Conditional Termination of Loops
over Heap-Allocated Data

Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, Guillermo
Román-Díez

DSIC, Complutense University of Madrid (UCM), Spain
DLSIIS, Technical University of Madrid (UPM), Spain

Abstract

Static analysis which takes into account the values of data stored in the heap
is considered complex and computationally intractable in practice. Thus, most
static analyzers do not keep track of object fields nor of array contents, i.e.,
they are heap-insensitive. In this article, we propose locality conditions for
soundly tracking heap-allocated data in Java (bytecode) programs, by means
of ghost non-heap allocated variables. This way, heap-insensitive analysis over
the transformed program can infer information on the original heap-allocated
data without sacrificing efficiency. If the locality conditions cannot be proven
unconditionally, we seek to generate aliasing preconditions which, when they
hold in the initial state, guarantee the termination of the program. Experimental
results show that we greatly improve the accuracy w.r.t. a heap-insensitive
analysis while the overhead introduced is reasonable.

Keywords: Static Analysis, Heap-Sensitive Analysis, Termination, Java
Bytecode, Program Transformation

1. Introduction

It is well known that shared mutable data structures, such as those stored in
the heap, are the bane of formal reasoning and static analysis (see e.g. [23, 11]).
This problem is exacerbated in object-oriented programs, since most data reside
in objects and arrays stored in the heap. Analyses which keep track (resp. do
not keep track) of heap-allocated data are referred to as heap-sensitive (resp.
heap-insensitive). In most cases, neither of the two extremes of using a fully
heap-insensitive analysis or a fully heap-sensitive analysis is acceptable. The
former produces too imprecise results and the latter is often computationally

Preliminary versions of some parts of this work were presented at FM’09 [4], SAS’10 [5]
and Bytecode’12 [10].

Email addresses: elvira@sip.ucm.es (Elvira Albert), puri@sip.ucm.es (Puri Arenas),
samir@fdi.ucm.es (Samir Genaim), german@fi.upm.es (Germán Puebla), groman@fi.upm.es
(Guillermo Román-Díez)
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 values of heap-allocated data are modeled. A well-known heuristic 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.

The approach we propose in this article is based on the observation that, by analyzing program fragments (or scopes), rather than the application as a whole, it is often possible to keep track of the values of heap-allocated data in a similar way as for non-heap-allocated variables. Such fragments can be built starting from methods, loops, or even blocks of contiguous sentences. Our final goal is to be able to instrument programs such that accesses to heap-allocated data are replaced with (or replicated by) equivalent accesses to, non-heap allocated, ghost variables whose values represent the values of the corresponding heap-allocated data. The instrumented program can then be input to any heap-insensitive static analysis, which can now obtain heap-sensitive information, since the ghost variables expose the heap-allocated values.

The kind of properties that can benefit from our approach are those which require a local or compositional reasoning, i.e., they require the inference of the property for certain fragments, rather than a global inference for the whole program execution. Termination, the target application of our article, is a property that requires such kind of local reasoning, where the scopes of interest are the loops. Basically, in order to prove termination, the analysis has to 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 [25] for the loop, which is a function which strictly decreases on a well-founded domain at each iteration of the loop and which ensures termination of the corresponding loop.

Obviously, not all heap-allocated data are transformable, i.e., their behaviour reproducible using ghost variables. In the most general characterization, the replacement is possible when two sufficient conditions hold within the scope: (a) the memory location where the heap-allocated data is stored does not change, i.e., the reference to such data remains constant, and (b) all accesses (if any) to such memory location are done through the same reference (and not through aliases). This characterization captures the situations in which heap-allocated data behave locally (i.e., like non heap-allocated variables) in the given scopes.

1.1. Summary of Contributions

The overall contribution is a practical method to perform heap-sensitive termination analysis by (a) first performing a pre-analysis which allows us to determine when heap-allocated data behave locally, (i.e., their behaviour is reproducible using ghost variables) and, otherwise, infer the conditions under which locality holds and, (b) then instrumenting the program with ghost variables whose contents represent the contents of the corresponding heap-allocated
data. Heap-insensitive termination analysis on the transformed program allows us to reason on data allocated in the heap through the ghost variables. Technically, our main contributions can be summarized as follows:

1. We first develop a semantic-based reference constancy analysis (instead of just performing syntactic checks) which infers the (constant) access paths to both fields and array elements in a uniform way.

2. We then present a general notion of locality for heap-allocated data which can be checked using the results obtained by the reference constancy analysis, and which determines if heap-allocated data behave as local variables.

3. Sometimes, heap-allocated data behave locally only under certain aliasing conditions. We introduce the notion of locality partition which, by assuming such aliasing conditions, guarantees the locality of the considered heap-allocated data.

4. Based on the notion of locality partition, we introduce a novel transformation which replaces the accesses to local heap-allocated data by non-heap allocated ghost variables. An interesting aspect of our conditional heap-sensitive analysis that we will show in the article is that we can improve the accuracy of the unconditional heap-insensitive analysis even in cases for which programs terminate unconditionally.

5. We then propose an approach for automatically inferring the aliasing preconditions that, when they hold in the initial state, guarantee the termination of the program under consideration.

6. We implement our approach in the COSTA system [8], a cost and termination analyzer for Java bytecode, and evaluate it on the Apache Commons Libraries [26].

1.2. Organization of the Article

This article is organized as follows. The next section briefly describes the language we consider, which is an intermediate (rule-based) representation of Java bytecode [22], and its semantics. Section 3 is devoted to presenting the reference constancy analysis for programs written in this language.

Section 4 introduces the heap-sensitive analysis in three main steps. We first define a simple locality condition which relies on the information inferred by the reference constancy analysis. Then, we discuss that such a condition might only hold under some aliasing (or not aliasing) conditions among the heap accesses. This leads to the notion of locality partition explained above. We can finally present a transformation which actually replaces the heap accesses which meet the locality condition by ghost variables for the given locality partition.

In Section 4, we do not specify how the termination (aliasing) preconditions can be generated. This is considered in Section 5 where we propose an approach to infer the conditions based on two notions of termination: local termination, which guarantees that the loops defined in a given scope $S$ are terminating, ignoring the termination behavior of loops defined in scopes that are called from $S$; and global termination, which guarantees that the loops of $S$ as well as those of scopes that are transitively called from $S$ are terminating. Basically,
preconditions are first inferred at a local termination level, and then they are combined to obtain the global preconditions.

Section 6 summarizes our experimental results performed on the Apache Commons Libraries [26]. We have analyzed all loops that contain guards with heap-accesses. Precision is greatly improved by the use of heap-sensitive analysis: namely using heap-insensitive analysis we can prove termination of 11.2% of loops, using unconditional heap-sensitive analysis this percentage increases to 75.9%, and using conditional heap-sensitive analysis we further increase it to 88%. The slowdown introduced by the unconditional heap-sensitive (w.r.t. heap-insensitive) is in most cases less than 2, while the overhead of the conditional (w.r.t. the unconditional heap-sensitive) is in most cases less than 1.25. Altogether, we argue that our experiments show that our approach pays off in practice.

Finally, Section 8 relates our approach to previous work and Section 9 concludes and points out several directions for future work.

2. A Simple Object-Oriented Imperative Bytecode Language

To formalize our analysis, we consider a simple rule-based imperative language (in the style of any of the languages in [6, 31, 21]). It has been shown that Java bytecode (and hence Java) can be automatically compiled into this intermediate language [6, 8]. When compared to analyzing the original bytecode, the key features 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 are useful for our transformation later.

2.1. Language Syntax

A rule-based program $P$ consists of a set of procedures and a set of classes. The set of class names $C$ defined in $P$ is denoted by $\text{classes}(P)$. We do not give an explicit syntax for defining classes, when needed, we simply use Java’s syntax. A procedure $p$ with $k$ input arguments $\bar{x} = \langle x_1, \ldots, x_k \rangle$ and $m$ output arguments $\bar{y} = \langle y_1, \ldots, y_m \rangle$ is defined by one or more guarded rules which adhere to this grammar:

\begin{verbatim}
rule ::= p(\bar{x}, \bar{y}) ← g, body.
g ::= true | exp_1 op exp_2 | type(x, C)
body ::= ε | b, body
b ::= x:=exp | x:=new C | x:=y.f | x.f:=y | x:=newarray(D, y) |
   | x[y]:=z | z:=x[y] | x:=arraylength(y) | q(\bar{x}, \bar{y})
exp ::= x | null | n | x aop y
aop ::= + | − | / | *
op ::= > | < | ≤ | ≥ | = | ≠
\end{verbatim}
where \( p(\bar{x}, \bar{y}) \) is the head of the rule; \( g \) its guard, which specifies conditions for the rule to be applicable; \( \text{body} \) the body of the rule; \( n \) an integer; \( x, y \) and \( z \) variables; \( f \) a field name and \( q(\bar{x}, \bar{y}) \) a procedure call by value. We assume that rules that belong to the same procedure have the same input and output parameter names.

The language supports class definition and includes instructions for object and array creation and manipulation. A class contains a finite set of typed field names, where a type can be (1) an integer; (2) a class \( C \in \text{classes}(P) \); or (3) an array whose elements are of type integer or \( \text{classes}(P) \).

We assume that the same field name, if used in different classes, has the same type. This can be done by automatically encoding the type into the field name. The set of all field names defined in \( P \) is denoted by \( \text{fields}(P) \). The instruction \text{new} \( C \) creates an object of type \( C \) and returns a reference to it, \text{newarray}(D, y) \) creates an array of \( y \) elements of type \( D \in \{ \text{int} \} \cup \text{classes}(P) \) and \text{arraylength}(y) \) returns the length of the array \( y \). For simplicity, we support only unidimensional arrays.

Classes, in our language, are in fact closer to records in C than classes in Java, as they encapsulate only fields and not methods, and do not use inheritance or interfaces as in Java. Such features are compiled to our language as we explain next. The translation from (Java) bytecode to the rule-based form is performed in two steps that are described in detail in [8]. First, a control flow graph is built for each method, where the virtual invocations are resolved using points-to analysis (this is why in our language classes contain only fields). 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 formalizing the analysis, the language does not include features of Java, such as exceptions, static fields, access control and primitive types besides integers, arrays and references. These features do not pose any technical difficulty to our analysis, but some of them (e.g., exceptions) require a more precise modeling of data. Our implementation deals with full sequential Java bytecode.

2.2. Language Semantics

First, we assume that programs have been verified for well-typedness. By well-typedness we mean that any program variable at a given program point can hold (in any possible execution) either a reference or an integer, but not both. We refer to such types as static types. The following four types are considered: \text{int} for integers, \text{ref} for object references, \text{aint} for references to arrays of integers and \text{aref} for references to arrays of objects. Given a variable \( x \), we let \text{stype}(x) \) denote its static type. Due to well-typedness, for the case of \( x[y]:=z \) and \( z:=x[y] \), the variable \( x \) can be (in any execution) either a reference to an array of integers or to an array of references (because \( z \) has always the same static type). For such case, we assume that \text{stype}(x) \) returns \text{aint} or \text{aref} respectively. Note that, for simplicity, we assume that variable \( x \) implicitly

\footnote{Note the loss of precision in the points-to analysis.}
contains information on the program point at which it appears so that its **stype** can uniquely identify this variable.

The execution of rule-based programs mimics standard bytecode [22]. The rules in Figure 1 define an operational semantics for the language (see [8] for more details). An **execution state** takes the form \( ar;h \), where \( ar \) is a stack of activation records, and \( h \) is a global heap. An **activation record** has the form \( \langle p, bc, tv \rangle \), where \( p \) is a procedure name, \( bc \) is a sequence of instructions, and \( tv \) is a variable mapping. Given a variable \( x \), \( tv(x) \) refers to the value of \( x \), and \( tv[x \mapsto v] \) updates \( tv \) by making \( tv(x) = v \) while \( tv \) remains the same for all other variables. A **heap** \( h \) is a partial map from an infinite set of memory locations (or reference) to objects. We use \( h[r \mapsto o] \) to denote the object referred to by \( r \) in \( h \). We use \( h[r \mapsto o] \) as a shortcut for \( h(o[r \mapsto o]) \) with \( o = h(r) \).

In rule (1), \( \text{eval}(\text{exp}, tv) \) returns the evaluation of the arithmetic or boolean expression \( \text{exp} \) for the values of the corresponding variables from \( tv \) in the standard way; for reference variables, it returns the reference. We assume that well-typing forbids pointer arithmetics. Rules (2), (3) and (4) deal with objects as expected. Procedure \( \text{newobject}(C) \) creates a new object of class \( C \) by initializing its fields to either 0 or null, depending on their types. Rules (5), (6), (7) and (8) account for arrays. For simplicity, an array of length \( v \) is modeled as an object \( o \) with a special (read-only) field \( \text{length} \) initialized to \( v \), and fields \( 1, \ldots, v \) which correspond to the array elements. The call \( \text{newarray}(D, v) \) creates an array of \( v \) elements initialized to 0 or null. Note that Rule (7) prevents “simulating” multi-dimensional arrays. Rule (9) (resp., (10)) corresponds to calling (resp., returning from) a procedure. The notation \( p[y', \bar{y}] \) records the association between the formal and actual return variables. \( \text{newenv} \) creates a new mapping of local variables for the method, where each variable is initialized to either 0 or null.

