Escape Analysis for Java™. Theory and Practice

BRUNO BLANCHET
Département d'informatique, École Normale Supérieure
Max-Planck Institut für Informatik

Escape analysis is a static analysis that determines whether the lifetime of data may exceed its static scope.

This paper first presents the design and correctness proof of an escape analysis for Java™. This analysis is interprocedural, context sensitive, and as flow sensitive as the static single assignment form. So, assignments to object fields are analyzed in a flow-insensitive manner. Since Java is an imperative language, the effect of assignments must be precisely determined. This goal is achieved thanks to our technique using two interdependent analyses, one forward, one backward. We introduce a new method to prove the correctness of this analysis, using aliases as an intermediate step. We use integers to represent the escaping parts of values, which leads to a fast and precise analysis.

Our implementation [Blanchet 1999], which applies to the whole Java language, is then presented. Escape analysis is applied to stack allocation and synchronization elimination. In our benchmarks, we stack allocate 13% to 95% of data, eliminate more than 20% of synchronizations on most programs (94% and 99% on two examples) and get up to 43% runtime decrease (21% on average). Our detailed experimental study on large programs shows that the improvement comes more from the decrease of the garbage collection and allocation times than from improvements on data locality, contrary to what happened for ML. This comes from the difference in the garbage collectors.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Optimization*; D.4.1 [**Operating Systems**]: Process Management—*Synchronization*; *Threads*; D.4.2 [**Operating Systems**]: Storage Management—*Allocation/deallocation strategies*

General Terms: Languages, Performance, Theory

Additional Key Words and Phrases: Java, optimization, stack allocation, static analysis, synchronization elimination

1. INTRODUCTION

Many languages use a garbage collector (GC) to make memory management easier for the programmer. A GC is even necessary for the Java programming language, since Java has been designed to be safe, so it cannot rely on the programmer to correctly deallocate objects. However, garbage collection is time consuming, especially with a mark and sweep collector as in the JDK until version 1.2. Therefore stack allocation may be an interesting alternative. However, it is only possible to stack allocate data if its lifetime does not exceed its static scope. The goal of escape

Author's address: Bruno Blanchet, Département d'Informatique, Ecole Normale Supérieure, 45 rue d'Ulm, 75230 Paris Cedex 05, France. E-mail: `Bruno.Blanchet@ens.fr`.
This work was partly done while the author was at INRIA Rocquencourt.

analysis is precisely to determine which objects can be stack allocated.

Our escape analysis is an abstract interpretation-based analysis [Cousot and Cousot 1977; 1979], which we have already applied to functional languages [Blanchet 1998]. Java has specific features, that make the analysis completely different from the version for functional languages:

(1) Java is an imperative language, so it makes an intensive use of assignments, which must therefore be precisely analyzed. This complicates the design of the analysis considerably: we have to use a bidirectional propagation (we have two analyses, one forward and one backward, which depend on each other). This also leads to an interesting proof technique: we use aliases as an intermediate step and iterate over the number of aliases composed transitively to prove the correctness. Such technique was not used in the case of functional languages, and is novel as far as we know.
(2) Java is an object-oriented language, so subtyping must be taken into account in the representation of escape information, which is computed from the types.

We not only wanted to have a precise analysis but also a fast analysis. That is why we use integers to represent the escaping part of objects. This representation is less costly than graphs for example. Our benchmarks demonstrate that it gives a very good compromise between precision and efficiency.

Escape analysis has two applications: an object o that does not escape from method m (i.e., whose lifetime is limited to the execution of m) can be stack allocated in m. Moreover, when our analysis claims that o does not escape from m, o is also local to the current thread, so we need not do synchronizations when calling a synchronized method on object o. This second optimization is important, because synchronization is a costly operation in the JDK. Moreover, there are many synchronizations even in single-threaded programs since libraries use synchronization to ensure thread-safety in all cases. Synchronization elimination could also be applied to other multithreaded languages.

Our implementation was presented, in [Blanchet 1999], with benchmark results but no correctness proof. This paper complements [Blanchet 1999] by explaining the design and correctness proof of our analysis and of the program transformations, while giving the necessary information on our implementation and benchmarks.

