Field-Sensitive Value Analysis by Field-Insensitive Analysis

Elvira Albert[1], Puri Arenas[1], Samir Genaim[1], and Germán Puebla[2]

[1] DSIC, Complutense University of Madrid (UCM), Spain
[2] CLIP, DLSIIS, Technical University of Madrid (UPM), Spain

Abstract. Shared and mutable data-structures pose major problems in static analysis and most analyzers are unable to keep track of the values of *numeric variables* stored in the heap. In this paper, we first identify sufficient conditions under which heap allocated numeric variables in object oriented programs (i.e., numeric fields) can be handled as non-heap allocated variables. Then, we present a static analysis to infer which numeric fields satisfy these conditions at the level of (sequential) *bytecode*. This allows instrumenting the code with *ghost* variables which make such numeric fields observable to any field-insensitive value analysis. Our experimental results in termination analysis show that we greatly enlarge the class of analyzable programs with a reasonable overhead.

1 Introduction

Static analyses which approximate the value of numeric variables have a large application field which includes its use for invariant generation, for finding ranking functions [15] which bound the number of iterations of loops in cost analysis, etc. Most existing *value* analyses are only applicable to numeric variables which satisfy two conditions: (1) all occurrences of a variable refer to the same memory location, and (2) memory locations can only be modified using the corresponding variable. Some notable exceptions are [8,11,10]. In general, the conditions above are not satisfied when numeric variables are stored in shared mutable data structures such as the heap. Condition (1) does not hold because memory locations (numeric variables) are accessed using reference variables, whose value can change during the execution. Condition (2) does not hold because a memory location can be modified using different references which are aliases and point to such memory location.

Example 1. Consider the following loop where *size* is a field of integer type:

while (x.f.size > 0) {*i=i+y.size;* *x.f.size=x.f.size-1;*}

This loop terminates in sequential execution because $x.f.size$ decreases at each iteration and, for any initial value of $x.f.size$, there are only a finite number of values which $x.f.size$ can take before reaching zero. Unfortunately, applying standard value analyses on numeric fields can produce wrong results, and further conditions are required. E.g., if we add the instruction $x{=}x.next$; within the loop body, the memory location pointed to by $x.f$ changes, invalidating Condition 1. Also, if we add $y.size$++; as $x.f$ and y may be aliases, Condition 2 is false. □

A. Cavalcanti and D. Dams (Eds.): FM 2009, LNCS 5850, pp. 370–386, 2009.

This paper presents a novel approach for approximating the value of *numeric fields* in object-oriented programs which greatly improves the precision over existing field-insensitive value analyses while introducing a reasonable overhead. Our approach is developed for object-oriented *bytecode*, i.e., code compiled for *virtual machines* such as the Java virtual machine [9] or .NET, and consists of the following steps: (1) partition the program to be analyzed into *scopes*, (2) identify *trackable* numeric fields which meet the above conditions and hence can be safely handled by field-insensitive value analysis, (3) transform the program by introducing local *ghost* variables whose values represent the values of the corresponding numeric fields, and (4) analyze the transformed program scope by scope using existing field-insensitive value analysis. This allows reusing the large body of work devoted to numerical static analysis: polyhedra [7], intervals [6], octagons [12], etc.

Example 2. Consider the loop in Ex. 1, with a single scope. There are three program points where a numeric field with signature *size* is accessed for reading and one where it is accessed for writing. In this paper, we develop a *Reference Constancy Analysis* (RCA for short) which is able to infer that the references used in all four accesses are *constant* in the sense that, in all iterations of the loop, such references do not change their value. For brevity, in the rest of the paper we say that an access is constant to indicate that the reference used in the corresponding program point is constant. Our analysis also provides a symbolic representation of such values. This allows determining that the two read accesses and the write access through $x.f.size$ not only are constant but also they have the same value in the three different program points. This is sufficient for guaranteeing Condition 1 above. Besides, since in the loop there are no other write accesses using the signature *size*, Condition 2 above is also guaranteed. Thus, we can safely introduce a ghost variable, which becomes local variable v, and corresponds to the value of the numeric field $x.f.size$ in all three program points. As regards the read access $y.size$, RCA is able to prove that it is constant. However, Condition (2) cannot be proved since by looking at the loop alone it is not possible to know whether $x.f$ and y are aliases. Therefore no ghost variable can be introduced for $y.size$. The transformed loop is as follows:

$v = x.f.size$; *while* $(v{>}0)$ $\{$*i=i+y.size; x.f.size=x.f.size-1; v=v-1;*$\}$

Read accesses to $x.f.size$ are replaced by equivalent accesses to the ghost variable v. For write accesses, we keep the original access and replicate it using the corresponding ghost variable. This is because there may be aliases for $x.f.size$ outside the loop which may need the value of the original numeric field. A standard value analysis can now infer that v decreases, which guarantees termination. □

2 The Bytecode Language in Rule-Based Form

Since reasoning about bytecode programs is complicated, it is customary to formalize analyses on intermediate representations of the bytecode (e.g., [18]). We consider a simplified form of the rule-based *recursive* language of [2]. A *bytecode program* consists of a set of *procedures* and classes. A procedure p with k input arguments $\bar{x}{=}x_1,\ldots,x_k$ and m output arguments $\bar{y}{=}y_1,\ldots,y_m$ is defined by one or more *guarded rules*. Without loss of generality, we assume that there are no two procedures with the same name and different number of arguments. Though Java bytecode methods only have one output argument, we allow multiple output arguments since, as discussed in Sec. 4, our program transformation may introduce additional output arguments. Rules are defined as:

$$\begin{array}{rlrl} \mathit{rule} ::= & p(\langle\bar{x}\rangle,\langle\bar{y}\rangle) \leftarrow g, b_1,\ldots,b_t & g ::= & \mathsf{true} \mid \mathit{bexp}_1\ \mathit{op}\ \mathit{bexp}_2 \mid \mathsf{type}(x,c) \\ b ::= & x{:=}\mathit{exp} \mid x{:=}\mathsf{new}\ c \mid x{:=}y.f \mid x.f{:=}y \mid q(\langle\bar{x}\rangle,\langle\bar{y}\rangle) & & \\ \mathit{bexp} ::= & x \mid \mathsf{null} \mid n & \mathit{exp} ::= & \mathit{bexp} \mid x{-}y \mid x{+}y \mid x{*}y \mid x/y \\ \mathit{op} ::= & > \ \mid\ < \ \mid\ \leq \ \mid\ \geq \ \mid\ = \ \mid\ \neq & & \end{array}$$

where $p(\langle\bar{x}\rangle,\langle\bar{y}\rangle)$ is the *head* of the rule; g its guard, i.e., necessary conditions for the rule to be applicable; $b_1,\ldots,b_t$ the body of the rule; n an integer; x and y variables; f a field signature (i.e., globally unique), and $q(\langle\bar{x}\rangle,\langle\bar{y}\rangle)$ a procedure call (by value). We often do not write guards which are true. The language supports class definition, object creation, field manipulation, and type comparison through the instruction $\mathsf{type}(x,c)$, which succeeds if the runtime class of x is exactly c. A class c is a finite set of typed field names, where the type can be integer or a class name. The key features of this language are: (1) *recursion* is the only iteration mechanism, (2) *guards* are the only form of conditional, (3) there is no operand stack, (4) objects can be seen as records, and the behaviour induced by dynamic dispatch is compiled into *dispatch* blocks guarded by type checks, and (5) rules may have *multiple* return values. The translation from (Java) bytecode to the rule-based form is performed in two steps. First, a *control flow graph* (CFG) is built. Second, a *rule* is defined for each block and the operand stack is *flattened* by considering its elements as local variables [2].

