A Practical Escape and Effect Analysis for Building Lightweight Method Summaries ⋆

Sigmund Cherem and Radu Rugina

Computer Science Department
Cornell University
Ithaca, NY 14853
{siggi,rugina}@cs.cornell.edu

Abstract. We present a unification-based, context-sensitive escape and effect analysis that infers lightweight method summaries describing heap effects. The analysis is parameterized on two values: k, indicating the heap depth beyond which objects escape; and b, a branching factor indicating the maximum number of fields per object that the analysis precisely tracks. Restricting these parameters to small values allows us to keep the method summaries lightweight and practical. Results collected from our implementation shows that the analysis scales well to large code bases such as the GNU Classpath libraries. They also show that summaries can help analysis clients approximate the effects of method calls, avoiding expensive inter-procedural computations, or imprecise worst-case assumptions.

1 Introduction

In the presence of method calls and heap allocation, program analyses must reason about the potential effects of invoking methods. When faced with this problem, typical analyses choose one of the two standard approaches: either perform an expensive inter-procedural analysis; or use a worst-case approximation of the possible method effects. The former affects the scalability and modularity of the analysis, whereas the latter can affect its precision. A middle ground is to summarize method behavior using effects [1] or other forms of method summaries, thus avoiding the costly inter-procedural computations or imprecise assumptions, at the expense of requiring method summaries to be provided from an external source. Summaries can be either supplied by a user and checked by the compiler; or computed automatically by a separate inference engine.

This paper proposes a practical inference algorithm that extracts lightweight method summaries (or signatures) to describe heap effects in Java programs. Our summaries serve as a foundation for other analyses, to approximate the effects of method calls. They can also be regarded as types; as such, they can be used to statically enforce a desired side-effect discipline, or for program understanding purposes.

This paper makes three contributions. First, it proposes lightweight method summaries in the form of *effect signatures* that concisely describe heap aliasing and heap access effects. Signatures provide information about objects being read or written, about returned objects, as well as about the aliasing effects of the method. Object are referred

⋆ This work was supported in part by NSF grants CCF-0541217 and CNS-0406345.

to by their reachability from the parameters. Our signatures use k-limiting [2] to bound the heap depth in signatures. Objects beyond the k limit escape the k-bounded heap and are conflated into a $\top$ value denoting the rest of the heap. We distinguish between objects reachable through different fields, but we use a branching factor b to limit the number of outgoing fields per object. The key aspect of our approach is that the k and b parameters control the size of the signatures: small values of these parameters make the signatures lightweight. In our experience, small signatures are humanly readable, and can help programmers quickly understand the overall heap behavior of methods without exploring their code.

The second contribution is a flow-insensitive unification-based, context-sensitive analysis that infers method summaries for given values of parameters k and b. For each method, the analysis computes two signatures: a static signature, describing the effects of calling that method; and a virtual signature, describing the effects of calling the method, or any method that overrides it.

Third, the paper presents an evaluation of method signatures. We perform a case study of effect signatures for the entire GNU Classpath Java libraries version 0.92, and present signature statistics. We also perform case studies involving two dataflow analysis clients that use method signatures for the analysis of method calls. Our results indicate that both analyses benefit from using method summaries, but the benefits of using higher values of the parameters depends on the client.

The rest of the paper is organized as follows. Section 2 introduces the effect signatures. Section 3 presents our escape analysis algorithm. Section 4 presents experimental results. Finally, we discuss related work in Section 5 and conclude in Section 6.

2 Effect Signatures

The effect signature of a method describes the heap objects that the method accesses (i.e., reads or writes), and the aliases it might create. Method signatures use the notion of k-limiting [2] to bound potion of the heap that they accurately model. More precisely, objects reachable from static fields or through more than k field accesses from the method parameters are conflated together, and their access or alias effects are combined. We say that these objects escape the k-limited heap area of the current method.

This definition extends the traditional notion of escaped objects. Escape analyses commonly divide objects in three categories: a) those that never escape the current method's scope, e.g., objects unreachable outside the method; b) those that may escape the current method's scope but can be *captured* in the callers, e.g., returned objects; and c) those that may escape the current scope and are unrecoverable, e.g., objects stored in static fields. In our work, objects beyond the k-limit are also unrecoverable.

For clarity, we first discuss signatures that do not distinguish between different kinds of accesses (i.e., read vs. write), or between different object fields. The discussion about accesses and field-sensitivity follows afterward.

Consider a method m with parameters p_0, ..., p_n and return value r. The k-level effect signature of m uses at most $k+1$ *attributes* for each parameter and for the return value, and is written as follows:

$$m \,:\, (\alpha_{0,0}, .., \alpha_{0,i_0}) \times ... \times (\alpha_{n,0}, .., \alpha_{n,i_n}) \rightarrow (\alpha_{r,0}, .., \alpha_{r,i_r})$$

Each attribute $\alpha_{i,j}$ is either $\perp$, $\top$, or a symbolic value. Bottom values $\perp$ correspond to parameters that have non-reference types (e.g., `void`, `int`, `char`, etc). Top values $\top$ correspond to objects that escape the k-limited heap. Symbolic values α describe abstract sets of objects (or regions). Different symbols refer to disjoint sets of objects. For each parameter p_j, the attributes in its sequence $\alpha_{j,0}, .., \alpha_{j,i_j}$ correspond to the accessed objects at depth 0, 1, .., i_j from p_j. If p_j has at most k attributes ($i_j < k$) and $\alpha_{j,i_j} \neq \top$, then the method does not access objects deeper that i_j levels from p_j. If p_j has $k+1$ attributes, the last attribute must be $\top$, indicating that the method accesses objects reachable from p_j that are beyond the k limit. Multiple occurrences of the same attribute in the signature indicate aliasing effects. Below we give several examples to illustrate how method signatures capture different forms of effects.

