Information and Software Technology 51 (2009) 1428–1439

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Fast and precise points-to analysis

Jonas Lundberg, Tobias Gutzmann *, Marcus Edvinsson, Welf Löwe

Software Technology Group, School of Mathematics and System Engineering, Växjö University, 351 95 Växjö, Sweden

ARTICLE INFO

Article history:
Available online 6 May 2009

Keywords:
Points-to analysis
Dataflow analysis
Escape analysis

ABSTRACT

Many software engineering applications require points-to analysis. These client applications range from optimizing compilers to integrated program development environments (IDEs) and from testing environments to reverse-engineering tools. Moreover, software engineering applications used in an edit-compile cycle need points-to analysis to be fast and precise.

In this article, we present a new context- and flow-sensitive approach to points-to analysis where calling contexts are distinguished by the points-to sets analyzed for their call target expressions. Compared to other well-known context-sensitive techniques it is faster in practice, on average, twice as fast as the call string approach and by an order of magnitude faster than the object-sensitive technique. In fact, it shows to be only marginally slower than a context-insensitive baseline analysis. At the same time, it provides higher precision than the call string technique and is similar in precision to the object-sensitive technique. We confirm these statements with experiments using a number of abstract precision metrics and a concrete client application: escape analysis.

1. Introduction

Points-to analysis is a static program analysis that extracts reference information from a given input program, e.g., possible targets of a call and possible objects referenced by a field. This reference information is an essential input to many types of client applications in optimizing compilers and software engineering tools.

Examples of such client applications are: *metrics analyses* computing coupling and cohesion between objects [1,2] and architectural recovery by *class clustering* proposing groupings of classes, either based on coupling and cohesion or directly on reference information [3,4]. Source code browsers compute *forward and backward slices* [5] of a program point which, in turn, requires reference information. In *software testing*, class dependencies determine the test order [6–8]. Reverse engineering of UML interaction diagrams requires very precise reference information in order to be useful [9]. Finally, static *design pattern detection* needs to identify the interaction among participating classes and object instances in order to exclude false positives [10].

In real life, where the program to be analyzed may contain hundreds or even thousands of classes, there is a tradeoff between the analysis precision and the analysis time and memory costs. This situation is problematic for user-interactive client analyses, where in many cases response time is essential and a delay of several minutes unacceptable. Especially for these client applications, scalably fast and precise points-to analysis is very much desirable.

The basis for many points-to analysis approaches, and program analysis in general, is the theory of monotone dataflow frameworks [11,12]. A program is represented by a program graph; its nodes correspond to program points, its edges to control and data dependencies between them. The analysis iteratively computes values for each node by merging values from predecessor nodes and by applying transfer functions representing the abstract program behavior at these nodes.

In a *context-insensitive* program analysis, analysis values of different call sites are propagated to the same method and get mixed there. The analysis value is then the merger of *all* calls targeting that method. A *context-sensitive* analysis addresses this source of imprecision by distinguishing between different calling contexts of a method. It analyzes a method separately for each calling context [13]. Context-sensitivity will therefore, in general, give a more precise analysis. The drawbacks are the increased memory cost that comes with maintaining a large number of contexts and their analysis values, and the increased analysis time required to reach a fixed point.

Context-sensitive approaches use a finite abstraction of the top sequence of the call stack possibly occurring at each call site in order to separate different call contexts. The two traditional approaches to define a context are referred to as the *call string* approach and the *functional* approach [14]. The call string approach defines a context by the first k callers, i.e., return addresses on the

* Corresponding author. Fax: +46 470 708021.
E-mail addresses: Jonas.Lundberg@vxu.se (J. Lundberg), Tobias.Gutzmann@vxu.se (T. Gutzmann), Marcus.Edvinsson@vxu.se (M. Edvinsson), Welf.Lowe@vxu.se (W. Löwe).

0950-5849/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2009.04.012

call stack top [15], referred to as the family of k-CFA (Control Flow Analysis). The functional approach uses some abstractions of the call site's actual parameters to distinguish different contexts [14,16]. Both the call string and the functional approaches were evaluated and put into a common framework by Grove et al. [16].

A functional approach designed for object-oriented languages is referred to as *object-sensitivity* [17,18]. It distinguishes contexts by separately analyzing the targeted method for each abstract object in the implicit *this*-parameter. Similarly to k-CFA, a family of k-object-sensitive algorithms distinguishing contexts by the top k abstract target objects on the call stack can be defined. In the two papers [17,18], a *simplified* version of 1-object-sensitivity was evaluated. Here, only method parameters and return values are treated context-sensitively. The authors report, compared to 1-CFA, increased precision of side-effect analysis and, to a lesser degree, call graph construction. Both approaches show similar costs in time and memory. These results generalize to variants where $k > 1$, which, however, are very costly in terms of memory and provide only a small increase in precision [19].

The contributions of this article are the following:

- We present a new functional approach to points-to analysis denoted *this-sensitivity*.
- We experimentally evaluate this-sensitivity by comparing it with two well-known context-sensitive approaches (1-CFA and a *complete* version of 1-object-sensitivity – complete in the sense that it does not merge analysis values of different contexts). Our measurements show that this-sensitivity (i) is twice as fast as 1-CFA and an order of magnitude faster than 1-object-sensitivity, (ii) requires less memory than 1-object-sensitivity, (iii) is more precise than 1-CFA, and (iv) is almost as precise as 1-object-sensitivity.
- We evaluate the precision by using one concrete client application (escape analysis) and two abstract precision metrics suites that cover different granularities and aspects of precision corresponding to two different types of client applications.

The remainder of this article is as follows: In Section 2, we outline *Points-to SSA*, our SSA-based program representation where each method is represented by a sparse method graph. Points-to SSA has been presented elsewhere [20], we include a brief version of this material for the understandability and completeness of this article. In Section 3, we present our context-sensitive *simulated execution* analysis technique where the analysis of a method (in a specific context) is interrupted when a call occurs, and later resumed when the analysis of the called method is completed. Then, our new context-sensitive points-to analysis is presented in Section 4. In Section 5, we present our precision metrics, experimental setup, and results. Finally, in Section 6, we discuss related work and conclude this article in Section 7.

2. Analysis values and program representation

In this section, we introduce the representation of analysis values, which are sets of abstract objects and a heap-memory abstraction, and our program representation called Points-to SSA. They are then used in the actual analysis algorithm, which is described in Section 3.

2.1. Analysis values

In points-to analysis, we need to represent references to abstract objects and an abstraction of the heap-memory.

An *abstract object* o is an analysis abstraction that represents one or more run-time objects. The mapping from run-time to abstract objects is called a *name schema*. In this article, we use the following name schema: each *syntactic creation point* s corresponds to a unique abstract object o_s. Thus, the set of all allocation sites in a program defines a finite set of abstract objects denoted O, and every abstract object $o_s \in O$ can be seen as an analysis abstraction representing *all* run-time objects created at the corresponding allocation site s in *any* execution of the analyzed program.

In the analysis, reference variables will in general hold references to more than one abstract object. Hence, we assume that each *points-to value* v in the analysis of a program is an element in the *points-to value lattice* $L_V = \{V, \sqcup, \sqcap, \top, \bot\}$ where $V = 2^O$ is the power set of O, $\top = O$, $\bot = \emptyset$, and $\sqcup, \sqcap$ are the set operations $\cup$ (union) and $\cap$ (intersection). The height of the points-to value lattice is $h_o = \mid O \mid$. We use the notation $Pt(a)$ to refer to the points-to value that is referenced by the expression a.

Each abstract object $o \in O$ has a unique set of *object fields* $[o,f] \in OF$ where $f \in F$ is a unique identifier of a field (capturing references). Each object field $[o,f]$ is in turn associated with a *memory slot* $([o,f], v)$ where v is a points-to value. A memory slot represents the abstract object references stored in object field $[o,f]$.

The abstraction of the heap-memory associated with an analyzed program, referred to as *abstract memory*, *Mem*, is defined as the set of all memory slots $([o,f], v)$. In our approach, we use a single global memory configuration. Our reason for introducing an abstract memory is not only to mimic the run-time behavior; it is a necessary construct to handle field store and load operations and the transport of abstract objects from one method to another that follows as a result of these operations. We think of the abstract memory as a mapping from object fields to points-to values. The memory is therefore equipped with two operations

$$Mem.get(OF) \rightarrow V \quad \text{and} \quad Mem.addTo(OF, V)$$

with the interpretation of reading the points-to value stored in an object field $[o,f] \in OF$, and merging the points-to value $v \in V$ with the points-to value already stored in an object field $[o,f] \in OF$, respectively. Note that we never override previously stored object field values in memory store operations, i.e., we never execute strong updates. Instead, we merge the new value with the old one using the points-to value lattice's join operation, i.e., we perform weak updates.

The abstract memory is updated as a side effect of the analysis. In order to quickly determine the fixed point, we use memory sizes indicating whether or not the memory has changed. In what follows, we refer to the size of the abstract memory as a *memory size* $x \in X = [0, h_m]$ where h_m is the maximum memory size. It corresponds to the case where all object fields contain all abstract objects, hence, $h_m = \mid OF \mid \cdot \mid O \mid$.