We now introduce some terminology used to define an *operational semantics* for rule-based bytecode. An *activation record* is of the form $\langle p, bc, tv\rangle$, where p is a procedure name, bc is a possibly empty sequence of instructions and tv a variable mapping. Executions proceed between *configurations* of the form $A; h$, where A is a stack of activation records (which grows leftward) and h is the *heap*, i.e., a partial mapping from an infinite set of *memory locations* to *objects*. We use $h(r)$ to denote the object referred to by r in h and $h[r \mapsto o]$ to indicate the result of updating the heap h by making $h(r) = o$. An object o is a pair consisting of the object class tag and a mapping from field names to values which is consistent with the types of the fields. We use $o.f$ or $o(f)$ to refer to the value of the field f in the object o, and $o[f{\mapsto}v]$ to set the value of $o.f$ to v. The operational semantics is quite standard and consists of the following rules:

$$(1)\ \frac{b \equiv x{:=}\mathit{exp}, \quad v = \mathit{eval}(\mathit{exp}, tv)}{\langle p, b{\cdot}bc, tv\rangle{\cdot}A; h \rightsquigarrow \langle p, bc, tv[x \mapsto v]\rangle{\cdot}A; h}$$

$$(2)\ \frac{b \equiv x{:=}\mathsf{new}\ c, \quad o{=}\mathsf{newobject}(c), \quad r{\notin}\mathit{dom}(h)}{\langle p, b{\cdot}bc, tv\rangle{\cdot}A; h \rightsquigarrow \langle p, bc, tv[x \mapsto r]\rangle{\cdot}A; h[r \mapsto o]}$$

$$(3)\ \frac{b \equiv x{:=}y.f, \quad tv(y) \in \mathrm{dom}(h), \quad o = h(tv(y))}{\langle p, b{\cdot}bc, tv\rangle{\cdot}A; h \rightsquigarrow \langle p, bc, tv[x \mapsto o.f]\rangle{\cdot}A; h}$$

$$(4)\ \frac{b \equiv x.f{:=}y, \quad r = tv(x) \in \mathrm{dom}(h), \quad o = h(r)}{\langle p, b{\cdot}bc, tv\rangle{\cdot}A; h \rightsquigarrow \langle p, bc, tv\rangle{\cdot}A; h[r \mapsto o[f \mapsto tv(y)]]}$$

$$(5)\ \frac{\begin{array}{c} b \equiv q(\langle \bar{x}\rangle, \langle \bar{y}\rangle),\ \text{there is a program rule } q(\langle \bar{x}'\rangle, \langle \bar{y}'\rangle) {\leftarrow} g, b_1, \cdots, b_t \\ \text{such that } tv'{=}newenv(q),\ \forall i.tv'(x_i') = tv(x_i),\ eval(g, tv') = true \end{array}}{\langle p, b{\cdot}bc, tv\rangle{\cdot}A; h \rightsquigarrow \langle q, b_1 \cdot \ldots \cdot b_t, tv'\rangle{\cdot}\langle p[\bar{y}, \bar{y}'], bc, tv\rangle{\cdot}A; h}$$

$$(6)\ \frac{}{\langle q, \epsilon, tv\rangle{\cdot}\langle p[\bar{y}, \bar{y}'], bc, tv'\rangle{\cdot}A; h \rightsquigarrow \langle p, bc, tv'[\bar{y} \mapsto tv(\bar{y}')]\rangle{\cdot}A; h}$$

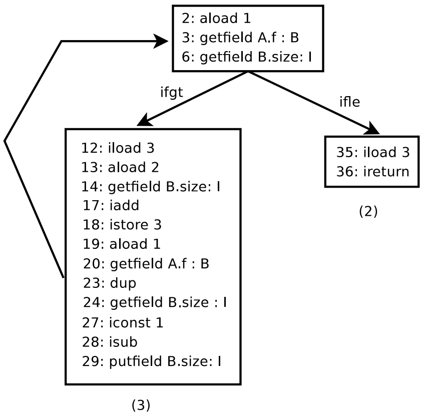
Intuitively, rule (1) accounts for all rules in the bytecode semantics which perform operations on variables. The evaluation $eval(exp, tv)$ returns the evaluation of the arithmetic or Boolean expression exp for the values of the corresponding variables from tv in the standard way, and for reference variables, it returns the reference. Rules (2), (3) and (4) deal with objects. We assume that newobject(c) creates a new object of class c and initializes its fields to either 0 or null, depending on their types. Rule (5) (resp., (6)) corresponds to calling (resp., returning from) a procedure. The notation $p[\bar{y}, \bar{y}']$ records the association between the formal and actual return variables. It is assumed that $newenv$ creates a new mapping of local variables for the corresponding method, where each variable is initialized as newobject does. Guards in different rules for the same procedure are always mutually exclusive. Execution is thus deterministic. An execution starts from an *initial configuration* $\langle start, p(\langle \bar{x}\rangle, \langle \bar{y}\rangle), tv\rangle; h$ and ends in a *final configuration* $\langle start, \epsilon, tv'\rangle; h'$ where $start$ is a marker for the initial entry which is guaranteed not to coincide with any procedure name, tv and h are initialized to suitable initial values, and tv' and h' include the final values. Program executions can be represented as *traces* $C_0 {\rightsquigarrow} C_1 {\rightsquigarrow} \cdots {\rightsquigarrow} C_\omega$, where C_ω is a final configuration. We use $C {\rightsquigarrow}^* C'$ to denote that the execution starting from C reaches C' in a finite number of steps. Non terminating executions have infinite traces.

Example 3. Consider the following rule-based form and bytecode (inside its CFG) corresponding to the method in Ex. 1 plus a final *return i*; instruction:

(1) $loop(\langle x, y, i\rangle, \langle r\rangle) {\leftarrow}$
$s_0{:=}x,$
$\textcircled{1} s_0{:=}s_0.f,$
$\textcircled{2} s_0{:=}s_0.size,$
$loop_c(\langle x, y, i, s_0\rangle, \langle r\rangle).$

(2) $loop_c(\langle x, y, i, s_0\rangle, \langle r\rangle) {\leftarrow}$
$\mathbf{s_0 \leq 0}, s_0{:=}i, r{:=}s_0.$

(3) $loop_c(\langle x, y, i, s_0\rangle, \langle r\rangle) {\leftarrow}$
$\mathbf{s_0 > 0}, s_0{:=}i, s_1{:=}y,$
$\textcircled{3} s_1{:=}s_1.size,$
$s_0{:=}s_0{+}s_1, i{:=}s_0, s_0{:=}x,$
$\textcircled{4} s_0{:=}s_0.f,\ \ s_1{:=}s_0$
$\textcircled{5} s_1{:=}s_1.size,$
$s_2{:=}1, s_1{:=}s_1{-}s_2,$
$\textcircled{6} s_0.size{:=}s_1, loop(\langle x, y, i\rangle, \langle r\rangle).$



```
2: aload 1
3: getfield A.f : B
6: getfield B.size: I
```

```
12: iload 3
13: aload 2
14: getfield B.size: I
17: iadd
18: istore 3
19: aload 1
20: getfield A.f : B
23: dup
24: getfield B.size : I
27: iconst 1
28: isub
29: putfield B.size: I
```

```
35: iload 3
36: ireturn
```

Variable names of the form s_i indicate that they originate from stack positions. Each block in the CFG is translated into a rule. The conditions on the edges become guards for the corresponding rules. Bytecode instructions are converted to a new representation. E.g., in the rule for block (2), the guard $\mathbf{s_0{\leq}0}$ corresponds to the condition ifle and iload 3 (3 refers to the third local variable i) is converted to $s_0{:=}i$. Instruction $s_1{:=}s_0$ corresponds to dup. Numbered circles are program point markers introduced for later reference. A is a class with a field of type B, and B is a class with an integer field. □

3 Reference Constancy Analysis

We present a *reference constancy analysis*, which aims at identifying reference variables which are constant at certain program points. The *program points* considered are the union of the program points of all program rules. All program points are made unique by numbering the program rules. The k-th program rule $p(\langle\bar{x}\rangle, \langle\bar{y}\rangle) \leftarrow g, b_1^k, \ldots, b_t^k$ has $t+1$ program points. The first one, $(k, 0)$, after the execution of the guard g and before the execution of b_1, then $(k, 1)$ between the execution of b_1 and b_2, until (k, t) after the execution of b_t. The analysis receives as input a program P and a procedure name p, which we refer to as *entry*. For any configuration $C = \langle q, b_i^k \cdot bc, tv\rangle \cdot A; h$ which is not initial, the program point to which C corresponds is $(k, i-1)$. Given a program P, we denote by $RF(P)$ (resp. $NF(P)$) the set of reference (resp. numeric) field signatures declared in P.