Heap-escape effects. Consider $k = 1$. The following is the signature of a method that doesn't escape its first parameter to the heap, but escapes its second parameter, and returns an escaped reference:

$$m \;:\; \alpha \times \top \rightarrow \top$$

In general, if the first attribute of a parameter is a value $\alpha \neq \top$, then the parameter does not escape the k-limited area. In addition, if α doesn't occur elsewhere in the signature, the parameter is never store in a heap field.

Allocator methods. The following is the signature of a method that doesn't store its first parameter in the heap, and allocates and returns a fresh object. This is the typical signature of `toString` methods in Java:

$$\texttt{toString} \;:\; \alpha \rightarrow \beta$$

In general, if the first attribute of the returned value is a symbol that doesn't occur elsewhere in the signature, then the method behaves as an object allocator.

Returned parameters. Returned parameters are described using the same attribute for the parameter and the return value. For instance, method `append(int)` in class `StringBuffer` returns its parameter, without storing it into the heap:

$$\texttt{append} : \alpha \times \perp \rightarrow \alpha$$

Signatures can also express cases where the returned value is one among several parameters. For instance, method `max` in `java.math.BigDecimal` can return either of its two arguments. This behavior is described as follows:

$$\texttt{max} : \alpha \times \alpha \rightarrow \alpha$$

Heap accesses. Method signatures can indicate the portion of the heap that methods access. Consider the following methods:

$$m_1 : \alpha \rightarrow \perp$$
$$m_2 : (\alpha, \top) \rightarrow \perp$$

Neither method escapes its parameter. However, m_1 doesn't access objects other than its first parameter, whereas m_2 does.

Consider the heap effects of functions *set(o,v)*, that assigns *v* to the field f of object o; and *get(o)* that retrieves the value of the same field. The 1-level signatures of these methods are:

$$\begin{aligned} set &: (\alpha, \top) \times \top \rightarrow \bot \\ get &: (\alpha, \top) \rightarrow \top \end{aligned}$$

Neither method escapes the receiver object (the first parameter), but *set* escapes its second argument, and *get* returns an escaped reference. In contrast, 2-level signatures are more accurate and indicate where objects are loaded from or stored into:

$$\begin{aligned} set &: (\alpha, \beta) \times \beta \rightarrow \bot \\ get &: (\alpha, \beta) \rightarrow \beta \end{aligned}$$

Here, α corresponds to the receiver object, and β to any objects that the fields of the receiver may reference during the execution of the method. The object loaded or stored is located one level deep from the receiver object.

In general, method signatures can describe the effects of methods that contain a combination of field load and store operations. For instance, the effects of `m(x,y)` with body "`x.f = y.f; return x`" is described by the following 2-level signature:

$$\mathtt{m} : (\alpha, \gamma) \times (\beta, \gamma) \rightarrow (\alpha, \gamma)$$

Modeling read and write effects. In the signatures above, an attribute α or $\top$ indicates that the method accesses, i.e., reads or writes, the corresponding portion of the heap. We refine our representation by tagging with a label that indicates the kind of access:

– a write access "w" shows that an object field has been written;
– a read-field access "r" indicates that an object field has been read;
– a read-address access "a" shows that the reference of an object has been read.

Tags form a lattice where $a \leq w$ and $a \leq r$, since accessing a field requires reading its reference. An rw tag indicates the combination of r and w tags: $r \leq rw$ and $w \leq rw$. Tags are placed on all attributes, including $\top$, and are shown in superscripts. The signatures of methods *set* and *get* become:

$$\begin{aligned} set &: (\alpha^w, \beta^a) \times \beta^a \rightarrow \bot \\ get &: (\alpha^r, \beta^a) \rightarrow \beta^a \end{aligned}$$

The signatures indicate that references to any of the objects described by β might be read, but their contents are not accessed.

Field sensitivity. We further refine signatures to describe the fields needed to reach objects in the signature. For example, the field-sensitive signature of *set* is:

$$set : (\alpha^w, f : \beta^a) \times \beta^a \rightarrow \bot$$

The signature says that the second argument might alias the field f of the receiver, but not any other field. When multiple fields are used, the signature lists each accessed field.

For example, a method *setFG* that receives two arguments and stores them in the f and g fields of the receiver object, would have the signature:

$$setFG : (\alpha^w, (f : \beta^a \mid g : \gamma^a)) \times \beta^a \times \gamma^a \to \bot$$

To maintain the signatures small we introduce a branching limit b, and restrict all attributes to have at most b different outgoing fields. When methods access more fields than the branching limit, fields are collapsed together, as in the field-insensitive case.