In order to apply the theory of monotone dataflow frameworks to memory size values as well, we introduce a lattice L_X referred to as the *memory size lattice*. The memory size lattice L_X is a single ascending chain of integers, i.e., $L_X = \{X, \sqcup, \sqcap, \top, \bot\}$ where $X = \{0, 1, 2, \ldots, h_m\}$, $\top = h_m$, $\bot = 0$, $x_1 \sqcup x_2 = \max(x_1, x_2)$, and $x_1 \sqcap x_2 = \min(x_1, x_2)$. The height of L_X is h_m.

2.2. Points-to SSA

Points-to SSA is our graph-based program representation. In Points-to SSA, local variables are resolved to dataflow edges connecting operations (nodes) that define variables to operations (nodes) that use these variables. As a result, every def-use relation via local variables is explicitly represented as an edge between the defining and using operations. Join-points in the control flow where several definitions may apply are modeled with special φ-nodes using possible definitions valid in the different branches and introducing new definitions.

```
public class L {
  V value = null;
  L next = null;

  public L (V v) {
    value = v;
  }

  public void append(V v) {
    if (next == null)
      next = new List(v);
    else
      next.append(v);
  }

  public void putAt(int n, V v) {
    int count = 0;
    L l = this;
    while (count < n) {
      l = l.next;
      count++;
    }
    l.value = v;
  }
}
```

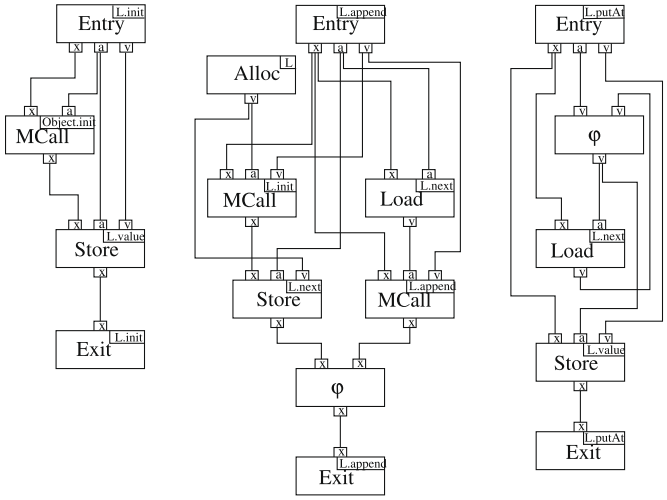


Fig. 1. Source code fragment and corresponding Points-to SSA graphs.

Fig. 1 shows a simple "Linked List" implementation (class L) and the corresponding Points-to SSA graphs. Each method is represented by a graph and each node in the graph represents an operation in the method. We have for example Entry and Exit nodes representing method entry/exit points, and Store and Load nodes representing field write/read operations. The ports at the top of a node represent operation input values (e.g., memory size x, target address values a, and the values v to store in the Store nodes) and the ports at the bottom represent operation results (e.g., a new memory size x in the Store nodes). Edges connecting node ports represent the flow of values from defining nodes (operation results) to using nodes (operation input values). More details regarding these notations will be presented later on.

Notice that the constructor L.init starts by calling its super constructor Object.init and that object creation, in L.append, is done in two steps: we first allocate an object of class L and then call the constructor L.init. φ-nodes are used in L.append to merge the memory size values from the two selective branches, and in L.putAt as the loop head of the iteration.

A Points-to SSA method graph can be seen as an abstraction of a method's semantics, an SSA graph representation specially designed for points-to analysis. It is an abstraction since we have removed all operations not directly related to reference computations, e.g., operations related to primitive types. Moreover, we abstracted from the semantics of the remaining operations by giving them an abstract analysis semantics.

Another feature of Points-to SSA is the use of memory edges to explicitly model (direct, indirect, and anti-) dependencies between different memory operations. An operation that may change the memory *defines* a new memory size value, and operations that may access this updated memory *use* the new memory size value. Thus, memory sizes are considered as data, and memory size edges have the same semantics – including the use of φ-nodes at join points – as def-use edges for other types of data. The introduction of memory size edges in Points-to SSA is important since they also imply a correct order in which the memory accessing operations are analyzed, which ensures that the analysis is a *flow-sensitive* abstraction of the semantics of the program. Flow-sensitivity will be discussed in Section 3.4.

A Points-to SSA method graph is now defined as a directed and ordered multi-graph $G = \{N, E, Entry, Exit\}$, where N is a set of Points-to SSA nodes, E is a set of Points-to SSA edges, *Entry* is a graph entry node satisfying $|pred(Entry)| = 0$, and *Exit* is a graph exit node satisfying $|succ(Exit)| = 0$.

The reference-related semantics of different language constructs (e.g., calls and field accesses) are described by a set of *operation node types*. Each node type n has a number of *in-ports* $in(n) = [in_1(n), \ldots, in_k(n)]$, and a number of *out-ports* $out(n) = [out_1(n), \ldots, out_l(n)]$. The in-ports represent input values to the operation in question, whereas the out-ports represent the results produced by the operation. All ports have a fixed *type* (V or X) and a current *analysis value* of that type ($v \in L_V$ or $x \in L_X$).

An edge $e = out_i(src) \rightarrow in_j(tgt)$ connects an out-port of a node *src* with an in-port of a node *tgt*. An edge may only connect out- and in-ports of the same type. An out-port $out_i(n)$ may be connected to one or more outgoing edges. An in-port $in_j(n)$ is always connected to a single incoming edge. The last property reflects our underlying SSA approach – each value has one, and only one, definition.

Certain node types have attributes that refer to node specific, static information. For example, each $Alloc^C$ node is decorated with a class identifier C that identifies the class of the object to be created.

Finally, each type of node is associated with a unique *analysis semantics* (or *transfer function*) which can be seen as a mapping from in-ports to out-ports that may have a side-effect on the memory. As an example, Algorithm 1 shows the analysis semantics for the $Store^f$ node, which abstracts the actual semantics of a field write statement $a.f = v$.

Algorithm 1. $Store^f : [x_{in}, a, v] \mapsto x_{out}$

$x_{out} = x_{in}$
for each $o \in Pt(a)$ **do**
  $prev = Mem.get([o,f])$
  if $v \not\sqsubseteq prev$ **then**
    $Mem.addTo([o,f], v)$
    $x_{out} = Mem.getSize()$
  end if
end for
return x_{out}

For each abstract object o in the address reference a, we look up the points-to value previously stored in object field $[o,f]$. If the new value to be stored changes the memory (i.e., if $v \not\sqsubseteq prev$), we merge v with the previous value and save the result. Notice also that we compute a new memory out-port value (a new memory size) if the memory has been changed during this operation.

3. Context-sensitive simulated execution

Our dataflow analysis technique, called *simulated execution*, is an abstract interpretation of the program based on the abstract analysis and program representation discussed in the previous section. It simulates the actual execution of a program where the analysis of a method is interrupted when a call occurs, and later resumed when the analysis of the called method was completed.

The simulated execution approach can be seen as a recursive interaction between the analysis of an individual Points-to SSA method graph and the transfer function associated with monomorphic calls, which handle the transition from one method to another. Polymorphic calls are handled as selections over possible target methods m_i, which are then processed as a sequence of monomorphic calls targeting m_i. The context-sensitive approach described here is a modification of the context-insensitive approach described in previous work [20]. The most noticeable difference is that we associate each monomorphic call targeting a method m with a number of contexts, and process m separately for each such context.

In Section 3.1, we describe the processing of individual method graphs. In Section 3.2, we describe the analysis of calls in a context. In Section 3.3, we present our context-sensitive handling of monomorphic calls. Our approach to intra- and inter-procedural flow-sensitivity is discussed in Section 3.4.

3.1. Method graph processing

For each method graph, we have a pre-computed node order that is determined by the data and memory dependencies between the nodes. We compute a topological sorting for forward edges. To order the nodes in loops, we use a so-called *interval analysis* [21,22] where we identify inner and outer loops and their loop heads (always φ-nodes).

The method processing starts in the method entry node, follows the node ordering, and iterates over loops until a fixed point is reached. Inner loops are stabilized before their outer loops. Consequences of this approach are: (1) All nodes in a method graph g_m are analyzed at least once every time method m is analyzed. (2) All nodes, except the loop head φ-nodes, have all their predecessor nodes updated before they are analyzed themselves. (3) The order in which the nodes are analyzed respects all control and data dependencies and is therefore an abstraction of the control-flow of an actual execution. The final point is a crucial step to assure flow-sensitivity in the SSA-based simulated execution technique.

The above properties of analyzing single method graphs is taken into consideration by *processMethod* as given in Algorithm 2. It should only be considered as a rough outline of the approach actually implemented. The idea is simple: We start by initializing the method entry node with the method input to be used in this particular method *activation*. We then analyze the method nodes repeatedly until we reach the method exit node. Therefore, we *compute* a node's transfer function given by the node type, *update* the successor in-ports, and determine the *next* node to analyze to get its values stable.

Algorithm 2. *processMethod* : $(m, [x_{in}, a, v_1, \ldots, v_n]) \mapsto [x_{out}, r]$

$n = m.entryNode$
$in(n) = [x_{in}, a, v_1, \ldots, v_n]$
do
  $n.computeTransferFunction()$
  $n.updateSuccs()$
  $n = n.next()$
while $n \neq m.exitNode$
return $in(n)$

pThe transition from one method to another is embedded in the statement *n.computeTransferFunction()* if n is of a monomorphic call type ($MCall^{m,cs_i}$). Note that the processing of a call in turn may lead to the analysis of the call target method m as defined in *processMethod*.