Definition 1 (access path function). *An* access path function *for a program P and an entry p is a syntactic construction of the form $l_j.f_1...f_n$, with $f_i \in RF(P)$ for $i = 1, \ldots, n$ and it represents a partial function from initial configurations to references. Given an initial configuration $C = \langle start, p(\langle\bar{x}\rangle, \langle\bar{y}\rangle), tv\rangle; h$ we define $l_j.f_1...f_n(C) \equiv h(\cdots(h(h(tv(l_j))(f_1))(f_2))\cdots)(f_n)$.*

Essentially, for determining the value of an access path in an initial configuration, we use the variable table and heap at such configuration in order to dereference w.r.t. the reference variable and reference fields in the access path. This function is undefined at paths that traverse objects which have not been allocated in the heap. Otherwise, it either returns a memory location in dom(h) or the value null. Equivalent notions have been defined for other languages (see, e.g. [1]).

Definition 2 (constant reference variable). *A reference variable z is* constant *at a program point (k, i) in a program P for an entry p w.r.t. the access path function $l_j.f_1 \ldots f_n$ if $\forall C \rightsquigarrow^* C'$ such that C is an initial configuration and $C' = \langle q, b_{i+1}^k \cdot bc, tv'\rangle \cdot A; h'$, we have $tv'(z) = l_j.f_1 \ldots f_n(C)$.*

Intuitively, a reference is constant w.r.t. an access path $l_j.f_1 \ldots f_n$ in a program point if, starting the execution from any initial configuration C, whenever we reach a configuration C' which corresponds to such program point, the reference always has the same value $l_j.f_1 \ldots f_n(C)$. Note that if execution reaches C' then $l_j.f_1 \ldots f_n(C)$ is defined since otherwise we must have attempted to dereference a null reference or a dangling pointer. In either case, the derivation would stop.

The idea behind RCA is similar in spirit to that of the classical numeric constant propagation analysis [6]. However, an important feature of RCA is that the values which are computed are not absolute constants but rather functions which, when provided with a particular initial configuration, return a fixed value in terms of the heap at the initial –and not the current– configuration.

Example 4. Consider the examples below (shown in Java source for clarity). We use l_1 and l_2 to represent the initial values of x and y, respectively.

while (x.f.getSize() $>$ 0)
  i+=y.getSize();
  x.f.setSize(x.f.getSize()-1);
ⓐ
	if ($k > 0$) then x=z else x=y;
	x.f=10;
	for(; i<x.f; i++)
	ⓑ b[i]=x.b[i];
		while (x != null) {
		  for(; x.c<n; x.c++)
		    value[x.c]++;
		ⓒx=x.next;}

while (x.size $<$ 10)
ⓓ {x.size++; x=x.next;}
	while (x.size $<$ 10)
	ⓔ {x.size++; acc+=y.size;}
		while (x.r.size $<$ 10)
		ⓕ {x.r.size++; y.r=z;}

Program ⓐ will be discussed later. In ⓑ, the reference x remains constant w.r.t. l_1 within the loop. However, if we consider the whole code fragment, x is no longer constant, since x can take two different values before the loop. In ⓒ, all occurrences of x are constant w.r.t. the same access path function, l_1, within the inner loop. However, x takes different values in different iterations of the outer loop, and thus x is not constant in the whole code fragment. In ⓓ, x is not constant because it is updated at each iteration of the loop. In ⓔ, x is constant w.r.t l_1 and y is constant w.r.t l_2, but it is unknown whether l_1 and l_2 are identical or not. In ⓕ, it cannot be ensured that $x.r$ is constant, since if x and y are aliases, updating $y.r$ changes $x.r$. □

3.1 A Global Reference Constancy Analysis for Bytecode

We assume familiarity with the concepts of *abstract interpretation* [6]. The basic idea of abstract interpretation is to infer information on programs by interpreting ("running") them using abstract values rather than concrete ones, thus obtaining safe approximations of the behavior of the program. Essentially, programs are interpreted over an *abstract domain* (D_α) which is simpler than the corresponding *concrete domain* (D). An abstract state in D_α is a finite representation of a possibly infinite set of actual states in D.

Definition 3 (access path). *An* access path *for a variable y at a given program point (k, j) is a syntactic construction which can take the forms:*

- ℓ_{any}*. Variable y is not guaranteed to be constant at (k, j).*
- ℓ_{num} *(resp. ℓ_{null}). Variable y holds a numeric value (resp.* null*) at (k, j).*
- $l_i.f_1 \ldots f_n$*. Variable y is* constant *w.r.t $l_i.f_1 \ldots f_n$ at (k, j).*

We use AP to denote the set of all access paths. Given $\ell_1, \ell_2 \in AP$, we define $\ell_1 \sqcup_{ap} \ell_2$ to be ℓ_2 if $\ell_1 = \ell_2$ and ℓ_{any} otherwise. An abstract state over a set of variables $\mathcal{V}$ and a set of reference fields $RF(P)$ has the form $\langle \phi, \theta \rangle$ where $\phi : \mathcal{V} \mapsto AP$ maps variables to access paths, and $\theta \subseteq RF(P)$ contains a set of reference field signatures which are guaranteed to be constant in the sense that such field has not been updated w.r.t. its value at the initial configuration in any object of the class where f is declared. We say $\langle \phi_1, \theta_1 \rangle \sqsubseteq_{as} \langle \phi_2, \theta_2 \rangle$ if $\theta_2 \subseteq \theta_1$ and $\forall x \in \mathcal{V}$

either $\phi_1(x) = \phi_2(x)$ or $\phi_2(x) = \ell_{\mathsf{any}}$. We define $\langle \phi_1, \theta_1 \rangle \sqcup_{as} \langle \phi_2, \theta_2 \rangle = \langle \phi, \theta_1 \cap \theta_2 \rangle$ s.t. $\phi(x) = \phi_1(x) \sqcup_{as} \phi_2(x)$. AS_d is the lattice $\langle AS, \top_{as}, \bot_{as}, \sqcup_{as}, \sqsubseteq_{as} \rangle$ where AS is the set of abstract states, $\top_{as}$ is the top of the lattice which is equal to $\langle \phi, \emptyset \rangle$ s.t $\forall x \in \mathcal{V}.\phi(x) = \ell_{\mathsf{any}}$, and $\bot_{as}$ is the bottom.

