren, L., Verbrugge, C.: A framework for optimizing Java using attributes. In: Sable Technical Report No. 2000-2. (2000)
5. Grant, B., Mock, M., Philipose, M., Chambers, C., Eggers, S.J.: DyC: an expressive annotation-directed dynamic compiler for C. *Theoretical Computer Science* **248** (2000) 147–199
6. Reig, F.: Annotations for portable intermediate languages. In Benton, N., Kennedy, A., eds.: *Electronic Notes in Theoretical Computer Science*. Volume 59., Elsevier Science Publishers (2001)
7. Whaley, J., Rinard, M.: Compositional pointer and escape analysis for Java programs. In: Proceedings of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, Denver, CO (1999)
8. Choi, J., Gupta, M., Serrano, M., Shreedhar, V., Midkiff, S.: Escape analysis for Java. In: ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). (1999)
9. Bogda, J., Hölzle, U.: Removing unnecessary synchronization in Java. In: ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). (1999)
10. Gay, D., Steensgaard, B.: Fast escape analysis and stack allocation for object-based programs. In: *Compiler Construction 2000*, Berlin, Germany (2000)
11. Hartmann, A., Amme, W., von Ronne, J., Franz, M.: Code annotation for safe and efficient dynamic object resolution. *Electronic Notes in Theoretical Computer Science* **82** (2003)
12. Lhoták, O., Hendren, L.: Run-time evaluation of opportunities for object inlining in Java. In: Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande (JGI-02), New York, ACM Press (2002) 175–184

13. Rehof, J., Mogensen, T.Æ.: Tractable constraints in finite semi-lattices. In Cousot, R., Schmidt, D.A., eds.: Third International Static Analysis Symposium (SAS). Volume 1145 of Lecture Notes in Computer Science., Springer (1996) 285–301
14. Java Grande Forum: The Java Grande Forum benchmark suite (2003)
15. Standard Performance Evaluation Corporation: SPEC JVM98 benchmarks. See online at <http://www.spec.org/osg/jvm98> for more information (1998)
16. Sălcianu, A.: Pointer analysis and its applications for Java programs. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA (2001)
17. Fitzgerald, R., Knoblock, T.B., Ruf, E., Steensgaard, B., Tarditi, D.: Marmot: an optimizing compiler for Java. *Software—Practice and Experience* **30** (2000) 199–232
18. Alpern, B., Attanasio, C.R., Barton, J.J., Cocchi, A., Hummel, S.F., Lieber, D., Ngo, T., Mergen, M., Shepherd, J.C., Smith, S.: Implementing Jalapeño in Java. In: Proceedings of the ACM SIGPLAN '99 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA). (1999)
19. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Second edn. Addison-Wesley, Reading, MA, USA (1999)
20. Ruggieri, C., Murtagh, T.P.: Lifetime analysis of dynamically allocated objects. In: Conference Record of the Conference on Principles of Programming Languages, ACM SIGACT and SIGPLAN, ACM Press (1988) 285–293
21. Park, Y.G., Goldberg, B.: Escape analysis on lists. In: Proceedings of the 5th ACM SIGPLAN Conference on Programming Language Design and Implementation. (1992) 116–127
22. Deutsch, A.: On the complexity of escape analysis. In: Conference Record of POPL '97: The 24TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM SIGACT and SIGPLAN, ACM Press (1997) 358–371
23. Blanchet, B.: Escape Analysis for Java(TM). Theory and Practice. *ACM Transactions on Programming Languages and Systems* **25** (2003) 713–775
24. Necula, G.C.: Proof-carrying code. In: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Paris, France (1997) 106–119
25. Necula, G.C., Lee, P.: The design and implementation of a certifying compiler. In: Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada (1998) 333–344 *SIGPLAN Notices* 33(5), May 1998.
26. Rinard, M.: Credible compilation. Technical Report MIT/LCS/TR-776, MIT (1999)
27. Rinard, M., Marinov, D.: Credible compilation with pointers. In: Proceedings of the FLoC Workshop on Run-Time Result Verification, Trento, Italy (1999)
28. Necula, G.C.: Translation validation for an optimizing compiler. *ACM SIGPLAN Notices* **35** (2000) 83–94
29. Franz, M., Krintz, C., Haldar, V., Stork, C.H.: Tamper-proof annotations, by design. Technical report, Department of Information and Computer Science, University of California, Irvine (2002)