3 Signature Inference Algorithm

The goal of the k-level escape and effect analysis is to compute method signatures for all methods. We consider programs with the following syntax:

$$\begin{array}{rl} \text{Locations:} & loc ::= x \mid x.f \mid C.f \\ \text{Expressions:} & e ::= loc \mid \texttt{null} \mid \texttt{new}\, C \mid m(x_0, .., x_n) \\ \text{Statements:} & s ::= loc = e \end{array}$$

Here, x ranges over variables, f over fields, C over classes, and m over methods. Variables include formal method parameters, denoted p_0, .., p_n. For virtual methods, p_0 is the reference to the receiver object. Expressions include static field accesses $C.f$ and instance field accesses $x.f$. To simplify the presentation, we assume that all expressions have reference types, and that methods always return a value. A throw statement "`throw x`" is represented as "`Exc.exc = x`" where `exc` is a static field of a special class `Exc`; catching an exception "`catch(Exception x)`" is represented as "`x = Exc.exc`"; and a return statement "`return x`" is modeled as an assignment to a special return variable: "`ret = x`". Arrays are modeled as a special field "`[]`".

The algorithm derives two signatures for each method m: a static signature $sig^S(m)$, that models a call to the method itself; and a virtual signature $sig^V(m)$ that models a virtual call that might be dispatched to the method or any of the methods that override it. This is motivated by the `invokespecial` bytecode instruction that statically calls a virtual method. The callee is determined statically using type information, even though the method is virtual and the call could have been dispatched. An important occurrence of this situation is the `<init>` method of `Object`, which is statically called after each object allocation. If one of the `<init>` methods escapes its receiver object, then `Object`'s `<init>` signature would be polluted and each object would escape right after allocation. The use of two signatures automatically solves this issue.

Figure 1 shows the k-level escape and effect analysis. The algorithm is formulated as a constraint-based analysis that uses unifications. The algorithm first performs an initialization, then analyzes each method by visiting each of its statements. Finally, it performs a context-sensitive instantiation of the callees' signatures into their callers.

Initially, each variable x is assigned a fresh attribute α representing the first attribute in its sequence. Each attribute has two pieces of information: the access tag (r, w, or a), and the maximum heap depth from any variable in the current scope. Subsequent attributes are generated *lazily*, as the analysis proceeds and determines that methods access deeper objects in the heap. Attributes are generated on-demand using a successor

Defintions:

$\overline{e}$ = first attribute of e

$sig(m)$ = signature of $m(p_1....p_n)$
$\overline{p_1} \times .. \times \overline{p_n} \rightarrow \overline{\texttt{ret}}$

$succ$ = lazy successor function

$$succ(\alpha_{[j]}, f) = \begin{cases} \alpha'_{[j+1]} & \text{if } \alpha_{[j]} \neq \top \\ & \text{and } j < k \\ \top & \text{otherwise} \end{cases}$$

Checking the branch limit:
If: $|\ \{f \mid hasSucc(\alpha_{[j]}, f)\}\ | > b$
Then: $unify(succ(\alpha_{[j]}, *), succ(\alpha_{[j]}, f))$

Initialization:

For each variable x:
$\overline{x} = \alpha_{[0]}$, fresh attribute $\alpha_{[0]}$ at depth 0

For each field access $x.f$:
$\overline{x.f} = succ(\overline{x}, f)$

For each static field $C.f$:
$\overline{C.f} = \top$

Set successor of $\top$:
$unify(\top, succ(\top, *))$

Recursive Unifications:

If: $unify(\alpha, \beta)$ and
($hasSucc(\alpha, f)$ or $hasSucc(\beta, f)$)
Then: $unify(succ(\alpha, f), succ(\beta, f))$

Intra-procedural constraints:

Build the static signature $sig^S(m)$:
For each assignment $loc = e$:
$unify(\overline{loc}, \overline{e})$ if $e \in \{x,\ x.f,\ C.f\}$
$a \in tag(\overline{x})$ if $e \in \{x\}$
$r \in tag(\overline{x})$ if $e \in \{x.f,\ m(x, ...)\}$
$r \in tag(\top)$ if $e \in \{C.f\}$
$w \in tag(\overline{x})$ if $loc \in \{x.f\}$
$w \in tag(\top)$ if $loc \in \{C.f\}$

Inter-procedural signature constraints:

For each static call $loc = m(y_0, .., y_n)$:
$sig^S(m) \leq \overline{y_0} \times ... \times \overline{y_n} \rightarrow \overline{loc}$

For each virtual call $loc = m(y_0, .., y_n)$:
$sig^V(m) \leq \overline{y_0} \times ... \times \overline{y_n} \rightarrow \overline{loc}$

For each method m : $sig^S(m) \leq sig^V(m)$

If m overrides m' : $sig^V(m) \leq sig^V(m')$

Signature embedding:

$\alpha_0 \times .. \times \alpha_n \rightarrow \alpha_r \leq \alpha'_0 \times .. \times \alpha'_n \rightarrow \alpha'_r$

If there exists μ such that:
$\mu(\top) = \top$
$\mu(\alpha_i) = \alpha'_i,\ \forall i = 0..n, r$
$succ(\alpha, f) = \beta \Rightarrow succ(\mu(\alpha), f) = \mu(\beta)$
$tag(\alpha) \leq tag(\mu(\alpha))$

Fig. 1. The k-level escape and effect analysis. An expression e has an attribute sequence starting with attribute $\overline{e}$. Successors in the sequence are obtained lazily with function $succ()$.

function *succ*. Given an attribute $\alpha_{[j]}$ at depth j, and a field f, function *succ* returns the successor attribute for that field if one exists. Otherwise, it creates a fresh successor $\alpha'_{[j+1]}$ at depth $j+1$. The successor function returns the $\top$ value when the maximum depth $j = k$ has been reached; and collapses all successors when the branching limit b is reached. A special field "*" denotes that all fields have been collapsed.