An execution for a program \( P \) starts from an initial configuration of the form \( \langle \text{start}, p(\bar{x}, \bar{y}), tv; h \rangle \), and ends in a final configuration \( \langle \text{start}, \epsilon, tv'; h' \rangle \), where:

1. \( \text{start} \) is an auxiliary name to indicate an initial activation record;
2. \( p(\bar{x}, \bar{y}) \) is a call to the procedure from which the execution starts;
3. \( h \) is an initial heap; and
4. \( tv \) is a variable mapping such that \( \text{dom}(tv) = \{ \bar{x} \} \cup \{ \bar{y} \} \), and all variables are initialized to an integer value, null or a reference to an object in \( h \).

Executions can be regarded as traces of the form \( C_0 \leadsto C_1 \leadsto \cdots \leadsto C_f \) (abbreviated \( C_0 \leadsto^* C_f \)), where \( C_f \) is a final configuration. Non-terminating executions have infinite traces.
To simplify the example, the method implements the Iterator interface. 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 Figure 2 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. Class List implements a linked list in the standard way. Finally, class UseIterator contains the method of interest m. The interesting features of this method are:
the combined use of fields and arrays, that it contains two nested loops which must be analyzed compositionally and that its termination can be only proven conditionally. For now, we focus on the inner while loop of method m. Its intermediate representation is made up of these three rules:

while((a, b, i, j), ( )) ←
  s0 := a,
  s1 := i,
  s0 := s0[s1],
while((a, b, i, j, s0), ( )).
while((a, b, i, j, s0), ( )) ← s0 ≤ 0.

The entry rule while receives as input parameters two references to the arrays a and b and two integer values i and j. Guards that are true are omitted in the example. An important point to note is that the accesses to the array are performed by pushing the values to the stack, which in the intermediate representation are just local variables. For instance, the first three instructions in the body of procedure while push the value of a[i] in the stack position s0. Therefore, the development of syntactic techniques (rather than semantics-based) to reason on these programs would be quite complex. Note that the termination of the method m can only be ensured if the array references a and b are different, i.e., point to different memory locations.
3. Reference Constancy Analysis

In this section, we develop a reference constancy analysis which allows us to obtain the access paths to the fields, arrays and array elements which are constant in the considered scopes. The idea behind this analysis is similar in spirit to that of the classical numeric constant propagation analysis [16]. However, in addition to numerical constants, the values computed by our analysis can include symbolic expressions that refer to locations in (the initial) heap. Such expressions encode as well the way that the corresponding memory locations are reached (e.g., the dereferenced fields).

Example 2. Consider the examples in Figure 3. S₁ and S₂ are used to delimit scopes. In A, the reference x remains constant within the scope of loop S₁ since its value does not change. However, if we consider the whole code fragment S₂, x is no longer constant, since x can take two different values before the loop. In B, all occurrences of x are constant within the scope S₁ of the inner loop. However, x takes different values in different iterations of the outer loop, and thus x is not constant in the whole scope S₂. In C, x is not constant because it is updated at each iteration of the loop. In D, it cannot be ensured that x[0].r is constant, since if x[0] and y are aliases, updating y.r changes x[0].r.

3.1. The Set of Access Paths

We start by defining the set of (symbolic) abstract values that our analysis assigns to each variable, at each program point. We refer to these values as access paths since they provide a symbolic representation of the access path to the memory locations. They are defined in terms of the (symbolic) input parameters of the initial call. We denote these parameters by \( \mathcal{L} = \{ l_1, \ldots, l_n \} \), where \( l_i \) represents the value of the \( i \)-th parameter. An access path will be denoted by \( A \) (possibly subscribed or primed), and it can be one of the following values:

1. \( n \in \mathbb{Z} \), which represents the corresponding integer;
2. \( A_{\text{null}} \), which represents the value null;
3. \(l_1 \cdot F_1 \cdots F_n\) or \(l_i[A'] \cdot F_1 \cdots F_n\), where each \(F_i\) is of the form \(f_i\) or \(f_i[A'']\) and \(f_i \in \text{fields}(P)\). Note that \(n\) can also be 0, in which case we do not access any field. Moreover, \(A'\) and \(A''\) are different from \(A_{null}\).

An access path \(A\) might refer to an integer or a value or a reference to an object or an array. Observe also that each \(l_i\) inherits the static type of its corresponding parameter. In addition to the access paths defined by the above rules, we use a special one, denoted by \(A_{any}\), that represents any value. Note that \(A_{any}\) cannot appear as part of any other access path.

**Example 3.** Intuitively, our analysis will assign to each variable (at each program point) an access path which describes its possible values whenever the execution reaches that point. Suppose that a variable \(x\), at some program point, is assigned the access path \(A\). Let us intuitively explain the meanings for some possible values of \(A\): (1) if \(A = 5\), then the value of \(x\) is equal to 5; (2) if \(A = l_2\), then the value of \(x\) is equal to the value of the second initial parameter; (3) if \(A = l_1.f\cdot g\), assuming that the first parameter points to an object \(o\), then \(x\) has the value of \(o.f\cdot g\) when evaluated in the initial state; (4) if \(A = l_2[l_i]\), assuming that the second initial parameter points to an array \(a\) and that the fourth parameter is an integer \(n\), then the value of \(x\) is like that of \(a[n]\) (again, when evaluated in the initial state); and (5) if \(A = A_{any}\), then the value of \(x\) can be any reference or integer value, depending on the type of \(x\).

The set of all access paths, w.r.t. a given set \(L\) of initial parameters, is denoted by \(AP(L)\). Given an access path \(A\), we denote by \(A[l_1/A_1, \ldots, l_n/A_n]\) the access path that results from simultaneously replacing each occurrence of \(l_i\) by \(A_i\). If the result includes \(A_{any}\), i.e., it is an invalid access path, then we assume that \(A[l_1/A_1, \ldots, l_n/A_n]\) is actually \(A_{any}\). This replacement operation extends for any entity that involves access paths. The set of access paths \(AP(L)\) is partially ordered by \(\subseteq\) such that for any \(A \in AP(L)\) we have \(A \subseteq A'\) if and only if \(A = A'\) or \(A' = A_{any}\). We let \(A_1 \sqsubseteq A_2\) be \(A_1\) if \(A_1 = A_2\); otherwise \(A_{any}\).

### 3.2. The Analysis

In order to assign access paths to variables at program point level, we first need to define such program points. For this, we assume that the program’s rules are uniquely numbered starting from 1. The \(k\)-th program rule \(p(x, y) \leftarrow g, b_1^k, \ldots, b_t^k\) has \(t + 1\) program points. The first one, \(k:1\), after the execution of the guard \(g\) and before the execution of \(b_1\), then \(k:2\) between the execution of \(b_1\) and \(b_2\), until \(k:t+1\) after the execution of \(b_t\). For an initial configuration \(C_0 = <\text{start}, p(x, y), tw>\), we assume that the call \(p(x, y)\) corresponds to program point 0:0. The set of all program points of a program \(P\), including 0:0, is denoted by \(pps(P)\).

The analysis receives as input a program \(P\) and a procedure name \(p\) which we refer to as the *entry*. We assume that \(p\) has \(n\) arguments, and their symbolic values, as above, are denoted by \(L = \{l_1, \ldots, l_n\}\). The analysis assigns to each program point \(k:j \in pps(P)\) an abstract state, from which it is possible to obtain
the access path of each variable, at any program point. Given a set of (typed) variables \( \mathcal{V} \), defined at a given program point, an abstract state over \( \mathcal{V} \) and \( \mathcal{L} \) has the form \( \langle \phi, \theta \rangle \), where \( \phi : \mathcal{V} \mapsto AP(\mathcal{L}) \) maps variables to access paths; and \( \theta \subseteq \text{fields}(P) \cup \{ \text{aint}, \text{aref} \} \) is a set of field names and array-types which are guaranteed to be constant, i.e., they are not modified in any execution that reaches the corresponding program point. Our main interest is in inferring \( \phi \), \( \theta \) is auxiliary to soundly construct \( \phi \) during the analysis.

We let \( \mathcal{S}(\mathcal{V}, \mathcal{L}) \) be the set of all abstract states, w.r.t. some \( \mathcal{V} \) and \( \mathcal{L} \). We say \( \langle \phi_1, \theta_1 \rangle \subseteq \langle \phi_2, \theta_2 \rangle \) if \( \theta_2 \subseteq \theta_1 \), and \( \phi_1(x) \subseteq_a \phi_2(x) \) for any \( x \in \mathcal{V} \). We let \( \langle \phi_1, \theta_1 \rangle \cup_a \langle \phi_2, \theta_2 \rangle = \langle \phi, \theta_1 \cap \theta_2 \rangle \) where \( \phi(x) = \phi_1(x) \cup_a \phi_2(x) \) for any \( x \in \mathcal{V} \).

We point out that \( \theta \) contains those field names and array-types which remain constant during the analysis. Hence, each time a field or an array component is modified, the corresponding signature is removed from \( \theta \) (see rules (2) and (9) in Figure 4). This is why \( \theta_2 \subseteq \theta_1 \) is required when defining \( \subseteq_a \). Similarly, when doing the operation \( \cup_a \) between two abstract states, the resulting abstract state must keep only those field signatures and array-types which are constant in both of them. Thus the new abstract state will contain the intersection of the corresponding \( \theta \)'s.

We denote by \( \mathcal{A}(\mathcal{V}, \mathcal{L}) \) the complete lattice \( \langle \mathcal{S}(\mathcal{V}, \mathcal{L}), \top_s, \bot_s, \cup_s, \cap_s, \subseteq_s \rangle \), where (1) the top element \( \top_s \) corresponds to \( \langle \phi, \emptyset \rangle \) in which \( \phi(x) = \mathcal{A}_{\text{any}} \) for any \( x \in \mathcal{V} \); and (2) the bottom element \( \bot_s \) is a symbolic value that represents an empty abstract state. Next we lift \( \mathcal{A}(\mathcal{V}, \mathcal{L}) \) in order to represent a set of abstract states, one for each \( k: j \in \text{pps}(P) \). We represent such states as sets of elements of the form \( k: j \mapsto \langle \phi, \theta \rangle \) (or \( k: j \mapsto \bot_s \)) where \( \langle \phi, \theta \rangle \in \mathcal{S}(\mathcal{V}_{k: j}, \mathcal{L}) \). Here \( \mathcal{V}_{k: j} \) is the set of (typed) variables that are available at program point \( k: j \). Such set must include an abstract state for each \( k: j \in \text{pps}(P) \). The set of all such states is denoted by \( \mathcal{S}_p \). We use \( \mathcal{A}_p \) to denote the complete lattice \( \langle \mathcal{S}_p, \top_p, \bot_p, \cup_p, \cap_p, \subseteq_p \rangle \) where \( \top_p, \bot_p, \cup_p, \cap_p, \subseteq_p \) are defined by lifting the corresponding ones of \( \mathcal{A}(\mathcal{V}_{k: j}, \mathcal{L}) \).

The analysis is based on a transfer function \( \tau \), depicted in Figure 4, that defines the effect of executing each (simple) instruction on a given abstract state \( \langle \phi, \theta \rangle \). Let us explain the different cases of the transfer function:

1. When a variable \( x \) is assigned the value of \( y \cdot f \), there are two cases. The first one corresponds to the case in which \( \phi(y) = \mathcal{A}_{\text{null}} \), i.e., a possible \text{null pointer exception}. In this case the transfer function returns \( \bot_s \), i.e., the empty abstract state. Otherwise, the transfer function updates the access path of \( x \) accordingly: if \( y \) is not \( \mathcal{A}_{\text{any}} \), and field \( f \) has not been updated so far, then the resulting access path is the concatenation of that of \( y \) with the symbol \( f \); otherwise \( \mathcal{A}_{\text{any}} \).

2. When a field \( f \) is assigned a value, then if \( \phi(x) = \mathcal{A}_{\text{null}} \) we proceed as in (1). Otherwise \( f \) is eliminated from \( \theta \), i.e., it is marked as a field that has been updated and it is not constant. Note that in any subsequent execution step, an access \( y \cdot f \) (of case (1)) will result in \( \mathcal{A}_{\text{any}} \).

3. This case simply updates the access path of \( x \) to the number \( n \).
<table>
<thead>
<tr>
<th>Instruction $b$</th>
<th>Transfer function $\tau(b, \langle \phi, \theta \rangle)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) $x := y.f$</td>
<td>$\bot_s$ if $\phi(y) = A_{\text{null}}$, otherwise $\langle \phi[x \mapsto A], \theta \rangle$</td>
</tr>
<tr>
<td>(2) $x.f := y$</td>
<td>$\bot_s$ if $\phi(x) = A_{\text{null}}$, otherwise $\langle \phi, \theta \backslash {f} \rangle$</td>
</tr>
<tr>
<td>(3) $x := n$</td>
<td>$\langle \phi[x \mapsto n], \theta \rangle$</td>
</tr>
<tr>
<td>(4) $x := \text{null}$</td>
<td>$\langle \phi[x \mapsto A_{\text{null}}], \theta \rangle$</td>
</tr>
<tr>
<td>(5) $x := y$</td>
<td>$\langle \phi[x \mapsto \phi(y)], \theta \rangle$</td>
</tr>
<tr>
<td>(6) $x := y \text{ aop } z$</td>
<td>$\langle \phi[x \mapsto A], \theta \rangle$</td>
</tr>
<tr>
<td>(7) $x := \text{newarray}(D, y)$</td>
<td>$\langle \phi[x \mapsto A_{\text{any}}], \theta \rangle$</td>
</tr>
<tr>
<td>(8) $x := y[z]$</td>
<td>$\bot_s$ if $\phi(y) = A_{\text{null}}$, otherwise $\langle \phi[x \mapsto A], \theta \rangle$</td>
</tr>
<tr>
<td>(9) $x[y] := z$</td>
<td>$\bot_s$ if $\phi(x) = A_{\text{null}}$, otherwise $\langle \phi, \theta \backslash {\text{stype}(x)} \rangle$</td>
</tr>
<tr>
<td>(10) $x := \text{new } C$</td>
<td>$\langle \phi[x \mapsto A_{\text{any}}], \theta \rangle$</td>
</tr>
<tr>
<td>(11) $x := \text{arraylength}(y)$</td>
<td>$\langle \phi[x \mapsto A_{\text{any}}], \theta \rangle$</td>
</tr>
<tr>
<td>(12) otherwise</td>
<td>$\langle \phi, \theta \rangle$</td>
</tr>
</tbody>
</table>

where we have the following conditions in rules:

1. If $f \in \theta \land \phi(y) \neq A_{\text{any}}$ then $A = \phi(y):f$; otherwise $A = A_{\text{any}}$.

6. If $\phi(y)$ and $\phi(z)$ are numbers, then $A = \phi(y) \text{ aop } \phi(z)$; otherwise $A = A_{\text{any}}$.

8. If $\text{stype}(y) \in \theta \land \phi(y) \neq A_{\text{any}} \land \phi(z) \neq A_{\text{any}}$ then $A = \phi(y)[\phi(z)]$; otherwise $A = A_{\text{any}}$.