RCA assigns an abstract state from AS to each program point by relying on the transfer function $\tau : \mathit{Instr} \times AS \mapsto AS$ depicted in the following table:

b	$\tau(b, \langle \phi, \theta \rangle)$	*conditions*	b	$\tau(b, \langle \phi, \theta \rangle)$	*conditions*
(1) $x{:=}y.f$	$\langle \phi[x \mapsto \ell], \theta \rangle$	$f \in RF(P)$	(6) $x{:=}\mathsf{null}$	$\langle \phi[x \mapsto \ell_{\mathsf{null}}], \theta \rangle$	
(2) $x.f{:=}y$	$\langle \phi, \theta \backslash \{f\}] \rangle$	$f \in RF(P)$	(7) $x{:=}exp$	$\langle \phi[x \mapsto \ell_{\mathsf{num}}], \theta \rangle$	$exp \not\equiv \mathsf{null}$
(3) $x{:=}y$	$\langle \phi[x \mapsto \phi(y)], \theta \rangle$		(8) $x \neq \mathsf{null}$	$\bot_{as}$	$\phi(x){=}\ell_{\mathsf{null}}$
(4) $x{:=}y.f$	$\langle \phi[x \mapsto \ell_{\mathsf{num}}], \theta \rangle$	$f \in NF(P)$	(9) $x.f{:=}y$	$\langle \phi, \theta \rangle$	$f{\in}NF(P)$
(5) $x{:=}\mathsf{new}\ c$	$\langle \phi[x \mapsto \ell_{\mathsf{any}}], \theta \rangle$		(10) *otherwise*	$\langle \phi, \theta \rangle$	

where in (1) ℓ is defined as: *if* $f \in \theta$ *and* $\phi(y) \neq \ell_{\mathit{any}}$ *then* $\ell = \phi(y).f$ *else* $\ell = \ell_{\mathit{any}}$ and *Instr* denotes the set of all possible instructions that can appear in the body of a rule. Note that $\tau(b, \bot_{as}) = \bot_{as}$. In (1), when a reference variable x is assigned the value of a (reference) field, the transfer function updates the access path of x accordingly. In (2), when a reference field with signature f is assigned a value, f is eliminated from θ, since we can no longer guarantee that fields with the f signature preserve their initial value. Note that if in a subsequent program point, a reference variable x is assigned a field with the f signature, then the access path for x becomes ℓ_{any} in rule (1). This is needed to guarantee correctness w.r.t. Condition 1 in Sec. 1. In (3), assignments between variables are handled by just assigning their access paths as well. This allows capturing equality of access paths: if the analysis computes the same access path function ℓ for two reference variables x and y then x and y refer to the same memory location or they are both null. Although this notion is related to aliasing, note that we do not propagate aliasing information among scopes, but rather we concentrate on computing aliasing information which is guaranteed to hold regardless of the contents of the heap at the initial configuration. This allows analyzing scopes separately. In (4) and (7) numeric variables are abstracted to ℓ_{num}. They will be the target of the subsequent value analysis performed after instrumentation. In (5), when a new object is created in a fresh memory location r which is associated to a reference variable x, x is given ℓ_{any} as access path, as r does not exist in the heap at the initial configuration. In (8), if the abstract state tells us that a variable x definitely has the value null and we encounter a guard which checks that x is not null then such guard is guaranteed to fail and the rest of the rule will not be executed. This is captured by the abstract state $\bot_{as}$ which represents unreachable configurations. The remaining instructions do not alter reference constancy information. The transfer function is used to define a set of data-flow equations, whose least solution provides the reference constancy information. Below, $\bar{\exists}\bar{w}$ denotes the projection on $\bar{w}$ (i.e., eliminates all variables not in $\bar{w}$).

Definition 4 (RCA). *Given a program P and an entry p, the set of reference constancy equations of P w.r.t. p, denoted E_P^p (or E_P or E when it is clear from the context), is defined as follows:*

1. *The entry p contributes the equation $p_{\downarrow}(\bar{x}) = \langle\phi, \theta\rangle$ where ϕ maps each reference variable x_i to a symbolic reference l_i and numeric variables to ℓ_{num}, and $\theta = RF(P)$;*
2. *Each rule $R^k \equiv p(\langle\bar{x}\rangle, \langle\bar{y}\rangle){\leftarrow}g, b_1^k, \ldots, b_n^k \in P$, s.t. $\bar{z} = vars(R)$, contributes:*
   - *(a) an initial equation $e_0^k(\bar{z}){=}\tau(g, \mathsf{init}(p_{\downarrow}(\bar{x}), \bar{z} \setminus \bar{x}))$ such that $\mathsf{init}(\langle\phi, \theta\rangle, \bar{v}) = \langle\phi[v_i{\mapsto}\ell], \theta\rangle$, where $\ell{=}\ell_{\mathsf{null}}$ if v_i is a reference variable, otherwise $\ell{=}\ell_{\mathsf{num}}$;*
   - *(b) for each b_j^k:*
     - *i. if b_j^k is an instruction, we generate $e_j^k(\bar{z}) = \tau(b_j^k, e_{j-1}^k(\bar{z}))$;*
     - *ii. if b_j^k is a call of the form $q(\langle\bar{w}\rangle, \langle\bar{s}\rangle)$, we generate $q_{\downarrow}(\bar{w}) = \bar{\exists}\bar{w}.e_{j-1}^k(\bar{z})$ and $e_j^k(\bar{z}) = \mathsf{extend}(e_{j-1}^k(\bar{z}), q_{\uparrow}(\bar{s}))$ s.t. $\mathsf{extend}(\langle\phi_1, \theta_1\rangle, \langle\phi_2, \theta_2\rangle) = \langle\phi_1[s_i \mapsto \phi_2(s_i)], \theta_1 \cap \theta_2\rangle$ for $s_i \in \mathrm{dom}(\phi_2)$;*
   - *(c) a final equation $p_{\uparrow}(\bar{y}) = \bar{\exists}\bar{y}.e_n^k(\bar{z})$.* □

Point 1 above indicates that on entry to $p(\bar{x})$ each x_i trivially holds its initial value l_i and all reference field signatures are constant. Point 2 traverses all rules in P. In Point 2a, we initialize the value of all variables in R^k which are not input arguments, which results in equation $e_0^k(\bar{z})$. Point 2(b)i states that if we have an instruction b_i^k, information for the next program point $e_i^k(\bar{z})$ can simply be obtained as $\tau(b_i^k, e_{i-1}^k(\bar{z}))$. Point 2(b)ii states that if b_i^k is a call $q(\bar{w}, \bar{s})$ then: the first equation declares a call $q_{\downarrow}(\bar{w})$ using $e_{i-1}^k(\bar{z})$ as initial input values, the second one uses the exit information of q, namely $q_{\uparrow}(\bar{s})$, together with the previously computed $e_{i-1}^k(\bar{z})$ in order to generate the analysis information at the next program point $e_i^k(\bar{x})$. In point 2c, we obtain information about the exit state for p, denoted $p_{\uparrow}(\bar{y})$, by removing all non-output variables from the information at the last program point in R^k, i.e., $e_n^k(\bar{z})$.

Once the set of equations E_P^p is generated, the analysis results are obtained by computing the least solution of E_P^p, which assigns an abstract element $\langle\phi, \theta\rangle \in AS$ to each equation. This can be done by bottom-up iterations, where we start from an initial solution $\bot_{as}$ for all equations and, then, at each iteration we use the results from the previous iteration in order to obtain a new solution for E_P^p. To ensure termination, new abstract states are merged with previous states using $\sqcup_{as}$ such that if a variable takes two different access paths, it becomes ℓ_{any}. As customary, $\sqcup_{as}$ merges the analysis results obtained for the different rules defining a procedure. The least solution of E_P^p over AS_d is denoted $I(E_P^p)$.

