From Object Fields to Local Variables: a Practical Approach to Field-Sensitive Analysis

E. Albert[1], P. Arenas[1], S. Genaim[1], G. Puebla[2] and D. Ramírez[2]

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

Abstract. Static analysis which takes into account the value of data stored in the heap is typically considered complex and computationally intractable in practice. Thus, most static analyzers do not keep track of *object fields* (or fields for short), i.e., they are *field-insensitive*. In this paper, we propose *locality conditions* for soundly converting fields into *local variables*. This way, field-insensitive analysis over the transformed program can infer information on the original fields. Our notion of locality is *context-sensitive* and can be applied both to numeric and reference fields. We propose then a *polyvariant* transformation which actually converts object fields meeting the locality condition into variables and which is able to generate multiple versions of code when this leads to increasing the amount of fields which satisfy the locality conditions. We have implemented our analysis within a termination analyzer for Java bytecode. Interestingly, we can now prove termination of programs which use *iterators* without the need of a sophisticated heap analysis.

1 Introduction

When data is stored in the heap, such as in object fields (numeric or references), keeping track of their value during static analysis becomes rather complex and computationally expensive. Analyses which keep track (resp. do not keep track) of object fields are referred to as *field-sensitive* (resp. *field-insensitive*). In most cases, neither of the two extremes of using a fully field-insensitive analysis or a fully field-sensitive analysis is acceptable. The former produces too imprecise results and the latter is often computationally intractable. There has been significant interest in developing techniques that result in a good balance between the accuracy of analysis and its associated computational cost. A number of heuristics exist which differ in how the value of fields is modeled. A well-known heuristics is *field-based* analysis, in which only one variable is used to model all instances of a field, regardless of the number of objects for the same class which may exist in the heap. This approach is efficient, but loses precision quickly.

Our work is inspired on an heuristics recently proposed in [3] for numeric fields. It is based on analyzing the behaviour of *program fragments* (or *scopes*) rather than the application as a whole, and modelling only those numeric fields whose behaviour is reproducible using local variables. In general, this is possible when two sufficient conditions hold within the scope: (a) the memory location

where the field is stored does not change, and (b) all accesses (if any) to such memory location are done through the same reference (and not through aliases). In [3], if both conditions hold, instructions involving the field access can be *replicated* by equivalent instructions using a local variable, which we refer to as *ghost* variable. This allows using a field-insensitive analysis in order to infer information on the fields by reasoning on their associated ghost variables.

Unfortunately, the techniques proposed in [3] for numeric fields are not effective to handle reference fields. Among other things, tracking reference fields by replicating instructions is problematic since it introduces undesired aliasing between fields and their ghost variables. Very briefly, to handle reference fields, the main open issues, which are contributions of this paper, are:

- *Locality condition:* the definition (and inference) of effective locality conditions for both numeric and reference fields. In contrast to [3], our locality conditions are context-sensitive and take must-aliasing context information into account. This allows us to consider as local certain field signatures which do not satisfy the locality condition otherwise.
- *Transformation:* an automatic transformation which *converts* object fields into ghost variables, based on the above locality. We propose a combination of context-sensitive locality with a *polyvariant* transformation which allows introducing multiple versions of the transformed scopes. This leads to a larger amount of field signatures which satisfy their locality condition.

Our approach is developed for object-oriented *bytecode*, i.e., code compiled for *virtual machines* such as the Java virtual machine [10] or .NET. It has been implemented in the COSTA system [4], a COSt and Termination Analyzer for Java Bytecode. Experimental evaluation has been performed on benchmarks which make extensive use of object fields and some of them use common patterns in object-oriented programming such as enumerators and iterators. They show that our analysis results in a very good balance between accuracy and efficiency.

2 Motivation: Field-Sensitive Termination Analysis

Automated techniques for proving termination are typically based on analyses which track *size* information, such as the value of numeric data or array indexes, or the size of data structures. Analysis should keep track of how the size of the data involved in loop guards changes when the loop goes through its iterations. This information is used for determining (the existence of) a *ranking function* for the loop, which is a function which strictly decreases on a well-founded domain at each iteration of the loop. This guarantees that the loop will be executed a finite number of times. For numeric data, termination analyzers rely on a *value* analysis which approximates the value of numeric variables (e.g. [7]). Some field-sensitive value analyses have been developed over the last years (see [11,3]). For heap-allocated data structures, *path-length* [15] is an abstract domain which provides a safe approximation of the length of the longest reference chain reachable from the variables of interest. This allows proving termination of loops which traverse acyclic data structures such as linked lists, trees, etc.

```
class Iter implements Iterator {
  List state; List aux;
  boolean hasNext() {
    return (this.state != null);}
  Object next() {
    List obj = this.state;
    this.state = obj.rest;
    return obj;}}
class Test {
  static void m(Iter x, Aux y, Aux z){
    while (x.hasNext()) x.next();
    y.f--;z.f--;}

  static void r(Iter x, Iter y, Aux z){
    Iter w=null;
    while (z.f > 0) {
      if ( z.f > 10 ) w=x else w=y;
      m(w,z,z);}}
  static void q(Iter x, Aux y, Aux z){
    m(x,y,z);}
  static void s(Iter x, Iter y, Aux w, Aux z){
    q(y,w,z); r(x,y,z); }}
  class Aux {int f;}
  class List {int data; List rest;}
```

Fig. 1. Iterator-like example.

Example 1. Our motivating example is shown in Fig. 1. This is the simplest example we found to motivate all aspects of our proposal. By now, we focus on method m. In object-oriented programming, the *iterator pattern* (also *enumerator*) is a design pattern in which the elements of a *collection* are traversed systematically using a *cursor*. The cursor points to the current element to be visited and there is a method, called next, which returns the current element and advances the cursor to the next element, if any. In order to simplify the example, the method next in Fig. 1 returns (the new value of) the cursor itself and not the element stored in the node. The important point, though, is that the state field is updated at each call to next. These kind of traversal patterns pose challenges in static analysis and effective solutions are required (see [17]). The challenges are mainly related to two issues: (1) Iterators are usually implemented using an auxiliary class which stores the cursor as a field (e.g., the "state" field). Hence, field-sensitive analysis is required; (2) The cursor is updated using a method call (e.g., within the "next" method). Hence, inter-procedural analysis is required.