Figure 4: Transfer function for reference constancy analysis

4. Similarly to the above case, it updates the access path of $x$ to $A_{\text{null}}$.

5. This case updates the access path of $x$ with that of $y$.

6. If the access path of $y$ and $z$ are numbers, then the access path of $x$ is updated to be the result of applying the corresponding arithmetic operator; otherwise it is updated to $A_{\text{any}}$ to indicate that it can be any number.

7. In this case, when creating a new array, the access path of $x$ is updated to $A_{\text{any}}$ to indicate that it can be any reference value.

8. If $\phi(y) = A_{\text{null}}$, i.e., a null pointer exception occurs, the transfer function returns the empty abstract state $\bot_s$. Otherwise if the access paths of $y$ and $z$ are not $A_{\text{any}}$, and it is guaranteed that the accessed array has not been modified (its static type is still in $\theta$), then the access path of $x$ is computed accordingly; otherwise it is $A_{\text{any}}$.

9. It eliminates the static type of array $x$ from $\theta$ whenever $\phi(x) \neq A_{\text{null}}$. It is analogue to case (2).

10. This case is similar to case (7). The access path of $x$ is updated to $A_{\text{any}}$, to indicate that its value might be any reference value.
Let us consider an abstract state \( \langle \phi_1, \theta_0 \rangle \) where \( \phi_1 = \{ a \mapsto \ell_1, i \mapsto \ell_2 \} \). Assume the fragment code \( z := a[i], x.f := z, w := x.f \). Then, by rule (8) in Figure 4, \( \tau(z := a[i], \langle \phi_0, \theta_0 \rangle) \) returns the new abstract state \( \langle \phi_1, \theta_0 \rangle \), where \( \phi_1 = \phi_0 \cup \{ z \mapsto \ell_1[\ell_2] \} \). Then, the application of rule (2) on the instruction \( x.f := z \) removes the field \( f \) from \( \theta_0 \) and returns \( \langle \phi_1, \{ \text{aint} \} \rangle \). Finally, with this last abstract state and rule (1) applied on the instruction \( w := x.f \) we compute \( \langle \phi_1 \cup \{ w \mapsto \text{A}_{\text{any}} \}, \{ \text{aint} \} \rangle \), what means that at this point only the array-type remains constant.

The analysis starts from an abstract state \( I^0_{\#} \in \bar{A}_P \) that assigns \( \perp \) to each program point \( k:j \in \text{pps}(P) \), except for 0:0 which is assigned the abstract state \( \langle \phi, \text{fields}(P) \cup \{ \text{aint, aref} \} \rangle \) where \( \phi \) maps each \( x_i \) (resp. \( y_i \)) of the initial call \( p(\bar{x}, \bar{y}) \) to \( \ell_i \) (resp. \( A_{\text{null}} \) or 0 depending on its type). Then, it iteratively computes \( I^i_{\#} = I^i_{\#} \cup \bigcup F^\#_p(I^i_{\#}) \) until it reaches a state in which \( I^{i+1}_{\#} = I^i_{\#} \). The operator \( F^\#_p : \mathcal{S}_P \rightarrow \mathcal{S}_P \) is defined as \( F^\#_p(X) = F^\#_1(X) \cup F^\#_2(X) \cup F^\#_3(X) \) where each \( F^\#_i \) is as follows:

\[
F^\#_1(X) = \begin{cases} 
  k:j+1 \mapsto \langle \phi', \theta' \rangle & \text{if } b^k_j \in P \text{ which is not a call} \\
  k:j \mapsto \langle \phi, \theta \rangle \in X & \text{if } \langle \phi', \theta' \rangle = \tau(b^k_j, \langle \phi, \theta \rangle) 
\end{cases}
\]

\[
F^\#_2(X) = \begin{cases} 
  k':1 \mapsto \langle \phi', \theta' \rangle & \text{if } b^k_j \equiv q(\bar{w}, \bar{z}) \in P, q \text{ is defined by rule} \\
  k:j \mapsto \langle \phi, \theta \rangle \in X & \text{if } k' \equiv q(\bar{x}, \bar{y}) \mapsto g, b^{k'}_1, \ldots, b^{k'}_r \in P, \\
  k:j \mapsto \langle \phi, \theta \rangle \in X & \text{if } \phi'' = \text{init}(k'), \phi' = \phi''[x_i \mapsto \phi(w_i)], \theta'' = \theta 
\end{cases}
\]

\[
F^\#_3(X) = \begin{cases} 
  k':j+1 \mapsto \langle \phi', \theta' \rangle & \text{if } p(\bar{x}, \bar{y}) \mapsto g, b^{k}_1, \ldots, b^{k'}_r \in P, b^{k'}_j \equiv p(\bar{w}, \bar{z}) \in P, \\
  k:j+1 \mapsto \langle \phi, \theta \rangle \in X, k':j \mapsto \langle \phi'', \theta'' \rangle \in X & \text{if } k:j+1 \mapsto \langle \phi, \theta \rangle \in X, k':j \mapsto \langle \phi'', \theta'' \rangle \in X, \\
  \phi' = \phi[z_i \mapsto \phi''(y_i)], \theta' = \theta'' 
\end{cases}
\]

Let us explain each \( F^\#_i \):

- \( F^\#_1 \) handles cases in which the instruction \( b^k_j \) is a simple instruction. It uses the current abstract state for program point \( k:j \) and the transfer function \( \tau \) in order to propagate the information to program point \( k:j+1 \).

- \( F^\#_2 \) handles cases in which \( k:j \) is a call \( q(\bar{w}, \bar{z}) \) to another procedure. In this case we propagate the abstract state of program point \( k:j \) to the entry program point \( k':1 \) for each rule \( k' \) that defines \( q \). This is done by first creating an initial mapping \( \phi'' \) that maps each variable of rule \( k' \) to either \( 0 \) or \( A_{\text{null}} \), depending on its type at that program point, and then modifying the access path of each \( x_i \) to be as that of the \( i \)-th actual parameter.

- In a similar way, \( F^\#_3 \) handles cases in which we propagate the return values to the calling context.

(11) Simply maps \( x \to \mathcal{A}_{\text{any}} \) since the length of the array can be any number.
while((a, b, i, j), ()) ←
    s0:=a,
    s1:=1,
    1s0:=s0[s1],
    while.((a, b, i, j, s0), ()),
    while.((a, b, i, j, s0), ()),
    while.((a, b, i, j, s0), ()),
       2s3:=s3[s4],
       s3:=s3 − 1,
       s1[s2]:=s3,
       s1:=b, s2:=j, s3:=b, s4:=j,
  3s3:=s3[s4],
  s3:=s3 + 1,
  s1[s2]:=s3,
while((a, b, i, j), ()),

{a→l1, b→l2, i→l3, j→l4}
{s0→l1, a→l1, b→l2, i→l3, j→l4}
{s1→l3, s0→l1, a→l1, b→l2, i→l3, j→l4}
{s1→l3, s0→Aany, a→l1, b→l2, i→l3, j→l4}
{s1→l3, s0→Aany, a→l1, b→l2, i→l3, j→l4}
{s0→Aany, a→l1, b→l2, i→l3, j→l4}
{s0→Aany, a→l1, b→l2, i→l3, j→l4}
{s4→l3, s3→l1, s2→l3, s1→l1} \cup X
{s4→l3, s3→Aany, s2→l3, s1→l1} \cup X
{s4→l3, s3→Aany, s2→l3, s1→l1} \cup X
{s4→l3, s3→Aany, s2→l3, s1→l1} \cup X
{s4→l4, s3→l2, s2→l4, s1→l2} \cup X
{s4→l4, s3→Aany, s2→l4, s1→l2} \cup X
{s4→l4, s3→Aany, s2→l4, s1→l2} \cup X
{s4→l4, s3→Aany, s2→l4, s1→l2} \cup X
{s4→l4, s3→Aany, s2→l4, s1→l2} \cup X
{s4→l4, s3→Aany, s2→l4, s1→l2} \cup X
{s4→l4, s3→Aany, s2→l4, s1→l2} \cup X
{s4→l4, s3→Aany, s2→l4, s1→l2} \cup X
{s4→l4, s3→Aany, s2→l4, s1→l2} \cup X
{s4→l4, s3→Aany, s2→l4, s1→l2} \cup X
{s4→l4, s3→Aany, s2→l4, s1→l2} \cup X
{s4→l4, s3→Aany, s2→l4, s1→l2} \cup X
{s4→l4, s3→Aany, s2→l4, s1→l2} \cup X

Figure 5: IR of the running example (left). Program point constancy information (right)

Example 5. Figure 5 shows to the left the intermediate representation of the while loop of Example 1 and to the right the abstract states, computed by the analysis, for some selected program points. Each abstract state corresponds to the result after analyzing the instructions in the corresponding line in the left-hand side. We use l1, l2, l3, l4 to refer to, respectively, the initial values of a, b, i and j. Observe that at program point 1, i.e., before executing s0:=s0[s1], the access paths assigned to s0 and s1 are respectively l1 and l3. This means that the corresponding array access will always refer to the memory location l1[3]. We will see that this piece of information is crucial to determine if the corresponding heap access is transformable into a local variable. Similarly, we can conclude that the array accesses at 2 and 3 will always refer to l1[3] and l2[4] respectively.

In what follows, we let I^#_p \in \mathcal{A}_p be the result of the analysis, which is computed iteratively as described above. It is actually the least fixpoint of \lambda X. I^0_p \sqcup_p F^#_p(X). We use \langle \phi_{k,j}, \theta_{k,j} \rangle to refer to the abstract state assigned to program point k,j in I^#_p. In addition, for a given procedure p, we define:

\langle \phi_p, \theta_p \rangle = \sqcup \{ \langle \phi, \theta \rangle \mid p(\bar{x}, \bar{y})\leftarrow g, b^p_k, \ldots, b^p_t \in P, k:t+1 \mapsto (\phi, \theta) \}

We refer to this abstract state as the summary of procedure p, which describes the access paths of its parameters upon exit from p in terms of the initial values that they take (see initial state I^0_p which assigns a different access path to each parameter). We assume that dom(\phi_p) always consists of variables with names \bar{x} and \bar{y} (just to avoid renamings).

14
Example 6. Let us consider the rules for while and while\(c\) in Figure 5. Note that because of the assignment \(s_1[s_2]:=s_3\) in the second rule of while\(c\), the value of \(\theta\) at the exit of both rules for while\(c\) will be \(\{\}\). Since while and while\(c\) are mutually recursive, then it is for sure that \(\theta\) will be empty also at the exit of while. Hence, it holds that \(\langle \phi_{\text{while}}, \theta_{\text{while}} \rangle = \langle \{a \mapsto l_1, b \mapsto l_2, i \mapsto l_3, j \mapsto l_4\}, \{\}\rangle \) and \(\langle \phi_{\text{while}}, \theta_{\text{while}} \rangle = \langle \{a \mapsto l_1, b \mapsto l_2, i \mapsto l_3, j \mapsto l_4, s_0 \mapsto A\text{ any}\}\rangle \).

3.3. Modular Analysis

References are often not globally constant, but they can be constant when we look at smaller fragments. Fortunately, the analysis can be applied modularly by partitioning the procedures (and therefore rules) of \(P\) into fragments which we refer to as scopes, provided that there are no mutual calls between scopes. The smaller the scopes are, the more precise the analysis result will be. Therefore, the strongly connected components (SCCs) of the program are the smallest scopes we can consider. We assume that each scope has a single entry. This is not a restriction since otherwise the analysis can be repeated for each entry.

Given a program \(P\), we let \(S_1, \ldots, S_n\) be the partitioning of its procedures into scopes, where the entry procedure of each \(S_i\) is \(p_i\). Since scopes are not mutually recursive, we can assume that if there is a call from a procedure in \(S_i\) to a procedure in \(S_j\), then \(i \geq j\). We refer to an inter-scope (resp. intra-scope) call as an external (resp. internal) call. Our aim is to apply the analysis of Section 3.2 in a modular way, by analyzing each scope separately, starting from \(S_1\), then \(S_2\), etc. Each scope \(S_k\) is analyzed by assuming an initial state, as in the non-modular case, but with a call to the entry procedure \(p_k(\bar{x}, \bar{y})\).

