Field Flow Sensitive Pointer and Escape Analysis for Java Using Heap Array SSA

Prakash Prabhu and Priti Shankar

Department of Computer Science and Automation,
Indian Institute of Science,
Bangalore 560012, India

Abstract. Context sensitive pointer analyses based on Whaley and Lam's *bddbddb* system have been shown to scale to large Java programs. We provide a technique to incorporate flow sensitivity for Java fields into one such analysis and obtain an escape analysis based on it. First, we express an intraprocedural *field flow sensitive analysis*, using Fink et al.'s Heap Array SSA form in Datalog. We then extend this analysis interprocedurally by introducing two new ϕ functions for Heap Array SSA Form and adding deduction rules corresponding to them. Adding a few more rules gives us an escape analysis. We describe two types of field flow sensitivity: *partial* (PFFS) and *full* (FFFS), the former without strong updates to fields and the latter with strong updates. We compare these analyses with two different (field flow insensitive) versions of Whaley-Lam analysis: one of which is flow sensitive for locals (FS) and the other, flow insensitive for locals (FIS). We have implemented this analysis on the *bddbddb* system while using the *SOOT* open source framework as a front end. We have run our analysis on a set of 15 Java programs. Our experimental results show that the time taken by our field flow sensitive analyses is comparable to that of the field flow insensitive versions while doing much better in some cases. Our PFFS analysis achieves average reductions of about 23% and 30% in the size of the points-to sets at load and store statements respectively and discovers 71% more "caller-captured" objects than FIS.

1 Introduction

A pointer analysis attempts to statically determine whether two variables may point to the same storage location at runtime. Many compiler optimizations like loop invariant code motion, parallelization and so on require precise pointer information in order to be effective. A precise pointer analysis can also obviate the need for a separate escape analysis. Of the various aspects of a pointer analysis for Java that affect its precision and scalability, two important ones are context sensitivity and flow sensitivity. A context sensitive analysis does not allow information from multiple calling contexts to interfere with each other. Although in the worst case, it can lead to exponential analysis times, context sensitivity is important, especially in case of programs with short and frequently invoked

M. Alpuente and G. Vidal (Eds.): SAS 2008, LNCS 5079, pp. 110–127, 2008.

methods. A flow sensitive analysis takes control flow into account while determining points to relations at various program points. An analysis could be flow sensitive for just scalars or for object fields too. Since it can avoid generation of spurious points to relations via non-existent control flow paths, flow sensitivity is important for precision of a pointer analysis. One of the most scalable context sensitive pointer analysis for Java is due to Whaley and Lam [1], based on the *bddbddb* system. However, it is flow sensitive just for locals and not for object fields. The analysis of Fink et al. [2], based on the Heap Array SSA form [2], is flow sensitive for both locals and fields. However, it is intraprocedural and context insensitive. In this work, we develop an analysis similar to that of Fink et al., extend it interprocedurally and integrate it into the context sensitive framework of Whaley and Lam. The contributions of this paper can be summarized as follows:

– We formulate two variants of a field flow sensitive analysis using the Heap Array SSA Form in Datalog: *partial field flow sensitive analysis* (PFFS) and *full field flow sensitive analysis* (FFFS). Section 2 gives an overview of the Heap Array SSA form and describes our formulation of intraprocedural field flow sensitive analysis in Datalog.
– We extend the Heap Array SSA form interprocedurally by introduction of two new ϕ functions: the *invocation* ϕ and the *return* ϕ function and use it to enhance PFFS and FFFS to work across methods. Section 3 describes these ϕ functions.
– We then incorporate interprocedural field flow sensitivity into the Whaley-Lam context sensitive analysis and derive an escape analysis based on this. This makes PFFS and FFFS both field flow and context sensitive. Both these analyses are described in Section 4.
– We experimentally study the effects of field flow sensitivity on the timing and precision of a context sensitive pointer and escape analysis. We do this by comparing PFFS and FFFS with two versions of the Whaley-Lam analysis [1]: one of which is flow sensitive for locals (FS) and the other flow insensitive for locals (FIS). Section 5 describes the implementation and gives the results.

2 Intraprocedural Field Flow Sensitivity

2.1 Heap Array SSA and Field Flow Sensitivity

A flow sensitive pointer analysis is costly both in terms of time and space since it has to compute and maintain a separate points-to graph at every program point. One way to reduce the cost is to use the SSA Form [3] for pointer analysis. But translating a program into SSA itself may require pointer analysis, due to the presence of pointer variables in the original program. Hasti and Horwitz [4] give an algorithm for performing flow sensitive analysis using SSA for C, while safely factoring the effect of pointers on the SSA translation. In case of Java, the use of fields gives rise to the same issues as the use of pointer variables in C. However, the normal SSA translation as applied to scalar variables does not give precise

```
public A foo()
{
    S1: u = new A(); // O1
    S2: v = new A(); // O2
    S3: u.f = v;
        ...
    S4: y = helper(u);
    S5: return y;
}

public A helper(A x)
{
    S6: r = new A(); // O3
    S7: ret = x.f;
    S8: ret.f = r;
    S9: return ret;
}
```

```
public void bar()
{
    S10: n = new A(); // O4
    S11: m = new A(); // O5
    S12: o = new A(); // O6
    S13: n.f = m;
         ...
    S14: p = helper(n);
    S15: q = n;
    S16: n.f = o;
         ...
    S17: n.f = m;
    S18: if (...) {
    S19:   q.f = o;
    S20:   q = m;
         }
}
```

Fig. 1. Example to illustrate Field Flow Sensitivity

results for Java fields. Consider the program in Figure 1 and the scalar SSA form of the *bar()* method, shown in Figure 2. It can be inferred from the points-to graph (Figure 3) that q at S15 points to O4 while at S20 it points to O5 (a scalar flow sensitive result), based on the SSA subscripts. However, we cannot infer from this graph that $n.f$ points to O5 at S13 while at S16 it points to O6 (a field flow insensitive result). This is because no distinction is made between the different instances of an object field (f in this case) at different program points. A *field flow sensitive analysis* is one which makes a distinction between field instances at different program points and is more precise than a field flow insensitive analysis.

A field flow sensitive analysis can be obtained by using an extended form of SSA, to handle object fields, called Heap Array SSA. Heap Array SSA is an application of the Array SSA Form [5], initially developed for arrays in C, to Java fields. In the Array SSA form, a new instance of an array is created every time one of its elements is defined. Since the new instance of the array has the correct value of only the element that was just defined, a function called the *define-ϕ* (dϕ) is inserted immediately after the assignment to the array element. The dϕ collects the newly defined values with the values of unmodified elements, available immediately before the assignment, into a new array instance.