3.2. Call processing

Our approach to analyzing individual calls (see Algorithm 3) describes the handling of a call to method m in a context ctx^m. For the understanding of our call processing, it is safe to assume that all calls to m are associated with only *one* context ctx^m, i.e., that we perform a context-insensitive analysis. This is generalized to more contexts in Section 3.3.

The processing of (recursive) method calls must guarantee that the analysis terminates and that the analysis values reach a global fixed point.

The crucial step to ensure termination is that each context ctx^m is associated with two attributes $prev\_args$ and $prev\_return$ where we store previous input and return values of the calls to m in that context ctx^m. The former of these attributes is used to decide whether we have seen a more general call targeting m in the same context ctx^m before, i.e., if $[x_{in}, a, v_1, \ldots, v_n] \sqsubseteq prev\_args$, in which case we interrupt the call processing and reuse the previous result from $prev\_return$.

Algorithm 3. $processCall(ctx^m, [x_{in}, a, v_1, \ldots, v_n]) \mapsto [x_{out}, r]$

– if ctx^m was already analyzed with larger parameters before
if $[x_{in}, a, v_1, \ldots, v_n] \sqsubseteq ctx^m.prev\_args$ **then**
  return $ctx^m.prev\_return$
end if
$ctx^m.prev\_args = ctx^m.prev\_args \sqcup [x_{in}, a, v_1, \ldots, v_n]$
– if ctx^m is on the analysis stack
if $ctx^m.is\_active$ **then**
  $ctx^m.is\_recursive = true$
  return $ctx^m.prev\_return$
end if
$ctx^m.is\_active = true$
$[x_{out}, r] = processMethod(m, ctx^m.prev\_args)$
– if ctx^m was not recursively called within *processMethod*
if $\neg\ ctx^m.is\_recursive$ **then**
  $ctx^m.prev\_return = [x_{out}, r]$
  $ctx^m.is\_active = false$
  return $[x_{out}, r]$
end if
– while ctx^m's recursive call results haven't reached fixed point
while $ctx^m.prev\_return \sqsubset [x_{out}, r]$ **do**
  $ctx^m.prev\_return = [x_{out}, r]$
  $[x_{out}, r] = processMethod(m, ctx^m.prev\_args)$
end while
$ctx^m.is\_recursive = false$
$ctx^m.is\_active = false$
return $[x_{out}, r]$

The alternative, a call targeting m in ctx^m with new arguments, leads to a new method activation where we process the target method m by invoking *processMethod* using the merged input $prev\_args \sqcup [x_{in}, a, v_1, \ldots, v_n]$. We also update the two attributes $prev\_args$ and $prev\_return$ in preparation for the next call targeting m in ctx^m.

Termination of our analysis is ensured since we incrementally merge our arguments $prev\_args \sqcup [x_{in}, a, v_1, \ldots, v_n]$ before we start processing a method m. Thus, the sequence of arguments $args_i$ used for a given context ctx^m forms an ascending chain satisfying

$$args_0 \sqsubset args_1 \sqsubset \ldots \sqsubset args_n.$$

Each such chain must have finite length since our value lattices have finite heights (both L_X and L_V are finite). Thus, each method can only be processed a finite number of times, and analysis termination is guaranteed. This argument also holds for calls involving recursion; terminations is guaranteed for these programs as well.

In order to guarantee that the fixed point is reached, especially in loops induced by recursive method calls, we need a few more attributes associated with each context: *is_active* is used to check if we are processing a call in a context that is currently being analyzed, i.e., if m is called recursively in ctx^m. In this case, we directly return *prev_return* for the recursive call and undefined $[0, \perp]$ if we have no previous results, respectively. Also we set *is_recursive* = *true* which indicates, upon return from *processMethod*, that we have seen a recursive call during *processMethod*. In this case, we need to stabilize the results by iteratively reinvoking *processMethod* until the fixed point is reached.

3.3. Transfer function of context-sensitive calls

The transfer function of monomorphic call nodes $MCall^{m,cs_i}$ is given in Algorithm 4. It completes the definition of the analysis semantics of a call from a call site $cs_i : r = a.m(v_1, \ldots, v_n)$. The algorithm first selects a set of contexts for each call: $selectContextsFor : [m, cs_i, a] \mapsto [ctx_1, \ldots, ctx_q]$. It creates new contexts if and only if they have not been created before when processing similar calls. This method will be described in more detail in Section 4.1 where we discuss different variants of context-sensitivity (leading to different implementations of *selectContextsFor*). It is, for the time being, sufficient to know that each call can be associated with a number of different contexts, i.e., we perform a (not yet further specified) context-sensitive analysis.

We assume that each context ctx^m is aware of the corresponding points-to value for the implicit variable *this*. In short, it is a singleton abstract object set $\{o^i\}, o^i \in Pt(a)$ for the 1-object-sensitive analysis, and the whole set $Pt(a)$ for context-insensitive, 1-this-sensitive and 1-CFA. This information is embodied in the assignment $this = ctx^m.getThis()$ that we are using to simplify the notations.

Algorithm 4. $MCall^{m,cs_i} : [x_{in}, a, v_1, \ldots, v_n] \mapsto [x_{out}, r]$

$Context[]ctxs = selectContextsFor(m, cs_i, a)$
$[x_{out}, r] = [0, \perp]$
for each $ctx^m \in ctxs$ **do**
  $this = ctx^m.getThis()$
  $args = [x_{in}, this, v_1, \ldots, v_n]$
  $[x_{out}, r] = processCall(ctx^m, args) \sqcup [x_{out}, r]$
end for
return $[x_{out}, r]$

Algorithm 4 lists the transfer function for MCall nodes. It computes all contexts *ctxs* and merges the analysis results of the individual calls to the target method m in each context ctx^m as returned from *processCall*.

3.4. Flow sensitivity

Flow-sensitivity is a concept that is frequently used, but there is no consensus as to its precise definition [23]. Informally, an analysis is flow-sensitive if it takes control-flow information into account [24]. Many people also require the use of so-called *strong* (or *killing*) updates as a criteria for flow-sensitivity [13]. Strong updates occur when an assignment supersedes (or kills the results of) an earlier assignment. The problem with strong updates is that they are only permitted if the ordering of the reads and writes of a given variable is sure and if the variable identifies a unique memory location. For local variables, these cases can be detected using a *def-use* analysis, i.e., an analysis that computes for every *definition* of a variable all *uses* of that variable along a definition free control-flow path.

Our SSA-based analysis has local (intra-procedural) flow-sensitivity in the strictest sense since our use of an SSA representation incorporates the def-use information needed to identify all places where strong updates of local variables are possible. That dataflow analysis on an SSA-based representation implies local flow-sensitivity has been demonstrated by Hasti and Horwitz [25].

Furthermore, our simulated execution based analysis has global (inter-procedural) flow-sensitivity in a more general sense since a memory accessing operation (call or field access) $a_1.x$ will never be affected by another memory access $a_2.x$ that is executed after $a_1.x$ in all runs of a program. This makes simulated execution strictly more precise than the frequently used flow-insensitive whole program points-to graph approach [16–19,26]. This statement was verified by experiments by Lundberg and Löwe, who also showed that a strict ordering of the two approaches, from a performance point of view, is impossible [20]. However, the SSA-based simulated execution approach was, on average, 22% faster.

4. Context sensitivity

The way we associate a call $a.m(\ldots)$ with a number of contexts under which the method m shall be analyzed depends on the call stack abstraction used. Each such abstraction defines a family of different context-sensitive analyses that can be parameterized by a call stack depth k. We only consider the case $k = 1$ in this article, hence, we can base the abstraction on the topmost stack frame, i.e., on the target and the return addresses and the actual call parameters of a call site. The case $k > 1$ is a matter of future work. However, previous experiments with 2-CFA and 2-object-sensitivity show a much increased memory cost but only a small increase in precision [19]. We have no reason to believe that our new approach should behave differently.

In this section, we present four different context definitions, denoted *Insens*, *CallSite*, *ObjSens*, and *ThisSens*. The former three represent the well known context-*in*sensitive, 1-CFA, and 1-object-sensitive approaches. The latter one is our new context-sensitive approach, *1-this-sensitivity*.

4.1. Context definitions

A *context definition* is a rule that associates a call with a set of contexts under which the target method should be analyzed. Actually, *ObjSens* is the only context definition (in this selection) that may associate a call with more than one context. Each context is in turn defined by a tuple; the tuple elements, its number and content, depend on what context definition we are using. In this article, we will use the following context definitions for a given call from a call site $cs_i : \ a.m(v_1, \ldots, v_n)$ where $Pt(a) = \{o^1, \ldots, o^p\}$.

Insens: $cs_i \mapsto \{(m)\}$
All calls targeting method m are mapped to the same context. This is the context-insensitive baseline approach.
CallSite: $cs_i \mapsto \{(m, cs_i)\}$
Calls from the same call site cs_i are mapped to the same context.
ObjSens: $cs_i \mapsto \{(m)\}$ if *m.isStatic*,
$\{(m, o^1), \ldots, (m, o^p)\}$ otherwise.
Calls targeting the same receiving abstract object $o^i \in Pt(a)$ are mapped to the same context. Static calls are handled context-insensitively.