In order to achieve this modularity, we extend the transfer function \(\tau\) for the case of external procedure calls, such that it uses the corresponding summaries. Let \(p(\bar{w}, \bar{s})\) be an external call, the result of \(\tau(p(\bar{w}, \bar{s}), \langle \phi, \theta \rangle)\) is \(\langle \phi', \theta' \rangle\) where:

1. \(\theta' = \theta \cap \theta_p\)
2. \(\forall z \in \text{dom}(\phi) \setminus \bar{s}, \text{ we have } \phi'(z) = \phi(z); \text{ otherwise}\)
3. \(\forall s_i \in \bar{s}, \text{ then } \phi'(s_i) = \text{ren}(\phi_p(y_i), \phi, \theta)\) where \(\text{ren}\) is:
   - \(\text{if } A \text{ includes a field or array-type } f \notin \theta \text{ then return } A\text{ any}\)
   - \(\text{else return } A[l_1/\phi(w_1), \ldots, l_n/\phi(w_n)]\)

Intuitively, in (1) fields and array-types that might be updated during the execution of \(p\) are eliminated in the calling context; in (2) variables in the calling context which are not output variables of \(p\) keep their current access paths; and, in (3) the access paths of the output variables \(\bar{s}\) are incorporated into the calling context.

Example 7. We demonstrate the modular analysis on the example of Figure 2. We focus on method next, on the inner while loop of Example 5, and on the (outer) while loop (while\(m\)) inside method \(m\), which calls method next and procedure while (and hence reuses their summaries). The translation of next and the outer loop into the intermediate representation is as follows:
next((this), (r)) ←

obj := this.state,

s₀ := obj.next,

this.state := s₀,

r := obj.

whileₗₘ((a, b, y, o, i, j), (o, i, j)) ←

s₀ := y,

hasNext((s₀), (s₀)),

whileₗₘ((a, b, y, o, i, j, s₀), (o, i, j)).

whileₗₘ((a, b, y, o, i, j, s₀), (a, i, j)).

Let us consider the scopes Sₘₜₐₖₚₜ, computes this summary for next:

⟨φₙₜₓₓₑᵦₙ, θₙₜₓ⟩ = { ⟨this→l₁, r→l₁.state⟩, {next, aint} }

The set θₙₜₓ does not include field state, which indicates that its memory location might be modified during the execution of next. The meaning of the access path r → l₁.state is that the returned value of the method next is equal to the value of dereferencing (upon entering the procedure) the first input argument using the field state. Let us explain how this summary is reused when analyzing the scope Sₙₜ. When reaching program point ③ for the first time, we will have the abstract state:

⟨φ₀, θ₀⟩ = { ⟨a→l₁, b→l₁, y→l₁, s₀→l₁, o→l₁, i→l₁, j→l₁⟩, {state, next, aint} }

Note that θ₀ includes all fields and array-types that appear in S₁, S₂ and S₃, since none has been updated so far. In order to incorporate the effect of executing the method next into the calling context, we apply the transfer function τ(next((s₀), (s₀)), ⟨φ₀, θ₀⟩), which results in:

⟨φ₁, θ₁⟩ = { ⟨a→l₁, b→l₁, y→l₁, s₀→l₁, o→l₁, i→l₁, j→l₁⟩, {next, aint} }

Now, the field state is not in θ₁. The access path s₀ → l₁.state is obtained by taking that of r, i.e., l₁.state and renaming l₁ to φ₀(s₀) = l₃. We take φ₀(s₀) since l₁ refers to the value of the first argument when calling the method next, which is s₀. In the next iteration of the analysis, we reach ③ with the abstract state:

⟨φ₂, θ₂⟩ = { ⟨a→l₁, b→l₁, y→l₁, s₀→l₁, o→l₁, i→l₁, j→l₁⟩, {next} }

Observe that aint is not included in θ₂, since it is removed when incorporating the summary of the inner while loop (computed from the analysis results shown in Example 5), which modifies an array of integers. Next, applying τ(next((s₀), (s₀)), ⟨φ₂, θ₂⟩) results in:

⟨φ₃, θ₃⟩ = { ⟨a→l₁, b→l₁, y→l₁, s₀→A_any, o→l₁, i→l₁, j→l₁⟩, {next} }

16
The access path \( s_0 \rightarrow A_{\text{any}} \) is also obtained from \( r \rightarrow l_1 \cdot \text{state} \) as before. However, in this case \text{ren} returns \( A_{\text{any}} \) since \( l_1 \cdot \text{state} \) includes the field state and state \( \notin \theta_3 \).

Given an access path \( A \neq A_{\text{any}} \) and an initial state \( C_0 = \langle \text{start}, p_i(x, y), tv \rangle ; h \), we let \( \llbracket A \rrbracket(C_0) \) be the value that corresponds to \( A \) in \( C_0 \). For instance, for local variables the value is obtained from the variable mapping \( \llbracket x_2 \rrbracket(C_0) = tv(x_2) \), for fields the heap must be accessed as well \( \llbracket x_2 \cdot f \rrbracket(C_0) = h(tv(x_2)) \cdot f \), etc. (See Appendix A for the exact definition). Now we state the soundness theorem, assuming that the analysis results for each program point \( k:j \) are obtained in a modular way as explained above.

**Theorem 1 (soundness).** Given a scope \( S_i \), a program point \( k:j \) in \( S_i \), and a variable \( x \in V_{k:j} \) such that \( \phi_{k:j}(x) = A \neq A_{\text{any}} \). Then, for any trace \( C_0 = \langle \text{start}, p_i(x, y), tv \rangle ; h \sim^* \langle q, bc, tv' \rangle \cdot ar; h' \) where \( bc \) corresponds to the program point \( k:j \), it holds that \( tv'(x) = \llbracket A \rrbracket(C_0) \).

4. Heap-Sensitive Analysis

This section presents the core of our method in three steps: we provide in Section 4.1 some auxiliary definitions and the basic notion of (unconditional) locality. Then, Section 4.2 introduces the notion of conditional partition that will let us transform a heap access by a ghost variable access under certain conditions. The use of ghost variables is a common technique in program proving. Finally, we present in Section 4.3 an automatic transformation that actually carries out the conversion.

4.1. Basic Locality

Let us first introduce two auxiliary notions which define the set of read and write access paths to fields and to arrays within a scope. Intuitively, such sets provide information on how a field or array-type is accessed in a scope (and in its reachable scopes). This information is needed in Section 4.3 for soundly tracking the values that are stored in such heap location.

**Definition 1.** Given a scope \( S \) and a field \( f \), the set of read access paths for \( f \) in \( S \), denoted by \( R(S, f) \), is defined as \( R(S, f) = R^0(S, f) \cup R^*(S, f) \) where

\[
R^0(S, f) = \{ A \mid b^k_{\equiv x:=y, f \in S, A = \phi_{k:j}(y), A \neq A_{\text{null}} } \}
\]

\[
R^*(S, f) = \{ A' \mid b^k_{\equiv q(x, y) \in S, q \in S' \neq S, A \in R(S', f), A' = \text{ren}(A, \phi_{k:j}, \theta_{k:j}) } \}
\]

The set of write access paths for \( f \) in \( S \), denoted \( W(S, f) \), is computed analogously, by considering instructions of the form \( y.f := x \).

Let us explain the above definition. In \( R^0(S, f) \), 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 transitively called from \( S \). This is done in \( R^*(S, f) \). For each call such that \( q \) is the entry of the scope \( S' \) we take \( R(S', f) \) and rename it according
to the calling context using ren as defined in Section 3.3. Note that all access paths in \( R(S, f) \) and \( W(S, f) \) refer to memory locations, they do not include \( A \) such that \( A \in \mathbb{Z} \) or \( A = A_{\text{null}} \). The above definition extends for arrays in a natural way.

**Definition 2.** Given a scope \( S \) and an array-type \( f \in \{ \text{aref}, \text{aint} \} \), the set of read access paths for \( f \) in \( S \), denoted by \( R(S, f) \), is defined as \( R(S, f) = R^\circ(S, f) \cup R^*(S, f) \) where

\[
R^\circ(S, f) = \{ A \mid b_j^k \equiv x := y[z] \in S, \text{stype}(y) = f, \phi_{k,j}(y) \not\in A_{\text{null}}, A = \phi_{k,j}(y)[\phi_{k,j}(z)] \}
\]

and \( R^*(S, f) \) is as in Definition 1. The set of write access paths for \( f \) in \( S \), denoted \( W(S, f) \), is computed analogously, by considering instructions of the form \( y[z] := x \).

**Example 8.** Using the results of the constancy analysis in Examples 5 and 7, we have that the read/write access sets are: \( R(S_1, \text{next}) = \{ l_1, \text{state} \} \) and \( W(S_1, \text{next}) = \emptyset \) and \( R(S_1, \text{state}) = W(S_1, \text{state}) = \{ l_1 \} \). In the scope of the inner loop, we have that \( R(S_2, \text{aint}) = W(S_2, \text{aint}) = \{ l_1[l_3], l_2[l_4] \} \), since the array content is read and modified using the references \( a[i] \) and \( b[j] \). In \( S_3 \), we have that \( R(S_3, \text{next}) = \{ l_3, \text{state} \} \), \( W(S_3, \text{next}) = \emptyset \), \( R(S_3, \text{state}) = W(S_3, \text{aint}) = W(S_3, \text{aint}) = \{ l_1[l_5], l_2[l_6] \} \).

Intuitively, in order to ensure a sound transformation, a field can be considered local in a scope \( S \) if all read and write accesses to it in all reachable scopes are performed through the same access path. This makes it safe to replace such heap access by a corresponding ghost variable.

**Definition 3 (locality).** Given a field or an array-type \( f \) and a scope \( S \), we say that \( f \) is local in \( S \) if \( R(S, f) \cup W(S, f) = \{ \} \).

**Example 9.** From the results computed in Example 8, we have that the array type \( \text{aint} \) is not local in the scope \( S_2 \) corresponding to the inner loop, because \( R(S_2, \text{aint}) = W(S_2, \text{aint}) = \{ l_1[l_3], l_2[l_4] \} \), where \( l_1, l_2, l_3, l_4 \) stand for \( a, i, b, j \) respectively, i.e., the union contains more than one element. But we have that \( \text{next} \) and state are local in both \( S_1 \) and \( S_3 \). Consider again the small examples in Figure 3: we have that in \( \text{X} \), field \( f \) is local in \( S_1 \) because \( R(S_1, f) = \{ l_1 \} \) and \( W(S_1, f) = \emptyset \). However, it is not local in \( S_2 \) because \( R(S_2, f) \cup W(S_2, f) = \{ A_{\text{any}} \} \), as \( x.f \) could be \( z.f \) or \( y.f \). In \( \text{Y} \), we have that \( c \) is local in \( S_1 \) because \( R(S_1, c) \cup W(S_1, c) = \{ l_1 \} \), while as before it is not local in \( S_2 \) because \( R(S_2, c) \cup W(S_2, c) = \{ A_{\text{any}} \} \). In \( \text{Z} \), size is not local because \( R(S_1, \text{size}) \cup W(S_1, \text{size}) = \{ A_{\text{any}} \} \). Also, in \( \text{D} \), we have that \( R(S_1, r) \cup W(S_1, r) = \{ A_{\text{any}} \} \).

4.2. Locality Partition

In ideal scenarios in which fields are unconditionally local, we can transform each local field access in the considered scope into an equivalent access using a ghost variable which exposes the value of the field. However, there are cases in
which the read and write sets do not provide enough information for tracking the values stored in the corresponding locations. In such cases, it is often possible to provide preconditions under which tracking such locations is possible. When such conditions are used, any property for the locations that we infer (e.g., using static analysis) is sound only for inputs that satisfy the precondition.

**Definition 4 (aliasing preconditions).** An aliasing precondition \( \varphi \) is a Boolean formula \( \bigvee_i (\bigwedge_j c_{ij}) \) where each \( c_{ij} \) is an atomic aliasing proposition of the form \( A_1 \approx A_2 \) or \( A_1 \not\approx A_2 \).

The meaning of \( A_1 \approx A_2 \) (resp. \( A_1 \not\approx A_2 \)) is, as expected, that \( A_1 \) and \( A_2 \) alias (resp. do not alias). For simplicity in the notation, we will use the term aliasing for access paths that represent memory locations as well as integer values. Note that, by definition, some propositions are valid, e.g., \( l_1 \approx l_1 \), and some are unsatisfiable, e.g., \( l_1 \not\approx l_1 \). Moreover, for any access path \( A \) we let \( A \not\approx A \) and \( A \approx A \) be both false. This is because such accesses are not constant. When an aliasing precondition \( \varphi \) implies another precondition \( \varphi' \) we write \( \varphi \models \varphi' \). We will be mainly interested in implied atomic aliasing propositions, e.g., \( \varphi \models A_1 \approx A_2 \) and \( \varphi \models A_1 \not\approx A_2 \).

**Example 10.** In our running example, we cannot precisely track the write accesses to the arrays \( (a \text{ or } b) \) in \( S_2 \) because the memory location accessed depends on an aliasing condition: if \( a \) and \( b \) point to the same array, the content of such array may be modified using both accesses, \( a[i] \) or \( b[j] \). Furthermore, if \( i \approx j \), both accesses are modifying exactly the same element of the array. Thus, the trackability of array accesses and, as a consequence, the number of ghost variables needed to track them depends on some preconditions that are given in terms of the initial parameters. E.g., assuming the precondition \( l_1[l_3] \not\approx l_2[l_4] \) over the read/write sets in Example 8, we will need two different ghost variables to safely represent these array references, because they are pointing to different memory locations. However, if we assume that \( l_1[l_3] \approx l_2[l_4] \), all accesses point to the same array element, so we just need one ghost variable to track both array accesses. Our method will try all possibilities in order to find all preconditions for which termination of the program can be shown.