The Array SSA form also carries over the *control-ϕ* (cϕ) function from the scalar SSA, inserted exactly at the same location (iterated dominance frontier) as done for scalar SSA. The cϕ merges the values of different instances of an array computed along distinct control paths. Heap Array SSA [2] applies the Array SSA form to Java objects. Accesses to an object field f are modeled by defining a one-dimensional heap array H^f . This heap array represents all instances of the field f that exists on the heap. Heap arrays are indexed by object references. A load of $p.f$ is modeled as read of element $H^f[p]$ and the store of $q.f$ is modeled

```
public void bar()
{
        n0 = new A(); // O4
        m0 = new A(); // O5
        o0 = new A(); // O6
        n1 = n0;
  S13:  n1.f = m0;
          ...
        p0 = helper(n1);
  S15:  q0 = n1;
        n2 = n1;
  S16:  n2.f = o0;
          ...
        n3 = n2;
        n3.f = m0;
        if (...) {
            q1 = q0;
  S19:      q1.f = o0;
  S20:      q2 = m0;
        }
        q3 = mφ(q0, q2);
}
```

Fig. 2. Scalar SSA form of the *bar()* method

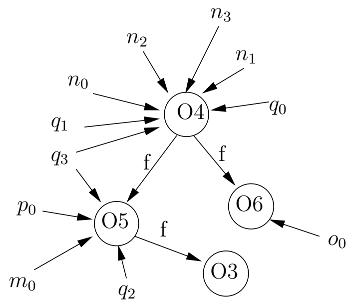

Fig. 3. Points to Graph for Scalar SSA Form of bar()

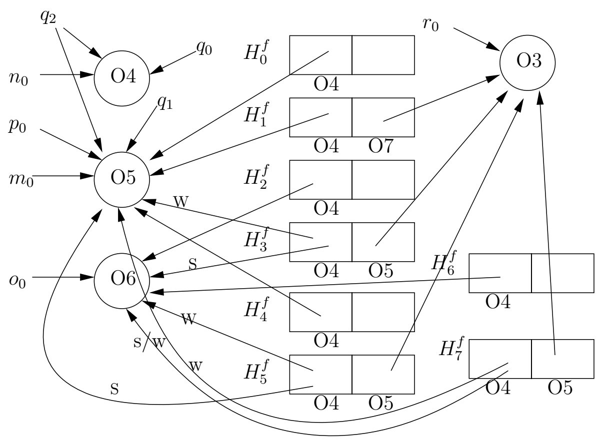

Fig. 4. Points to Relations using Heap Array SSA for *bar()*

as a write of element $H^f[q]$. Figure 5 shows the Heap Array SSA form for the program seen earlier. The mϕ function is the traditional ϕ function used for scalars [3]. Converting a Java program into Heap Array SSA form and running a flow insensitive analysis algorithm on it generates a flow sensitive pointer analysis result for both fields and locals.

2.2 Field Flow Sensitive Analysis as Logic Programs

Pointer analysis can naturally be expressed in a logic programming language [6] like Datalog. The Java statements that affect the points-to relations are given as input relations to the logic program while the points-to relation is the output generated by the analysis. There is one input relation to represent every type of statement in a Java program. Every source statement in the Java program is encoded as a unique tuple (row) in the corresponding input relation. The transfer functions are represented as deduction rules. Whaley and Lam have developed *bddbddb*, a scalable system for solving Datalog programs and implemented a context sensitive version of Anderson's pointer analysis [7] on it. We apply Anderson's analysis

```
public A foo()
{
    u_0 = new A(); // O1
    v_0 = new A(); // O2
    H^f_8[u_0] = v_0;
        ...
    y_0 = helper(u_0);
    H^f_9 = rφ(H^f_12{e_2}, H^f_8);
    return y_0;
}
public A helper(A x)
{
    H^f_10 = iφ(H^f_0{e_1}, H^h_8{e_2});
    r_0 = new A(); // O3
    ret_0 = H^f_10[x];
    H^f_11[ret_0] = r_0;
    H^f_12 = dφ(H^f_11, H^f_10);
    return ret_0;
}
```

```
public void bar()
{
       n_0 = new A(); // O4
       m_0 = new A(); // O5
       o_0 = new A(); // O6
 S13:  H^f_0[n_0] = m_0;
         ...
       p_0 = helper(n_0);
       H^f_1 = rφ(H^f_12{e_1}, H^f_0);
       q_0 = n_0;
 S16:  H^f_2[n_0] = o_0;
       H^f_3 = dφ(H^f_2, H^f_1);
         ...
       H^f_4[n_0] = m_0;
       H^f_5 = dφ(H^f_4, H^f_3);
       if (...) {
 S19:      H^f_6[q_0] = o_0;
           H^f_7 = dφ(H^f_6, H^f_5);
           q_1 = m_0;
       }
       q_2 = mφ(q_1, q_0);
       H^f_8 = cφ(H^f_7, H^f_5);
}
```

Fig. 5. Heap Array SSA Form for the program in Figure 1

over the Heap Array SSA form of a Java program to obtain a field flow sensitive analysis. The precision of the resulting analysis is as good as any field flow sensitive analysis that is based on points-to graphs [8][9]. The main advantage of using Heap Array SSA is that it obviates the need to maintain a separate points-to graph at every program point and thereby effecting a more scalable analysis.

We formulate two variants of this analysis in Datalog: partial field flow sensitive analysis (PFFS) and full field flow sensitive analysis (FFFS). All heap objects are abstracted by their allocation sites. Two points-to sets, *vPtsTo* and *hPtsTo*, are associated with scalar variables and heap array elements (heap arrays themselves are indexed by objects) respectively. These sets hold the objects pointed to by them. At the end of analysis, *vPtsTo* and *hPtsTo* together give a field flow sensitive points-to result. We use Whaley and Lam's notation for logic programs [1]. V and H represent the domain of scalars and heap objects (object numbers) respectively. F is the domain of all fields. The numeric subscripts inserted by the Heap Array SSA transformation are represented from N, the set of natural numbers. The relations used in our analysis have an attribute to accommodate the SSA numbers of the heap arrays. For local variables, the SSA subscripts are a part of the variable name itself. Table 1 lists for every input relation, the tuple representation of a particular source statement. For an output relation, it specifies the tuple representation for a particular element of the derived points-to set.The first four deduction rules of the analysis capture the effect of *new*, *assign*, *load* and *store* statements and are very similar to the rules for the non-SSA form, the only difference being the SSA numbers for heap arrays:

$$vPtsTo(v_1, h) \quad :- \quad new(v_1, h) \tag{1}$$
$$vPtsTo(v_1, h) \quad :- \quad assign(v_1, v_2),\ vPtsTo(v_2, h) \tag{2}$$

$$vPtsTo(v_1, h_2) \quad :- \quad load(v_1, f, s_0, v_b),\ vPtsTo(v_b, h_1),\ hPtsTo(f, s_0, h_1, h_2) \tag{3}$$
$$hPtsTo(f, s_0, h_1, h_2) \quad :- \quad store(v_1, f, s_0, v_b),\ vPtsTo(v_b, h_1),\ vPtsTo(v_1, h_2) \tag{4}$$