ThisSens: $cs_i \mapsto \{(m)\}$ if *m.isStatic*,
$\{(m, Pt(a))\}$ otherwise.
Calls targeting the same points-to set $Pt(a)$ are mapped to the same context. Static calls are handled context-insensitively.

This-sensitivity (*ThisSens*) is to our knowledge new. In contrast to object-sensitivity, which analyzes a method separately for each *abstract object* reaching the implicit *this*-variable, this-sensitivity analyzes a method separately for each *set of abstract objects* reaching *this*. For example, given a (non-static) call $a.m(v_1)$ with $Pt(a) = \{o^1, o^2\}$, *ThisSens* would map it to the single context $(m, \{o^1, o^2\})$, whereas *ObjSens* would map it to the two contexts $(m, \{o^1\})$ and $(m, \{o^2\})$. We discuss the differences between these two approaches in more detail in the following.

4.2. Object- and this-sensitivity

The object-sensitive approach has been thoroughly studied during the last years, and there seems to be an agreement that this technique is particularly well suited for the analysis of object-oriented programs [17–19]. The difference between our new this-sensitivity and object-sensitivity is that we are using a different context definition for a given call site. Therefore, we compare the two approaches in more detail. 1-CFA is left out of the discussion since it has been compared to object-sensitivity before [18,19].

4.2.1. Analysis precision

It is not obvious which of the two techniques, object- or this-sensitivity, is more precise. In what follows, we will present two scenarios where one technique provides higher precision than the other and vice versa. This proves that neither of the two approaches is strictly more precise than the other.

The first example shows a situation where this-sensitivity provides higher precision:

Example 1. Method m:

$m(V\ v)$ {return v;}$\mapsto V$

Call 1:

$Pt(a_1) = \{o_a^1\},\ Pt(v_1) = \{o_v^1\}$
$r_1 = a_1.m(v_1)$

Call 2:

$Pt(a_2) = \{o_a^1, o_a^2\},\ Pt(v_2) = \{o_v^2\}$
$r_2 = a_2.m(v_2)$

We have two consecutive calls targeting the same method m, which just returns the provided argument. The two calls target expressions a_1 and a_2, whose respective points-to sets both contain the abstract object o_a^1, and a_2 also o_a^2. In an object-sensitive analysis, both calls target the context (m, o_a^1), and the return values get mixed in the second call:

$$Pt(r_1) = \{o_v^1\} \quad \textit{and}\ Pt(r_2) = \{o_v^1, o_v^2\}.$$

Here, the points-to set v_1 from call 1 gets mixed into the points-to set of r_2, as the call $a_2.m()$ is analyzed under both the contexts $(m, \{o_v^1\})$ and $(m, \{o_v^2\})$.

In a this-sensitive analysis, the two calls target different contexts:

$$Pt(r_1) = \{o_v^1\} \quad \text{and}\ Pt(r_2) = \{o_v^2\}.$$

Here, $a_2.m()$ is analyzed under the single context $(m, \{o_v^1, o_v^2\})$, so that no mixing of return values occurs.

The second example shows a situation where object-sensitivity provides higher precision:

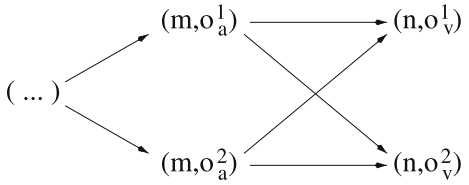
Example 2. Method m:

$m(\ )\ \{V\ v = this.f;\ v.n();\}$

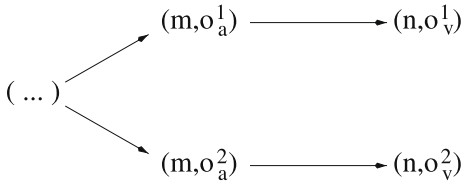
Call:

$Pt(a) = \{o_a^1, o_a^2\}$
$Pt([o_a^1, f]) = \{o_v^1\}$
$Pt([o_a^2, f]) = \{o_v^2\}$
$a.m()$

Here, the method m reads from memory and calls another method n on the read-result. In the this-sensitive approach, the call $a.m()$ targets the context $(m, \{o_a^1, o_a^2\})$ that reads $\{o_v^1, o_v^2\}$ from memory and then calls the context $(n, \{o_v^1, o_v^2\})$. This gives the upper of the following two call graphs.



This-sensitive call graph.



Object-sensitive call graph.

In the object-sensitive approach, the call $a.m()$ is analyzed under two different contexts (m, o_a^1) and (m, o_a^2), which read $\{o_v^1\}$ and $\{o_v^2\}$, respectively, and then call the contexts (n, o_v^1) and (n, o_v^2), respectively. This leads to the lower of the two object call graphs given above. That is, the object-sensitive approach gives a more precise result in this example since the results of the memory read operation $v = this.f$ never get mixed in this scenario.

Thus, neither of the two approaches is strictly more precise than the other. Our experiments show that these approaches have similar precision in practice, cf. Section 5. However, these two examples nicely illustrate the fundamental differences between the two approaches. This-sensitivity, with the potential to use many more contexts for a given method, is better in separating two different calls targeting the same method. Object-sensitivity, where each abstract object is treated separately, has a more precise handling of operations (calls and field accesses) targeting the implicit variable *this*.

4.2.2. Analysis cost

Object-sensitivity may use a number of contexts polynomial in the number of abstract objects, whereas this-sensitivity might use an exponential number of contexts and requires therefore, in theory, an exponential amount of memory and time. In practice, however, our experiments show that the number of contexts used by this-sensitivity is (on average) less than half of the number used by object-sensitivity. Furthermore, a low number of contexts used does not only reduce the memory requirements, but also speeds up the analysis by reducing the amount of processing required to reach the fixed point. This non-obvious experimental observation

is one of the major reasons why this-sensitivity is an order of magnitude faster than object-sensitivity, cf. Section 5.8.

Finally, it is straightforward to add an *ad hoc* mechanism that recognizes situations in which the number of contexts explodes. For instance, when a given method has been associated with more than N contexts, N any constant that appears appropriate, the analysis may be widened, e.g., by merging contexts. This is the approach that should be used in any non-experimental implementation of this-sensitivity, but since none of the programs studied in this article shows exponential behavior for this-sensitivity, we omit a deeper discussion of such possible mechanisms.

5. Experiments

In this section, we evaluate our new approach, this-sensitivity, by comparing it with 1-CFA and 1-object-sensitivity. In fact, we compare our implementations thereof, which we refer to as *ThisSens*, *CallSite*, and *ObjSens*, respectively. The differences between the theory and our implementation are discussed in Section 5.1.

We have defined a set of general precision metrics relevant for a large number of different client applications. More precisely, we tried to identify two types of client applications that require different granularity of reference information as their input. These two types are presented in Section 5.2 along with relevant metrics for each type. Section 5.3 presents escape analysis, a concrete client analysis that requires points-to information. Our experimental setup is presented in Section 5.4, and the results of our experiments are discussed in Sections 5.5–5.8.

5.1. Implementation details

Our Java implementation of the presented analysis reads and analyzes Java bytecode. We use the *Soot* framework, version 2.3.0, as our bytecode reader [27]. We then use the *Shimple* format provided by *Soot* as the starting point to construct the SSA-based graphs for the individual methods.

In the context-sensitive analysis, all contexts are equipped with a node-to-values map where all current analysis values are saved. Using this approach, we avoid cloning the method graphs. Moreover, our points-to set implementation is similar to the hybrid-set implementation that comes with *Soot*, and we make sure that we never save multiple copies of identical sets.

We use stubs to simulate the behavior of native methods. We have a conservative handling of exceptions, threads, and methods in the Java class library dealing with array manipulation (e.g., `java.lang.System.arraycopy`). Our analysis implementation is currently incomplete in the following sense: (1) It does not handle features related to dynamic class loading and reflection correctly. To our knowledge, no feasible approach to handling these features is known. (2) Native methods returning `String` and `StringBuffer` objects are treated like allocation sites. Methods returning `void` are considered to be side-effect free, and all other (non-`void`) methods return $\perp$.

5.2. Two types of client applications

The reference information that can be extracted from a program using static points-to analysis is in most cases used as input to different *client applications*. These client applications can be further divided into different *domains* such as compiler optimizations, software development, and reverse engineering. In this section, we will take an orthogonal approach and try to focus on what *granularity* of reference information the client applications need rather than their domain.

The first type of client application that we have identified, denoted *SourceCode* clients, is primarily interested in source code entities and reference relations between them, i.e., in relations between source code entities like classes, methods, fields, and statements, that hold for any execution of the program and for all instances of a class. Examples are all client applications that require a call graph as input, i.e., most types of inter-procedural program analysis. Other examples of *SourceCode* clients are: *virtual call resolution* to avoid dynamic dispatch and facilitate method inlining [19,28], *cast safety analysis* to avoid unnecessary run-time type checking [19,29], *metrics-based analyses* to compute coupling and cohesion metrics involving members and classes [1,2], *source code browsers* in IDEs [30,31] that need to resolve various source code references, and *software testing* where class dependencies determine the test order [6–8].