Note that every \( A \in R(S, f) \cup W(S, f) \) is associated to a set of program points in which the corresponding field/array access appear. The set of all such program points is denoted by \( \text{rwpps}(A) \).

**Definition 5 (locality partition).** Given a field or an array-type \( f \), a scope \( S \), and an aliasing precondition \( \varphi \), we say that a partition \( G_1, \ldots, G_n \) of \( R(S, f) \cup W(S, f) \) is a locality partition for \( f \) w.r.t. \( \varphi \) if:

1. \( \forall 1 \leq i \leq n. \forall A_1, A_2 \in G_i. \varphi \models A_1 \approx A_2 ; \text{ and} \)
2. \( \forall 1 \leq i < j \leq n. \forall A_1 \in G_i, \forall A_2 \in G_j. \varphi \models A_1 \not\approx A_2. \)
3. \( \forall 1 \leq i < j \leq n. \forall A_1 \in G_i, \forall A_2 \in G_j. \text{rwpps}(A_1) \cap \text{rwpps}(A_2) = \emptyset. \)

We denote the locality partition by \( P^f_\varphi \).
Let us explain the above definition: Condition (1) requires that the access paths in each $G_i$ alias, i.e., they all refer to the same memory location. Condition (2) requires that access paths from different partitions do not alias, i.e., they refer to different memory locations. The main idea is that now each component $G_i$ can be used to track the value stored in a corresponding memory location by means of a different ghost variable. Condition (3) requires that every (trackable) access in the program always refers to the same memory location. This condition is added for simplifying the presentation and it could be omitted if we use a polyvariant transformation \[5\] which clones the code for each calling pattern.

We say that the memory location induced by $G_i$ is trackable.

If $A_{\text{any}} \in R(S, f) \cup W(S, f)$ then there is no partition that satisfies the above definition. This is because in (1) we can take $A_1 = A_2 = A_{\text{any}}$ for which $A_1 \approx A_2$ does not hold. Observe that Definition 3 induces a locality partition w.r.t. the aliasing precondition true, i.e., the heap access is unconditionally local.

**Example 11.** Partitions can be built by considering all possible equalities and disequalities of the elements in $R(S, f) \cup W(S, f)$. Consider the read/write access sets in $S_2$ in Example 8, the following two locality partitions can be generated:

1. $G_1 = \{l_1[l_3], l_2[l_4]\}$ which gives us the precondition $\psi_1 = \{l_1[l_3] \approx l_2[l_4]\}$, or if we refer to the source code variables, then $\psi_1 = \{a[i] \approx b[j]\}$; (2) $G_1 = \{l_1[l_3]\}, G_2 = \{l_2[l_4]\}$ which gives us the precondition $\psi_2 = \{l_1[l_3] \neq l_2[l_4]\}$ and equivalently, using the source code variables, $\psi_2 = \{a[i] \neq b[j]\}$.

### 4.3. Automatic Transformation

In addition to identifying when memory locations are trackable w.r.t. a given precondition $\varphi$, we need to find a way to actually track them. Our approach is based on instrumenting the program with extra local (ghost) variables that expose the values of those locations to a heap-insensitive analysis as follows:

1. (1) for each trackable location induced by $G_i$, we introduce a ghost variable $g$; (2) when the content of the memory location is modified, we modify $g$ accordingly; and (3) when the memory location is read, we read the value from $g$. This approach has one clear advantage: there is no need to change existing static analysis tools to make them heap-sensitive, we simply apply them on the transformed program, and then the properties inferred for the ghost variables hold also for the corresponding memory locations. In the following we use $S^*$ to refer to the union of $S$ and all other scopes reachable from $S$.

**Definition 6 (locality transformation).** Given a scope $S$, and a corresponding locality partition $P_\varphi = \langle G_1, \ldots, G_n \rangle$. The instrumented program $T(P_\varphi)$ is obtained by transforming all rules of $S^*$ as follows:

1. Let $\bar{g} = \langle g_1, \ldots, g_n \rangle$ be $n$ different ghost variable names;
2. Every procedure call or rule head $p(\bar{x}, \bar{y})$ is replaced by $p(\bar{x} \cdot \bar{g}, \bar{y} \cdot \bar{g})$; and
3. For every $A \in G_i$, and $k:j \in \text{rwpps}(A)$, the field or array access at program point $k:j$ is replaced by $g_i$. 

20
Given \( k \) different fields \( f_1, \ldots, f_k \) with locality partitions \( P_{f_1}^{\phi_1}, \ldots, P_{f_k}^{\phi_k} \), we let \( T(P_{f_1}^{\phi_1}, \ldots, P_{f_k}^{\phi_k}) \) be the program obtained by applying the above steps on each \( P_{f_i}^{\phi_i} \), iteratively.

Let us explain the above transformation: in step (1) we define the ghost variables \( g_i \), where \( g_i \) will be used to track the content of the memory location induced by \( G_i \); in (2) we add the ghost variables as input and output arguments to all rules in \( \mathcal{S}^* \); and in (3) we simply replace accesses to the memory locations by accesses to the corresponding ghost variables. When several fields or arrays are going to be transformed, the instrumented program is obtained by applying the transformation on each corresponding locality partition iteratively. This is safe since \( f_i \neq f_k \), which guarantees that the different partitions refer to different memory locations.

**Example 12.** Using the preconditions and partitions of Example 11, we apply Definition 6 twice, once for each precondition and obtain the two versions depicted in the two columns of Figure 6, where \( \overline{\text{arg}} \) stands for \( a, b, i, j \). The one in the first column corresponds to the transformation for \( \psi_1 \), and has one ghost variable \( g_1 \), and the one in the second column corresponds the one for \( \psi_2 \) and has two ghost variables \( g_1 \) and \( g_2 \). Observe that the second version always terminates, while the first one might not because both array accesses, \( a[i] \) and \( b[j] \), modify the same array location. Therefore, using the transformed program, a heap-insensitive termination analyzer would infer that the while loop at hand terminates for the precondition \( \psi_2 \), i.e. \( \{ a \neq b \lor i \neq j \} \).

We say that a configuration \( C \) satisfies an aliasing precondition \( \varphi \), denoted by \( C \models \varphi \), iff for any \( A_1 \) and \( A_2 \) such that \( \varphi \models A_1 \approx A_2 \) (resp. \( \varphi \models A_1 \not\approx A_2 \)), it holds that \( \llbracket A_1 \rrbracket(C) = \llbracket A_2 \rrbracket(C) \) (resp. \( \llbracket A_1 \rrbracket(C) \neq \llbracket A_2 \rrbracket(C) \)). The following soundness theorem states that any reachable state in the original program, has a corresponding “equivalent” one in the transformed program. Given a ghost variable \( g_i \), we let \( A_{g_i} \) be the memory location that \( g_i \) tracks.
Theorem 2. Let $S$ be a scope with an entry $p$, $P_{\phi_1}^f, \ldots, P_{\phi_k}^f$ be locality partitions such that $f_i \neq f_j$, $C_0 = (\text{start}, p(\bar{x}, \bar{y}), tv_0); h_0$ such that $C_0 \models \phi_1 \wedge \ldots \wedge \phi_k$, and $C'_0 = (\text{start}, p(\bar{x} \cdot \bar{g}, \bar{y} \cdot \bar{g}), tv'_0); h_0$ such that $tv'_0$ extends $tv_0$ with $tv'(g_1) = [\mathbb{A}_g][C_0]$; If $C_0 \sim^n (g, bc_n, tv_n) \cdot ar; h_n$ is a possible execution using $S$, then $C'_0 \sim^n (g, bc_n, tv'_n) \cdot ar; h'_n$ is a possible execution using the corresponding $T(P_{\phi_1}^f, \ldots, P_{\phi_k}^f)$ and $tv_n(x) = tv'_n(x)$ for any $x \in \text{dom}(tv_n)$.

An interesting aspect of using locality partitions is that we can improve the accuracy of the unconditional heap-insensitive analysis even in cases for which programs terminate unconditionally. The reason is that considering the partitions separately gives us a kind of disjunctive reasoning that we lack in the basic locality transformation. Let us see an example.

Example 13. Consider the following method $p$ which receives as parameters two objects of type $A$. Assume that class $A$ has an integer field $f$.

\begin{verbatim}
void p(A x, A y) { 
  while (x.f > 0) {
    x.f--; 
    y.f--; 
  }
}
\end{verbatim}

This loop (and hence the method) always terminates. However, by applying the locality condition, we cannot prove it. The reason is that the read and write access paths for $f$ are $R(S, f) = W(S, f) = \{l_1, l_2\}$, where $S$ corresponds to the scope of the while loop. Thus, it cannot be transformed. Let us consider the locality partitions $P_{\phi_1}^f$ and $P_{\phi_2}^f$, where the first one consists of the unique set $\{l_1, l_2\}$ and $\phi_1 = \{l_1 \equiv l_2\}$, and the second one contains two sets $\{l_1\}, \{l_2\}$ and $\phi_2 = \{l_1 \not\equiv l_2\}$. The locality transformation for $P_{\phi_1}^f$ and $P_{\phi_2}^f$ gives respectively:

\begin{verbatim}
(void precondition $\phi_1$) (void precondition $\phi_2$)

| void p(A x, A y, int g) { | void p(A x, A y, int g_x, int g_y) { |
| while (g > 0) { | while (g_x > 0) { |
|     g--; |     g_x--; |
|     g--; |     g_y--; |
| } } |
| } |

Heap-insensitive termination analysis can prove termination for both methods.

4.4. Heuristics for References

In the above transformation, we track all possible heap accesses. However, it is also safe not to track all of them. This means that we can select some of the $G_i$ to be transformed. This is especially interesting when the heap accesses are reference fields or arrays of references. In these cases, it is usually a good heuristic to track the locations that are used for traversing the data structures.
(also called cursors), but do not track reference fields which are part of the data structure itself. This distinction can often be discovered because cursors are both read and updated, hence $G_i \cap R(S, f) \neq \emptyset$ and $G_i \cap W(S, f) \neq \emptyset$, while references that are part of the data structure are typically only read, i.e., $G_i \cap W(S, f) = \emptyset$.

**Example 14.** In order to prove termination of methods that use method `next` to traverse the list (e.g., method `m`), we need to infer that the “size of state” decreases every time we execute method `next`. We use the path-length abstraction \[30\] (i.e., the longest reachable path from the considered reference) as a size measure to check how the size of non-cyclic data structures is modified. By applying Definition 5 (or simply Definition 3), both reference fields `state` and `next` are local in the scope of the method unconditionally. Thus, it is possible to convert them into respective ghost variables $\alpha$ (for `state`) and $\beta$ (for `next`). The rule defining `next` of Example 7 would be transformed into:

\[
\text{next}(\langle \text{this}, \alpha, \beta, \delta, \eta \rangle, \langle \phi, \gamma, \zeta \rangle) \leftarrow \text{obj} : = \alpha, \delta : = \phi, \gamma : = \gamma, \zeta : = \zeta.
\]

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

\[
\text{next}(\langle \text{this}, \alpha, \beta \rangle, \langle \phi, \gamma \rangle) \leftarrow \text{obj} : = \alpha, \delta : = \text{obj}.\text{next}, \gamma : = \delta, \zeta : = \zeta.
\]

The path-length abstraction approximates now the effect of the instructions by \{\text{obj} = \alpha, \delta < \text{obj}, \gamma = \gamma, \zeta = \zeta\} which implies $\gamma < \alpha$. The important point is that, in the second constraint, when accessing a field of an acyclic data structure, the corresponding path-length decreases. This enables us to prove termination of loops that use `next` (like in `m`) by relying only on the field-insensitive version of path-length (note that \[30\] is not field-sensitive).

In summary, our approach can be used to prove termination of programs which use common patterns in OO languages such as iterators and enumerators by using a heuristics which tries to transform only those reference heap accesses which are used as cursors to the data structures. This is achieved by requiring that the field (or array) is both read and written in the scope.

### 5. Inference of Termination Preconditions

In this section, we describe our approach for inferring aliasing preconditions that, when they hold in the initial state, guarantee the termination of the program under consideration. Our approach is defined in two stages: (i) we first find preconditions that guarantee local termination, i.e., they guarantee that the loops defined in a given scope $S$ are terminating, ignoring the termination behavior of loops defined in scopes that are called from $S$; and (ii) in a second
step, we use the local preconditions in order to obtain conditions on global termination which guarantee that the loops of $S$ as well as those of scopes that are transitively called from $S$ are terminating. Note that if $S$ does not call any other scope, then local and global termination are equivalent for $S$.