The semantics of the rules are as follows :

- **Rule (1) for new** $v_1 = new\ \ h()$: Creates the initial points-to relation for the scalar variables.
- **Rule (2) for assign** $v_1 = v_2$: Updates the points-to set for v_1 based on the inclusion property: $points-to(v_2) \subseteq points-to(v_1)$.
- **Rule (3) for load** $v_1 = H^f_{s_0}[v_b]$: Updates the points-to set for v_1 using the points-to set of $H^f_{s_0}[h_1]$ for every object h_1 that is pointed to by the index variable v_b of the Heap array instance $H^f_{s_f}$.
- **Rule (4) for store** $H^f_{s_0}[v_b] = v_1$: Acts similar to rule (3), the data flow being in the opposite direction in this case.

The next two rules correspond to the $c\phi$ and the $m\phi$ statement:

$$hPtsTo(f, s_0, h_1, h_2) \quad :- \quad cphi(f, s_0, s_1),\ hPtsTo(f, s_1, h_1, h_2) \tag{5}$$
$$vPtsTo(v_0, h) \quad :- \quad mphi(v_0, v_1),\ vPtsTo(v_1, h) \tag{6}$$

The semantics of these rules are:

- **Rule (5) for cϕ:** $H^f_{s_0} = c\phi(H^f_{s_1}, H^f_{s_2}, ..., H^f_{s_n})$: The c$\phi$ statement is represented as a set of tuples (f, s_0, s_i) $\forall i$ such that $1 \leq i \leq n$ in the *cphi* relation since all the arguments of cϕ are symmetric with respect to the lhs heap array instance. The effect of the rule (5) is to merge the points-to sets corresponding to all valid object indices of its arguments into the rhs heap array instance.
- **Rule (6) for mϕ:** [1] $v_0 = m\phi(v_1, ..., v_n)$ Performs the merge of points-to sets for scalars.

To complete the analysis, we need to add rules corresponding to the dϕ statement. Based on the type of rules for modeling dϕ, we distinguish two types of field flow sensitivity:

Partial Field Flow Sensitivity (PFFS). We define a partial field flow sensitive analysis as one that performs only weak updates to heap array elements. To obtain PFFS, the rules required to model dϕ statement are simple and are

[1] In the implementation, this rule is replaced by rule (2) for assigns, since a mϕ can be modeled as a set of assignment statements.

Table 1. Relations used in the Field Flow Sensitive Analysis

Source Statement/ Pointer Semantics	Tuple(s) Representation	Relations	Type
$v_1 = new\ \ h()$	(v_1, h)	new(v: V, h: H)	input
$v_1 = v_2$	(v_1, v_2)	$assign$(v_1: V, v_2: V)	input
$v_2 = H^f_{s_0}[v_1]$	(v_2, f, s_0, v_1)	$load$(v: V, f: F, s: N, b: V)	input
$H^f_{s_0}[v_1] = v_2$	(v_2, f, s_0, v_1)	$store$(v: V, f: F, s: N, b: V)	input
$H^f_{s_0} = d\phi(H^f_{s_1}, H^f_{s_2})$	(f, s_0, s_1, s_2)	$dphi$(f: F, s_0: N, s_1: N, s_2: N)	input
$H^f_{s_0} = c\phi(H^f_{s_1}, ..., H^f_{s_n})$	(f, s_0, s_i) $\forall i$ such that $1 \leq i \leq n$	$cphi$(f: F, s_0: N, s_1: N)	input
$v_0 = m\phi(v_1, ..., v_n)$	(v_0, v_i) $\forall i$ such that $1 \leq i \leq n$	$mphi$(v_0: V, v_1: V)	input
$v_0 \rightarrow h$	(v_0, h)	$vPtsTo$(v: V, h: H)	output
$H^f_{s_0}[h_1] \rightarrow h_2$	(f, s_0, h_1, h_2)	$hPtsTo$(f: F, s_0: N, h_1: H, h_2: H)	output

similar to that for the cϕ statement. Without strong updates, useful information can still be obtained since the points-to relation of heap arrays that appear at a later point in the control flow do not interfere with the points-to relations of the heap array at the current program point. The rules for achieving PFFS are:

$$hPtsTo(f, s_0, h_1, h_2) \quad :- \quad dphi(f, s_0, s_1, -),\ hPtsTo(f, s_1, h_1, h_2) \qquad (7)$$

$$hPtsTo(f, s_0, h_1, h_2) \quad :- \quad dphi(f, s_0, -, s_2),\ hPtsTo(f, s_2, h_1, h_2) \qquad (8)$$

The semantics of these rules are:

- **Rules (7) and (8) for dϕ:** $H^f_{s_0} = d\phi(H^f_{s_1}, H^f_{s_2})$: Merge the points-to sets corresponding to *all* the heap array indices of both the argument heap arrays $H^f_{s_1}$ and $H^f_{s_2}$ and gather them into the lhs heap array instance $H^f_{s_0}$. As no pointed object is ever evicted (killed) from a heap array at a store, only a weak update to fields is done.

Full Field Flow Sensitivity (FFFS). We define a fully field flow sensitive analysis as one that performs strong updates to heap array elements. Hence, FFFS is more precise than PFFS. However, a strong update can be applied at a store statement $v_b.f = v_2$ only under two conditions:

1. v_b points to a single abstract heap object that represents only one concrete object at runtime.
2. The method in which the abstract object is allocated should not be a part any loop or recursive call chain.[2]

[2] This is because we do not have any information about the predicate conditions for loops/recursive method invocations and have to conservatively infer that an object can be allocated more than once, preventing the application of a strong update.