We aim at inferring that the while loop in method m terminates. This can be proven by showing that the path-length of the structure pointed to by the cursor (i.e., x.state) decreases at each iteration. Proving this automatically is far from trivial, since many situations have to be considered by a static analysis. For example, if the value of x is modified in the loop body, the analysis must infer that the loop might not terminate since the memory location pointed to by x.state changes (see condition (a) in Sec. 1). The path-length abstract domain, and its corresponding abstract semantics, as defined in [15] is field-insensitive in the sense that the elements of such domain describe path-length relations among local variables only and not among reference fields. Thus, analysis results do not provide explicit information about the path-length of reference fields.

Example 2. In the loop in method m in Fig. 1, the path-length of x cannot be guaranteed to decrease when the loop goes through its iterations. This is because x might reach its maximal chain through the field aux and not state. However, the path-length of x.state decreases, which in turn can be used to prove termination of the loop. To infer such information, we need an analysis which is able to model the path-length of x.state and not that of x. Namely, we need a field-sensitive analysis based on path-length, which is one of our main goals in this paper.

The basic idea in our approach is to replace field accesses by accesses to the corresponding *ghost* variables whenever they meet the *locality condition* which will be formalized later. This will help us achieve the two challenges mentioned above: (1) make the path-length analysis field-sensitive, (2) have an inter-procedural analysis by using ghost variables as output variables in method calls.

3 A Simple Imperative Bytecode

We formalize our analysis for a simple *rule-based* imperative language [3] which is similar in nature to other representations of bytecode [16,9]. A *rule-based program* consists of a set of *procedures* and a set of 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*. Rules adhere to this grammar:

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

where $p(\langle \bar{x} \rangle, \langle \bar{y} \rangle)$ is the *head* of the rule; g its guard, which specifies conditions for the rule to be applicable; $b_1, \ldots, b_n$ the body of the rule; n an integer; x and y variables; f a field name, and $q(\langle \bar{x} \rangle, \langle \bar{y} \rangle)$ a procedure call by value. The language supports class definition and includes instructions for object creation, field manipulation, and type comparison through the instruction $\mathsf{type}(x, \mathrm{C})$, 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 which facilitate the formalization of the analysis are: (1) *recursion* is the only iterative mechanism, (2) *guards* are the only form of conditional, (3) there is no operand stack, (4) objects can be regarded as records, and the behavior induced by dynamic dispatch in the original bytecode program is compiled into *dispatch* rules guarded by a type check and (5) rules may have *multiple output* parameters which is useful for our transformation. The translation from (Java) bytecode to the rule-based form is performed in two steps [4]. First, a control flow graph is built. Second, a *procedure* is defined for each basic block in the graph and the operand stack is *flattened* by considering its elements as additional local variables. For simplicity, our language does not include advanced features of Java, but our implementation deals with full *sequential* Java bytecode.

Example 3. Fig. 2 shows the rule-based representation of our running example. Procedure m corresponds to method m, which first invokes procedure *while* as defined in rules ⑥–⑧. Observe that loops are extracted into separate procedures. Upon return from the *while* loop, the assignment $s_0{:=}y.f$ pushes the value of the numeric field $y.f$ on the stack. Then, this value is decremented by one and the result is assigned back to $y.f$. When a procedure is defined by more than one rule, each rule is *guarded* by a (mutually exclusive) condition. E.g., procedure r_2 is defined by rules ⑪ and ⑫. They correspond to checking the condition of the *while* loop in method r and are guarded by the conditions $\mathbf{s_0} > \mathbf{0}$ and $\mathbf{s_0} \leq \mathbf{0}$.

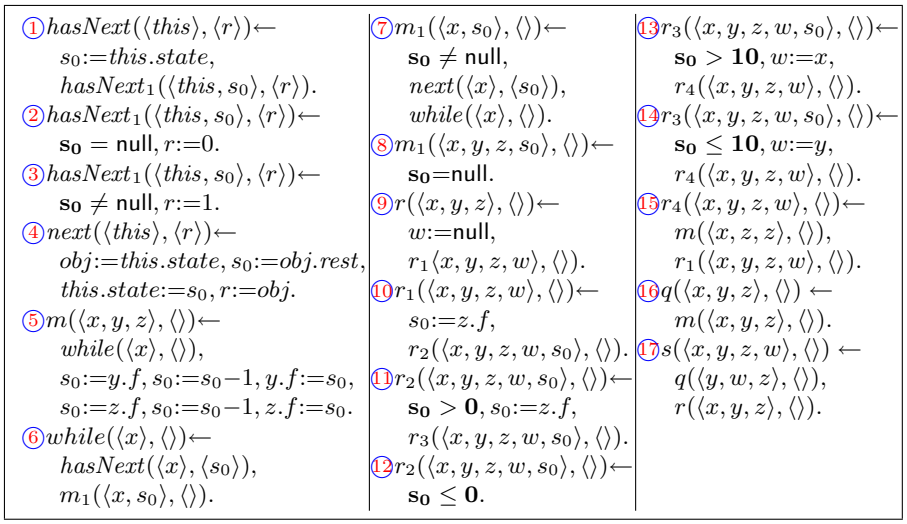

Fig. 2. Intermediate representation of running example in Fig. 1.

In the first case, the loop body is executed. In the second case execution exits the loop. Another important observation is that all rules have input and output parameters, which might be empty. The analysis is developed on the intermediate representation, hence all references to the example in the following are to this representation and not to the source code.

The execution of rule-based programs mimics standard bytecode [10]. A thorough explanation of the latter is outside the scope of this paper. An *operational semantics* for rule-based programs and our correctness proofs can be found in an online technical report `http://costa.ls.fi.upm.es/Papers/sas10.pdf`.