5.1. Inference of Local Termination Preconditions

Given a scope $S$, the purpose of this section is to describe how to infer an aliasing precondition $\varphi$ which guarantees the local termination of $S$. Algorithm 1 outlines the steps to generate $\varphi$. At line 1, we choose the set of fields of interest $F_S = \{f_1, \ldots, f_k\}$; generate all possible locality partitions $P^*$ for $F_S$; $\varphi = \text{false}$; foreach $(P_{f_1}^{\varphi_1}, \ldots, P_{f_k}^{\varphi_k}) \in P^*$ do if $\text{Termin}$ returns true on $T(P_{f_1}^{\varphi_1}, \ldots, P_{f_k}^{\varphi_k})$ then $\varphi = \varphi \lor (\varphi_1 \land \cdots \land \varphi_k)$; end end return $\varphi$;

Algorithm 1: Inference of local termination preconditions for a scope $S$

Input: A scope $S$

Output: Local termination preconditions

1. Choose the set of fields of interest $F_S = \{f_1, \ldots, f_k\}$;
2. Generate all possible locality partitions $P^*$ for $F_S$;
3. $\varphi = \text{false}$;
4. foreach $(P_{f_1}^{\varphi_1}, \ldots, P_{f_k}^{\varphi_k}) \in P^*$ do
5. if $\text{Termin}$ returns true on $T(P_{f_1}^{\varphi_1}, \ldots, P_{f_k}^{\varphi_k})$ then
6. $\varphi = \varphi \lor (\varphi_1 \land \cdots \land \varphi_k)$;
7. end
8. end
9. return $\varphi$;

...
however, it needs to transform them in order to track the modifications that invoked scopes might perform on the sizes of data. The answer of \texttt{Termin}, as expected, is not definitive, it might be \textit{yes} or \textit{don’t-know}. For each locality partition of the involved fields (line 4), we run the local termination analyzer on the program resulting from applying the locality transformation in Definition 6 w.r.t. such locality partition. If \texttt{Termin} returns \textit{true}, according to Theorem 2, \( S \) locally terminates when the precondition holds in the input state. In line 6, we perform the disjunction of the current result with the preconditions obtained from the previous partitions such that the final result is the disjunction of all preconditions for which the program terminates.

**Example 15.** Let us apply Algorithm 1 on the scope \( S_2 \) (inner loop) of our running example. Step 1 gives us the type \( \texttt{aint} \). At line 2, the two partitions of Example 11 are generated. Thus, the \texttt{foreach} loop performs two iterations. When considering the locality partition \( P_{\text{aint}} \) where the precondition is \( \psi_2 = \{ l_1 \neq l_2 \lor l_3 \neq l_4 \} \), the transformed program of Figure 6 (right) is constructed and \texttt{Termin} returns \textit{true} in line 5. Hence, \( \varphi \) (initialized to \textit{false}) takes now the value \( \psi_2 \). In the next iteration, the locality partition \( P_{\text{aint}} \) is considered and the transformed program of Figure 6 (left) is constructed. In this case, \texttt{Termin} returns \textit{don’t-know} and hence \( \varphi \) remains with the value \( \psi_2 \) assigned in the previous iteration, which is returned as result in line 9.

### 5.2. Inference of Global Termination Preconditions

Let us first explain intuitively how Algorithm 2 infers global termination preconditions. Consider a scope \( S \) that includes a call \( b_{ij}^k = p(x, \bar{y}) \in S \) to a procedure \( p \) that is defined in a different scope \( S' \). Moreover, assume that \( S' \) does not call procedures that are defined in other scopes. In a first step, we have inferred local termination preconditions \( \varphi_1 \) and \( \varphi_2 \) for \( S \) and \( S' \), respectively, by means of Algorithm 1. In this step, we will combine \( \varphi_1 \) and \( \varphi_2 \) into global termination preconditions \( \psi_1 \) and \( \psi_2 \), respectively. For \( S' \), clearly we can take \( \psi_2 = \varphi_2 \), since it does not call any other scope. For \( S \), we seek a precondition \( \psi_1 \) such that \( \psi_1 \models \varphi_1 \); and \( \psi_2 \) holds whenever the execution reaches the call to \( p \). We take \( \psi_1 = \varphi_1 \land \psi_2' \), where \( \psi_2' \) is obtained from \( \psi_2 \) by replacing each \( l_i \) by \( \phi_{k,i}(x_i) \). Intuitively, \( \psi_2' \) is the global termination precondition of \( p \), but expressed in terms of the input to the entry of \( S \). This process is applied for all scopes in reverse topological order, i.e., starting from the one which does not
call any other scope, until the one that includes the main entry procedure.

<table>
<thead>
<tr>
<th>Input</th>
<th>Scopes $S_1, \ldots, S_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>Local and Global termination preconditions</td>
</tr>
</tbody>
</table>

```plaintext
for $i \leftarrow 1$ to $n$ do
  $\text{LC}[i] = \text{LocalCondTermin}(S_i)$;
  $\phi \leftarrow \text{LC}[i]$;
  foreach external call $b_j^i \equiv p(\bar{x}, \bar{y}) \in S_i$ where $p \in S_h$ do
    $\phi \leftarrow \phi \land \psi[l_1/l_1, \phi_{k_1}(x_1), \ldots, l_m/l_3 \phi_{k_3}(x_m)]$ where $\psi = \text{GC}[h]$;
  end
  $\text{GC}[i] \leftarrow \phi$;
end
```

Algorithm 2: Computing termination preconditions

Algorithm 2 outlines the main steps to generate global termination precondition for all scopes. The outer loop iterates over the scopes in reverse topological order and computes a global termination precondition for each scope $S_i$. At line 2, we compute a local termination condition $\text{LocalCondTermin}(S_i)$ for $S_i$ using Algorithm 1. This condition is also used at line 3 as its initial global termination precondition. Then, the inner loop traverses all calls to external scopes, for each such call, at line 5, it translates the global termination condition of the corresponding $S_h$ to be in terms of the input of $S_i$, and adds it to the global precondition of $S_i$. When the algorithm terminates, $\text{LC}[i]$ and $\text{GC}[i]$ will be, respectively, the local and global termination precondition for the scope $S_i$.

**Example 16.** Consider the analysis of the scope $S$ corresponding to method $m$ in Figure 2. Inferring global termination of $m$ requires the analysis of all scopes invoked in the method. Let us focus on $S_1$ (method next), $S_2$ (the inner loop) and $S_3$ (the outer loop). The application of Algorithm 1 on $S_1$ computes the precondition true (see Example 14). Algorithm 1 on $S_2$ computes the precondition $\phi = \{l_1 \not\approx l_2 \lor l_3 \not\approx l_4\}$ (see Example 15). For scope $S_3$, Algorithm 2 is able to prove termination (ignoring the inner loop) unconditionally by using a unique ghost variable for the field $\text{state}$ and considering the size relations inferred in Example 14. Once all local conditions for $S_1$, $S_2$ and $S_3$ have been computed, Algorithm 2 proceeds as follows: $\text{LC}[1]$ and $\text{LC}[2]$ are initialized to values true and $\phi$, respectively. As there are no external calls in $S_1$ nor in $S_2$, the foreach loop at line 4 is not executed for any of the scopes, and $\text{GC}[1] = \text{true}$ and $\text{GC}[2] = \phi$ (line 7). In the iteration on $S_3$, $\text{GC}[3]$ is initialized to $\phi' = \text{true}$. The interesting point is in the call to the inner loop, for which it is required a renaming according to the access path information $\phi$ stored at the program point just before the inner loop. According to Examples 7 and 14, we have $\{i \mapsto l_1/\text{state-data}, j \mapsto l_3/\text{state-data}\} \in \phi$. Line 5 of Algorithm 2 computes $\phi' = \phi[l_1/l_1, l_2/l_2, l_3/l_3/\text{state-data}, l_4/l_3/\text{state-data}]$, which results in $\phi' = \{l_1 \not\approx l_2 \lor l_3/\text{state-data} \not\approx l_3/\text{state-data}\} = \{l_1 \not\approx l_2\}$ as global precondition for $S_3$, and then for method $m$. 26
6. Experiments

We have implemented our technique in COSTA [8], a COSt and Termination Analyzer for Java bytecode. The implementation can be tried out at http://costa.ls.fi.upm.es by setting the enable_field_sensitive_analysis option in the manual configuration to conditional. Also, we have an option conditional_heuristic which, if set to references, applies the heuristics in Section 4.4 to treat reference data.

In order to assess the practicality of the approach on realistic programs, we have tried to infer the termination of all methods that contain heap accesses in their guards for all classes in the Apache Commons Project [26] libraries. These are libraries widely used for the development of industrial applications. The analysis is applied at “class” level, i.e., if the analyzed method invokes other methods defined in the same class, they are transitively analyzed, while those methods that belong to a different class are ignored by the analysis. The main reason for performing class level analysis is to avoid the analysis of the Java standard libraries. This is known to pose several difficulties for termination analysis which are outside the scope of this paper. Also, it should be noted that class level analysis gives us a better control on the results obtained. Basically, if we fail to analyze a method in one class, we do not propagate this failure to other methods that invoke it outside the class. Experiments have been performed on an Intel(R) Core(TM)2 Duo CPU P8700 2.53GHz with 4GB of RAM running Linux 3.2.0.

We have analyzed all methods (505 in total) from the Apache Commons Project libraries containing loops with heap accesses. We have analyzed each method using the three different approaches implemented in COSTA, (1) heap-insensitive analysis, that ignores the information stored in the heap, (2) unconditional heap-sensitive, and, (3) conditional heap-sensitive analysis (which generates locality partitions and composes the termination preconditions). We have run the analysis with a timeout for each method of 3, 6 and 7 minutes respectively for the heap-insensitive, unconditional heap-sensitive and conditional heap-sensitive analyses. Out of the 505 methods, 74 have exceeded the time out: 36 of them already exceeded the timeout in the heap-insensitive analysis, while the remaining ones exceeded it when adding the heap-sensitive flag (for both conditional and unconditional).

Table 1 summarizes our experimental results for the 431 methods which do not exceed the time out. They contain in total 564 loops with heap accesses in their guards. The first column shows the library name. Column $\#_M$ shows the number of methods, $\#_L$ the number of loops analyzed for each library and $\#_B$ the number of bytecode instructions for the analyzed code. Columns within Termination show the number of methods for which COSTA can guarantee termination for the three approaches, $T_I$ for heap-insensitive, $T_U$, for unconditional heap-sensitive and $T_C$ for conditional heap-sensitive analysis. Our results clearly show that the precision gained by using heap-sensitive analysis is quite significant, going from 8.6% of methods that can be proven terminating using heap-insensitive analysis to 71.9% for unconditional heap-sensitive analysis, and,
Table 1: Experimental evaluation on Apache libraries

<table>
<thead>
<tr>
<th>Library</th>
<th>Statistics</th>
<th>Termination</th>
<th>$\mathcal{O}$ Unconditional</th>
<th>$\mathcal{O}$ Conditional</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$M$</td>
<td>$B$</td>
<td>$L$</td>
<td>$T_A$</td>
</tr>
<tr>
<td>BCEL</td>
<td>59</td>
<td>4216</td>
<td>76</td>
<td>67.8</td>
</tr>
<tr>
<td>BeanUtils</td>
<td>3</td>
<td>467</td>
<td>4</td>
<td>3.5</td>
</tr>
<tr>
<td>Betwixt</td>
<td>2</td>
<td>44</td>
<td>2</td>
<td>0.3</td>
</tr>
<tr>
<td>Chain</td>
<td>1</td>
<td>377</td>
<td>1</td>
<td>1.2</td>
</tr>
<tr>
<td>Collections</td>
<td>25</td>
<td>1411</td>
<td>25</td>
<td>4.8</td>
</tr>
<tr>
<td>Compass</td>
<td>6</td>
<td>802</td>
<td>9</td>
<td>5.7</td>
</tr>
<tr>
<td>Configuration</td>
<td>11</td>
<td>985</td>
<td>11</td>
<td>6.3</td>
</tr>
<tr>
<td>DBCP</td>
<td>2</td>
<td>911</td>
<td>2</td>
<td>4.4</td>
</tr>
<tr>
<td>Digester</td>
<td>4</td>
<td>359</td>
<td>4</td>
<td>6.3</td>
</tr>
<tr>
<td>Discovery</td>
<td>2</td>
<td>68</td>
<td>2</td>
<td>0.1</td>
</tr>
<tr>
<td>EL</td>
<td>14</td>
<td>1190</td>
<td>18</td>
<td>11.0</td>
</tr>
<tr>
<td>IO</td>
<td>8</td>
<td>504</td>
<td>8</td>
<td>1.8</td>
</tr>
<tr>
<td>JCI</td>
<td>1</td>
<td>68</td>
<td>1</td>
<td>0.2</td>
</tr>
<tr>
<td>JCS</td>
<td>19</td>
<td>967</td>
<td>20</td>
<td>24.3</td>
</tr>
<tr>
<td>Jexl</td>
<td>14</td>
<td>976</td>
<td>14</td>
<td>5.4</td>
</tr>
<tr>
<td>Lang</td>
<td>14</td>
<td>1798</td>
<td>22</td>
<td>49.0</td>
</tr>
<tr>
<td>Math</td>
<td>192</td>
<td>23870</td>
<td>274</td>
<td>317.8</td>
</tr>
<tr>
<td>Net</td>
<td>3</td>
<td>187</td>
<td>4</td>
<td>1.2</td>
</tr>
<tr>
<td>Pool</td>
<td>21</td>
<td>1205</td>
<td>21</td>
<td>3.9</td>
</tr>
<tr>
<td>Sanselan</td>
<td>23</td>
<td>2872</td>
<td>38</td>
<td>117.7</td>
</tr>
<tr>
<td>SCXML</td>
<td>1</td>
<td>101</td>
<td>2</td>
<td>0.6</td>
</tr>
<tr>
<td>Transaction</td>
<td>1</td>
<td>38</td>
<td>1</td>
<td>0.2</td>
</tr>
<tr>
<td>Validator</td>
<td>5</td>
<td>368</td>
<td>5</td>
<td>2.8</td>
</tr>
<tr>
<td>Summary</td>
<td>431</td>
<td>43038</td>
<td>564</td>
<td>636</td>
</tr>
<tr>
<td>Percentages</td>
<td>8.6%</td>
<td>71.9%</td>
<td>81.4%</td>
<td>35.0%</td>
</tr>
</tbody>
</table>

to 81.4% for conditional heap-sensitive analysis. One remarkable issue is that all methods that can be proven terminating only using conditional heap-sensitive analysis do not require aliasing preconditions because all partitions generated terminate unconditionally. That means that even if these methods terminate unconditionally, they require conditional analysis because the basic locality is not enough to guarantee the soundness of the transformation (see Example 13).