Another type of client application, denoted *ObjectIdentity* clients, is primarily interested in individual objects and references to individual objects. Examples of *ObjectIdentity* clients are: *side-effect analysis* that computes the set of object fields that may be modified during the execution of a statement [17,18,32], *escape analysis* that identifies method-local (or thread-local) objects to improve garbage collection (and to remove synchronization operations) [33–35], *Memory leak debugging* to identify references that prevent garbage collection [26], static *design pattern detection* to identify the interaction among possible participating objects [10], reverse engineering of UML *interaction diagrams* [9], and architectural recovery by *class clustering* to avoid erroneous groupings of classes/instances [3,4].

5.2.1. Application entities

In order to avoid taking into account results due to the same set of Java library and Java Virtual Machine (JVM) start-up classes over and over again, we decided to use the following method when applying our metrics suites on the results of the points-to analysis: we selected a subset of all classes in each benchmark program and denoted them *application classes*. A simple name filter on the fully qualified class names did this job. For example, the application classes of `xsltcl.2` are all those classes having a name starting with `org.apache`. Members defined in these classes are denoted *application members* and abstract objects corresponding to allocations of these classes are denoted *application objects*. We did not consider any class from the Java standard library as an application class in any of the benchmark programs.

5.2.2. The SourceCode metrics suite

The set of precision metrics presented here is most relevant for client applications of type *SourceCode*. These metrics are frequently used when evaluating different approaches to points-to analysis.

- *Node, Edge*: The number of nodes (methods) and edges (calls) in a call graph where at least one of the participants (caller or callee) is an application method.
- *PCall*: The number of potentially polymorphic call sites located in an application class.
- *Cast*: The number of casts (located in an application class) potentially failing at run-time.

The call graph related metrics, *Node* and *Edge*, are relevant for inter-procedural analyses. The other two are related to *method inlining* and *cast safety analysis*.

5.2.3. The ObjectIdentity metrics suite

The set of metrics presented here is most relevant for client applications of type *ObjectIdentity*.

Table 1
Benchmark information and context-insensitive results.

Program	*General*				*SourceCode*				*ObjectIdentity*				*Escape*	
	Class	Method	Object	Time (s)	Node	Edge	PCall	Cast	ONode	OEdge	Heap	Enter	Esc	Esc*
Antlr	420	2222	4716	4.6	882	3099	408	52	7274	328060	167255	50915	1997	98927
Bloat	646	4504	8177	137.1	3007	17468	1049	1138	57418	8203696	589118	1443312	3322	840927
Chart	1002	6262	13893	50.4	1271	3564	127	143	15311	131056	32884	384550	6796	1160933
Eclipse	889	5299	10419	17.6	1218	3387	152	143	28094	428931	14490	203440	2528	566378
Fop	436	2094	4375	3.4	331	552	26	19	5395	82021	42980	21477	1836	117111
Javac	490	3424	5343	35.9	1747	6865	675	477	28296	529629	109326	415455	2419	1526559
Javadoc	606	3423	6231	35.6	1242	4257	175	122	32946	514460	202011	399820	2933	602102
Jython	677	4777	8688	142.8	2465	8510	466	412	62308	3912919	339520	749715	3170	3370322
Ps	571	2867	5400	6.2	1164	10736	296	574	12362	602239	337790	134165	2822	130729
Sablecc-j	995	5940	8697	27.1	2101	18508	506	348	25384	456902	137846	426971	3562	888819
Soot-c	933	4044	6270	19.2	2894	14599	1048	723	20187	1038555	85332	383451	2782	251474
Emma2.0	856	4904	10070	117.8	1770	5001	112	118	55408	1340111	552556	722392	4745	5407805
Javacc3.2	311	2136	8507	6.1	1093	3554	39	404	9986	523167	24938	81499	2111	101447
Jess4.5	364	1825	3976	5.4	732	2385	50	75	6530	315898	73247	43605	2088	226636
Pmd3.2	556	3320	5301	5.2	1742	4323	64	392	16131	261652	43883	89907	2557	209580
Xsltc1.2	717	3758	8493	43.0	2027	10346	530	532	62305	3629243	303933	701675	4850	764186

- *ONode, OEdge*: The Application Object Member Graph (AOMG) is a graph consisting of two node types: object methods $[o,m]$ and object fields $[o,f]$, and three edge types: object call $[o^i,m_p] \rightarrow [o^j,m_q]$, object field store $[o^i,m] \rightarrow [o^j,f]$, and object field load $[o^i,f] \rightarrow [o^j,m]$. *ONode* and *OEdge* is the number of nodes and edges in an AOMG where at least one of the participants is an application object member.
- *Heap*: The number of abstract objects referenced by the application object fields. That is, we have summed up the sizes of all points-to sets stored in all application object fields.
- *Enter*: The number of abstract objects entering an application method. That is, we have counted the number of different abstract objects that enter an application method (i.e., out-port values for entry, field load, and call nodes) and summed these up.

The AOMGs are easy to derive since we know the set of abstract objects referenced by the implicit variable *this* in each method (or context), and we know the targets of all member accesses *a.x*. A small number of *OEdge* indicates small *this* value sets as well as a precise resolution of member accesses (relevant in, e.g., reverse engineering of UML *interaction diagrams*).

The *Heap* metrics can be seen as the size of the abstract heap associated with the application objects. It is a metric that puts all focus on the precision of the memory store operation and is of direct relevance for a number of memory management optimizations (e.g., *side-effect* analysis).

Enter focuses on the flow of abstract objects between different parts of a program. A low value indicates a precise analysis that narrows down the flow of abstract objects from one part of the program to another (e.g., *object tracing*).

5.3. Escape analysis

Escape analysis is a static program analysis that finds abstract objects escaping from the methods of a program. We say that an object escapes from a method *m* if it is created *during* the execution of *m* and is still accessible *after* the execution of *m*. An object escapes *m* if it is returned by *m*, assigned under *m*'s execution to a field of another escaping object, or assigned to an object allocated outside *m*. Static variables are considered fields of a virtual object allocated outside any method.

On top of our points-to analysis, we have implemented a standard escape analysis [35,36]. Our implementation uses points-to analysis as a black box preparing for an easy switching from one context-sensitive analysis technique to another.

Escape analysis has, among others, successfully been applied to the stack-allocation of heap-objects [36–38]: a non-escaping object *o*, which is – outside any loop – heap-allocated in a method *m*, can as well be allocated on the stack-frame of *m* avoiding the garbage collection of *o*. Our metrics *Esc(m)* counts the number of abstract objects allocated in and escaping from *m* by unifying the sets of objects escaping any context of *m*. Lower values are better since more objects could be stack-allocated.

The above idea has also been generalized before: an object allocated outside a loop in *m*, and escaping from *m*, can be stack-allocated in a *transitive* caller method *n*, if (i) it does not escape from *n*, (ii) *n* dominates *m*, and (iii) each execution of *n* corresponds to at most one execution of *m*. Therefore, our second metric $Esc^*(m)$ counts the number of abstract objects allocated *transitively* in *m*, and escaping from *m* by unifying the sets of objects escaping any context of *m*. Lower values are again better since more objects could be potentially stack allocated.

5.4. Experimental setup

We have used a benchmark containing 16 different programs. Since we analyze Java bytecode, we characterize the size of a program in terms of "number of classes and methods" rather than "lines of code"; our benchmark programs range from 311 to 1002 classes. All programs are presented in Table 1.

The programs in the upper half of the table are taken from well-known test suites [39–41], and we have picked all those programs that were (i) larger than 300 classes, and (ii) freely available on the Internet. In the lower half, we have our own set of "more recent" test programs, which are also freely available. All programs are analyzed using version 1.4.2 of the Java standard library, and all experimental data presented in this article is the median value of three runs on the same computer (Dell PowerEdge 1850, 6GB RAM, Dual Intel Xeon 3.2 GHz under Linux x86-64, kernel 2.6.22.1).

Table 1 contains data taken from our context-insensitive analysis *Insens*. This set of data will be our baseline result which we compare the context-sensitive results with. The first section *General* is provided to give a rough overview of the different programs, and shows the number of used classes (*Class*), the number of reachable methods (*Method*), and the number of abstract objects (*Object*). It further lists the analysis time (*Time*), i.e., the time needed to perform the points-to analysis. The times

Table 2
Results relevant for *SourceCode* clients.