Example 5. Let *loop* be the entry of the program in Ex. 3. The initial equation for *loop* is $loop_{\downarrow}(x, y, i){=}\langle\{x{\mapsto}l_1, y{\mapsto}l_2, i{\mapsto}\ell_{\mathsf{num}}\}, \{f\}\rangle$. The equations contributed by rule 1, where $\bar{z}{=}\{x, y, i, s_0, r\}$, $\bar{w}{=}\{x, y, i, s_0\}$ and $\bar{w}'{=}\{x, y, i\}$ are shown at the end of the example. The fixed point computation proceeds as follows. From $loop_{\downarrow}(\bar{w}')$ we generate e_0^1, which adds the initial values for s_0 and r. Then e_0^1 is used to learn the information for e_1^1 and so on. Note the change in value of s_0 in equations e_1^1 and e_2^1. Once we learn the information for e_3^1, we declare that we have a call to procedure $loop_c$. This is done by equation $loop_{c\downarrow}(\bar{w})$, which in turn activates (in the next iteration) the computation for the equations for $loop_c$ (rules 2 and 3). In each iteration, the new information is merged with that of the previous iterations using $\sqcup_{as}$. Column "analysis results" shows the information obtained once a fixpoint has been reached.

rule 1	analysis result
$e_0^1(\bar{z})=\tau(\mathsf{true}, \mathsf{init}(loop_{\downarrow}(\bar{w}'), \{s_0, r\}))$	$\langle\{x\mapsto l_1, y\mapsto l_2, s_0\mapsto \ell_{\mathsf{null}}, i\mapsto \ell_{\mathsf{num}}, r\mapsto \ell_{\mathsf{num}}\}, \{f\}\rangle$
$e_1^1(\bar{z})=\tau(s_0{:=}x, e_0^1(\bar{z}))$	$\langle\{x\mapsto l_1, y\mapsto l_2, \underline{s_0\mapsto l_1}, i\mapsto \ell_{\mathsf{num}}, r\mapsto \ell_{\mathsf{num}}\}, \{f\}\rangle$
$e_2^1(\bar{z})=\tau(s_0{:=}s_0.f, e_1^1(\bar{z}))$	$\langle\{x\mapsto l_1, y\mapsto l_2, \underline{s_0\mapsto l_1.f}, i\mapsto \ell_{\mathsf{num}}, r\mapsto \ell_{\mathsf{num}}\}, \{f\}\rangle$
$e_3^1(\bar{z})=\tau(s_0{:=}s_0.size, e_2^1(\bar{z}))$	$\langle\{x\mapsto l_1, y\mapsto l_2, s_0\mapsto \ell_{\mathsf{num}}, i\mapsto \ell_{\mathsf{num}}, r\mapsto \ell_{\mathsf{num}}\}, \{f\}\rangle$
$loop_{c\downarrow}(\bar{w})=\bar{\exists}\bar{w}.e_3^1(\bar{z})$	$\langle\{x\mapsto l_1, y\mapsto l_2, s_0\mapsto \ell_{\mathsf{num}}, i\mapsto \ell_{\mathsf{num}}\}, \{f\}\rangle$
$e_4^1(\bar{z})=\mathsf{extend}(e_3^1(\bar{z}), loop_{c\uparrow}(r))$	$\langle\{x\mapsto l_1, y\mapsto l_2, s_0\mapsto \ell_{\mathsf{num}}, i\mapsto \ell_{\mathsf{num}}, r\mapsto \ell_{\mathsf{num}}\}, \{f\}\rangle$
$loop_{\uparrow}(r)=\bar{\exists}r.e_4^1(\bar{z})$	$\langle\{r\mapsto \ell_{\mathsf{num}}\}, \{f\}\rangle$

3.2 Compositional Reference Constancy Analysis

We now present a *compositional* RCA which can be used for analyzing different parts of the program, i.e., scopes, separately.

Example 6. Consider the program ⓐ in Ex. 4. It is similar to the one in Ex. 3, but we introduce two auxiliary methods *getSize* and *setSize* defined, resp., as follows: "*int getSize(){return this.size;}*" and "*void setSize(int n){this.size=n;}*" and whose rule-based forms are, "$getSize(\langle this\rangle, \langle r\rangle){\leftarrow} s_0{:=}this, s_0{:=}s_0.size, r{:=}s_0$" and "$setSize(\langle this, n\rangle, \langle\rangle){\leftarrow} s_0{:=}this, s_1{:=}n, s_0.size{:=}s_1$", respectively. Now, the read accesses to field *size* (at program points ②, ③, and ⑤) are replaced by calls to *getSize* and the write access at program point ⑥ by a call to *setSize*. Note that now, in the whole program, instead of three, we only have one read access ($s_0{:=}s_0.size$, in the body of *getSize*) to the *size* field. Unfortunately, s_0 is not constant at that program point, as it sometimes has the value y (when calling from program point ③) and sometimes $x.f$ (when calling from program points ② and ⑤). Instead of giving up, compositional analysis should let us analyze *getSize* separately and infer that s_0 is constant within each call to *getSize*. □

The first step for achieving compositionality is to split the program P into scopes $S_1, \ldots, S_n$ by partitioning the procedures (and therefore rules) in P into groups such that there are no mutual calls (directly or indirectly) between any two different groups. Therefore, the strongly connected components (or SCCs) of the program are the smallest scopes we should consider. For the sake of simplicity, we assume that each scope S has a single entry p. This is not a restriction, as we can repeat the analysis for each entry separately. Scopes are analyzed in a reverse topological order. Since there are no cycles among scopes, when analyzing a scope S, we have already analyzed all scopes reachable from S.

The only change required in the analysis presented in Sect. 3.1 is to modify the transfer function in order to handle calls to procedures defined in external scopes. Let $q(\langle\bar{w}\rangle, \langle\bar{s}\rangle)$ be a call to a procedure defined in $S' \neq S$ for which we have computed $q_{\uparrow}(\bar{s}) = \langle\phi', \theta'\rangle \in I(E^q_{S'})$. To avoid variable renamings, we assume that such answer $q_{\uparrow}$ is returned with the same variable names. Now, we define the transfer function for this call as $\tau(q(\langle\bar{w}\rangle, \langle\bar{s}\rangle), \langle\phi, \theta\rangle) = \langle\phi'', \theta''\rangle$ where:

(1) $\theta'' = \theta \cap \theta'$;

(2) we distinguish three kinds of variables to define ϕ'':
(2.1) $\forall z \in \mathrm{dom}(\phi) \setminus \bar{s}$, we have $\phi''(z) = \phi(z)$; otherwise
(2.2) $\forall z \in \mathrm{dom}(\phi)$ which is numeric, $\phi''(z) = \ell_{\mathsf{num}}$; otherwise
(2.3) $\forall s_k \in \bar{s}$ if $\phi'(s_k) = l_j.f_1 \dots f_n \wedge \{f_1, \dots, f_n\} \subseteq \theta \wedge \phi(w_j) \neq \ell_{\mathsf{any}}$ then $\phi''(s_k) = \phi(w_j).f_1 \dots f_n$, else $\phi''(s_k) = \ell_{\mathsf{any}}$.

Intuitively, field updates that might occur in the execution of q are learned in (1). Variables which are not output variables of q (2.1) are not affected by this step. In point (2.2), output numeric variables become ℓ_{num}. In (2.3), the answer for reference output variables of q is *renamed* to use them in this calling context. For this, we need to use the access paths computed for the input variables to perform the renaming on the output variables. We require that the involved field signatures are in θ and that the access path for w_j is not ℓ_{any}.