When the analysis unifies two attributes, it merges their access tags, and takes the maximum heap depth. Unifications are recursive, so that unifying two attributes requires unifying each of their corresponding successors. To ensure laziness, this is done only if at least one of them has a successor. As usual, unifications can be implemented efficiently using union-find structures.

After initialization, the algorithm performs an intra-procedural computation of static method signatures. For each assignment, the analysis unifies the attributes of the expressions in the assignment. It also sets the appropriate access tags, according to the semantics of assignments, as shown in the top right corner of Figure 1.

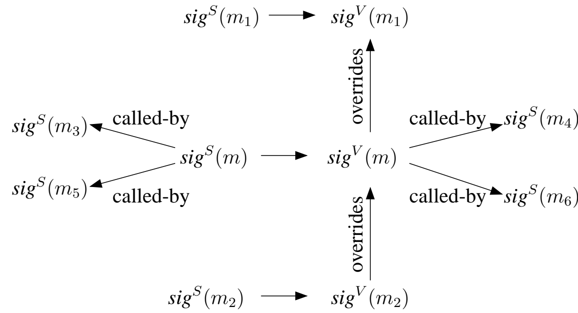

Fig. 2. Dependences between static and virtual method signatures for a method m .

Finally, the analysis runs the inter-procedural part of the algorithm. The analysis imposes two kinds of inter-procedural signature constraints: call-site and overriding constraints. We use a notion of signature embedding (or subtyping) to describe the fact that the effects in a signature are reflected in another signature. We write $sig \leq sig'$ to denote that signature *sig* is embedded in *sig*$'$. Signature embedding is defined in the lower right part in Figure 1. With this definition, the analysis requires that: 1) each static signature of a method must be embedded in the virtual signature of that method; 2) the virtual signature of a method must be embedded in the virtual signature of the method it directly overrides, if any; and 3) at each call site, the appropriate signature of the callee must be embedded into the call site signature. Hence, the analysis uses the overriding relations to determine possible targets at each call. The analysis is context-sensitive: it instantiates the signature of the callee at each call site via embedding.

Inter-procedural signature constraints are graphically illustrated in Figure 2. In the presence of recursive functions, signature constraints become circular and require a fixed-point-computation. The analysis uses a worklist algorithm to solve the constraints. Initially, all methods are added to the worklist. At each step, the algorithm removes a method m from the worklist and enforces the embedding constraints corresponding to the incoming edges to m in Figure 2. If the signature of m changes, then the analysis adds to the worklist the signatures of all methods on the outgoing edges from m.

Treatment of recursive structures. With the analysis presented so far, objects belonging to cyclic structures always escape the k-limited heap, and objects in recursive structures typically escape, too. We briefly sketch an extension that improves the analysis precision in such cases. The idea is to regard the escape signatures as graphs where nodes are attributes, and edges model the successor relations. Using this conceptual representation, signatures in the standard algorithm are k-bounded DAGs. In the extended algorithm, they are k-bounded cyclic graphs. This is done by changing the meaning of depth: the depth of each attribute is the depth in the spanning tree of the graph. The algorithm is changed so that, after the *succ* or *unify* functions are applied, the depths of *all* attributes are recomputed; those beyond the k limit are conflated into $\top$. This extension allows the analysis to treat all elements of a recursive structure such as a list as non-escaping, by representing them using a single non-top attribute having a self loop.

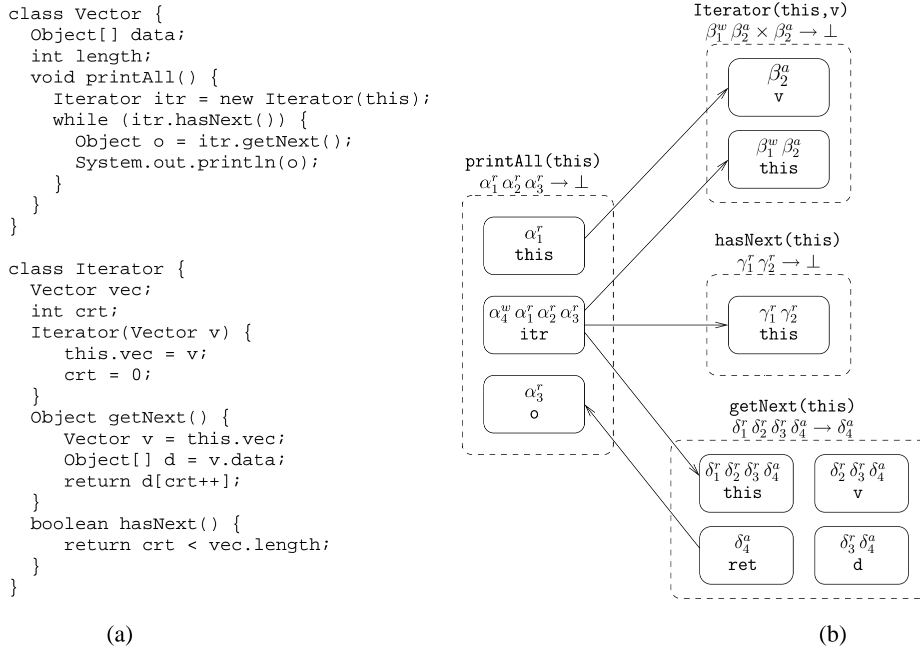

Fig. 3. Escape and effect analysis with $k = 4$, $b = 0$: a) example program; and b) analysis results for each method. The call to `System.out.println()` is omitted; this method reads the content of the object passed in.