Program	CallSite				ObjSens				ThisSens			
	Node	Edge	PCall	Cast	Node	Edge	PCall	Cast	Node	Edge	PCall	Cast
Antlr	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
Bloat	0.996	0.979	0.884	0.993	0.996	0.978	0.883	0.970	0.996	0.978	0.883	0.970
Chart	0.998	0.981	0.984	0.916	0.990	0.904	0.969	0.979	0.989	0.904	0.969	0.979
Eclipse	0.996	0.991	0.993	0.839	0.991	0.987	1.000	0.839	0.991	0.987	1.000	0.839
Fop	1.000	1.000	1.000	0.895	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
Javac	1.000	0.997	0.999	0.983	1.000	0.996	0.988	0.973	1.000	0.996	0.990	0.977
Javadoc	0.999	0.999	0.994	0.893	0.994	0.985	0.994	0.918	0.994	0.985	0.994	0.918
Jython	0.995	0.988	0.931	0.998	0.994	0.986	0.925	0.995	0.994	0.986	0.923	0.995
Ps	1.000	1.000	0.997	0.988	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
Sablecc-j	0.998	0.291	0.834	0.989	0.997	0.286	0.751	0.986	0.997	0.286	0.751	0.986
Soot-c	0.998	0.969	0.948	0.981	0.998	0.965	0.924	0.981	0.998	0.965	0.924	0.981
Emma2.0	0.984	0.986	0.973	0.966	0.976	0.973	0.973	0.958	0.976	0.973	0.973	0.958
Javacc3.2	1.000	1.000	1.000	1.000	0.999	1.000	1.000	1.000	0.999	1.000	1.000	1.000
Jess4.5	1.000	1.000	1.000	1.000	0.999	0.999	1.000	1.000	0.999	0.999	1.000	1.000
Pmd3.2	0.998	0.999	0.969	0.990	0.998	0.999	0.969	0.990	0.998	0.999	0.969	0.990
Xsltc1.2	1.000	1.000	1.000	0.996	0.996	0.987	1.000	0.996	0.996	0.987	1.000	0.996
Average	**0.998**	**0.949**	**0.969**	**0.964**	**0.995**	**0.940**	**0.961**	**0.974**	**0.995**	**0.940**	**0.961**	**0.974**
Median	**0.999**	**0.998**	**0.994**	**0.988**	**0.997**	**0.987**	**0.991**	**0.986**	**0.997**	**0.987**	**0.992**	**0.986**

Table 3
Results relevant for *ObjectIdentity* clients.

Program	CallSite				ObjSens				ThisSens			
	ONode	OEdge	Heap	Enter	ONode	OEdge	Heap	Enter	ONode	OEdge	Heap	Enter
Antlr	0.836	0.302	0.891	0.446	0.792	0.055	0.024	0.216	0.808	0.255	0.024	0.216
Bloat	0.776	0.357	0.680	0.621	0.770	0.214	0.673	0.620	0.770	0.321	0.673	0.620
Chart	0.811	0.692	0.524	0.765	0.792	0.349	0.485	0.825	0.807	0.637	0.485	0.826
Eclipse	0.409	0.122	0.165	0.241	0.362	0.099	0.145	0.192	0.362	0.100	0.145	0.191
Fop	0.779	0.379	0.904	0.810	0.735	0.070	0.109	0.735	0.735	0.271	0.109	0.735
Javac	0.883	0.619	0.535	0.841	0.359	0.234	0.167	0.203	0.372	0.350	0.182	0.202
Javadoc	0.855	0.482	0.668	0.752	0.294	0.129	0.099	0.185	0.294	0.159	0.099	0.185
Jython	0.713	0.527	0.243	0.411	0.647	0.123	0.112	0.344	0.655	0.463	0.112	0.342
Ps	0.928	0.828	0.727	0.947	0.922	0.616	0.021	0.916	0.922	0.825	0.021	0.916
Sablecc-j	0.727	0.145	0.108	0.277	0.573	0.094	0.026	0.180	0.573	0.109	0.026	0.180
Soot-c	0.753	0.234	0.901	0.491	0.750	0.164	0.768	0.472	0.750	0.229	0.768	0.472
Emma2.0	0.743	0.331	0.839	0.534	0.632	0.078	0.081	0.409	0.664	0.273	0.080	0.408
Javacc3.2	0.739	0.079	0.484	0.407	0.736	0.045	0.231	0.393	0.736	0.077	0.231	0.393
Jess4.5	0.887	0.399	0.872	0.688	0.860	0.209	0.059	0.676	0.878	0.385	0.059	0.675
Pmd3.2	0.829	0.501	0.846	0.743	0.753	0.379	0.389	0.622	0.768	0.490	0.389	0.624
Xsltc1.2	0.901	0.327	0.599	0.804	0.620	0.123	0.194	0.277	0.630	0.207	0.195	0.277
Average	**0.786**	**0.395**	**0.624**	**0.611**	**0.662**	**0.186**	**0.224**	**0.453**	**0.670**	**0.322**	**0.225**	**0.453**
Median	**0.795**	**0.368**	**0.674**	**0.655**	**0.735**	**0.126**	**0.129**	**0.401**	**0.735**	**0.272**	**0.128**	**0.400**

required for Points-to SSA graph construction and analysis setup are not included[1].

The sections *SourceCode*, *ObjectIdentity*, and *Escape* show the context-insensitive results for the precision metrics that we introduced earlier.

5.5. The SourceCode metrics suite — results

In this section, we present the first set of results when measuring the precision using the *SourceCode* metrics suite. The results related to our context-sensitive approaches *CallSite*, *ObjSens* and *ThisSens* are presented in Table 2. All results in this and the following tables are given as a multiple of the context-insensitive results presented in Table 1. For example, for the `sablecc-j` benchmark, the number of unresolved polymorphic calls in analysis *CallSite* is $506 \times 0.834 = 422$, where 506 is the number for the metrics *PCall* given in Table 1. Tables 2–5 use both the mean (*average*) and median (*median*) values to report the overall results. The median values are included to reduce the effects of outliers (e.g., `sablecc-j` in Table 2), which are over-emphasized in *average*.

First, there is no significant difference between the three approaches for this set of metrics. Second, they only provide slightly better results than the context-insensitive analysis, with one major exception: in the benchmark `sablecc-j`, the number of call graph edges is reduced by 71% when using a context-sensitive analysis. Thus, client applications of type *SourceCode* would probably not notice any significant change for most programs.

Our results are in agreement with those of Lhoták and Hendren [19], who also explained the outlier `sablecc-j`: they traced this increase in precision to the map-implementation of `sablecc-j`, where different maps store different types of objects, but all maps use the same kind of generic map entry object. Thus, in the context-insensitive analysis, the contents of all maps get mixed, and consequently, the analysis cannot compute that methods like `toString()` and `equals()` are called for only some but not all of the objects stored in such maps.

[1] The longest graph construction time we measured was for `Chart` (48.2 s). A large part of this time (92%) was spent on file reading and in third party components (Soot).

Table 4
The escape analysis results.

Program	*CallSite*		*ObjSens*		*ThisSens*	
	Esc	*Esc**	*Esc*	*Esc**	*Esc*	*Esc**
Antlr	0.999	0.993	0.973	0.971	0.973	0.971
Bloat	0.956	0.754	0.952	0.753	0.952	0.751
Chart	0.982	0.985	0.979	0.958	0.979	0.958
Eclipse	0.987	0.632	0.945	0.624	0.945	0.625
Fop	0.991	0.992	0.930	0.980	0.930	0.981
Javac	0.957	0.923	0.953	0.909	0.953	0.909
Javadoc	0.964	0.984	0.922	0.902	0.922	0.902
Jython	0.983	0.960	0.951	0.883	0.951	0.878
Ps	0.988	0.967	0.953	0.801	0.967	0.905
Sablecc-j	0.980	0.962	0.883	0.747	0.883	0.751
Soot-c	0.999	0.525	0.999	0.984	0.999	0.984
Emma2.0	0.990	0.980	0.749	0.222	0.750	0.221
Javacc3.2	0.998	0.996	0.990	0.987	0.990	0.987
Jess4.5	0.978	0.975	0.907	0.835	0.907	0.837
Pmd3.2	0.971	0.926	0.971	0.916	0.971	0.914
Xsltc1.2	0.993	0.955	0.992	0.762	0.992	0.761
Average	**0.982**	**0.907**	**0.940**	**0.827**	**0.941**	**0.833**
Median	**0.985**	**0.965**	**0.953**	**0.893**	**0.953**	**0.904**

Table 5
Used contexts and analysis time.

Program	CallSite		ObjSens		ThisSens	
	Ctx	Time	Ctx	Time	Ctx	Time
Antlr	5.06	1.80	5.80	8.80	5.24	1.28
Bloat	7.95	5.99	15.89	127.15	8.90	2.94
Chart	4.65	1.94	18.21	40.11	6.36	1.14
Eclipse	4.33	1.76	12.06	13.74	5.70	0.90
Fop	3.60	1.19	12.99	4.81	5.18	1.38
Javac	4.59	4.05	8.91	14.70	4.55	1.65
Javadoc	4.63	1.65	11.05	9.51	5.37	0.71
Jython	4.83	2.06	17.43	26.25	6.88	1.00
Ps	7.60	2.02	10.81	10.94	4.60	1.08
Sablecc-j	3.91	1.44	10.02	19.39	5.25	0.91
Soot-c	7.14	4.48	7.78	14.84	4.58	0.93
Emma2.0	4.43	1.00	21.37	19.59	6.32	0.24
Javacc3.2	8.04	1.91	16.74	17.83	9.44	1.53
Jess4.5	4.19	2.07	10.56	12.31	5.39	1.55
Pmd3.2	3.73	2.11	9.97	15.37	4.04	1.26
Xsltc1.2	6.81	5.40	19.21	54.45	7.99	2.15
Average	**5.34**	**2.55**	**13.05**	**25.61**	**5.99**	**1.29**
Median	**4.64**	**1.98**	**11.56**	**15.11**	**5.38**	**1.20**

5.6. The ObjectIdentity metrics suite — results

In this section, we present the results when measuring the precision using the *ObjectIdentity* metrics suite. The results are presented in Table 3.