4 Preliminaries: Inference of Constant Access Paths

When transforming a field f into a local variable in a given code fragment, a necessary condition is that whenever f is accessed, using $x.f$, during the execution of that fragment, the variable x must refer to the same heap location (see condition (a) in Sec. 1). This property is known as *reference constancy* (or local aliasing) [3,1]. This section summarizes the *reference constancy analysis* of [3] that we use in order to approximate when a reference variable is constant at a certain program point. Since the analysis keeps information for each program point, we first make all program points unique as follows. The k-th rule $p(\langle\bar{x}\rangle,\langle\bar{y}\rangle)\leftarrow g, b_1^k,\ldots,b_n^k$ has $n{+}1$ program points. The first one, $(k,1)$, after the execution of the guard g and before the execution of b_1, until $(k,n{+}1)$ after the execution of b_n. This analysis receives a code fragment S (or scope), together with an entry procedure $p(\langle\bar{x}\rangle,\langle\bar{y}\rangle)$ from which we want to start the analysis. It assumes that each reference variable x_i points to an initial memory location represented by the symbol l_i, and each integer variable has the (symbolic) value ℓ_{num} representing any integer. The result of the analysis associates each variable at each program point with an *access path*. For a procedure with n input arguments, the entry is written as $p(l_1,\ldots,l_n)$.

Definition 1 (access path). *Given a program P with an entry $p(l_1, \ldots, l_n)$, an* access path ℓ *for a variable y at program point (k, j) is a syntactic construction which can take the forms:*

- ℓ_{any}. *Variable y might point to any heap location at (k, j).*
- ℓ_{num} *(resp. ℓ_{null}). Variable y holds a numeric value (resp.* null*) at (k, j).*
- $l_i.f_1 \ldots f_h$. *Variable y always refers to the same heap location represented by $l_i.f_1 \ldots f_h$ whenever (k, j) is reached.*

we use $acc\_path(y, b^k_j)$ to refer to the access path of y before instruction b^k_j.

Intuitively, the access path $l_i.f_1 \ldots f_h$ of y refers to the heap location which results from dereferencing the i-th input argument x_i using $f_1 \ldots f_h$ in the initial heap. In other words, variable y *must alias* with $x_i.f_1 \ldots f_n$ (w.r.t. to the initial heap) whenever the execution reaches (k, j).

Example 4. Consider an execution which starts from a call to procedure m in Fig. 2. During such execution, the reference x is constant. Thus, x always refers to the same memory location within method m, which, in this case, is equal to the initial value of the first argument of m. Importantly, the content of this location can change during execution. Indeed, x is constant and thus $x.state$ always refers to the same location in the heap. However, the value stored at $x.state$ (which is in turn a reference) is modified at each iteration of the while loop. In contrast, reference $x.state.rest$ is not constant in m, since $this.state$ refers to different locations during execution. Reference constancy analysis is the component that automatically infers this information. More concretely, applying it to rules ① − ③ w.r.t. the entry $hasNext(l_1)$, it infers that at program point $(1, 1)$ the variable *this* is constant and always refers to l_1. Applying it to ④, it infers that: at program points $(4, 1)$, $(4, 3)$ and $(4, 4)$ the variable *this* is constant and always refers to l_1; at $(4, 2)$ variable *obj* is constant and always refers to $l_1.state$; and at $(4, 3)$, variable s_0 is constant and always refers to $l_1.state.rest$.

Clearly, references are often not globally constant, but they can be constant when we look at smaller fragments. For example, when considering an execution that starts from m, i.e., $m(l_1, l_2, l_3)$, then variable *this* in *next* and *hasNext* always refers to the same heap location l_1. However, if we consider an execution that starts from s, i.e., $s(l_1, l_2, l_3, l_4)$, then variable *this* in *next* and *hasNext* might refer to different locations l_1 or l_2, depending on how we reach the corresponding program points (through r or q). As a consequence, from a global point of view, the accesses to *state* are not constant in such execution, though they are constant if we look at each sub-execution alone. Fortunately, the analysis can be applied compositionally [3] by partitioning the procedures (and therefore rules) of P into groups which we refer to as *scopes*, provided that there are no mutual calls between scopes. Therefore, the strongly connected components (SCCs) of the program are the smallest scopes we can consider. In [3], the choice of scopes directly affects the precision and an optimal strategy does not exist: sometimes enlarging one scope might improve the precision at one program point and make it worse at another program point. Instead, in this paper, due to the context-sensitive nature of the analysis, information is propagated among the scopes and

the maximal precision is guaranteed when scopes are as small as possible, i.e., at the level of SCCs. In the examples, sometimes we enlarge them to simplify the presentation. Moreover, we assume that each SCC has a single entry, this is not a restriction since otherwise the analysis can be repeated for each entry.

Example 5. The rules in Fig. 2 can be divided into the following scopes: $S_{hasNext} = \{hasNext, hasNext_1\}$, $S_{next} = \{next\}$, $S_m = \{m, while, m_1\}$, $S_r = \{r, r_1, r_2, r_3, r_4\}$, $S_q = \{q\}$ and $S_s = \{s\}$. A possible reverse topological order for the scopes of Ex. 5 is $S_{hasNext}$, S_{next}, S_m, S_r, S_q and S_s. Therefore, compositional analysis starts from $S_{hasNext}$ and S_{next} as explained in Ex. 4. Then, S_m is analyzed w.r.t. the initial call $m(l_1, l_2, l_3)$, next, S_r w.r.t. $r(l_1, l_2, l_3)$ and so on. When the call from r to m is reached, the analysis uses the reference constancy inferred for m and adapts it to the current context. This way, the reference to *state* is proven to be constant, as we have justified above. As expected, the analysis cannot guarantee that the access to *rest* is constant.

In order to decide whether a field f can be considered local in a scope S, we have to inspect all possible accesses to f in any possible execution that starts from the entry of S. Note that these accesses can appear directly in S or in a scope that is reachable from it (transitive scope). We use S^* to refer to the union of S and all other scopes reachable from S, and $S(p)$ (resp. $S(b_j^k)$) to refer to the scope in which the procedure p (resp. instruction b_j^k) is defined (resp. appears). We distinguish between access for reading the field value from those that modify its value. Given a scope S and a field signature f, the set of *read* (resp. *write*) *access paths* for f in S, denoted $R(S, f)$ (resp. $W(S, f)$), is the set of access paths of all variables y used for reading (resp. modifying) a field with the signature f, i.e., $x{:=}y.f$ (resp. $y.f{:=}x$), in S^*. Note that if S has calls to other scopes, for each call $b_j^k \equiv q(\langle \bar{w} \rangle, \langle \bar{z} \rangle) \in S$ such that $S(q) \neq S$, we should adapt the read (resp. write) access paths $R(S(q), f)$ (resp. $W(S(q), f)$) to the calling context by taking into account *aliasing* information. Let us see an example.