One way to model the dϕ statement to allow for strong updates is by the following rules:

$$\begin{aligned} hPtsTo(f, s_0, h_1, h_2) &:- \; dphi(f, s_0, s_1, -),\ hPtsTo(f, s_1, h_1, h_2) && (9)\\ hPtsTo(f, s_0, h_1, h_2) &:- \; store(-, f, s_1, v_b),\ nonsingular(v_b),\\ &\quad dphi(f, s_0, s_1, s_2),\ hPtsTo(f, s_2, h_1, h_2),\\ &\quad hPtsTo(f, s_1, h_1, -) && (10)\\ hPtsTo(f, s_0, h_1, h_2) &:- \; mayBeInLoop(h_1),\ dphi(f, s_0, s_1, s_2),\\ &\quad hPtsTo(f, s_2, h_1, h_2),\ hPtsTo(f, s_1, h_1, -) && (11)\\ hPtsTo(f, s_0, h_1, h_2) &:- \; dphi(f, s_0, s_1, s_2),\ hPtsTo(f, s_2, h_1, h_2),\\ &\quad !commonIndex(f, s_1, s_2, h_1) && (12)\\ nonsingular(v_1) &:- \; vPtsTo(v_1, h_1),\ vPtsTo(v_1,\ h_2), h_1 != h_2 && (13) \end{aligned}$$

The semantics of these rules are:
Rules (9), (10), (11), (12) and (13) for dϕ: $H^f_{s_0} = d\phi(H^f_{s_1}, H^f_{s_2})$: Whenever a dϕ statement is encountered, the points-to sets for lhs heap array instance $H^f_{s_0}$ is constructed from its arguments $H^f_{s_1}$ and $H^f_{s_2}$ as follows:

- All the points-to sets for object indices of $H^f_{s_1}$ are carried over to $H^f_{s_0}$. The ordering of the arguments for the dϕ is important here: The heap array instance $H^f_{s_1}$ corresponds to the one that is defined in the store statement immediately before this dϕ statement. Rule (9), which performs the derivation of $H^f_{s_0}$ from $H^f_{s_1}$, depends on this ordering to work.
- The points-to sets from $H^f_{s_2}$, for object indices common to $H^f_{s_1}$ and $H^f_{s_2}$, are conditionally carried over to $H^f_{s_0}$ using rules (10) and (11). These rules represent the negation of the conditions required to satisfy a strong update (kill) for a store statement. The *nonsingular* relation determines whether a variable may point to more than one heap object and is derived using rule (13). When the base variable v_b of the store statement is in the *nonsingular* relation, the points-to sets from $H^f_{s_2}$ go into $H^f_{s_0}$, using rule (10). The *mayBeInLoop* relation, on the other hand, is an input relation, which represents all the allocation sites which may be executed more than once. The heap objects abstracted at these sites may not represent a single runtime object. We compute this input relation using a control flow analysis provided by *SOOT*. Whenever a target heap object h_1 of a store is in the *mayBeInLoop* relation, the points-to sets from $H^f_{s_2}$ go into $H^f_{s_0}$, using rule (11).
- The points-to sets from $H^f_{s_2}$, for object indices not common to $H^f_{s_1}$ and $H^f_{s_2}$, are carried over to $H^f_{s_0}$ using rule (12). The computation of the common indices itself requires a pointer analysis due its inherent recursive nature.
- **Strong Updates and Non stratified logic programs:** Consider, for a moment, the following rule as a replacement for (12):

$$\begin{aligned} hPtsTo(f, s_0, h_1, h_2) &:- \; dphi(f, s_0, s_1, s_2),\ hPtsTo(f, s_2, h_1, h_2),\\ &\quad !hPtsTo(f, s_1, h_1, -) \end{aligned}$$

This rule uses recursion with negation on *hPtsTo* relation to compute itself. This rule would result in a *non-stratified* Datalog program, which is currently not supported by *bddbddb*. The way *bddbddb* evaluates Datalog programs is by constructing a *predicate dependency graph* (PDG), where each node represents a relation and an edge $a \rightarrow b$ exists if a is the head relation of a rule which has b as a subgoal relation. Also, an edge is labeled as negative if the subgoal relation has a negation in the rule. The PDG is then divided into different strongly connected components (SCC) and the relations within each SCC are wholly computed by doing a fixed point iteration over the rules representing the edges within the SCC. The whole program is then evaluated in the topological ordering of these SCCs. A Datalog program becomes non-stratified if there exists a SCC with a negative edge. In this case, there is a cycle from *hPtsTo* to itself. XSB [10], a system which supports non-stratified programs using well-founded semantics,[3] operates at the tuple level and is not as scalable as *bddbddb* which operates on complete relations by representing them as BDDs and using efficient BDD operations. Our approach here is to pre-compute the *commonIndex* relation, which represents an over-approximation of the set of common object indices for the two argument heap arrays $H^f_{s_2}$ and $H^f_{s_1}$ of the dϕ and get a stratified Datalog program. The *commonIndex* relation is pre-computed using a field flow insensitive analysis pass, PFFS, in our case:

$$\begin{aligned} commonIndex(f, s_1, s_2, h_1) \quad :- \quad & dphi(f, -, s_1, s_2),\ hPtsTo(f, s_1, h_1, -), \\ & hPtsTo(f, s_2, h_1, -) \end{aligned} \tag{14}$$

Consider the points-to relations obtained by the field flow sensitive analysis for the *bar()* method as seen in Figure 4. The edges labeled w (weak) are the additional edges inferred by PFFS which FFFS does not infer. Both FFFS and PFFS infer the s (strong) edges. Both the analyses infer that at S13, $n.f$ points to O5 ($n_0 \rightarrow O4$ and $H^f_0[O4] \rightarrow O5$) while at S16, $n.f$ points to O6 ($n_0 \rightarrow O4$ and $H^f_2[O4] \rightarrow O6$), which is a field flow sensitive result. However, if there was a reference to $n.f$ after S16, PFFS would infer that $n.f$ may point to either O5 or O6 (due to the w edge: $H^f_3[O4] \rightarrow O6$) while FFFS would say that $n.f$ may point to only O6 (the single s edge: $H^f_3[O4] \rightarrow O6$).

3 Interprocedural Field Flow Sensitivity: iϕ and rϕ

To obtain field flow sensitivity in the presence of method calls, we have to take into account: (a) Effects of field updates made by a called method visible to the caller at the point of return (b) Flow of correct object field values into a called method depending on the invocation site from where it is called. We introduce two new ϕ functions to extend the Heap Array SSA Form interprocedurally: (a) The invocation ϕ function, iϕ (b) The return ϕ function, rϕ.