First, all three approaches are much more precise than context-insensitive analysis. These results indicate that client applications of type *ObjectIdentity* are likely to benefit from using a context-sensitive analysis. The results for the metrics *Heap* show a significant precision improvement (38–78%) when using a context-sensitive analysis, a result that indicates a considerably improved precision in the handling of memory store operations. This comes as no surprise for the two functional approaches (both come with a 78% improvement), since, due to encapsulation, almost every memory store operation is done using the implicit variable *this*. It was also shown by Milanova et al. [17,18] that this type of memory related client application can benefit from context-sensitive analysis. Our experiments confirm these results.

Second, the two functional approaches, on average and median, are clearly more precise than *CallSite*. This- and object-sensitivity have in common that the contexts in which a method is analyzed depends on the value of the implicit *this*-variable. That is, the analysis values of two calls targeting disjoint *this*-sets are never mixed. This is important in object-oriented programs where a large part of all field accesses and calls targets *this*. The focus on *this* makes this- and object-sensitivity more precise in the analysis of object-oriented programs than CFA, which was designed for the analysis of functional and imperative programs. This is also in agreement with previous results comparing object-sensitivity and the call string approach [18,19]. However, our use of the metrics suite that focus on individual objects makes the difference between the two types of context definitions more obvious. It seems likely that client applications that require precise information about individual objects and their interaction would benefit from using one of the two functional approaches.

Finally, the two functional approaches provide almost identical results in three out of four metrics. The major difference is in the metrics *OEdge*, where *ObjSens* is considerably more precise. The higher precision is probably due to a more precise handling of memory load operations via the implicit variable *this*, a situation that is very common in most programs. This situation was studied in Example 2, Section 4.2.

5.7. Escape Analysis — results

The picture that the abstract points-to metrics show is confirmed by the concrete client application escape analysis, as shown in Table 4. The results measured by *Esc* and *Esc** improve clearly (up to 6% and 17%, respectively, on average) when the precision of the underlying points-to analysis improves.

This confirms our assumption that client analyses which are interested in individual objects and their relations improve when based on more precise context-sensitive points-to results.

In one case (emma2.0), the overall number *Esc* of objects defined in and escaping from a method is reduced to 75% and for *Esc*$*$, the number of objects defined in a method or its transitive descendants and escaping, is reduced to 22%.

It should come at no surprise that the reduction in the direct escapes *Esc* is somewhat smaller than the reduction in the transitive escapes *Esc**: an object that is identified as not escaping a method *m* in which it is allocated directly, cannot escape from a caller of *m* either. Hence, small improvements in *Esc* can imply larger differences in *Esc**.

Again, the two functional approaches are more precise than *CallSite*, which is an effect that our abstract points-to metrics already indicate and explain. The two functional approaches *ObjSens* and *ThisSens* provide again almost identical results.

5.8. Time and memory measurements

Table 5 shows the number of used contexts (*Ctx*) and the analysis time required for each approach. We have used the number of used contexts, rather than a direct memory measurement, as our memory cost metrics. This is based on the assumption that the memory cost of maintaining *N* contexts for a given method *m* is $O(N)$, which is true for our implementation and most other implementations that we know of. Furthermore, it is simple and implementation and machine independent. *Ctx* is the average number of used contexts per method, and time is again given as a multiple of the context-insensitive results presented in Table 1.

The first thing to notice is that *ThisSens* is about twice as fast as *CallSite*, and more than an order of magnitude faster than *ObjSens*, on average and median. The time measurements also show that *ThisSens* is, on average (median), only 29% (20%) slower than our context-insensitive analysis *Insens*, a number that is likely to be accepted by client applications that can make use of the improved precision. The fact that *ThisSens* is even faster than *Insens* on a

number of benchmarks (e.g., `sablecc-j` and `emma2.0`) can be considered as a positive side-effect of the improved memory store precision (as manifested by the *Heap* results). Improved memory store precision implies fewer memory changes, which consecutively implies that the fixed point iteration stabilizes faster, cf. Section 3.

ObjSens is also much more costly than the other two approaches when it comes to the number of used contexts (*Ctx*). It requires, on average, 2.2 (2.4) times as many contexts as *ThisSens* (*CallSite*).

This is a bit surprising since, remembering from Section 4.2, that this-sensitive analysis potentially requires an exponential number of contexts in theory. In practice, on the other hand, it turns out that, on average, the number of used *this*-sets is smaller than the number of receiving abstract objects. Furthermore, a low number of used contexts does not only reduce the memory requirements, but even speeds up the analysis by reducing the processing required to reach a fixed point. This non-obvious observation is the key to understand why this-sensitivity outperforms object-sensitivity despite of an exponential worst-case scenario. Another reason is that object-sensitivity may associate a call site with more than one context. Thus, a call $a.m(\ldots)$, where $N = |Pt(a)|$, may require that method m (and all its callees transitively) be processed N times, one for each abstract object in $Pt(a)$. That is, a single call targeting a large points-to set where $N \gg 1$ may generate a cascade of new contexts to process. This problem does not occur in this-sensitivity where each call always targets a single context, which also simplifies the analysis implementation considerably.

Finally, our results where object-sensitivity is more than an order of magnitude slower than the context-insensitive analysis may at first glance seem contradictory to the results presented by Milanova et al. [18], where object-sensitivity was only slightly slower. The explanation is simple: they presented object-sensitivity in theory, but implemented a simplified version where only a fraction of the program (the method parameter nodes) was treated in a context-sensitive manner; all other node types were treated insensitively. This simplification will of course speed up the analysis at the cost of some precision loss. Our implementation follows the theory and treats *all* method contexts separately.

Although we have not experienced any sign of exponential behavior in our experiments[2], we still recommend any non-experimental implementation of this-sensitivity to use a guarded approach (cf. discussion in Section 4.2) to guarantee a polynomial behavior for any input program.

6. Related work

A context-insensitive version of the SSA-based simulated execution approach used in this article was presented before [20]. Our program representation Points-to SSA is closely related to Memory SSA [42,43]. Memory SSA is an extension to the traditional approach to SSA [21,44].

Points-to SSA is a graph-based representation specially designed for points-to analysis where we have removed all operations not directly related to reference computations. A similar approach (a sparse graph designed for points-to analysis) was used by Binkley and Lyle [45] to compute program slices in C programs. It is, however, not SSA-based.

The number of papers explicitly dealing with context-sensitive points-to analysis of object-oriented programs is growing [17–19,26,46,47]. The active research within this area demonstrates its expected potential to improve the analysis precision. The papers experiment with different context definitions and techniques to reduce the memory cost associated with having multiple contexts for a given method. It should also be noted that many approaches targeting object-oriented programs have an "imperative counterpart", which often pre-dates the object-oriented work. People interested in more general reviews of the area should take a look at the papers of Hind [24] and Ryder [13].

Many authors use a call string approach and approximative method summaries to reduce the cost of having multiple contexts [46,47]. Sometimes, ordered binary decision diagrams (OBDD) are used to efficiently exploit commonalities among similar contexts [19,26,42], which allows handling of a very large number of contexts at reasonable memory cost.

Milanova et al. [17,18] present the object-sensitive technique as discussed earlier. The object-sensitive and call string approaches were compared in various works [18,19]. These works also show that 1-object-sensitivity scales to programs containing hundreds of classes.

Whaley et al. [26] present a *k*-call-string based analysis with no fixed upper limit (k) that only takes acyclic call paths into account. Calls within strongly connected components are treated context-insensitively. They report reasonable analysis costs due to their use of OBDDs to handle all contexts. However, their analysis technique was later on compared to 1-object-sensitivity, which was found to be "clearly better" both in terms of precision and scalability [19]. 1-object-sensitivity, in turn, is similar in precision to 1-this-sensitivity but much more costly, as shown in this article.

7. Summary, conclusion, and future work

In this article, we present a new context-sensitive approach to points-to analysis where the target context associated with a call site $a.m(\ldots)$ is determined by the pair $(m, Pt(a))$, where $Pt(a)$ is the points-to set of the target expression a. Hence, we distinguish analysis contexts of a method by its implicit variable *this*. We have therefore named it *this-sensitivity*. It is a modified version of object-sensitivity presented by Milanova et al. [17,18].

We have experimentally evaluated 1-this-sensitivity by comparing it with two well-known context-sensitive approaches (1-object-sensitivity and 1-CFA). Our measurements show that 1-this-sensitivity is much faster than the other two. It is in fact, on average (median), only 35% (21%) slower than our context-insensitive analysis.

We have used two different abstract metrics suites to evaluate the precision of our new approach. Each metrics suite is targeted to a specific group of client applications.

The first metrics suite, denoted *SourceCode*, is most relevant for client applications that are primarily interested in source code entities and reference relations between them, i.e., in relations that hold for all instances of a class. An example is call graph construction. The second metrics suite, denoted *ObjectIdentity*, is most relevant for client applications that are primarily interested in *individual objects* and references to individual objects. Examples of *ObjectIdentity* clients are *side-effect analysis* and reverse engineering of UML *interaction diagrams*. Finally, we have implemented a client application – escape analysis – representing a concrete *ObjectIdentity* client.

The experiments using the *SourceCode* metrics suite show no significant difference between the three context-sensitive approaches. Furthermore, they only provide insignificantly better results than the context-insensitive analysis.

The experiments using the *ObjectIdentity* metrics suite show that all three context-sensitive approaches are much more precise than the context-insensitive analysis. Furthermore, 1-object-sensitivity and 1-this-sensitivity are clearly more precise than 1-CFA. This is in agreement with previous results, where 1-object-sensitivity and 1-CFA were compared with each other [18,19].