Example 6. The read and write sets for f, *state* and *rest* w.r.t. the scopes of Ex. 5 are:

	$R(S_i, f)$	$R(S_i, state)$	$R(S_i, rest)$	$W(S_i, f)$	$W(S_i, state)$	$W(S_i, rest)$
$S_{hasNext}$	{}	$\{l_1\}$	{}	{}	{}	{}
S_{next}	{}	$\{l_1\}$	$\{l_1.state\}$	{}	$\{l_1\}$	{}
S_m	$\{l_2, l_3\}$	$\{l_1\}$	$\{\ell_{\mathsf{any}}\}$	$\{l_2, l_3\}$	$\{l_1\}$	{}
S_r	$\{l_3\}$	$\{\ell_{\mathsf{any}}\}$	{}	$\{l_3\}$	$\{\ell_{\mathsf{any}}\}$	{}
S_q	$\{l_2, l_3\}$	$\{l_1\}$	$\{\ell_{\mathsf{any}}\}$	$\{l_2, l_3\}$	$\{l_1\}$	{}
S_s	$\{l_3, l_4\}$	$\{\ell_{\mathsf{any}}\}$	$\{\ell_{\mathsf{any}}\}$	$\{l_3, l_4\}$	$\{\ell_{\mathsf{any}}\}$	{}

An essential observation is that, in S_m, we have two different read access paths l_2 and l_3 to f. However, when we adapt this information to the calling context from r, they become the same due to the aliasing of the last two arguments in the call to m from r. Instead, when we adapt this information to the calling context from q, they remain as two different access paths. Finally, when computing the sets for S_s, since we have a call to q followed by one to r, we have to merge their information and assume that there are two different access paths for f that correspond to those through the third and fourth argument of s.

5 Locality Conditions for Numeric and Reference Fields

Intuitively, in order to ensure a sound monovariant transformation, a field signature can be considered *local in a scope* S if all read and write accesses to it in *all* reachable scopes (i.e., S^*) are performed through the same access path.

Example 7. According to the above intuition, the field f is not local in m since it is not guaranteed that l_2 and l_3 (i.e., the access paths for the second and third arguments) are aliased. Therefore, f is not considered as local in S_r (since $S_m \in S_r^*$) and the termination of the while loop in r cannot be proven. However, when we invoke m within the loop body of r, we have knowledge that they are actually aliased and f could be considered local in this context.

As in [3], when applying the reference constancy analysis (and computing the read and write sets), we have assumed no aliasing information about the arguments in the entry to each SCC, i.e., we *do not know* if two (or more) input variables point to the same location. Obviously, this assumption has direct consequences on proving locality, as it happens in the example above. The following definition introduces the notion of *call pattern*, which provides *must aliasing* information and which will be used to specify entry procedures.

Definition 2 (call pattern). *A call pattern ρ for a procedure p with n arguments is a partition of $\{1, \ldots, n\}$. We denote by ρ_i the set $X \in \rho$ s.t. $i \in X$.*

Intuitively, a call pattern ρ states that each set of arguments $X \in \rho$ are guaranteed to be aliased. In what follows, we denote the most general call pattern as $\rho_\top = \{\{1\}, \ldots, \{n\}\}$, since it does not have any aliasing information.

Example 8. The call pattern for m when called from r is ρ=$\{\{1\}, \{2, 3\}\}$, which reflects that the 2^{nd} and 3^{rd} arguments are aliased. The call pattern for m when called from q is $\rho_\top$ in which no two arguments are guaranteed to be aliased.

The reference constancy analysis [3] described in Sec. 4 is applied w.r.t. $\rho_\top$. In order to obtain access path information (and read and write sets) w.r.t. a given initial call pattern ρ, a straightforward approach is to re-analyze the given scope taking into account the aliasing information in ρ. Since the analysis is compositional, another approach is to reuse the read and write access paths inferred w.r.t. $\rho_\top$ and adapt them (i.e., *rename* them) to each particular call pattern ρ. This is clearly more efficient since we can analyze the scope once and reuse the results when new contexts have to be considered. In theory, re-analyzing can be more precise, but in practice reusing the results is precise enough for our needs. The next definition provides a *renaming* operation. By convention, when two arguments are aliased, we rename them to have the same name of the one with smaller index. This is captured by the use of *min*.

Definition 3 (renaming). *Given a call pattern ρ and an access path $\ell \equiv l_i.f_1 \ldots f_n$, $\rho(\ell)$ is the* renamed access path *$l_k.f_1 \ldots f_n$ where $k = min(\rho_i)$. For a set of access paths A, $\rho(A)$ is the set obtained by renaming all elements of A.*

Example 9. Renaming the set of access paths $\{l_2, l_3\}$ obtained in Ex. 7 w.r.t. ρ of Ex. 8 results in $\{l_2\}$. This is because $\rho(l_2){=}l_2$ and $\rho(l_3){=}l_2$, by the convention of *min* above. It corresponds to the intuition that when calling m from r, all accesses to field f are through the same memory location, as explained in Ex. 7.

Renaming is used in the context-sensitive *locality* condition to obtain the read and write sets of a given scope w.r.t. a call pattern, using the context-insensitive sets. It corresponds to the context-sensitive version of condition (*b*) in Sec. 1.

Definition 4 (general locality). *A field signature f is local in a scope S w.r.t. a call pattern ρ, if $\rho(R(S,f)) \cup \rho(W(S,f)) = \{\ell\}$ and $\ell \neq \ell_{any}$.*

Example 10. If we consider S_m w.r.t. the call pattern $\{\{1\},\{2,3\}\}$ then f becomes local in S_m, since l_2 and l_3 refer to the same memory location and, therefore, it is local for S_r. Considering f local in S_r is essential for proving the termination of the while loop in r. This is because by tracking the value of f we infer that the loop counter decreases at each iteration. However, making f local in all contexts is not sound, as when x and y are not aliased, then each field decreases by one, and when there are aliases, it decreases by two. Hence, in order to take full advantage of context-sensitivity, we need a *polyvariant* transformation which generates two versions for procedure m (and its successors).