We also evaluate the efficiency of the analysis. Column $T_A$ shows the time (in seconds) taken by the heap-insensitive analysis. Columns within $\mathcal{O}$ Unconditional show the overhead introduced by the unconditional heap-sensitive approach over the heap-insensitive analysis time. Columns $#_{1.5}$, $#_{2}$ and $#_{2+}$ show the number of methods with an overhead under 1.5, between 1.5 – 2 and greater than 2, respectively, over heap-insensitive analysis. We can observe that 70% (35 + 35) of the methods have an overhead under 2. This percentage is clearly acceptable for the gain of precision obtained by adding heap-sensitivity, from 8.6% to 71.9%. Conditional heap-sensitivity adds an additional overhead w.r.t. unconditional heap-sensitive approach. This can be seen in $\mathcal{O}$ Conditional, which shows the overhead introduced by the conditional approach w.r.t. the unconditional one. Columns $#_{1.25}$, $#_{1.5}$ and $#_{1.5+}$ show the number of methods that have an overhead under 1.25, between 1.25 – 1.5 and greater than 1.5.
respectively. Note that 68.4\% of the methods are analyzed with an overhead under 1.25, which is quite small, while we gain some further precision, namely from 71.9\% to 81.4\% (41 methods). Altogether, we argue that our results show that the accuracy gained by heap-sensitive analysis is very significant while the overhead is reasonable and, thus, our approach pays off in practice.

7. Comparison with other Termination Tools

The goal of this section is to study whether the technique described in this paper is significant not only for the conditional termination of a program but also for the unconditional heap-sensitive termination analysis of Java programs. For this purpose, we compare the unconditional heap-sensitive approach described in this paper, implemented as an extension of COSTA [8], with two state-of-the-art termination tools, AProVE [24] and Julia [30]. AProVE is a termination analysis tool which transforms a Java bytecode program into a term rewriting system, and Julia transforms it into a Constraint Logic Program (CLP), and then they study the termination of the transformed programs.

The comparison of the tools is performed by analyzing a program that contains a basic loop which terminates unconditionally and uses one heap access ($H$) in its loop condition, decrementing its value within the loop:

\[
\begin{align*}
\text{p} \{ \ldots \} & \{
\text{while } (H > 0) \{
\text{H --;}
\} \}
\end{align*}
\]

For performing the comparison, $H$ will be replaced by several forms of heap accesses, such as field accesses, array accesses and their combinations, and the resulting program is analyzed for each tool. Each time we analyze three different programs, (1) a program that only contains a method with the basic loop (method $p$), without an explicit initialization of the heap; (2) a main method that explicitly creates the heap before invoking the method with the loop; and, (3) a modification of the loop which instead of decrementing the heap value inside the loop, it calls another method ($q$) which decrements the value stored in the heap. Let us see an example of the programs obtained by instantiating $H$ with concrete code that accesses $x.f$:

\[
\begin{align*}
(1) & \quad p \{ \text{Obj x} \} \{
\text{while } (x.f > 0) \{
\text{x.f --;}
\} \}
\end{align*}
\]

\[
\begin{align*}
(2) & \quad \text{main} \{ \ldots \} \{
\text{Obj x = new Obj ();}
\text{x.f = 10;}
\text{p(x); //x.p2(x);} \}
\end{align*}
\]

\[
\begin{align*}
(3) & \quad p2 \{ \text{Obj x} \} \{
\text{while } (x.f > 0) \{
\text{q(x); x.f --;}
\} \}
\end{align*}
\]

In what follows, variables are named using this convention: $i, j$ represent integer variables; $a$ an array of integers; $x, y, z$ reference variables; and, $xs$ an array of
Table 2: Termination tools comparison

<table>
<thead>
<tr>
<th></th>
<th>(\mathcal{H})</th>
<th>COSTA</th>
<th>AProVE</th>
<th>Julia</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>x.f_i</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>(2)</td>
<td>x.f_y.f_i</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>(3)</td>
<td>x.f_y.f_z.f_i</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>(4)</td>
<td>a[i]</td>
<td>yes</td>
<td>yes</td>
<td>unk</td>
</tr>
<tr>
<td>(5)</td>
<td>f_a[i]</td>
<td>yes</td>
<td>yes</td>
<td>unk</td>
</tr>
<tr>
<td>(6)</td>
<td>a[x.f_i]</td>
<td>yes</td>
<td>yes</td>
<td>unk</td>
</tr>
<tr>
<td>(7)</td>
<td>x.f_a[x.f_i]</td>
<td>yes</td>
<td>yes</td>
<td>unk</td>
</tr>
<tr>
<td>(8)</td>
<td>x.s[i].f_i</td>
<td>yes</td>
<td>yes</td>
<td>unk</td>
</tr>
<tr>
<td>(9)</td>
<td>a[i][j]</td>
<td>unk</td>
<td>unk</td>
<td>unk</td>
</tr>
<tr>
<td>(10)</td>
<td>iterator</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

In the results displayed in Table 2, we can observe that, also for the unconditional case, our approach improves the capabilities of state-of-the-art termination analyzers. COSTA and AProVE behave similarly only when the analysis starts from a main method and thus it is possible to symbolically execute the instructions that involve heap accesses. However, it is very important to notice that modular analysis, and analysis of incomplete code, requires a scope-based approach like ours, where the analysis is able to analyze methods independently from their used context. On the other hand, the table shows that Julia cannot handle loops whose termination depends on data stored in arrays, even if the arrays have been explicitly initialized in a main method. Column \(T_p\) for the Julia analyzer is not evaluated because Julia requires a main method to analyze the program. Finally none of the studied tools can handle multidimensional arrays. Thus, given the obtained results, we argue that the precision achieved by using our unconditional heap-sensitive approach is significant and it allows analyzing programs that are not complete (such as library methods) and modular analysis.

8. Related Work

Traditionally, existing approaches to reason on shared mutable data structures either track all possible updates of heap-allocated data (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 heap-allocated updates but rather only those which behave like non heap-allocated variables.
As our experiments show, our approach is sufficiently precise for scope-based reasoning while introducing a reasonable overhead, as required in important applications, such as termination and resource analysis.

Miné’s [23] value analysis for C takes a different approach by enriching the abstract domain to make the analysis field-sensitive. The motivation here is different from ours, his analysis is developed to improve points-to analysis in the presence of pointer arithmetics. Similarly, [14] enriches a numeric abstract domain with alien expressions (field accesses). Without additional information, such as reference constancy analysis, this domain would be rather limited (imprecise) for bytecode.

We have applied our approach to the context of termination analysis. In this sense, our work continues and improves over the stream of work on termination analysis of object-oriented bytecode programs [4, 24, 2, 29, 27]. For numeric data, termination analyzers rely on a value analysis which approximates the value of numeric variables (e.g. [17]). Some field-sensitive value analyses have been developed over the last years (see the work of Miné mentioned above [23]).

For heap-allocated data structures, path-length [29] 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. However, the path-length abstract domain, and its corresponding abstract semantics, as defined in [29] 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 heap-allocated data. This article extends our previous work [4, 5] by handling array contents and proving termination conditionally, i.e., inferring aliasing preconditions under which termination can be proven.

As regards the reference constancy analysis, equivalent notions have been defined for other languages (see [1] and its references) and for different purposes. Our work adapts and extends such analyses to consider arrays and fields in a uniform way. Also, the analysis in this paper generalizes our previous reference constancy analyses [4] which infers information only on class fields, to consider arrays, integer variables and arithmetic expressions. There is also a lot of work devoted to inferring shape invariants of data-structures (see, e.g., [28], [18]). These techniques can, in principle, be used for inferring accurate reference constancy information, but at a much higher performance cost since they target more complex properties. These approaches are sometimes limited to some predefined data-structures such as linked lists.

There are termination analysis techniques [12] that rely on shape analysis in order to track the depth (i.e., the path-length) of data-structures to which variables points-to. This means that they would fail to observe a decrease in the depth of a data-structure pointed to by a field when such decrease does not imply a corresponding one at the level of a variable. For example, in the iterator example shown in Section 4.4, we have seen that given an object $x$ of type ListIter, it is necessary an analysis which is able to model the path-length of field $x$.state and not that of $x$. Namely, we need a heap-sensitive analysis.
based on path-length, which is one of our contributions in this article. Applying these techniques on our transformed programs, we expect them to infer the required information without any modification to their analyses. In order to track numerical values that are stored in fields, these approaches require further modifications [19].

Finally, conditional termination has been considered before in [13, 15]. In these works, the focus is in inferring termination conditions on the numerical variables, and not on the reference variables as we do.

9. Conclusions and Future Work

Heap sensitiveness is considered currently one of the main challenges in static analyses of object-oriented languages. We have presented a novel practical approach to heap-sensitive analysis which handles arrays and object fields (numeric and references) in a uniform way. The basic idea is to partition the program into fragments and track heap-allocated data by means of corresponding ghost variables which expose their values at each fragment, whenever such conversion is sound. If the conversion is not sound for any context, our technique can infer automatically aliasing conditions which guarantee termination if they hold in the initial state. The conditional transformation introduces a kind of disjunctive reasoning in our approach that allows us to improve the accuracy to prove termination of certain loops that indeed terminate unconditionally but the unconditional heap-sensitive approach cannot prove it.

In the future, we plan to extend our work in several directions. We want to consider a concurrent language and study how our analysis has to be adapted to produce sound and precise results in the presence of concurrency. A first step in this direction has been already taken in [3] for a language based on the concurrent objects paradigm [20]. We plan to extend our language with thread-based concurrency instead. Another objective is to improve the efficiency of our analyzer. For this purpose, we want to make field-sensitive analysis incremental in such a way that code can be analyzed during software development and, when the code is modified, the incremental analysis recomputes the least possible information. Incremental resource analysis [9] has been developed in the heap-insensitive context.

Acknowledgments

We are grateful to the anonymous referees for their comments and suggestions that have greatly improved the quality of the article. This work was funded in part by the Information & Communication Technologies program of the European Commission, Future and Emerging Technologies (FET), under the ICT-231620 HATS project, by the Spanish Ministry of Science and Innovation (MICINN) under the TIN2008-05624, TIN2012-38137 and PRI-AIBDE-2011-0900 projects, by UCM-BSCH-GR35/10-A-910502 grant and by the Madrid Regional Government under the S2009TIC-1465 PROMETIDOS-CM project.


Appendix A. Proofs

Appendix A.1. Proof of Theorem 1

Let us formally define \([\mathcal{A}](C)\), i.e., how to interpret the access path \(\mathcal{A} \neq \mathcal{A}_{\text{any}}\) in a configuration \(C = \langle q, bc, tv \rangle \cdot ar; h\):

\[
\begin{align*}
[\mathcal{A}_{\text{null}}](C) &= \text{null} \\
[n](C) &= n \\
[l\mathcal{A}, f_1, \ldots, f_n](C) &= [f_1, \ldots, f_n](C, tv(x_i)) \\
[l\mathcal{A}](C) &= [f_1, \ldots, f_n](C, h(tv(x_i)).v) \text{ where } v = [\mathcal{A}](C) \\
[f, F_1, \ldots, F_n](C, x) &= [F_1, \ldots, F_n](C, h(x).f) \\
[f\mathcal{A}, F_1, \ldots, F_n](C, x) &= [F_1, \ldots, F_n](C, h(h(x).f).v) \text{ where } v = [\mathcal{A}](C) \\
[\epsilon](C, x) &= x
\end{align*}
\]

The first two cases are straightforward. The third (resp. sixth) case evaluates \(l_i\) (resp. \(l_i[\mathcal{A}]\)), and then makes a recursive call to dereference (when \(n > 0\)) the corresponding object with \(f_1 \ldots f_n\), which is done iteratively by the last three cases.

Let us consider first the analysis of a standalone scope \(S\), i.e., a scope that does not call procedures from other scopes. Afterwards, we consider the modular case. Assume a given standalone scope \(S\) with an entry procedure \(p\), the analysis is applied iteratively using \(I^\#_{i+1} = I^\#_i \cup F^\#_{i+1}(I^\#_i)\), starting from an abstract state \(I^\#_0\) that maps every program point \(k\cdot j\) of \(S\) to \(\bot\), except for program point \(0\cdot 0\) which is mapped to the initial abstract state \(\langle \phi, \text{fields}(P) \cup \{\text{aint, aref}\} \rangle\) in which \(\phi(x_i) = l_i\); and \(\phi(y_i) = 0\) or \(\mathcal{A}_{\text{null}}\) depending on the type of \(y_i\).

Similarly to \(F^\#\), we define \(F(X) = \{C' : C \in X, C \rightsquigarrow C'\}, I_{i+1} = I_i \cup F(I_i)\), and \(I_0 = \{C_0\}\) where \(C_0 = \langle \text{start}, p(\bar{x}, \bar{y}), tv_0; h_0\rangle\). Each \(I_i\) represents that set of reachable states in at most \(i\) derivation steps, when starting from \(C_0\). For simplicity, we assume that each \(y_i\) in the initial state is either 0 or null, this does not restrict the correctness statements since these values are never used (they are overwritten upon exit).