Example 7. We now split the program in Ex. 6 in three scopes: $S_1 = \{\mathit{getSize}\}$, $S_2 = \{\mathit{setSize}\}$ and $S_3 = \{\mathit{loop}, \mathit{loop}_c\}$. The analysis of S_1 results in $\mathit{getSize}_{\uparrow}(r) = \langle \{r \mapsto \ell_{\mathsf{num}}\}, \{f\} \rangle \in I(E_{S_1})$. The analysis of S_3 generates $E_{S_3}^{\mathit{loop}}$, which is as the one in Ex. 5 except for the equation that refers to the *size* field. In particular, equation $e_4^1(\bar{z})$ is replaced by: $e_4^1(\bar{z}) = \tau(\mathit{getSize}(\langle s_0 \rangle, \langle s_0 \rangle), e_3^1(\bar{z}))$. It results in the same value for e_4^1 as in Ex. 5, i.e., compositional analysis allows considering *size* constant in *getSize* without losing accuracy when composing the results. Thus, as for the program in Ex. 3, we conclude that in the calls to $\mathit{getSize}(\langle \mathit{this} \rangle, \langle r \rangle)$, *this* has the value $l_1.f$ at program points ②,⑤ and the value l_2 at ③. Reasoning similarly, we get that for the call to $\mathit{setSize}(\langle \mathit{this}, n \rangle, \langle \rangle)$, variable *this* always has the value $l_1.f$ at program point ⑥. □

Theorem 1 (soundness). *Let S be a scope and p be an entry. For any equation $e_i^k(\bar{x}) = \langle \phi, \theta \rangle \in I(E_S^p)$ and variable $z \in dom(\phi)$, if $\phi(z) = l_j.f_1 \dots f_n$, then z is constant w.r.t. $l_j.f_1 \dots f_n$ at program point (k, i).* □

Our method is parametric w.r.t. the choice of scopes. As a rule of thumb, the larger scopes are, the more context information we can propagate in the subsequent value analysis, but the less likely that numeric field accesses can be considered trackable. As motivated above, scopes should not be larger than methods, unless they are mutually recursive. For cost and termination, defining the scopes by first computing the SCCs and then grouping non-recursive SCCs that form a chain (consecutive SCCs in topological order) works well in practice.

4 An Instrumentation for Tracking Numeric Fields

We now identify sufficient conditions for instrumenting the program by adding *ghost* variables which correspond to the value of numeric fields. As for RCA, we define the instrumentation in a compositional way, guided by a set of scopes.

4.1 Finding Trackable Numeric Fields

Given an instruction b_j^k and a reference variable y, we use $\mathit{acc\_path}(y, b_j^k) = \ell$ as a shortcut for $e_{j-1}^k(\bar{x}) = \langle \phi, \theta \rangle \in I(E_S) \wedge \phi(y) = \ell$. We use S^* to refer to the union of S and all other scopes reachable from S. Given a scope S and a numeric field signature f, the set of *read access paths* for f in S, denoted $R(S, f)$, is the set of access paths of all variables y used for reading (i.e., instructions of the form $x{:=}y.f$) a field with the f signature in S^*. $R^+(S, f)$ denotes the set of access paths that originate from read accesses in S, and $R^*(S, f)$ those which originate from read accesses in $S^*\backslash\{S\}$. Thus, $R(S, f) = R^+(S, f) \cup R^*(S, f)$ where:

$$R^+(S,f) = \left\{ acc\_path(y, b_j^k) \,\middle|\, b_j^k \equiv x{:=}y.f \in S \right\}$$

$$R^*(S,f) = \left\{ \ell' \,\middle|\, \begin{array}{l} b_j^k \equiv q(\langle\bar{x}\rangle, \langle\bar{y}\rangle) \in S, \quad q \in S' \neq S, \quad \ell \in R(S', f) \\ \text{if } \ell = l_h.p \text{ then } \ell' = acc\_path(x_h, b_{j-1}^k).p \text{ else } \ell' = \ell_{\mathsf{any}} \end{array} \right\}$$

In $R^+(S, f)$, for each access $x{:=}y.f$, we add the access path that the analysis has computed for y. Computing the read access paths for a scope S requires computing the read access paths for all other scopes in S^*. Since scopes subsume SCCs, read access paths can be computed in reverse topological order without iterating. For each call $q(\langle\bar{x}\rangle, \langle\bar{y}\rangle)$ such that q is the entry of scope S', we take $R(S', f)$ and rename it according to the calling context. This requires renaming each l_h using the access path of x_h at b_{j-1}^k. The set of write access paths for f in S, denoted $W(S, f)$, is computed in a similar way by just considering the access path of all variables y in instructions of the form $y.f{:=}x$, instead of $x{:=}y.f$.

Example 8. Following Ex. 7, the set $R^+(S_1, size) = \{l_1\}$, due to the instruction $s_0 = s_0.size$, and $R^*(S_1, size) = \emptyset$, since S_1 does not have calls to other scopes. $R^+(S_2, size) = R^*(S_2, size) = \emptyset$ since no read accesses to the field *size* occur in *setSize*. Also, $R^+(S_3, size) = \emptyset$, since S_3 does not access *size* directly, but $R^*(S_3, size) = \{l_1.f, l_2\}$. Note that $l_1.f$ originates from program point ② and ⑤ and l_2 from program point ③. Finally, $W(S_1, size) = \emptyset$, $W(S_2, size) = \{l_1\}$ and $W(S_3, size) = \{l_1.f\}$, where $l_1.f$ originates from program point ⑥. □

Definition 5 (trackable numeric field signature). *Given a scope S from a program P and a numeric field signature $f \in NF(P)$, f is* trackable *in scope S if (1) f is trackable in all scopes in $S^*\backslash\{S\}$ and one of the following conditions holds: (2) $W(S, f) = \emptyset$; or (3) $W(S, f) = \{\ell\}$ and ℓ is of the form $l_j.f_1 \ldots f_n$.*

Condition (1) is required in order to have a sound transformation, as we cannot track accesses which are not trackable in transitively reachable scopes. Then, Condition (2) refers to scenarios where we do not have any write access to f, like example ⓑ. In such case, the value of numeric fields read through (possibly) different access paths will be stored in different ghost variables. Condition (3) requires that all write accesses are done through the same path, like examples ⓐ, ⓒ (inner loop) and ⓔ. This is the reason why the field accesses in the examples ⓓ and ⓕ are not trackable. An essential point is that, though it is allowed to have read accesses to f through access paths different from ℓ, they cannot be tracked. This is the case in the read access $y.size$ in example ⓔ.

4.2 Instrumenting Trackable Numeric Fields

The following transformation describes how to instrument a scope S with ghost variables for the different trackable uses of a numeric field f:

1. If f is not trackable go to 4
2. **Add Arguments:** each head or call $p(\langle\bar{x}\rangle, \langle\bar{y}\rangle)$ such that $p \in S$ is converted to $p(\langle\bar{x}{\cdot}\bar{v}_r\rangle, \langle\bar{y}{\cdot}\bar{v}_w\rangle)$ with $\bar{v}_w = \{v_{\ell.f} \mid \ell \in W(S,f)\}$ and
   (a) if $W(S,f) = \emptyset$ then $\bar{v}_r = \{v_{\ell.f} \mid \ell{\neq}\ell_{\mathsf{any}} \in R(S,f)\}$
   (b) if $W(S,f) = \{\ell\}$ then if $\ell \in R(S,f)$ then $\bar{v}_r = \{v_{\ell.f}\}$ else $\bar{v}_r = \emptyset$
3. **Replicate Field Accesses:**
   (a) each $b_j^k \equiv y.f{:=}x \in S$ produces a subsequent assignment $v_{\ell.f}{:=}x$, if $\mathit{acc\_path}(y, b_j^k) = \ell$
   (b) each $b_j^k \equiv x{:=}y.f \in S$ produces a subsequent assignment $x{:=}v_{\ell.f}$, if $\mathit{acc\_path}(y, b_j^k) = \ell \neq \ell_{\mathsf{any}} \wedge W(S,f) \subseteq \{\ell\}$
4. **Handle External Calls:** Let $b_j^k \equiv q(\bar{x}, \bar{y}) \in S$ be an external call, and $q(\langle\bar{x}'{\cdot}\bar{v}_r'\rangle, \langle\bar{y}'{\cdot}\bar{v}_w'\rangle)$ be the head of the definition of q after transforming its corresponding scope. The call is translated to $q(\langle\bar{x}{\cdot}\bar{v}_r\rangle, \langle\bar{y}{\cdot}\bar{v}_w\rangle)$ where, given a variable $v_\ell' \in \bar{v}_r' \cup \bar{v}_w'$ with $\ell = l_h.f_1 \ldots f_n.f$, its corresponding variable v_m is:
   (a) if $\mathit{acc\_path}(x_h, b_{j-1}^k) = \ell_{\mathsf{any}}$ or f is not trackable in S, then $v_m = *$;
   (b) otherwise, $m = \mathit{acc\_path}(x_h, b_{j-1}^k).f_1 \ldots f_n.f$.