An important observation is that, for reference fields, it is not always a good idea to transform all fields which satisfy the general locality condition above.

Example 11. By applying Def. 4, both reference fields *state* and *rest* are local in S_{next}. Thus, it is possible to convert them into respective ghost variables v_s (for *state*) and v_r (for *rest*). Intuitively, rule ④ in Ex. 3 would be transformed into:

$$next(\langle this, v_s, v_r\rangle, \langle v_s, v_r\rangle) \leftarrow obj{:=}v_s, s_0{:=}v_r, v_s{:=}s_0, r{:=}obj.$$

For which we cannot infer that the path-length of v_s in the output is smaller than that of v_s in the input. In particular, the path-length abstraction approximates the effect of the instructions by the constraints $\{obj{=}v_s, s_0{=}v_r, v'_s{=}s_0, r{=}obj\}$. Primed variables are due to a single static assignment. The problem is that the transformation replaces the assignment $s_0{:=}obj.rest$ with $s_0{:=}v_r$. Such assignment is crucial for proving that the path-length of v_s decreases at each call to *next*. If, instead, we transform this rule w.r.t. the field *state* only:

$$next(\langle this, v_s\rangle, \langle r, v_s\rangle) \leftarrow obj{:=}v_s, s_0{:=}obj.rest, v_s{:=}s_0, r{:=}obj.$$

and the path-length abstraction approximates the effect of the instructions by $\{obj{=}v_s, s_0{<}obj, v'_s{=}s_0, r{=}obj\}$ which implies $v'_s{<}v_s$. Therefore, termination (of the corresponding loop) can be proven relying only on the field-insensitive version of path-length. Note that, in the second constraint, when accessing a field of an acyclic data structure, the corresponding path-length decreases.

Now we introduce a locality condition which is more restrictive than that in Def. 4, called *reference locality*. This condition is interesting because it only holds for field accesses which perform heap updates, but it does not hold for

other cases. Thus, it often solves the problem of too aggressive transformation, shown above. This is achieved by requiring that the field signature is both read and written in the scope. Intuitively, this heuristics is effective for tracking the references that are used as cursors for traversing the data structures and not reference fields which are part of the data structure itself.

Definition 5 (reference locality). *A field signature f is local in a scope S w.r.t. a call pattern ρ, if $\rho(R(S,f)) = \rho(W(S,f)) = \{\ell\}$ and $\ell \neq \ell_{\mathsf{any}}$.*

While reference locality is more effective than general locality for reference fields, in the case of numeric fields, general locality is more appropriate than reference locality. For example, numeric fields are often used to bound the loop iterations. For these cases, reference locality is not sufficient, since the field is read but not updated. Since numeric and reference fields can be distinguished by their signature, we apply general locality to numeric fields and reference locality to reference fields without problems. In what follows, we use *locality* to refer to either general or reference locality, according to the corresponding field signature.

Example 12. Field *rest* is not local in S_{next}, according to Def. 5. Field *state* is local in S_{next} and S_m, but not in $S_{hasNext}$.

6 Polyvariant Transformation of Fields to Local Variables

Our transformation of object fields to local variables is performed in two steps. First, we infer *polyvariance declarations* which indicate the (multiple) versions we need to generate of each scope to achieve a larger amount of field signatures which satisfy their locality condition. Then, we carry out a (polyvariant) transformation based on the polyvariance declarations.

We first define an auxiliary operation which, given a scope S and a call pattern ρ, infers the *induced* call patterns to the external procedures.

Definition 6 (induced call pattern). *Given a call pattern ρ for a scope S, and a call $b_k^j = q(\langle \bar{x} \rangle, \langle \bar{y} \rangle) \in S$ such that $S(q) \neq S$, the call pattern for $S(q)$ induced by b_k^j, denoted $\rho(b_k^j)$, is obtained as follows:*
(1) generate the tuple $\langle \ell_1, \ldots, \ell_n \rangle$ where $\ell_i = \rho(acc\_path(b_k^j, x_i))$; and
(2) i and h belong to the same set in $\rho(b_k^j)$ if and only if $\ell_i = \ell_h \neq \ell_{\mathsf{any}}$.

Example 13. Consider the scope S_r and a call pattern $\rho = \{\{1\},\{2\},\{3\}\}$. The call pattern induced by $b_1^{15} \equiv m(\langle x, z, z \rangle, \langle\rangle)$ is computed as follows: (1) using the access path information, we compute the access paths for the arguments $\langle x, z, z \rangle$ which in this case are $\langle l_1, l_3, l_3 \rangle$; (2) $\rho(b_1^{15})$ is defined such that arguments i and j are in the same set if the access paths of the i-th and the j-th arguments are equal. Namely, we obtain $\rho(b_1^{15}) = \{\{1\},\{2,3\}\}$ as induced call pattern.

Now, we are interested in finding out the maximal polyvariance level which must be generated for each scope. Intuitively, starting from the entry procedure, we will traverse all reachable scopes in a top-down manner by applying the *polyvariance operator* defined below. This operator distinguishes two sets of fields:

– F_{pred} is the set of field signatures which are local for the predecessor scope;
– F_{curr} is the set of tuples which contain a field signature and its access path, which are local in the current scope and not in the predecessor one.

This distinction is required since before calling a scope, the fields in F_{curr} should be initialized to the appropriate values, and upon exit the corresponding heap locations should be modified. Intuitively, the operator works on a tuple $\langle S, F_{pred}, \rho\rangle$ formed by a scope identifier S, a set of predecessor fields F_{pred}, and a call pattern ρ. At each iteration, a *polyvariance declaration* of the form $\langle S, F_{pred}, F_{curr}, \rho\rangle$ is generated for the current scope, where the local fields in F_{curr} for context ρ are added. The operator transitively applies to all reachable scopes from S.