Example. Figure 3 shows an example of inter-procedural k-level effect analysis. This example uses a `Vector` class and an `Iterator` for vectors. The `Vector` class has a method `printAll()` that uses an `Iterator` to print all of the elements in the vector. The right part of the figure shows the analysis result for $k = 4$ and $b = 0$ for methods `printAll`, `getNext`, `hasNext`, and for the constructor `Iterator`. For each method, the dashed box groups the variables of that method. The arrows show the inter-procedural constraints at the call sites to `getNext`, `hasNext`, and `Iterator`. The escape attributes of each variable are shown above the variable. Method signatures are shown above each method's box.

The limit of $k = 4$ allows the analysis to precisely identify the accessed objects. For instance, the 4-level signature of `getNext` identifies that the returned object is an object at depth 3 from the receiver; and that no object beyond that level is accessed.

Java Features. We briefly discuss several other Java features.

Interface, abstract, static, and native methods. Interface and abstract methods only have a virtual signature, since they cannot be called statically. Similarly, static methods only have a static signature. For native methods, we manually wrote signatures to model their effects.

Threads and calls by reflection. All objects passed to child threads, or passed as arguments to calls by reflection are marked as escaped. This is done by providing hand-

written signatures for reflection methods (e.g. `forName` in `java.lang.Class`) and forking methods (i.e. `start0` in `java.lang.Thread`).

Dynamic class loading. Our current analysis system is static. However, we believe that support for dynamic class loading could be provided in one of the following two ways: 1) either use signatures as extended method types, include them in class files, and extend the bytecode verifier to type-check signatures at run-time; or 2) extend the JIT compiler and derive new signatures as new classes get loaded.

4 Results

We have implemented the escape and effect analysis in a compilation system for Java using the Soot infrastructure version 2.2.2 [3]. We have conducted several case studies using programs from the SPECjvm98 benchmark suite [4] and the GNU Classpath Java libraries version 0.92 (available at: `www.gnu.org/software/classpath`). Table 1 presents the sizes of the evaluated programs in kilobytes of Java bytecodes, as well as the number of classes and methods in each program. We excluded the SPEC program `mpegaudio` because it uses field names and method names containing non-printable characters, that certain components of our system were not able to handle.

The libraries and the application code were analyzed separately. Our system analyzed libraries first, and generated signature files containing signatures for all library methods. A separate signature file was generated for each parameter combination. The signature files are in text format; they are humanly readable and can be manually edited [1]. Next, when analyzing an application, the system first loaded signatures from the appropriate signature file. Our tool automatically detected when the application changed library method signatures via overriding. This was the case for a few methods in the SPEC test harness that extended classes in `java.io` or `java.awt`. To preserve soundness, we have manually edited the violating signatures (13 in total) and reanalyzed the libraries. No violations were then reported with the new signatures.

The remaining of this section presents three case studies: a study of library methods and two analysis clients. The signatures in these studies use values of k between 1 to 4, and values of b of $0, 1, 2, 5$, and 15. The extension for cyclic and recursive structures discussed in Section 3 has only been used in the first study; in the other two studies it was disabled because it did not impact the analysis clients.

4.1 Case Study 1: Library Methods

The first case study evaluates the analysis of the GNU Classpath Java libraries and presents statistics regarding the method effects in these libraries. In this experiment we used a Dual Xeon 3 Ghz machine with 4Gb of memory, running Linux.

Analysis cost. Table 2 presents the analysis times for the entire GNU Classpath libraries, for different values of k and b. These results demonstrate the scalability of our approach: the analysis can analyze a code base containing more than 50K methods and 16 Mb of Java bytecodes in 1 minute or less. For example, for $k = 3$ and $b = 5$, the

[1] The signature files can be browsed online at: http://www.cs.cornell.edu/projects/frex/sigs

Program	Size(Kb)	Classes	Methods
compress	68	12	44
jess	319	151	690
raytrace	110	25	176
db	67	3	34
javac	579	176	1190
mtrt	110	26	180
jack	180	56	316
Total SPECjvm98	1433	473	2851
GNU Classpath v.092	16815	6586	54436

Table 1. Application sizes.

$b \backslash k$	1	2	3	4
0	23	25	25	26
1	25	27	28	32
2	26	28	35	39
5	33	35	37	43
15	34	41	66	280

Table 2. Analysis times (in seconds) for the GNU Classpath libraries.

analysis takes less than 40 seconds. In comparison, the time needed by Soot just to load the class files from disk and build the intermediate representation is about 6 minutes, an order of magnitude larger. Memory consumption is also a concern when performing whole-program analysis on a very large code base. Loading the intermediate representation requires about 900Mb of memory. The memory needed by our analysis ranged from 100Mb to 300Mb, for different values of the parameters.

Signature information. To better understand the escape behavior of methods in the Java libraries, we have collected statistics about reference parameters and returned values in library methods. Reference parameters can be classified in four categories: escaped, stored, returned, and borrowed. Escaped parameters are reachable from static fields, thrown exceptions, or objects beyond the k-limit. Stored parameters are reachable through one or more field dereferences (within the k-limit) from another parameter. All of the other parameters are borrowed. If a reference is passed into a borrowed parameter of a method, then the method does not create additional copies of that reference.