[3] It provides a form of 3-valued evaluation for logic programs.

The iϕ function models the flow of values into a method corresponding to the invocation site from where it is called. It selects the exact points-to set that exists at the invocation site (at the point of call), on the basis of the invocation edge in the call graph. This is in contrast with the $c\phi$ function which merges the points-to sets flowing via different control flow paths. The iϕ function has the following form:

$$H^f_{s_{in}} = i\phi(H^f_{s_1}\{e_1\}, H^f_{s_2}\{e_2\}, ..., H^f_{s_n}\{e_n\})$$

where $H^f_{s_1}, H^f_{s_2}, ..., H^f_{s_n}$ are the heap array instances that *dominate*[4] the point of call and $e_1, e_2, ..., e_n$ are the corresponding invocation edges in the call graph.

The rϕ function models the merge of the points-to sets of the heap array that existed before the call and the points-to set of the heap arrays that were modified by the call. The updated heap array after the call models the effect of all the methods transitively called in the call chain. It has the following form:

$$H^f_{s_r} = r\phi(H^f_{s_1}\{e_1\}, H^f_{s_2}\{e_2\}, ..., H^f_{s_n}\{e_n\}, H^f_{s_{local}})$$

The presence of more than one edge in the rϕ function is due to virtual method invocations. $H^f_{s_i}$, $\forall i$ such that $1 \leq i \leq n$ is the dominating heap array instance that is in effect at the end of the called method, with corresponding invocation edge e_i. Thus, for every possible concrete method that can be called at the call site, rϕ collects the heap array instances for the field f that dominate the end of the called method and merges the points-to sets into $H^f_{s_r}$. Those objects whose field f has not been modified get their points-to sets into $H^f_{s_r}$ from $H^f_{s_{local}}$, the heap array instance for f that dominates the immediate program point before the call site in the calling method.

The $i\phi$'s are placed at the entry point of a method. Only those heap arrays that are *used* or *modified* in the current method and all the methods it calls transitively require an $i\phi$ at the entry point. Similarly, an $r\phi$ is required only for those heap arrays which are *modified* transitively by a method call. The list of heap arrays for which $r\phi$ and $i\phi$ are required can be determined while performing a single traversal (in reverse topological order of nodes of the call graph) on the interprocedural control flow graph of the program. For the example in Figure 5, an $i\phi$ is placed for f in the *helper()* while rϕ's are placed after calls to *helper()* in *foo()* and *bar()*. The lhs heap arrays of these ϕ functions are also renamed as a part of Cytron's SSA renaming step [3].

Although the previous step would have determined the placement of the $i\phi$ and $r\phi$, still we have to determine their arguments and rename the heap array instances used in the arguments. This is achieved by plugging in the heap array instances that dominate the *point of call* and those that dominate the *callee's exit points* into the arguments of $i\phi$ and $r\phi$ respectively. For the example in Figure 5, the arguments of the iϕ in *helper*() method are H^f_0 and H^f_8, the heap arrays that dominate

---

[4] By definition of the *dominates* relation [3], there is only one dominating heap array instance at any program point for every field in the Heap Array SSA form.

Table 2. Additional Relations for Interprocedural Field Flow and Context Sensitivity

Source Statement/ Pointer Semantics	Tuple(s) Representation	Relation	Type
An invocation i from context c_1 to m in context c_2	(c_1, i, c_2, m)	$IE_c(c_1$: C, i: I, c_2: C, m: M)	input
$H^f_{s_{in}} = i\phi(H^f_{s_1}\{e_1\}, ..., H^f_{s_n}\{e_n\})$	$(f, s_{in}, s_i, e_i)\ \forall i$ such that $1 \leq i \leq n$	$iphi(f$: F, s_{in}: Z, s_i: Z, e_i: I)	input
$H^f_{s_r} = r\phi(H^f_{s_1}\{e_1\}, ..., H^f_{s_n}\{e_n\}, H^f_{s_l})$	$(f, s_r, s_l, s_i, e_i)\ \forall i$ such that $1 \leq i \leq n$	$rphi(f$: F, s_r: Z, s_l: Z, s_i: Z, e_i : I)	input
In context c_1, $v_1 \to h$	(c_1, v_1, h)	$vPtsTo(c$: C, v: V, h: H)	output
In context c_1, $H^f_{s_0}[h_1] \to h_2$	(c_1, f, s_0, h_1, h_2)	$hPtsTo(c$: C, f: F, s_f: N, h_1: H, h_2: H)	output

the points, in *bar()* and *foo()* respectively, at which $helper()$ is called. Similarly the argument of the rϕ's for f is H^f_{12}, the dominating heap array instance in $helper()$ at the point of return. The overall placement of the *phi* functions ($m\phi$, $d\phi$, $c\phi$, $i\phi$ and $r\phi$) and their renaming are performed in the following order:

1. Place $d\phi$, $c\phi$ and $m\phi$ functions using dominance frontiers as in [3]
2. Place the $r\phi$ and $i\phi$ functions.
3. Apply Cytron's Algorithm [3] to rename the Heap Array Instances (results of $d\phi$, $c\phi$, $r\phi$, $i\phi$ and arguments of $d\phi$ and $c\phi$) and local variables (results and arguments of $m\phi$).
4. Use the dominating heap array instances at exit points of methods and call sites to plug in the arguments for $r\phi$ and $i\phi$.

4 Combined Field Flow and Context Sensitivity

4.1 Pointer Analysis

Using the iϕ and rϕ functions, we incorporate interprocedural field flow sensitivity into the Whaley-Lam context sensitive analysis [1] algorithm. We describe only those relations (Table 2) and rules that pertain to the iϕ and rϕ statements. The rest of the relations and rules are context sensitive extensions of those mentioned in Section 2.2 and those for parameter and return value bindings, invocation edge representations [1]. I is the domain of invocation edges, M represents all the methods and C is the domain of context numbers. Two main relations new to this analysis are *iphi* and *rphi*, while *vPtsTo* and *hPtsTo* now have an additional attribute for context numbers. The IE_c relation represents context sensitive invocation edges, computed using $SOOT$'s pre-computed call graph and context numbering scheme of Whaley and Lam [1]. In this scheme, every method is assigned a unique context number for every distinct calling context.[5] The deduction rules and their semantics are as follows:

$$hPtsTo(c_2, f, s_i, h_1, h_2) \quad :- \quad iphi(f, s_i, s_1, i), IE_c(c_1, i, c_2, -), hPtsTo(c_1, f, s_1, h_1, h_2). \tag{15}$$

$$hPtsTo(c_1, f, s_r, h_1, h_2) \quad :- \quad rphi(f, s_r, -, s_i, i), IE_c(c_1, i, c_2, -), hPtsTo(c_2, f, s_i, h_1, h_2). \tag{16}$$

$$hPtsTo(c_1, f, s_r, h_1, h_2) \quad :- \quad rphi(f, s_r, s_l, -, i), IE_c(c_1, i, -, -), hPtsTo(c_1, f, s_l, h_1, h_2). \tag{17}$$