Definition 7 (polyvariance). *Given a program P with an entry procedure p and call pattern ρ, the set of all versions in P is $\mathcal{V}_P = \mathcal{P}ol(\langle S(p), \emptyset, \rho\rangle)$ s.t.*

$$\mathcal{P}ol(\langle S, F_{pred}, \rho\rangle) = \{\langle S, F_{pred}, F_{curr}, \rho\rangle\} \bigcup \{\mathcal{P}ol(\langle S(q), F, \rho(b^j_k)\rangle) \mid b^j_j \equiv q(\langle\bar{x}\rangle, \langle\bar{y}\rangle) \text{ is external call in } S\}$$

where F_{curr} and F are defined as follows:

– $F_{curr} = \{\langle f, \ell\rangle \mid f$ *is local in* S *w.r.t.* ρ *with an access path* ℓ *and* $f \notin F_{pred}\}$,
– $F = (F_{pred} \cup \{f \mid \langle f, \ell\rangle \in F_{curr}\}) \cap \mathsf{fields}(S^*(q))$ *where* $\mathsf{fields}(S^*(q))$ *is the set of fields that are actually accessed in* $S^*(q)$.

Since there are no mutual calls between any scopes, termination is guaranteed.

Example 14. The polyvariance declarations obtained by iteratively applying $\mathcal{P}ol$ starting from $\mathcal{P}ol(\langle S_s, \emptyset, \{\{1\},\{2\},\{3\},\{4\}\}\rangle)$ are:

Id	S	F_{pred}	F_{curr}	ρ
1	S_s	$\emptyset$	$\emptyset$	{{1},{2},{3},{4}}
2	S_q	$\emptyset$	$\{l_1.state\}$	{{1},{2},{3}}
3	S_r	$\emptyset$	$\{l_3.f\}$	{{1},{2},{3}}
4	S_m	$\{f\}$	$\{l_1.state\}$	{{1},{2,3}}

Id	S	F_{pred}	F_{curr}	ρ
5	S_m	$\{state\}$	$\emptyset$	{{1},{2},{3}}
6	S_{next}	$\{state\}$	$\emptyset$	{{1}}
7	$S_{hasNext}$	$\{state\}$	$\emptyset$	{{1}}

Each line defines a polyvariance declaration $\langle S, F_{pred}, F_{curr}, \rho\rangle$ as in Def. 7. The first column associates to each version a unique Id that will be used when transforming the program. The only scope with more than one version is S_m, which has two, with identifiers 4 and 5. The call patterns in such versions are different and in this case they result in different fields being local.

In general, the set of polyvariance declarations obtained can include some versions which do not result in further fields being local. Before materializing the polyvariant program, it is possible to introduce a minimization phase (see, e.g., [13]) which is able to reduce the number of versions without losing opportunities for considering fields local. This can be done using well-known algorithms [8] for minimization of deterministic finite automata.

Given a program P and the set of all polyvariance declarations $\mathcal{V}_P$, we can proceed to transform the program. We assume that each version has a unique identifier (a positive integer as in the example above) which will be used in order to avoid name clashing when cloning the code. The instrumentation is done by

1. Given $F{=}\{f|\langle f,\ell\rangle{\in}F_{curr}\}$, we let $\bar{v}{=}\{v_f|f{\in}F_{pred}{\cup}F\}$ be a tuple of ghost variable names.
2. **Add arguments to internal calls:** each head of a rule or a call $p(\langle\bar{x}\rangle,\langle\bar{y}\rangle)$ where p is defined in S is replaced by $p{\cdot}\texttt{i}(\langle\bar{x}\cdot\bar{v}\rangle,\langle\bar{y}\cdot\bar{v}\rangle)$.
3. **Transform field accesses:** each access $x.f$ is replaced by v_f.
4. **Handle external calls:** let $b_j^k \equiv q(\langle\bar{x}\rangle,\langle\bar{y}\rangle) \in S$ such that $S(q) \neq S$.
   - Lookup the (unique) version $\langle S(q), F_{pred}\cup F, F', \rho(b_j^k)\}\rangle$ of $S(q)$ that matches the calling context and has a unique identifier `id`.
   - Let $\bar{v}' = \{v_f|f \in F_{pred}\cup F\}\cup\{v_f|\langle f,\ell\rangle \in F'\}$.

   Then, we transform b_j^k as follows:
   (a) **Initialization**: $\forall\langle f, l_h.f_1\ldots f_n\rangle \in F'$ we add an initialization statement (before the call) $v_f{:=}x_h.f_1\ldots f_n.f$;
   (b) **Call**: we add the modified call $q{\cdot}\texttt{id}(\langle\bar{x}\cdot\bar{v}'\rangle,\langle\bar{y}\cdot\bar{v}'\rangle)$
   (c) **Recovery**: $\forall\langle f, l_h.f_1\ldots f_n\rangle \in F'$ we add a recovering statement (after the call) $x_h.f_1\ldots f_n.f{:=}v_f$;

Fig. 3. Transformation of a Polyvariance Declaration with Identifier `i`

cloning the original code of each specification in $\mathcal{V}_P$. The clone for a polyvariance declaration $\langle S, F_{pred}, F_{curr}, \rho\rangle \in \mathcal{V}_P$ with identifier `i` is done using the algorithm in Fig. 3. The four steps of the instrumentation work as follows:

(1) This step generates new unique variable names $\bar{v}$ for the local heap locations to be tracked in the scope S. Since the variable name v_f is associated to the field signature f (note that in bytecode field signatures include class and package information), we can retrieve it at any point we need it later.
(2) This step adds the identifier `i`, as well as the tuple of ghost variables (generated in the previous step) as input and output variables to all rules which belong to the scope S in order to carry their values around during execution.
(3) This step replaces the actual heap accesses (i.e., read and write field accesses) by accesses to their corresponding ghost variables. Namely, an access $x.f$ is replaced by the ghost variable v_f which corresponds to f.
(4) Finally, we transform external calls. The main point is to consider the correct version `id` for that calling context by looking at the polyvariance declarations. Then, the call to q is replaced as follows:
   4a We first need to initialize the ghost variables which are local in $S(q)$ but not in S, namely the variables in F'.
   4b We add a call to q which includes the ghost variables $\bar{v}'$.
   4c After returning from the call, we *recover* the value of the memory locations that correspond to ghost variables which are local in $S(q)$ but not in S, i.e., we put their value back in the heap.