Return values are divided in five groups: escaped, stored, loaded, parameter, and fresh. Stored return values are objects that, besides being returned, are also reachable from a parameter's field. Loaded return values are objects that were reachable from a parameter's field even before executing the method. Write effects play a key role in distinguishing loaded values from stored values: if an object is represented by an attribute with only read effects, all of the object's fields may be loaded but are never stored. Fresh returned values denote new objects returned by allocator methods.

Table 3 shows the distribution of reference parameters and return values among all signatures generated for the GNU library. The following summarizes our findings:

- A large fraction (roughly 69%) of the method parameters are just borrowed;
- An escape analysis with $k \geq 2$ can identify that about 6% of the parameters are being stored in the field of another parameter;
- Few of the parameters (1%) are returned, and few of the returned objects 4% come from the parameters;
- A large fraction of methods, 42%, are allocator methods that return fresh objects. An additional 14% of the returned objects are loaded from a parameter field;
- Signatures improve for higher values of k and b. However, increasing values of these parameters yield diminishing returns. Signatures can be further improved using the more precise treatment of cycles and recursive structures.

			Parameter values				Returned values				
k	b	model cycles	Escaped	Stored	Returned	Borrowed	Escaped	Stored	Loaded	Returned param.	Fresh
1	0	no	30.7%	0%	0.9%	68.4%	53.8%	0%	0%	4.3%	41.9%
2	2	no	24.3%	5.9%	0.9%	68.9%	39.3%	0.7%	13.6%	4.3%	42.1%
3	5	no	23.2%	6.9%	0.9%	69.0%	36.8%	1.1%	15.5%	4.3%	42.3%
4	15	no	22.3%	7.5%	1.0%	69.2%	35.4%	1.4%	16.2%	4.4%	42.6%
4	15	yes	17.9%	11.8%	1.0%	69.3%	33.9%	2.3%	16.8%	4.4%	42.6%

Table 3. Distribution of parameters and return values in static signatures.

4.2 Case Study 2: Variable Uniqueness Analysis

The second case study evaluates method summaries in the context of a dataflow analysis client aimed at identifying unique variables. A variable is unique if it holds the only live reference to the object it points to. The uniqueness information is used by a compiler to provide compile-time memory management for Java programs, by automatically inserting *free* statements when the program updates a unique variable. In this study, method summaries are used to improve the analysis precision at method calls.

The uniqueness analysis is formulated as a forward dataflow analysis. At each program point, the analysis computes a partition of variables, i.e., a set of disjoint sets of variables. A heap object referenced by a variable in a set can be referenced only by other variables in the same set. When a set contains a single variable, that variable is unique. At each allocation site *x = new()*, the analysis creates a new partition $\{x\}$. The transfer functions remove variables when they are updated; move them to other sets when they are copied; or kill entire sets when variable references are stored into the heap.

In the absence of method signatures, method calls are also treated conservatively: variable sets are killed when a reference in the set is passed as an argument or returned from a method. When method summaries are available, the analysis of method calls is enhanced in three ways. First, calls to allocator method are treated as allocation sites. Second, a set is not killed when a variable in the set is passed as an argument to a method, but the method signature indicates that the corresponding parameter is just borrowed. Finally, the analysis models returned parameters as assignments in the caller.

Evaluation. We ran two versions of the uniqueness analysis, with and without method summaries, and compared the amount of memory freed using these analyses. We also compared these memory savings to those obtained from a sophisticated inter-procedural shape analysis that we previously developed [5]. For a fair comparison, we used the same machine as in our previous work, a 2Ghz Pentium with 1Gb of memory.

When using worst-case assumptions for method calls, the uniqueness analysis took 2 seconds for all the the benchmarks together. The compiler inserted 86 frees, allowing the deallocation of 4% of the total memory. When using the method summaries of $k = 1$ and $b = 0$ to model method calls, the analysis took 3 seconds, inserted 710 free statements, and enabled the deallocation of 36% of the total memory. In comparison, shape analysis [5] is able to reclaim up to 54% of the memory, but it is significantly more expensive, requiring a total of 11 minutes for the analysis of all of the benchmarks. Table 4 shows a breakdown of the memory savings for each benchmark, using each of

Program	Memory (Mb)			Savings (%)		Shape
	Total	Summaries		Summaries		analysis
	allocated	No	Yes	No	Yes	savings[5]
compress	111	111	111	0%	0%	78%
jess	305	305	246	0%	19%	19%
raytrace	161	138	32	14%	80%	81%
db	81	81	37	0%	54%	86%
javac	240	238	230	1%	4%	14%
mtrt	170	147	41	13%	75%	77%
jack	313	313	257	0%	17%	24%
Average				4%	36%	54%
Total analysis time				2sec	3sec	11 min

Table 4. Case Study 2: Maximum memory usage with no GC, and total analysis times.

$b\backslash k$	1	2	3	4
0	6%	6%	7%	7%
1	7%	11%	13%	13%
2	8%	11%	13%	13%
5	8%	11%	14%	14%
15	8%	11%	14%	14%

Table 5. Case Study 3: Additional redundant loads for the SPECjvm98 programs.

the three analyses. For some applications, such as *db* and *jess*, the additional saving when using summaries are due to the ability of recognizing allocator methods. We also experimented with larger values $k > 1$ and $b > 0$, but although the compiler added a few more frees, the memory usage of the transformed programs was unchanged.