- **Rule (15) for** $i\phi$: This rule models the effect of the $i\phi$. When there is a change in context from c_1 to c_2 due to an invocation i, the heap array instance for a field f in context c_2 (the lhs of the $i\phi$ with SSA number s_i) inherits its points-to set from the heap array instance in c_1 before the call was made (argument s_1 corresponding to the invocation edge i in the $i\phi$ statement). The presence of IE_c makes sure that points-to set of multiple calling contexts don't interfere with each other.
- **Rules (16) and (17) for** $r\phi$: These rules are similar to Rules (7) and (8) that model the effect of $d\phi$ statement in PFFS. Rule (16) makes sure that the points-to sets of the heap array instance (s_i in context c_2) from a virtual method invocation (invocation edge i) are merged into that of lhs heap array instance (s_r) in context c_1. Rule (17) ensures that the lhs heap array instance gets the points-to sets from the local heap array instance that dominates the call site (s_l in context c_1).

For the only iϕ in our example, H^f_{10} in *helper()* inherits its points to sets from H^f_0 along edge e_1 (called from *bar()*) and from H^f_8 along edge e_2 (called from *foo()*) in separate contexts. Hence ret_0 points to O5 and O2 in two distinct contexts and consequently, y_0 and p_0 point to O2 and O5 respectively.

4.2 Escape Analysis for Methods

Escape Analysis [8][9] is a compiler analysis technique which identifies objects that are local to a particular method. For such objects, the compiler can perform stack-allocation, which helps to speed up programs by lessening the burden on the garbage collector. Escape analysis works by determining whether an object may escape a method and if an object does not escape a method ("captured"), it can be allocated on the method's stack frame. Adding a few more relations (Table 3) and rules to the analysis of Section 4.1 gives an escape analysis. We encode the heap array instances that dominate the exit points of methods in

[5] For eg, if a call to method *helper()* by method *bar()* is represented by an invocation edge e_1 in the call graph, and *bar*() is in a calling context with context number c_1 while *helper*() is in a context numbered c_2, this invocation would be represented by the tuple $IE_c(c_1, e_1, c_2, helper)$.

Table 3. Additional Relations for Escape Analysis

Relation	Type	Tuple Semantics
$dominatingHA$(m: M, f: F, s: Z)	input	Heap Array Instance s for field f dominates exits of method m
$formal(m : M, z : Z, v : V)$	input	Formal parameter z of method m is represented by variable v
$threadParam$(v: V)	input	v is passed as parameter to a thread
$callEdge$(m_1: M, m_2: M)	input	A call edge $m_1 \rightarrow m_2$ exists in the call graph
$allocated$(h: H, m: M)	input	Object h allocated within m
$classNode$(c: V, h: H)	input	Class c is given a heap number h
$escapes$(h: H, m: M)	output	Object h escapes m, h need not be allocated within m
$aEscapes$(h: H, m: M)	output	Object h, allocated within m, escapes m
$captured$(h: H, m: M)	output	Object h, allocated within m, does not escape m
$callerCaptured$(h: H, m: M)	output	Object h, allocated within m, escapes m, but does not escape an immediate caller of m

a relation ($dominatingHA$) and every class is uniquely numbered in the heap objects' domain.

$$escapes(h_2, m) \quad :- \quad dominatingHA(m, f, s_0), classNode(-, h_1), hPtsTo(-, f, s_0, h_1, h_2) \tag{18}$$

$$escapes(h, m) \quad :- \quad vPtsTo(-, v, h), formal(m, -, v). \tag{19}$$

$$escapes(h, m) \quad :- \quad return(m, v), vPtsTo(-, v, h). \tag{20}$$

$$escapes(h, -) \quad :- \quad threadParam(v), vPtsTo(-, v, h). \tag{21}$$

$$escapes(h_2, m) \quad :- \quad escapes(h_1, m), hPtsTo(-, f, s_0, h_1, h_2), dominatingHA(m, f, s_0). \tag{22}$$

$$aEscapes(h, m) \quad :- \quad escapes(h, m), allocated(h, m). \tag{23}$$

$$captured(h, m) \quad :- \quad !aEscapes(h, m), allocated(h, m). \tag{24}$$

$$callerCaptured(h, m_1) \quad :- \quad !escapes(h, m_1), aEscapes(h, m_2), callEdge(m_1, m_2). \tag{25}$$

The semantics of these rules are as follows :

- **Rules (18) to (21) : Direct Escape**: These rules determine the objects that directly escape a method m: objects whose reference is stored in a static class variable (Rule 18), objects representing the parameters (Rule 19), objects returned from a method (Rule 20) and thread objects and objects passed to thread methods (Rule 21)
- **Rules (22) to (25): Indirect Escape, Capture and Caller Capture**: Rules (22)-(24) compute the indirectly escaping objects, ie, those reachable via a sequence of object references from a directly escaped object. The remaining objects are '*captured*', ie, those that are inaccessible outside their method of allocation. Finally, Rule (25) computes the *caller-captured* objects, which escape their method of allocation but are captured within a caller method. Such caller-captured objects can be stack allocated in the caller's stack.