Note that in points 4a and 4c, it is required to relate the field access which is known to be local within such call (say f) and the actual reference to it in the current context. This is done by using the access paths as follows. If a field f is local in S and it is always referenced through $l_i.f_1\ldots f_n$, then when calling $q(\langle\bar{w}\rangle,\langle\bar{z}\rangle)$, the initial value of the corresponding ghost variable should be initialized to $w_i.f_1\ldots f_n.f$. This is because l_i refers to the location to which the i-th argument points when calling q.

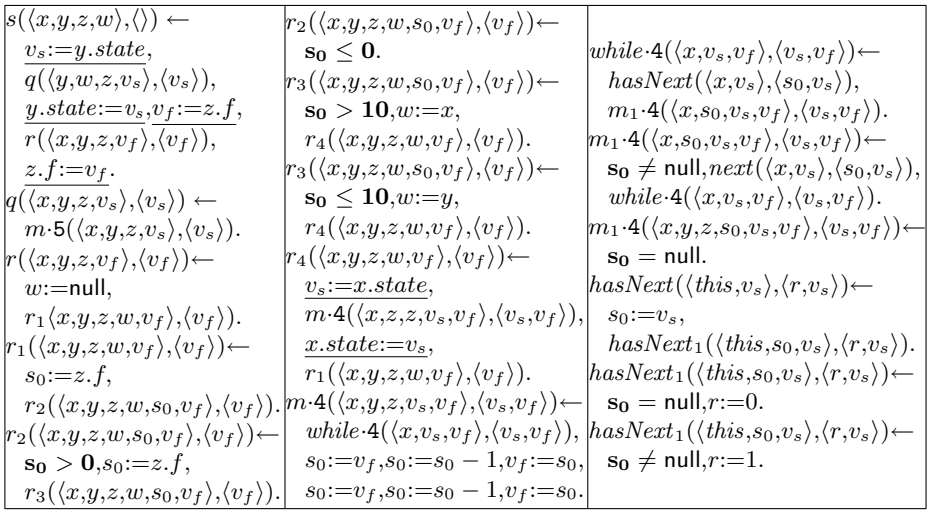

Fig. 4. Polyvariant Transformation of Running Example (excerpt)

Example 15. Fig. 4 shows the transformed program for the declarations of Ex. 14. For simplicity, when a scope has only one version, we do not introduce new names for the corresponding procedures. Procedure *next* is not shown, it is as in Ex. 11. Procedure *hasNext* now incorporates a ghost variable v_s that tracks the value of the corresponding *state* field. Note that r calls $m{\cdot}\mathsf{4}$ while q calls $m{\cdot}\mathsf{5}$. For version 4 of m, both f and *state* are considered local and therefore we have the ghost variables v_s and v_f. Version 5 of m is not shown for lack of space, it is equivalent to version 4 but without any reference to v_f since it is not local in that context. Now, all methods can be proven terminating by using a field-insensitive analysis.

In practice, generating multiple versions for a given scope S might be expensive to analyze. However, two points should be noted: (1) when no accuracy is gained by the use of polyvariance, i.e., when the locality information is identical for all calling contexts, then the transformation behaves as monovariant; (2) when further accuracy is achieved by the use of polyvariance, a *context-sensitive*, but monovariant transformation can be preferred for efficiency reasons by simply declaring as local only those fields which are local in versions.

7 Experiments

We have integrated our method in COSTA [4], a cost and termination analyzer for Java bytecode, as a pre-process to the existing field-insensitive analysis. It can be tried out at: `http://costa.ls.fi.upm.es`. The different approaches can be selected by setting the option enable_field_sensitive to: "trackable" for using the setting of [3]; "mono_local" for context-insensitive and monovariant transformation; and "poly_local" for context-sensitive and polyvariant transformation. In Table 1 we evaluate the precision and performance of the proposed techniques by analyzing three sets of programs. The first set contains loops from the JOlden suite [6] whose termination can be proven only by tracking reference fields. They are challenging because they contain reference-intensive kernels and

Bench.	#R	#R$_p$	#L	#L$_p$	L$_{ins}$	L$_{tr}$	L$_{mono}$	L$_{poly}$	T$_{ins}$	O$_{tr}$	O$_{mono}$	O$_{poly}$
bh	1759	1759	21	21	16	16	16	21	230466	1.54	1.25	1.50
em3d	1015	1015	13	13	1	3	3	13	17129	1.43	1.24	1.55
health	1364	1364	11	11	6	6	6	11	21449	2.23	1.65	2.00
java.util	593	593	26	26	3	24	24	24	17617	1.55	1.62	1.72
java.lang	231	231	14	14	5	14	13	13	2592	1.69	1.38	1.52
java.beans	113	113	3	3	0	3	3	3	3320	1.07	1.09	1.15
java.math	278	278	12	12	3	11	11	11	15761	1.07	1.05	1.12
java.awt	1974	1974	102	102	25	100	100	100	64576	1.25	1.21	1.55
java.io	187	187	4	4	2	4	4	4	2576	2.72	2.16	3.47
run-ex	40	61	2	3	0	0	1	3	300	1.28	1.25	2.24
num-poly	71	151	4	8	0	0	1	8	576	1.27	1.26	3.33
nest-poly	86	125	8	10	1	4	7	10	580	1.61	1.61	3.02
loop-poly	16	29	1	2	0	0	0	2	112	1.25	1.25	2.61

Table 1. Accuracy and Efficiency of four Analysis Settings in COSTA

use enumerators. The next set consists of the loops which access numeric field in their guards for all classes in the subpackages of "`java`" of SUN's J2SE 1.4.2. Almost all these loops had been proven terminating using the trackable profile [3]. Hence, our challenge is to keep the (nearly optimal) accuracy and comparable efficiency as [3]. The last set consists of programs which require a context-sensitive and polyvariant transformation (source code is available in the website above).