In summary, the uniqueness analysis with method summaries runs much faster than a full-blown inter-procedural shape analysis; it is more precise compared to using worst-case assumptions; and higher values of k and b do not bring more benefits.

4.3 Case Study 3: Redundant Loads

The third study is a dataflow analysis aimed at identifying redundant loads. We ran this analysis with and without using method summaries, and compared the total number of identified redundant loads.

The analysis is a forward dataflow analysis that computes a set of available loads at each program point. The analysis is similar to the standard available expressions analysis and uses set intersection as the merge operator. A load statement $x = y.f$ is redundant if $y.f$ is available before the statement.

At method calls, the analysis uses the method signatures to determine the fields that the callees might update. A method updates field f if the signature of that method contains an attribute with write effects and outgoing edge f or "*". The analysis preserves a load $x.f$ if it determines that the callee doesn't update field f.

Evaluation. Table 5 shows the analysis results for the SPECjvm98 benchmarks. For each value of k and b, the table shows the number of additional redundant loads that the analysis has identified, as a percentage of the number of redundant loads identified when using worst-case assumptions at method calls. The results indicate an increase of up to 14% more redundant loads. Unlike in the previous case study, higher values of k and b lead to performance improvements in the client analysis. However, there seems to be no additional improvement beyond $k = 3$ and $b = 5$. For this experiment we used the same machine as in the previous experiment. The analysis time for any parameter combination took less than 2 seconds per benchmark.

Algorithm	FS	CS	Unif	k	b	Method Summaries	Largest Java app. analyzed Name	Size(Mb)
Choi et. al. [8]	✓	✓		∞	∞	✓	Trans	0.5
Whaley, Rinard [9]	✓	✓		∞	∞	✓	Pbob	0.3
Blanchet [10]		✓		∞	0	✓	Jess	0.4
Gay, Steensgaard [11]		✓		1	∞		Marmot	1.5[3]
Bogda, Hoelzle [12]		✓	✓	2	∞	✓	Javac	0.6
Ruf [13]		✓[1]	✓	∞	∞	✓	Marmot	1.5[3]
O'Callahan [14]		✓	✓	∞	∞	✓	Ladybug	0.4
Cherem, Rugina [15]		✓	✓	∞	∞	✓	Javac	0.6
Whaley, Lam [16]		✓		∞	∞		Gruntspud	0.7
Milanova et al. [17]		✓[2]		∞	∞		Soot-1.beta.4	1.1
Sridharan,Bodík [18]		✓		∞	∞		Sablecc-j	2.4
Wilson, Lam [19]	✓	✓		∞	∞	✓	—	—
Liang, Harrold [20]		✓	✓	∞	∞	✓	—	—
Fähndrich et al. [21]		✓	✓	∞	∞		—	—
Lattner,Adve [22]		✓[1]	✓	∞	∞	✓	—	—
This paper		✓	✓	any	any	✓	Classpath v0.92	16.3

Table 6. Comparison of related pointer and escape analyses. Columns indicate: flow-sensitivity (FS); context-sensitivity (CS); unification-based analyses (Unif); the k and b limits; whether method summaries are computed; and the largest application analyzed (only for Java analyses). Notes: [1] context-insensitive within an SCC; [2] object-sensitive; [3] estimated bound.

5 Related Work

Type and effect systems. Type and effect systems have been originally proposed in the seminal work of Gifford and Lucassen on type and effect systems for mostly functional languages [1]. Effect annotations for Java have been proposed in several systems, including JML [6], a specification language that allows specifying pure methods, and assignable locations that a method can mutate; or AliasJava [7], a system that supports an annotation `lent` that indicates non-escaping method parameters. Our effect signatures are richer and cover a larger set of effects compared to these systems. In addition to read/write effects, our signatures can describe method allocator effects, aliasing effects in the shallow heap, or returned parameters. Furthermore, our work focuses on the efficient static inference of such effects, rather than on type-checking effect annotations.

Escape and pointer analysis. Escape and pointer analyses has been an active area of research for many years. A large number of algorithms have been proposed in the past two decades. Table 6 summarizes a relevant subset of these algorithms and classifies them according to their features, shown in the columns of the table. The distinctive feature of our algorithm is that it is parameterized on the values of parameters k and b. These values can be tuned to trade precision for efficiency, and vice-versa.

Most analyses use infinite depth and field branching ($k = b = \infty$). Exceptions include the analysis of Gay and Steensgaard [11] where objects escape when they are stored in the heap, i.e. $k = 1$; the analysis of Bodga and Hoelzle [12], with $k = 2$; and the analysis of Blanchet [10] which identifies objects by their heap depth, hence $b = 0$. In essence, our parameterized escape analysis generalizes all of these analyses.

Escape analyses have been traditionally used to identify objects that do not escape their method or thread scopes, thus enabling stack allocation or synchronization elimination optimizations [11, 9, 12, 10]. Pointer analyses have been mainly concerned with building points-to sets, or resolving alias queries [21, 16–18]. Other pointer analyses have been used to summarize method effects [22, 15, 23]. Our work focuses on this last use of pointer analysis. Therefore, the ability of the analysis to compute and build procedure summaries becomes an important aspect, as effects can be expressed more naturally using summaries. In existing algorithms, method summaries are points-to graphs [20, 13, 22, 15]; type representations of points-to graphs [14]; or pairs of input-output points-to graphs, in the case of flow-sensitive analyses [9, 19, 23]. Pointer analyses that do not compute method summaries require an additional MOD analysis to translate the points-to set information into side-effect information for each method [17]. In contrast to all of the existing analyses that compute method summaries, our analysis has the ability to control the sizes of the summaries, by tuning the values of k and b. For small values of k and b, the analysis becomes scalable and efficient, and signatures become lightweight and humanly readable.