The scopes in a program are instrumented in a reverse topological order. For simplicity, in the presentation, a scope S is instrumented iteratively, once for each field in $\mathit{NF}(P)$. However, in the implementation, each scope is instrumented just once, simultaneously for all field signatures. The key features in our instrumentation are: (i) Ghost variables have names of the form $v_{l.f}$, where l is an access path function and f a numeric field. (ii) If the field access is not trackable in the current scope, then it is not safe to propagate the value of numeric fields to/from external calls. To handle this, we use the mark '$*$' which, at the input, should be interpreted as a random integer and, at the output, it indicates that we should ignore the corresponding output value when we return from a call. This syntax can be easily supported by modifying rules 5 and 6 in the semantics, and treating it in value analysis is straightforward. (iii) When there are updates to a field signature, we can only track read accesses which refer to the same access path function used for the updates (see explanation of condition 3 in Def. 5).

Intuitively, each step in the instrumentation of a scope S w.r.t. a field signature f is: (1) If f is not trackable in S, we only need to instrument external calls (step 4a) by ignoring the value of ghost variables. E.g., when we instrument the calling scope to the loop in example b, we cannot track the value of the field $x.f$. (2) Input and output ghost variables are added as follows. For output ghost variables, the definition of trackable ensures that there is at most one access path in the write set. For the input ones, if there are no write accesses, we can track all their possible read uses (step 2a); otherwise, we can only track the accesses through the same access path (step 2b), hence we have at most one variable. The same arguments are also added to internal calls. (3) We replicate field accesses with accesses to its corresponding ghost variable. The condition $W(S,f) \subseteq \{\ell\}$ takes care of issue (iii) above. (4) For calls to other scopes, it is guaranteed that they have been already instrumented. We have to look up at the reference constancy information to find out which ghost variables we must use in the calling context, step 4b. In step 4a, if the field is not trackable or its access path is not constant, it is not safe to track its value.

Example 9. We first transform S_1 in Ex. 7 w.r.t. *size*. Recall that $R(S_1, size) = \{l_1\}$ and $W(S_1, size) = \emptyset$. Thus, we add an input variable v_1 for the ghost variable $v_{l_1.size}$, resulting in: “ $getSize(\langle this, \underline{v_1}\rangle, \langle r\rangle) \leftarrow s_0{:=}this, \underline{s_0{:=}v_1}, r{:=}s_0$”. Note that we have replaced the read access statement $s_0{=}s_0.size$ by $s_0{=}v_1$, which reads the ghost variable v_1. Similarly, the transformation of S_2 w.r.t. *size* generates the rule: “$setSize(\langle this, n\rangle, \langle v_1\rangle) \leftarrow s_0{:=}this, s_1{:=}n, s_0.size{:=}s_1, \underline{v_1{:=}s_1}$”, where now v_1 is an output value which stores the modification of $v_{l_1.size}$. Note that the write access $s_0.size{:=}s_1$ is replicated using variable v_1, which results in the additional statement $v_1{:=}s_1$. This corresponds to the intuition shown in the instrumentation of the Java code in Sec. 1, though it is more sophisticated as we have an inter-procedural transformation which allows multiple output variables. Hence, it could not be directly done in the original Java program. The instrumented version of rules (1), (2) and (3) of S_3 is:

$$\begin{array}{l} (1)\ loop(\langle x, y, i, \underline{v_1}\rangle, \langle r, \underline{v_1}\rangle) \leftarrow \\ \quad s_0{:=}x, s_0{:=}s_0.f, \\ \quad getSize(\langle s_0, \underline{v_1}\rangle, \langle s_0\rangle), \\ \quad loop_c(\langle x, y, i, s_0, \underline{v_1}\rangle, \langle r, \underline{v_1}\rangle). \end{array}$$

$$\begin{array}{l} (2)\ loop_c(\langle x, y, i, s_0, \underline{v_1}\rangle, \langle r, \underline{v_1}\rangle) \leftarrow \\ \quad \mathbf{s_0} \leq \mathbf{0}, s_0{:=}i, r{:=}s_0. \end{array}$$

$$\begin{array}{l} (3)\ loop_c(\langle x, y, i, s_0, \underline{v_1}\rangle, \langle r, \underline{v_1}\rangle) \leftarrow \\ \quad \mathbf{s_0} > \mathbf{0}, s_0{:=}i, s_1{:=}y, getSize(\langle s_1, *\rangle, \langle s_1\rangle), \\ \quad s_0{:=}s_0 + s_1, i{:=}s_0, s_0{:=}x, s_0{:=}s_0.f, \\ \quad s_1{:=}s_0, getSize(\langle s_1, \underline{v_1}\rangle, \langle s_1\rangle), \\ \quad s_2{:=}1, s_1{:=}s_1 - s_2, setSize(\langle s_0, s_1\rangle, \langle \underline{v_1}\rangle), \\ \quad loop(\langle this, x, y, i, \underline{v_1}\rangle, \langle r, \underline{v_1}\rangle). \end{array}$$

Since the write set is $\{l_1.f\}$, only one variable v_1 can be added for the read access $l_1.f$ (i.e., ghost variable $v_{l_1.f.size}$) and we cannot track the one corresponding to the read access l_2 (step 2b). An important point is that, in the calls to *getSize*, we use either v_1 or $*$ depending on the access path of the first argument, computed in step 4b. Field-insensitive value analysis of the instrumented program is now able to infer that v_1 (i.e., $x.f.size$) is decreasing and has 0 as lower limit. This is due to the fact that, for $getSize(\langle this, v_1\rangle, \langle r\rangle)$, field-insensitive value analysis can now infer that $r = v_1$ (which corresponds to *this.size*) and for $setSize(\langle this, n\rangle, \langle v_1\rangle)$ it infers that v_1 decreases by one. Cost and termination analyses hence succeed to bound the number of loop iterations by the ranking function v_1. □

The following theorem guarantees that we can safely use the instrumented program for value analysis instead of the original one.

Theorem 2. *Let P be a program, P_F be its instrumentation for $NF(P)$, and $C = \langle start, p(\langle \bar{x}\rangle, \langle \bar{y}\rangle), tv\rangle; h$ an initial configuration. If there is a trace t of the form $C \leadsto_P^n C_n$ then there exists a trace t' of the form $C' \leadsto_{P_F}^m C_m$ s.t. $C'{=}\langle start, p(\langle \bar{x} \cdot \bar{*}\rangle, \langle \bar{y} \cdot \bar{*}\rangle),\ tv\rangle; h$; $m \geq n$; s.t. if we remove all ghost variables and states that originate from the instrumentation from t', we obtain t.* □

Even though the instrumented program may have non-deterministic behaviour due to ghost variables whose values are unknown ('$*$'), this does not introduce a loss of precision w.r.t. field-insensitive value analysis, since such unknowns correspond to numeric fields which are also unknown in field-insensitive analysis.

5 Experiments in the COSTA System

COSTA [2] is a static analyzer able to prove termination and obtain upper bounds on resource usage for a relatively large class of Java bytecode programs. We have integrated our method in COSTA as a pre-process to the existing field-insensitive value analysis. In order to assess its practicality on realistic programs, we have tried to infer termination of all the loops which contain numeric field accesses in their guards for all classes in the subpackages of "`java`" of the Sun's implementation of J2SE 1.4.2. In total, we have found 133 methods which contain loops of this form, which we have taken as entries. COSTA has an application extraction algorithm (or class analysis) which pulls methods transitively used from each entry. COSTA failed to analyze 11 methods because when analyzing context-independently, it is required to analyze more methods than it can handle.