[2] The maximum memory consumption we encountered was 1.2 GB when analyzing *chart* with ObjSens.

The results of the concrete client application escape analysis also improve significantly when a more precise context-sensitive analysis is used. Again, 1-object-sensitivity and 1-this-sensitivity provide almost identical results and are slightly more precise than 1-CFA.

Finally, 1-object-sensitivity and 1-this-sensitivity provide almost identical results on escape analysis and in three out of four *ObjectIdentity* metrics. The major difference lies in the metrics indicating a precise resolution of object member accesses. Here, 1-object-sensitivity is significantly more precise. However, this comes at the price of being more than an order of magnitude slower than 1-this-sensitivity.

In conclusion, we recommend 1-this-sensitivity if the client application requires quite precise points-to information and the analysis is on an edit-compile cycle. We recommend 1-object-sensitivity only if the client application requires very precise points-to information and the analysis could be executed as a batch job. Finally, client applications of type *SourceCode* can probably safely avoid the trouble of adding any kind of context-sensitivity to their analysis.

We are currently working on three different approaches to extend 1-this-sensitivity to the next "level of precision" by (1) implementing support for k-this-sensitivity, (2) implementing support for context-sensitive abstract objects, and (3) including all arguments, not only *this*, into our functional context definitions. We refer to the latter as argument-sensitivity and consider it to be an orthogonal next "level of precision".

References

[1] S.R. Chidamber, C.F. Kemerer, A metrics suite for object-oriented design, IEEE Transactions on Software Engineering 20 (6) (1994) 476–493.
[2] B. Henderson-Sellers, Object-Oriented Metrics: Measures of Complexity, Prentice Hall Object-Oriented Series, Prentice Hall, Upper Saddle River, New Jersey, 1999.
[3] S. Mancoridis, B. Mitchell, Y. Chen, E. Ganser, Bunch: a clustering tool for the recovery and maintenance of software system structures, in: Proceedings of the International Conference on Software Maintenance (ICSM'99), 1999, pp. 50–59.
[4] M. Salah, T. Denton, S. Mancoridis, A. Shokoufandeh, F. Vokolos, Scenariographer: a tool for reverse engineering class usage scenarios from method invocation sequences, in: Proceedings of the International Conference on Software Maintenance (ICSM'05), 2005, pp. 155–164.
[5] S. Horwitz, T. Reps, D. Binkley, Interprocedural slicing using dependence graphs, ACM Transactions on Programming Languages and Systems 12 (1) (1990) 26–60.
[6] R. Binder, Testing Object-Oriented Systems: Models, Patterns, and Tools, Addison-Wesley, 1999.
[7] K. Tai, J. Daniels, Interclass test order for object-oriented software, Journal of Object-Oriented Programming 12 (2) (1999) 18–25.
[8] A. Milanova, A. Rountev, B.G. Ryder, Constructing precise object relation diagrams, in: Proceedings of the International Conference on Software Maintenance (ICSM'02), 2002, pp. 586–595.
[9] P. Tonella, A. Potrich, Reverse engineering of the interaction diagrams from C++ code, in: Proceedings of the International Conference on Software Maintenance (ICSM'03), IEEE Computer Society, Washington, DC, USA, 2003, p. 159.
[10] J. Seemann, J.W. von Gudenberg, Pattern-based design recovery of Java software, in: Proceedings of the International Symposium on Foundations of Software Engineering (FSE'98), 1998, pp. 10–16.
[11] T. Marlowe, B. Ryder, Properties of data flow frameworks: a unified model, Acta Informatica 28 (1990) 121–163.
[12] F. Nielsen, H.R. Nielsen, C. Hankin, Principles of Program Analysis, second ed., Springer, 2005.
[13] B.G. Ryder, Dimensions of precision in reference analysis of object-oriented programming languages, in: International Conference on Compiler Construction (CC'03), LNCS, vol. 2622, Springer, 2003, pp. 126–137.
[14] M. Sharir, A. Pnueli, Two approaches to interprocedural data flow analysis, in: S. Muchnick, N. Jones (Eds.), Program Flow Analysis: Theory and Applications, Prentice Hall, 1981.
[15] O. Shivers, Control-flow analysis of higher-order languages, Technical Report, Ph.D. Thesis. Carnegie-Mellon University, CMU-CS-91-145, 1991.
[16] D. Grove, G. DeFouw, J. Dean, C. Chambers, Call graph construction in object-oriented languages, in: Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'97), 1997, pp. 108–124.
[17] A. Milanova, A. Rountev, B.G. Ryder, Parameterized object sensitivity for points-to and side-effect analyses for Java, in: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'02), 2002, pp. 1–11.
[18] A. Milanova, A. Rountev, B.G. Ryder, Parameterized object sensitivity for points-to analysis for Java, ACM Transactions on Software Engineering and Methodology 14 (1) (2005) 1–41.
[19] O. Lhoták, L. Hendren, Context-sensitive points-to analysis: is it worth it?, in: A. Mycroft, A. Zeller (Eds.), International Conference on Compiler Construction (CC'06), LNCS, vol. 3923, Springer, 2006, pp. 47–64.
[20] J. Lundberg, W. Löwe, A scalable flow-sensitive points-to analysis, Technical Report, Matematiska och systemtekniska institutionen, Vöxjö Universitet, 2007, URL <http://w3.msi.vxu.se/wlo/files/goos07.pdf>.
[21] S.S. Muchnick, Advanced Compiler Design Implementation, Morgan Kaufmann Publishers, San Francisco, California, 1997.
[22] A. Aho, R. Sethi, J. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986.
[23] T.J. Marlowe, B.G. Ryder, M.G. Burke, Defining flow sensitivity for data flow problems, Laboratory of Computer Science Research Technical Report Number LCSR-TR-249.
[24] M. Hind, Pointer analysis: haven't we solved this problem yet? in: Workshop on Program Analysis for Software Tools and Engineering (PASTE'01), 2001, pp. 54–61.
[25] R. Hasti, S. Horwitz, Using static single assignment form to improve flow-insensitive pointer analysis, in: Proceedings of the Conference on Programming Language Design and Implementation (PLDI'98), 1998, pp. 97–105.
[26] J. Whaley, M.S. Lam, Cloning-based context-sensitive pointer alias analysis using binary decision diagrams, in: Proceedings of the Conference on Programming Language Design and Implementation (PLDI'04), 2004, pp. 131–144.
[27] Soot: a Java Optimization Framework, URL <http://www.sable.mcgill.ca/soot>.
[28] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, T. Nakatani, A study of devirtualization techniques for a Java Just-In-Time compiler, in: Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00), 2000, pp. 294–310.
[29] R. O'Callahan, The generalized aliasing as a basis for software tools, Ph.D. Thesis. Carnegie Mellon University (December 2000).
[30] Eclipse IDE, URL <http://www.eclipse.org/>.
[31] X-develop IDE, URL <http://www.omnicore.com/>.
[32] L.R. Clausen, A Java byte code optimizer using side-effect analysis, Concurrency: Practice and Experience 9 (11) (1999) 1031–1045.
[33] B. Blanchet, Escape analysis for object-oriented languages: application to Java, in: Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99), 1999, pp. 20–34.
[34] J. Bogda, U. Hölzle, Removing unnecessary synchronization in Java, in: Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99), 1999, pp. 35–46.
[35] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, S. Midkiff, Escape analysis for Java, in: Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99), 1999, pp. 1–19.
[36] D. Gay, B. Steensgaard, Fast escape analysis and stack allocation for object-based programs, in: Proceedings of the 9th International Conference on Compiler Construction (CC'00), LNCS, vol. 1781, Springer, 2000, pp. 82–93.
[37] B. Blanchet, Escape analysis for Java: theory and practice, ACM Transactions on Programming Languages and Systems 25 (6) (2003) 713–775. http://doi.acm.org/10.1145/945885.945886.
[38] J. Whaley, M. Rinard, Compositional pointer and escape analysis for Java programs, in: Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99), 1999, pp. 187–206.
[39] Ashes Suite, URL <http://www.sable.mcgill.ca/ashes>.
[40] SPEC JVM98, URL <http://www.spec.org/osg/jvm98>.
[41] DaCapo suite, URL <http://www.dacapobench.org> (2006).
[42] M. Trapp, Optimierung objektorientierter programme, Ph.D. Thesis. Universität Karlsruhe (December 1999).
[43] M. Trapp, G. Lindenmaier, B. Boesler, Documentation of the intermediate representation Firm, Technical Report 1999-14, Fakultät für Informatik, Universität Karlsruhe, Germany, 1999.
[44] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, K. Zadeck, Efficiently computing static single assignment form and the control dependence graph, ACM Transactions on Programming Languages and Systems 13 (4) (1991) 451–490.
[45] D. Binkley, J.R. Lyle, Application of the pointer state subgraph to static program slicing, Journal of Systems and Software 40 (1) (1998) 17–27.
[46] R. Chatterjee, B. Ryder, W. Landi, Relevant context inference, in: Symposium on Principles of Programming Languages (POPL'99), 1999, pp. 133–146.
[47] E. Ruf, Effective synchronization removal for Java, in: Proceedings of the Conference on Programming Language Design and Implementation (PLDI'00), 2000, pp. 208–218.