Although all analyses in Table 6 exhibit a certain degree of context-sensitivity, some use restricted forms. Some analyses treat recursive cycles in a context-insensitive manner to avoid fixed-point computations for recursive procedures [13, 22]. For large code bases such as the Java libraries that have large recursive cycles, large portions of the code will end up being analyzed in a context-insensitive manner. Object-sensitive analyses [17] use another restricted form where calling contexts distinguish only the receiver object. Our analysis uses the general, unrestricted notion of context-sensitivity, and uses a context-sensitive heap abstraction.

As shown in the last column of Table 6, our case study on the GNU Classpath libraries involves a whole-program code base larger than those experimented with in previous escape or pointer analysis studies. This demonstrates the scalability of our escape analysis using lightweight summaries.

6 Conclusions

We have proposed lightweight method signatures to summarize heap aliasing and heap access effects. We also have presented a very efficient unification-based, context-sensitive algorithm to derive such signatures. We have demonstrated the scalability of signature inference a large code base, and shown that computed summaries can help analysis clients to approximate the effects of methods calls and avoid worst-case assumptions.

References

1. Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: Proceedings of the Symposium on the Principles of Programming Languages. (1988)
2. Jones, N., Muchnick, S.: Flow analysis and optimization of Lisp-like structures. In: Conference Record of the Symposium on the Principles of Programming Languages, San Antonio, TX (1979)
3. Vallee-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: Soot - a Java optimization framework. In: CASCON '99, Toronto, CA (1999)

4. Uniejewski, J.: SPEC Benchmark Suite: Designed for today's advanced systems. SPEC Newsletter Volume 1, Issue 1, SPEC (1989)
5. Cherem, S., Rugina, R.: Compile-time deallocation of individual objects. In: Proceedings of the International Symposium on Memory Management, Ottawa, Canada (2006)
6. Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G., Leino, R., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer **7**(3) (2005) 212–232
7. Aldrich, J., Kostadinov, V., Chambers, C.: Alias annotations for program understanding. In: Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications, Seattle, WA (2002)
8. Choi, J.D., Gupta, M., Serrano, M.J., Sreedhar, V.C., Midkiff, S.P.: Stack allocation and synchronization optimizations for java using escape analysis. ACM Trans. Program. Lang. Syst. **25**(6) (2003) 876–910
9. Whaley, J., Rinard, M.: Compositional pointer and escape analysis for Java programs. In: Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications, Denver, CO (1999)
10. Blanchet, B.: Escape analysis for Java: Theory and practice. ACM Transactions on Programming Languages and Systems **25**(6) (2003) 713–775
11. Gay, D., Steensgaard, B.: Fast escape analysis and stack allocation for object-based programs. In: Proceedings of the International Conference on Compiler Construction, Berlin, Germany (2000)
12. Bogda, J., Hoelzle, U.: Removing unnecessary synchronization in Java. In: Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications, Denver, CO (1999)
13. Ruf, E.: Effective synchronization removal for Java. In: Proceedings of the Conference on Program Language Design and Implementation, Vancouver, Canada (2000)
14. O'Callahan, R.: Generalized Aliasing as a Basis for Program Analysis Tools. PhD thesis, School of Computer Science, Carnegie Mellon Univ. (2001)
15. Cherem, S., Rugina, R.: Region analysis and transformation for Java programs. In: Proceedings of the International Symposium on Memory Management, Vancouver, Canada (2004)
16. Whaley, J., Lam, M.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: Proceedings of the Conference on Program Language Design and Implementation. (2004)
17. Milanova, A., Rountev, A., Ryder, B.: Parameterized object sensitivity for points-to analysis for Java. ACM Transactions Softw. Eng. Methodol. **14**(1) (2005) 1–41
18. Sridharan, M., Bodík, R.: Refinement-based context-sensitive points-to analysis for java. In: Proceedings of the Conference on Program Language Design and Implementation. (2006)
19. Wilson, R., Lam, M.: Efficient context-sensitive pointer analysis for C programs. In: Proceedings of the Conference on Program Language Design and Implementation. (1995)
20. Liang, D., Harrold, M.: Efficient points-to analysis for whole-program analysis. In: Proceedings of the Symposium on the Foundations of Software Engineering, Toulouse,France (1999)
21. Fähndrich, M., Rehof, J., Das, M.: Scalable context-sensitive flow analysis using instantiation constraints. In: Proceedings of the Conference on Program Language Design and Implementation, Vancouver, Canada (2000)
22. Lattner, C., Adve, V.: Data Structure Analysis: An Efficient Context-Sensitive Heap Analysis. Tech. Report UIUCDCS-R-2003-2340, Computer Science Dept., Univ. of Illinois at Urbana-Champaign (2003)
23. Salcianu, A., Rinard, M.: Purity and side effect analysis for Java programs. In: Proceedings of the Conference on Verification, Model Checking and Abstract Interpretation. (2005)