Name	Description	Byte codes
jip	Java Interactive Profiler	210K
umldot	UML Diagram Creator	142K
jython	Python Interpreter	295K
jsch	Implementation of SSH	282K
java_cup	Parser Generator	152K
jlex	Lexical Analyzer Generator	91K
check	Checker for JVM	46K
jess	Java Expert Shell	13K
cst	Hashing Implementation	32K
si	Small Interpreter	24K
compress	Modified Lampel-Ziv method	21K
raytrace	Ray tracer	65K
db	Memory Resident Database	13K
anagram	Anagram Generator	9K
mtrt	A variant of raytrace	1K

Fig. 6. Benchmark Programs used

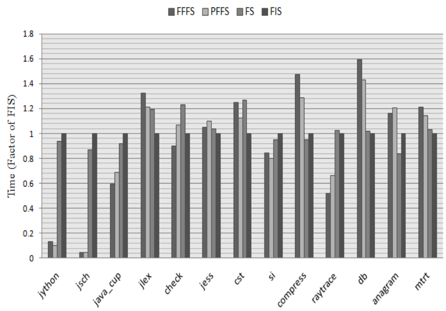

Fig. 7. Time of Analysis

5 Experimental Results

We have implemented both PFFS and FFFS on the *bddbddb* system, using $SOOT$ as the front end to generate Heap Array SSA and the input relations. We ran our analyses on some of the popular Java programs from SourceForge and SPEC JVM 98 benchmark suite (Figure 6). All the programs were run in whole program mode in with a precomputed call graph in $SOOT$.[6] The analyses were done on 4 CPU 3.20 GHz Intel Pentium IV PC with 2 GB of RAM running Ubuntu Linux.

We compare our analyses with two field flow insensitive versions of the Whaley-Lam analysis [1]: one of which is flow sensitive for locals (FS) and the other flow insensitive for locals (FIS). The Whaley-Lam analysis is context sensitive. The FS analysis uses the scalar SSA form to obtain flow sensitivity for locals while FIS does not make use of SSA. The comparisons are based on: (a) Time of analysis (b) Precision in terms of size of points-to sets at load/store statements (c) Number of objects found to be captured in the escape analysis. Figure 7 shows the relative time taken for the four analyses, normalized with respect to FIS. Figure 8 shows the average number of objects pointed to by an object field $x.f$ at a load $y = x.f$ (which we call the *loadPts* set), again as a factor of FIS. This is computed based on the *vPtsTo* relation for x at the load and the *hPtsTo* relation for objects pointed to by x. The average number of objects pointed to by a scalar variable x at a store $x.f = y$ (the *storePts* set), inferred by each analysis is shown in Figure 9.

The time taken by the field flow sensitive versions are comparable to FS and FIS for most programs, while doing much better in some cases. Also, for two

[6] Although we use a precomputed call graph here, this analysis can be combined with a on-the-fly call graph construction using the techniques employed in [1].

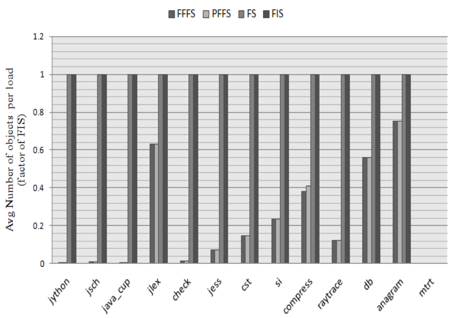

Fig. 8. Average Size of *loadPts* set

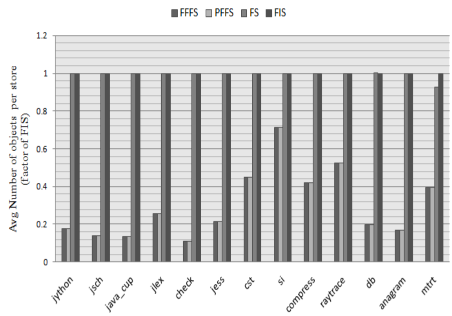

Fig. 9. Average Size of *storePts* set

programs, *jip* and *umldot*, the JVM ran out of memory while running FIS and FS. In terms of precision, both PFFS and FFFS reduce the size of *loadPts* set to about 23% of the size computed by FIS, averaged over all programs. The size of the *storePts* set is reduced to about 30% of the size computed by FIS.

Three observations can be made from these plots: Firstly, these results illustrate the importance of field flow sensitivity in a context sensitive analysis, especially in programs written in a object oriented programming language like Java where short and frequently invoked methods are the common case [11]. A field flow sensitive analysis takes advantage of longer interprocedural program paths that have been identified as distinct from each other by a context sensitive analysis. In the absence of field flow sensitivity, even though a context sensitive analysis identifies longer distinct interprocedural paths, the points-to sets of object fields at various points along the path are merged. Secondly, programs for which the size of *loadPts* as computed by PFFS is less than 10% as that computed by FIS, the analysis time for PFFS is also much less than FIS. Hence field flow sensitivity not only helps in getting a precise pointer analysis result, but also helps in reducing analysis times to some extent, by avoiding the computation of spurious points-to relations (*hPtsTo*) across methods. Finally, as is evident from the size of *loadPts* and *storePts* computed by FFFS and PFFS, there is very little gain in precision by using FFFS as compared to PFFS. This counter-intuitive result can be explained by the following observation: In the absence of (a) support for non stratified queries and (b) an accurate model of the heap, the conditions for strong updates (all of which are required for correctness) are very strong and their scope is limited to only a few field assignments that always update a single runtime object and that too at most once. Although the developers of *bddbddb* did not find the need for non stratified queries for program analysis,[7] the use of Heap array SSA to perform aggressive strong updates does illustrate an instance where non stratified queries are of

[7] We quote, from [12]: "In our experience designing Datalog programs for program analysis, we have yet to find a need for non-stratifiable queries".

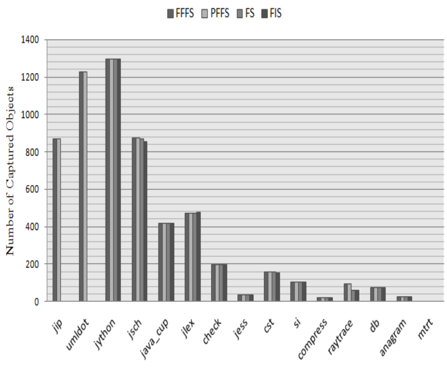

Fig. 10. No. of captured objects

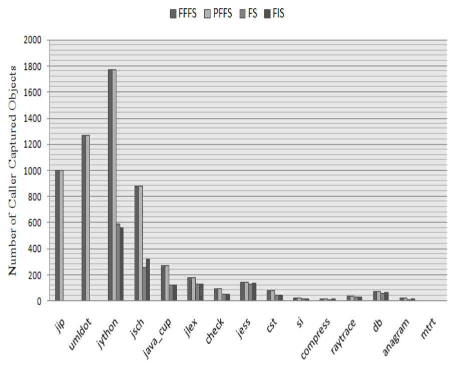

Fig. 11. No. of recaptured objects

importance to a program analysis. In addition, an accurate model of the heap can greatly help in improving precision. A shape analysis builds better abstractions of structure of the heap and enables strong updates much more freely than is currently possible. One of the most popular shape analysis algorithms [13] is based on three-valued logic and to formulate this analysis in the logic programming paradigm, we might need to adopt a different kind of semantics like the well founded semantics, as done in XSB [10], which also handles non-stratified queries.