In order to show that Theorem 1 is correct for the standalone case, it is enough to show that the following holds:

For any \(C_0 = \langle \text{start}, p(\bar{x}, \bar{y}), tv_0; h_0\rangle; h_0, \text{ if } C = \langle q, bc, tv \rangle \cdot ar; h \in I_i, \text{ and } bc\text{ corresponds to program point } k\cdot j\text{ (different from } 0\cdot 0), \text{ then } k\cdot j \rightarrow (\phi, \theta) \in I^\#_i\) such that

1. \(\phi(x) = \mathcal{A} \neq \mathcal{A}_{\text{any}} \Rightarrow tv(x) = [\mathcal{A}](C_0);\)
2. \(\text{if } f \in \theta, \text{ then:}\)
   2.1. \(\text{if } f \in \text{fields}(P), r \in \text{dom}(h_0), \text{ and } h(r)\text{ is an object that has a field with name } f, \text{ then } h_0(r).f = h(r).f;\)
   2.2. \(\text{if } f \in \{\text{aint, aref}\}, r \in \text{dom}(h_0), \text{ and } h_0(r)\text{ is an array of static type } f, \text{ then the elements of } h_0(r)\text{ and } h(r)\text{ have the same values};\)
We say that \(\langle \phi, \theta \rangle\) correctly approximates \(C\). Note that our main interest is in showing that (1) holds, (2) is an auxiliary statements required for doing this.

We prove the above claim, for an arbitrary \(C_0\), by induction on the number of iterations when computing \(I_i\) and \(I^{\#}_i\) (i.e., on \(i\)). The claim trivially holds for \(i = 0\). We assume it holds for \(I_i\) and \(I^{\#}_i\), and now we show that it holds for \(I_{i+1}\) and \(I^{\#}_{i+1}\).

Pick an arbitrary \(C' = \langle q', bc', tv'\rangle \cdot ar'; h' \in I_{i+1}\), and let \(bc'\) correspond to program point \(k:j\). Assume \(C' \notin I_i\), otherwise the claim holds by the induction hypothesis. Thus \(C' \in F(I_i)\), i.e., we have \(C = \langle q, bc, tv\rangle \cdot ar; h \in I_i\) such that \(C \sim C'\). We prove that the claim holds for such case by considering all possible ways to move from \(C\) to \(C'\). In particular we prove that there is \(\langle \phi, \theta \rangle\) in \(F(I^{\#}_{i+1})\), associated to the program point of \(bc'\), that correctly approximates \(C'\).

First note that, by the induction hypothesis, there is \(k:j \mapsto \langle \phi, \theta \rangle\in I^{\#}_i\) that correctly approximates \(C\), i.e., satisfies conditions (1) and (2) for \(C\). In what follows, when we refer to \(\langle \phi, \theta \rangle\), we mean exactly this one.

Let us start by considering the simple instructions, i.e., those that do not correspond to calling (or returning from) a procedure. For such cases, using \(\langle \phi, \theta \rangle\) and \(F^{\#}_i\), we compute \(k:j+1 \mapsto \langle \phi', \theta' \rangle\). We claim that \(\langle \phi', \theta' \rangle\) correctly approximates \(C'\).

Case 1: \(x:=y.f\). We have \(\theta' = \theta\), and \(\phi'\) is different from \(\phi\) only by the value of \(x\). Trivially, Condition (2) holds since the heap is not modified, and Condition (1) holds for any variable which is different from \(x\). We show that Condition (1) holds also for \(x\). If \(\phi'(x) = A_{any}\) then Condition (1) trivially holds for \(x\). The other possibility is that \(\phi'(x) = A.f\), in which case \(tv(y) = tv'(y) = A \neq A_{any}\) and \(f \in \theta\), then we have \(tv'(x) = tv(y).f = [A](C_0).f = [A.f](C_0)\).

Case 2: \(x.f:=y\). Clearly Condition (1) holds since no variable is updated. Also Condition (2) holds since the only field that has been modified is removed from the set \(\theta\).

Case 3: \(x:=n\). Clearly Conditions (1) holds for any variable different from \(x\), and it trivially holds for \(x\) since its access path is the number \(n\). Condition (2) also holds since these instructions does not modify the heap.

Case 4: \(x:=\text{null}\). Straightforward using the same reasoning as in Case 3.

Case 5: \(x:=y\). Clearly (1) holds for any variable different from \(x\). It holds also for \(x\) because \(tv'(x) = tv(y) = [\phi(y)](C_0) = [\phi(x)](C_0)\).

Case 6: \(x:=y\ aop\ z\). Clearly Condition (1) holds for any variable different from \(x\). If \(\phi'(x) = A_{any}\) then it trivially holds for \(x\) also. If \(\phi'(x) \neq A_{any}\), then \(\phi(y)\) and \(\phi(z)\) must be numbers, and thus \(tv'(x) = tv(y) + tv(z) = [\phi(x)](C_0) + [\phi(z)](C_0) = [\phi'(x)](C_0)\).

Case 7: \(x:=\text{newarray}(D, y)\). Clearly Condition (1) holds for any variable different from \(x\). It holds also for \(x\) since \(\phi'(x) = A_{any}\). Condition (2) holds since no field or array-type is modified.
Case 8: \( x := y[z] \). Analogue to Case 1.

Case 9: \( x[y] := z \). Analogue to Case 2.

Case 10: \( x := \text{new } C \). Like Case 7.

Case 11: \( x := \text{arraylength}(y) \). Clearly Condition (1) holds for any variable different from \( x \). It holds also for \( x \) since \( \phi'(x) = A_{\text{any}} \). Condition (2) holds since no field or array-type is modified.

Now we consider the case of a procedure call. For this, we assume that \( C \) is of the form \( \langle q, q' \cdot (\bar{w}, z) \cdot bc', tv') \cdot ar \cdot h \rangle \), and we make an execution step using a corresponding rule \( q'(\bar{x}, y) \rightarrow g, b'_k, \ldots, b'_1 \in P \). Thus, according to the language’s semantics we get \( C' = \langle q', b'_k, \ldots, b'_1, tv' \rangle \cdot q, q[y/\bar{z}] \cdot bc', tv') \cdot ar \cdot h \) where \( tv' \) maps every variable to either 0 or null, except for the formal parameters \( \bar{x} \) whose values are obtained from the actual ones \( \bar{w} \). Note that \( C' \) corresponds to program point \( k' \cdot 1 \). In the abstract setting, using \( \langle \phi, \theta \rangle \), rule \( F^\#_2 \) we compute \( k' \cdot 1 \rightarrow \langle \phi', \theta' \rangle \). We claim that \( \langle \phi', \theta' \rangle \) correctly approximates \( C' \). This is because Condition (2) holds since \( \theta' = \theta \) and the derivation step does not modify any field or array; and Condition (1) also holds since all we do is to copy the values of the access paths of actual parameters from those of the formal parameters, the rest of local variables are initialized to 0 or \( A_{\text{null}} \) for which Condition (1) trivially holds.

Now we consider the case of a return from a procedure. For this we assume that \( C \) is of the form \( \langle q, q' \cdot (\bar{w}, z) \cdot bc', tv') \cdot ar \cdot h \rangle \), which according to the language’s semantics can, in a single step, only lead to \( C' = \langle q', bc', tv' \rangle \cdot ar \cdot h \) where \( tv' \) maps all variables as in \( tv'' \), except for the output variables \( \bar{z} \) for which we have \( tv'(z_i) = tv(y_i) \). Now, note that there must be an abstract state \( C'' \) of the form \( \langle q', q(\bar{w}, z) \cdot bc', tv'' \rangle \cdot ar \cdot h' \) in \( I_i \) from which we have (transitively) obtained \( C \). Assume that \( C'' \) corresponds to program point \( k' \cdot j \). Thus, \( C'' \) corresponds to program point \( k' \cdot j + 1 \). By the induction hypothesis, \( I''_j \) must include \( k' \cdot j \rightarrow \langle \phi'', \theta'' \rangle \) that correctly approximates \( C'' \). It is easy to see that, using \( F^\#_j \), together with \( \langle \phi, \theta \rangle \) and \( \langle \phi'', \theta'' \rangle \), we get \( k' \cdot j + 1 \rightarrow \langle \phi', \theta' \rangle \) that correctly approximates \( C' \).

We have proved that whenever \( F(I_i) \) introduces a state \( C' \), then \( F^\#(I''_j) \) introduces an abstract state \( \langle \phi', \theta' \rangle \) that correctly approximates \( C' \). Merging \( F^\#(I''_j) \) with \( I''_j \) using \( \cup_p \) keeps this approximation correct since the only changes that can happen are: the access path of a variable \( x \) is upgraded to \( A_{\text{any}} \), or a field or array-type \( f \) is removed from \( \theta \). In both cases, respectively, Conditions (1) and (2) still hold.

This concludes the proof for the stand alone case, and next we consider the modular case.

Let us first note the following immediate consequence of Theorem 1. Assume that we have analyzed a scope \( S \) (that does not call any other scope), and that
we have obtained the following summary for procedure $p$:

$$\langle \phi_p, \theta_p \rangle = \bigcup_s \{ \langle \phi, \theta \rangle \mid p(x, y) \leftarrow g, b_k \_i, \ldots, b_k \_b \in P, k := 1 \mapsto \langle \phi, \theta \rangle \}$$

Now let $C \sim C'$ where $C = \langle q, p(w, z) \cdot bc, tv \cdot ar; h, C' = \langle q, bc, tv' \cdot ar; h' \rangle$, and $q$ (the procedure currently executed in $C$) is not in the same scope of $p$, i.e., $p$ is an external call. Then, clearly, if $\phi_p(y_i) = A \neq A_{\text{any}}$, we have $tv'(z_i) = [A](C)$.

Now let us change the definition of $F(X)$ in order to collect the reachable states that correspond to program points in a given scope only. This can be done by changing $F$ such that when $C = \langle q, p(w, z) \cdot bc, tv \cdot ar; h \rangle$ and $p$ is external, then instead of making a single derivation step we use $C \sim C'$ where $C' = \langle q, bc, tv' \cdot ar; h' \rangle$, i.e., we execute $p$ completely.

Now we claim that Theorem 1 still holds when applied to a given scope $S$. The proof is the same as for the standalone scope, however, we should extend it to handle calls to external procedures.

Assuming the $C$ corresponds to program point $k; j$, and that $k; j \mapsto \langle \phi, \theta \rangle \in I^\#_\theta$ correctly approximates $C$, we show that $\langle \phi', \theta' \rangle = \tau(p(w, z), \langle \phi, \theta \rangle)$ correctly approximates $C'$. Clearly Condition (2) holds since $\theta' = \theta \cap \theta_p$. Condition (1) clearly holds for any variable not in $z$. Next, we prove that it holds also for those variables in $z$.

Applying the definition of $\tau$ for external calls, we get that each $\phi'(z_j)$ equals to $\text{ren}(\phi_p(y_i), \phi, \theta)$. According to the definition of $\text{ren}$, if $\phi_p(y_i)$ includes a field or array-array type $\phi \notin \theta$ we get $\phi'(z_i) = A_{\text{any}}$, so for such case Condition (1) trivially holds. Assume that there is no such $f$, i.e., we are in the else branch of $\text{ren}$. In such case, if $\phi_p(y_i) = A_{\text{any}}$, or it includes $l_j$ such that $\phi(w_j)$ is $A_{\text{any}}$, then $\phi'(z_i) = A_{\text{any}}$, so for this case too Condition (1) trivially holds. Now, assume this is not the case, then $\phi'(z_i) = A'$ where $A'$ is obtained from $\phi_p(y_i)$ by replacing each $l_j$ by $\phi(w_n)$. We prove that $tv(z_i) = [A'](C_0)$: As we have commented above (at the beginning of the proof of the modular analysis) we have $tv(z_i) = [\phi_p(y_i)](C)$. Now when evaluating $[\phi_p(y_i)](C)$, we reach base-cases in which we need to evaluate $[l_j](C)$, which is equal to $tv(x_i)$, and by the induction hypothesis it is equal to $[\phi(z)](C_0)$. Thus, $[\phi_p(y_i)](C)$ is equal to $[A'](C_0)$.

**Appendix A.2. Proof of Theorem 2**

The correctness of this Theorem is straightforward given (1) the correctness of the access path analysis; and (2) the definition of locality partitions. We explain this below.

Let us first assume a single locality partition $P_{f_1}^{\varphi_1} = \langle G_1, \ldots, G_n \rangle$. Being a locality partition, as in Definition 5, in any execution that starts from a state $C_0$, in which $\varphi_1$ holds, the following is guaranteed:

1. The access paths in $G_1, \ldots, G_n$ refer to different heap locations;
2. We can identify all statically heap accesses instructions in the program (either read or write) that refer to each heap location induced by each $G_i$.
3. A heap access instruction that accesses the heap location induced by $G_i$, will always access this specific location during the execution. It is not possible that it accesses a different location. This is guaranteed by Condition 3 of Definition 5.

The above points basically state that each heap location induced by $G_i$ can be tracked, i.e., we can statically identify all heap access instructions in the program that read or modify it. This in turn means that we can simulate this specific location with a global variable $g_i$ as follows:

1. We set the initial value of $g_i$ to the value of the corresponding location in $C_0$; and
2. Any instruction that refers to that heap location is changed to refer to $g_i$.

Moreover, since every heap access in the program is associated, if any, with a single $G_i$, we can do the above transformation simultaneously for $\langle G_1, \ldots, G_n \rangle$. This is exactly what we do in Definition 6. The variables $\langle g_1, \ldots, g_n \rangle$ are basically global variables since the are added as input and output to all rules.

Given the above, the case of several locality partitions $P_{f_1}^{\tau_1}, \ldots, P_{f_k}^{\tau_k}$ becomes straightforward. The fact that each $f_1, \ldots, f_n$ refers to different array (static) type or field signature guarantees that they cannot refer to a common heap location. Thus, the above transformation can be safely applied simultaneously for all partitions.