For each benchmark, we provide the size of the code to be analyzed, given as number of rules #**R**. Column #**R**$_p$ contains the number of rules after the polyvariant transformation which, as can be seen, increases only for the last set of benchmarks. Column #**L** is the number of loops to be analyzed and #**L**$_p$ the same after the polyvariant transformation. Column **L**$_{ins}$ shows the number of loops for which COSTA has been able to prove termination using a field-insensitive analysis. Note that, even if all entries correspond to loops which involve fields in their guards, they can contain inner loops which might not and, hence, can be proven terminating using a field-insensitive analysis. This is the case of many loops for benchmark bh. Columns **L**$_{tr}$, **L**$_{mono}$ and **L**$_{poly}$ show the number of loops for which COSTA has been able to find a ranking function using, respectively, the trackable, mono_local and poly_local profiles as described above. Note that when using the poly_local option, the number of loops to compare to is #**L**$_p$ since the polyvariant transformation might increase the number of loops in #**L**.

As regards accuracy, it can be observed that for the benchmarks in the JOlden suite, trackable and mono_local behave similarly to a field-insensitive analysis. This is because most of the examples use iterators. Using poly_local, we prove termination of all of them. In this case, it can be observed from column #**R**$_p$ that context-sensitivity is required, however, the polyvariant transformation does not generate more than one version for any example. As regards the J2SE set, the profile trackable is already very accurate since these loops contain only numeric fields. Except for one loop in java.lang which involves several numeric fields, our profiles are as accurate as trackable. All examples in the last set include many reference fields and, as expected, the trackable does not perform well. Although

mono_local improves the accuracy in many loops polyvariance is required to prove termination. The profile poly_local proves termination of all loops in this set.

The next set of columns evaluate performance. The experiments have been performed on an Intel Core 2 Duo 1.86GHz with 2GB of RAM. Column $\mathbf{T}_{ins}$ is the total time (in seconds) for field-insensitive analysis, and the other columns show the slowdown introduced by the corresponding field-sensitive analysis w.r.t. $\mathbf{T}_{ins}$. The overhead introduced by trackable and mono_local is comparable and, in most cases, is less than two. The overhead of poly_local is larger for the examples which require multiple versions and it increases with the size of the transformed program in $\#\mathbf{R}_p$. We argue that our results are promising since the overhead introduced is reasonable in return for the significant accuracy gains obtained.

8 Conclusions and Related Work

Field sensitiveness is considered currently one of the main challenges in static analyses of object-oriented languages. We have presented a novel practical approach to field-sensitive analysis which handles all object fields (numeric and references) in a uniform way. The basic idea is to partition the program into fragments and convert object fields into local variables at each fragment, whenever such conversion is sound. The transformation can be guided by a context-sensitive analysis able to determine that an object field can be safely replaced by a local variable only for a specific context. This, when combined with a polyvariant transformation, achieves a very good balance between accuracy and efficiency.

Our work continues and improves over the stream of work on termination analysis of object-oriented bytecode programs [3,12,2,15,14]. The heuristic of treating fields as local variables in order to perform field-sensitive analysis by means of field-insensitive analysis was proposed by [3]. However, there are essential differences between both approaches. The most important one is that [3] handles only numeric fields and it is not effective to handle reference fields. A main problem is that [3] *replicates* numeric fields with equivalent local variables, instead of *replacing* them as we have to do to handle references as well, e.g., an instruction like $y.ref{:=}x$ is followed (i.e., replicated) by $v_{ref}{:=}x$. Replicating instructions, when applied to reference fields, makes $y.ref$ and v_{ref} alias, and therefore the path-length relations of v_{ref} will be affected by those of $y.ref$. In particular, further updates to $y.ref$ will force losing useful path-length information about v_{ref}, since the abstract field update (see [15]) loses valuable path-length information on anything that shares with y. Handling reference fields is essential in object-oriented programs, as witnessed in our examples and experimental results. When applied to numeric fields, the fact that our transformation is context-sensitive and polyvariant can make it more accurate in some cases.

Techniques which rely on separation logic in order to track the depth (i.e., the path-length) of data-structures [5] would have the same limitation as path-length based techniques, if they are applied in a field-insensitive manner. This is because the depth of a variable x does not necessarily decrease when the depth of one of its field decreases (see Sec. 2). However, by applying these techniques

on our transformed programs, we expect them to infer the required information without any modification to their analyses.

References

1. A. Aiken, J. S. Foster, J. Kodumal, and T. Terauchi. Checking and inferring local non-aliasing. In *Proc.of PDLI' 03*, pages 129–140. ACM, 2003.
2. E. Albert, P. Arenas, M. Codish, S. Genaim, G. Puebla, and D. Zanardini. Termination Analysis of Java Bytecode. In *FMOODS*, LNCS 5051, pages 2–18, 2008.
3. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Field-Sensitive Value Analysis by Field-Insensitive Analysis. In *FM'09*, LNCS 5850, pages 370–386. Springer, 2009.
4. E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Resource usage analysis and its application to resource certification. In *FOSAD 2007/2008/2009 Tutorial Lectures*, LNCS 5705, pages 258–288. Springer, 2009.
5. J. Berdine, B. Cook, D. Distefano, and P. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *Proc. CAV*, 2006.
6. B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *IEEE PACT*, pages 280–291, 2001.
7. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. POPL*. ACM, 1978.
8. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.
9. H. Lehner and P. Müller. Formal translation of bytecode into BoogiePL. In *Bytecode'07*, ENTCS, pages 35–50. Elsevier, 2007.
10. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
11. A. Miné. Field-sensitive value analysis of embedded c programs with union types and pointer arithmetics. In *LCTES*, 2006.
12. C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl. Termination Analysis of Java Bytecode by Term Rewriting. In J. Waldmann, editor, *International Workshop on Termination WST'09*, Leipzig, Germany, June 2009.
13. G. Puebla and M. Hermenegildo. Abstract Multiple Specialization and its Application to Program Parallelization. *JLP*, 41(2&3):279–316, November 1999.
14. S. Rossignoli and F. Spoto. Detecting Non-Cyclicity by Abstract Compilation into Boolean Functions. In *VMCAI*, LNCS 3855. Springer, 2006.
15. F. Spoto, P. Hill, and E. Payet. Path-length analysis of object-oriented programs. In *EAAI'06*, ENTCS. Elsevier, 2006.
16. R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *CASCON'99*, pages 125–135, 1999.
17. S. Xia, M. Fähndrich, and F. Logozzo. Inferring Dataflow Properties of User Defined Table Processors. In *SAS*, LNCS 5673, pages 19–35. Springer, 2009.