The *captured* relation sizes are shown in Figure 10. This relation computes the captured state of an object with respect to its method of allocation. As seen from the graph, the number of objects captured in their method of allocation is almost the same for the field flow sensitive and insensitive versions. This is surprising given that PFFS/FFFS have an extra level of precision and hence should have discovered more captured objects. Figure 11 compares the number of *caller-captured* objects (ie, objects that escape their method of allocation, but are caught in their immediate caller) discovered by the four analyses. PFFS improved the number of caller-captured objects by an average of about 71% compared to FIS (again there was not much gain in using FFFS over PFFS). Such caller-captured objects can be stack allocated in the calling method's stack. This is beneficial when coupled with partial specialization of Java methods [14]. Since the computation of *caller-captured* objects takes into account the flow sensitivity of field assignments across two methods, PFFS gives a better *caller-captured* set than *captured*, especially in the presence of context sensitivity. This is because an object that escapes its method of allocation can be captured in more than one of its callers (each in a separate calling context) and an interprocedural field flow sensitive analysis with its dominating heap array information can help in discovering such objects better than a field flow insensitive analysis with no control flow and dominance information.

6 Related Work

Our work is inspired by Whaley and Lam's work on context sensitive analysis [1] and work by Fink et al. [2] on Heap Array SSA. Whaley and Lam also specify a thread escape analysis while we have used the pointer analysis results to infer a *method* escape analysis. One of the earliest context-insensitive and flow-insensitive pointer analysis was due to Anderson [7] which is inclusion based, solved using subset constraints. The analysis of Emami et al [15], formulated for C, is both context and flow sensitive. It computes both *may* and *must* pointer relations and context sensitivity is handled by regarding every path in the call graph as a separate context (cloning). Our analysis is a *may* pointer analysis while for context sensitivity the cloned paths are represented by BDDs using *bddbddb*. Reps describes techniques for performing interprocedural analysis using logic databases and gives the relation between context-free reachability, logic programs and constraint based analyses [6]. All program analysis problems whose logic programs are chain programs are equivalent to a context-free reachability problem. Sridharan and Bodik express a context sensitive, flow insensitive pointer analysis for Java as a context-free reachability (CFL) problem [16]. We could not use CFL due to the presence of logic program rules which were not chain rules, for eg, that for *dphi* function rule that adds the field flow sensitivity.

Milanova et al [17] propose *object sensitivity* as a substitute for context sensitivity using call strings. Object names are used, instead of context numbers, to distinguish the pointer analysis results of a method which can be invoked on them. Object names represent an abstraction of a sequence of object allocation sites on which methods can be invoked. Their analysis is flow insensitive. The field flow sensitive portion of our analysis could be used with object sensitivity instead of context sensitivity. Whaley and Rinard's escape analysis [8], similar to Choi et al's analysis [9], maintains a points-to escape graph at every point in the program and hence is fully field flow sensitive. However, complete context and field flow sensitivity is maintained only for objects that do not escape a method. The pointer information for objects that escape a method are merged.

7 Conclusions

In this paper, we presented two variants of a field flow sensitive analysis for Java using the Heap Array SSA Form in Datalog. We have extended the Heap Array SSA form interprocedurally to obtain field flow sensitivity in the presence of context sensitivity and derived an escape analysis based on this. We have implemented our analysis using *SOOT* and *bddbddb*. Our results indicate that partial field flow sensitivity obtains significant improvements in the precision of a context sensitive analysis and helps to identify more captured objects at higher levels in the call chain. Strong updates do not seem to lead to any further gain in precision in current system. The running times of our analysis are comparable to a field flow insensitive analysis, while in some cases running much faster than the latter.

Acknowledgments

We thank John Whaley for the *bddbddb* system and clarifying some of our doubts regarding his paper. We thank the people at McGill for the *SOOT* framework.

References

1. Whaley, J., Lam, M.S.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: Programming language design and implementation, pp. 131–144 (2004)
2. Fink, S.J., Knobe, K., Sarkar, V.: Unified analysis of array and object references in strongly typed languages. In: Static Analysis Symposium, pp. 155–174 (2000)
3. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. 13(4), 451–490 (1991)
4. Hasti, R., Horwitz, S.: Using static single assignment form to improve flow-insensitive pointer analysis. In: Programming language design and implementation, pp. 97–105 (1998)
5. Knobe, K., Sarkar, V.: Array SSA form and its use in parallelization. In: Symposium on Principles of Programming Languages, pp. 107–120 (1998)
6. Reps, T.W.: Program analysis via graph reachability. In: International Logic Programming Symposium, pp. 5–19 (1997)
7. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. PhD thesis, DIKU, University of Copenhagen (May 1994)
8. Whaley, J., Rinard, M.: Compositional pointer and escape analysis for Java programs. In: Object-oriented programming, systems, languages, and applications, pp. 187–206 (1999)
9. Choi, J.D., Gupta, M., Serrano, M., Sreedhar, V.C., Midkiff, S.: Escape analysis for Java. In: Object-oriented programming, systems, languages, and applications, pp. 1–19 (1999)
10. Sagonas, K., Swift, T., Warren, D.S.: XSB as an efficient deductive database engine. In: International conference on Management of data, pp. 442–453 (1994)
11. Budimlic, Z., Kennedy, K.: Optimizing Java: theory and practice. Concurrency: Practice and Experience 9(6), 445–463 (1997)
12. Whaley, J., Avots, D., Carbin, M., Lam, M.S.: Using Datalog and binary decision diagrams for program analysis. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, Springer, Heidelberg (2005)
13. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3–valued logic. In: Symposium on Principles of Programming Languages, pp. 105–118 (1999)
14. Schultz, U.P., Lawall, J.L., Consel, C.: Automatic program specialization for Java. ACM Trans. Program. Lang. Syst. 25(4), 452–499 (2003)
15. Emami, M., Ghiya, R., Hendren, L.J.: Context-sensitive interprocedural points-to analysis in the presence of function pointers. In: Programming language design and implementation, pp. 242–256 (1994)
16. Sridharan, M., Bodík, R.: Refinement-based context-sensitive points-to analysis for Java. In: Programming language design and implementation, pp. 387–400 (2006)
17. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to and side-effect analyses for Java. In: International Symposium on Software testing and analysis, pp. 1–11 (2002)