Bench.	**Ru**	$\mathbf{L}_n$	$\mathbf{R}_s$	$\mathbf{R}_i$	$\mathbf{T}_{rca}$	$\mathbf{T}_{tr}$	$\mathbf{T}_{gh}$	$\mathbf{T}_i$	$\mathbf{T}_s$	**SD**
lang	315	13	13	0	0.12	0.01	0.02	3.33	5.47	1.64
util	685	24	24	0	0.58	7.88	4.90	20.21	39.36	1.95
beans	90	3	3	0	0.05	0.00	0.00	1.42	1.65	1.16
math	662	15	12	1	0.22	0.18	0.17	7.84	9.84	1.26
text	1743	24	20	1	0.79	0.34	0.37	37.33	141.04	3.78
awt	4524	90	87	0	2.44	7.56	7.59	98.56	248.49	2.52
io	716	6	5	2	0.61	0.49	0.27	17.79	23.94	1.35
security	58	1	1	0	0.03	0.01	0.00	0.90	0.98	1.09
total	8793	176	165	4	4.84	16.47	13.32	187.38	470.77	2.51

The above table shows our experimental results for the 122 methods which COSTA can handle which belong to the packages whose name appears in the first column. For each package, we provide the size of the code to be analyzed, given as number of rules (**Ru**), the number of loops ($\mathbf{L}_n$) analyzed in each package which contain numeric field accesses in their guards. The column $\mathbf{R}_s$ shows the number of loops involving numeric guards for which COSTA has been able to find a ranking function using our proposed approach to field-sensitive analysis. Column $\mathbf{R}_i$ shows the same for field-insensitive analysis. It can be observed that, before applying our technique, COSTA could prove termination of only 4 of the 176 loops. In those 4 loops it is possible to prove termination using a field-insensitive analyzer because, for example, termination is guaranteed by reaching exceptional states. When we apply our approach to field-sensitive analysis, we prove termination of 165 of the 176 loops. It is also worth mentioning that only in 3 loops we fail to prove termination because the numeric field in the guard is not trackable (in particular, the reference is not constant). In the other 8 loops, though the fields are trackable, we failed due to limitations of the underlying termination techniques used in COSTA, and which are not related to our approach. In most cases, the problem is that the termination condition does not depend on the size of the data structure, but rather on the particular value stored at some location within the data structure, and also to the use of linear arithmetic operations.

The next set of columns evaluate time efficiency. The experiments have been performed on an Intel Core 2 Quad Q9300 at 2.5GHz with 1.95GB of RAM, running Linux 2.6.27-11. Analysis times are shown in seconds. The time of the RCA is shown in $\mathbf{T}_{rca}$. Columns $\mathbf{T}_{tr}$ and $\mathbf{T}_{gh}$ show, resp., the times to infer the trackability condition and to instrument the program with ghost variables. We have observed that the examples which require more time to infer trackability always involve a high number of numeric fields and thus the transformation also has to consider a high number of ghost variables. The total analysis time of the field-sensitive analysis, which includes the previous three columns is in $\mathbf{T}_s$. The field-insensitive analysis time is shown in $\mathbf{T}_i$. Finally, the **SD** column shows the slowdown introduced by field-sensitive analysis. The total overhead is 2.51. We argue that our results are quite positive since the overhead introduced is reasonable in return for the quite significant accuracy gains obtained.

6 Conclusions and Related Work

This paper proposes, to the best of our knowledge, the first static analysis to support *numeric fields* in cost and termination analysis of object-oriented bytecode. A complementary analysis for *reference fields* is [17]. Traditionally, existing approaches to reason on shared mutable data structures either track all possible updates of fields (endangering efficiency) or abstract all field updates into a single element (sacrificing accuracy). Our work does not fall into either category, as it does not track all field updates but rather only those which behave like non heap-allocated variables. Miné's [11] value analysis for C takes a different approach by enriching the abstract domain to make the analysis field-sensitive. His motivation is different from ours, such analysis is developed to improve points-to analysis in the presence of pointer arithmetics. We argue that our approach is sufficiently precise for context-independent analysis as required by important applications of value analysis such as termination analysis, while introducing a reasonable overhead. Also, [4] enriches a numeric abstract domain with alien expressions (field accesses). Without additional information, such as our RCA, this domain would be rather limited (imprecise) for bytecode. The notion of `restricted` variables used in [1] for C programs is related to our notion of reference constancy. However, [1] imposes more restrictive conditions, namely it avoids global pointers to be used locally and local copies to escape from the local context and, thus, it does not imply our reference constancy condition. In general, more accurate aliasing analysis (see [1] and its references) can be used to improve the precision of our analysis when computing the read and write sets, but at a higher performance cost and, besides, such further precision might not be required in practice for analyzing subprograms context-independently. Must-aliasing (aliases at program points) does not imply constancy of references, since the values of two variables might but still alias at the program point of interest. However, infering transitive relations of must-aliasing, i.e., between variables at different program points might enable the inference of constancy information, but this results in a much more expensive analysis than ours. Our work shares its motivation with the evolving field of *local reasoning* [14], such as separation logic [16] and regional logic [3] which provide expressive frameworks to reason about programs with shared mutable data structures. While our goals are more restricted, our technique has the advantage of allowing fully automatic inference.

Acknowledgments. This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* and IST-231620 *HATS* projects, by the Spanish Ministry of Education (MEC) under the TIN-2005-09207 *MERIT*, TIN-2008-05624 *DOVES* and HI2008-0153 (Acción Integrada) projects, and the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project.

References

1. Aiken, A., Foster, J.S., Kodumal, J., Terauchi, T.: Checking and inferring local non-aliasing. In: Proc.of PDLI 2003, pp. 129–140. ACM, New York (2003)
2. Albert, E., Arenas, P., Codish, M., Genaim, S., Puebla, G., Zanardini, D.: Termination Analysis of Java Bytecode. In: Barthe, G., de Boer, F.S. (eds.) FMOODS 2008. LNCS, vol. 5051, pp. 2–18. Springer, Heidelberg (2008)
3. Banerjee, A., Naumann, D., Rosenberg, S.: Regional logic for local reasoning about global invariants. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 387–411. Springer, Heidelberg (2008)
4. Chang, B.-Y.E., Leino, K.R.M.: Abstract interpretation with alien expressions and heap structures. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 147–163. Springer, Heidelberg (2005)
5. Codish, M., Taboch, C.: A semantic basis for the termination analysis of logic programs. J. Log. Program. 41(1), 103–123 (1999)
6. Cousot, P., Cousot, R.: Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: POPL 1977, pp. 238–252. ACM, New York (1977)
7. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proc. POPL. ACM, New York (1978)
8. Deutsch, A.: Interprocedural may-alias analysis for pointers: Beyond k-limiting. In: PLDI, pp. 230–241 (1994)
9. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Addison-Wesley, Reading (1996)
10. Logozzo, F.: Cibai: An abstract interpretation-based static analyzer for modular analysis and verification of java classes. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 283–298. Springer, Heidelberg (2007)
11. Miné, A.: Field-sensitive value analysis of embedded c programs with union types and pointer arithmetics. In: LCTES (2006)
12. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation 19(1), 31–100 (2006)
13. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis, 2nd edn. Springer, Heidelberg (2005)

14. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local Reasoning about Programs that Alter Data Structures. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, p. 1. Springer, Heidelberg (2001)
15. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004)
16. Reynolds, J.C.: Separation Logic: A Logic for Shared Mutable Data Structures. In: LICS, pp. 55–74 (2002)
17. Spoto, F., Hill, P., Payet, E.: Path-length analysis of object-oriented programs. In: EAAI 2006. ENTCS. Elsevier, Amsterdam (2006)
18. Vallee-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: Soot - a Java optimization framework. In: CASCON 1999, pp. 125–135 (1999)