1.1 Related Work

1.1.1 *Escape analysis and compile-time garbage collection for functional languages.* This paper is in the line of previous work on escape analysis for functional languages [Park and Goldberg 1992; Deutsch 1997; Blanchet 1998] that represent the escaping part of data by integers. Park and Goldberg [1992] consider lists as the only data type, and do not handle assignments, [Deutsch 1997; Blanchet 1998] consider that a value escapes as soon as it is stored in another value, which is enough in the case of functional languages. Deutsch has greatly improved the complexity of Park and Goldberg's analysis, reducing it to $\mathcal{O}(n \log^2 n)$, with exactly the same results for first-order expressions (there is an unavoidable loss of precision in the higher-order case). Our escape analysis for Java handles assignments much more precisely than those analyses for functional languages, but still remains very fast: its complexity is $\mathcal{O}(np^2n_i)$, where n is the size of the static single assignment form of the program, p the maximum number of parameters of a method, and the number of iterations n_i is very small in practice (at most 17 and on average 3.9 in our benchmarks; see Section 11 for more details).

Hughes [1992] introduces integer levels to represent the escaping part of data. He does not perform stack allocation, but explicit deallocation, in a purely functional language. The work closest to Hughes's is [Inoue et al. 1988] in which only the top of lists is explicitly deallocated, but experimental results are given.

Mohnen [1995a; 1995b] describes an escape analysis for a functional language with recursive types, but the analyzed language does not contain imperative operations. His analysis is more precise but also has a higher worst-case complexity than our integer escape analyses. (The complexity of his analysis is $\mathcal{O}(D^3p^2Fn)$ where D is the maximum size of the graph of a type, p the maximum number of parameters of a function, F the number of functions, n the size of the program. In particular, D can be large in practice, reaching several thousands in our experiments.)

Alias analysis [Deutsch 1994], reference counting [Hederman 1988; Hudak 1986], storage use analysis [Serrano and Feeley 1996], which is similar to [Jones and Muchnick 1982; Shivers 1988; Harrison 1989; Deutsch 1990] can be applied to stack allocation though at a much higher worst-case complexity. (For example, the storage use analysis [Serrano and Feeley 1996] is in $\mathcal{O}(n^4)$. Moreover, escape analysis is context-sensitive, whereas making storage use analysis context-sensitive increases again its complexity.)

Another allocation optimization has been suggested in [Tofte and Talpin 1993; Aiken et al. 1995; Birkedal et al. 1996]: region allocation. All objects are allocated in heap regions whose size is not known statically in general, but for which we know when they can be deallocated. Regions can therefore be deallocated without GC. This analysis solves a more general problem than ours, but at the cost of an increased worst-case complexity. In fact, on many programs, opportunities for stack allocation outnumber opportunities for region allocation, as noticed in [Birkedal et al. 1996]. Our analysis could be extended to region allocation.

1.1.2 *Escape analysis and similar analyses for stack allocation and synchronization elimination in Java.* McDowell [1998] gives experimental evidence that stack allocation in Java programs would be interesting, but does not actually allocate data on the stack.

Gay and Steensgaard [2000] apply an escape analysis to stack allocation in Java, but consider that an object escapes as soon as it is stored in another object. This is a very simple and fast analysis, but it lacks precision.

Other escape analyses for Java are more precise but also much more computation intensive. Ruggieri and Murtagh [1988] give an analysis algorithm that can be applied to stack allocation in object oriented languages, but it is much more complex than ours (at least $\mathcal{O}(n^4)$), and no implementation is mentioned.

Bogda and Hölzle [1999] use escape analysis for synchronization elimination. This analysis is more precise than ours in some cases since it can distinguish different fields of objects. It is however less precise in other cases: it cannot detect objects that are transitively reachable only from local objects if two or more dereferences are needed to reach the object.

Aldrich et al. [1999] eliminate synchronization on thread local objects with an analysis based on a preliminary alias analysis. However, their results are limited by the lack of specialization. They also eliminate synchronizations when a monitor is always protected from concurrent access by another monitor. We do not investigate this approach here, since eliminating synchronizations on thread local objects already deals with most cases.

Other escape analyses are based on graphs: Whaley and Rinard [1999] use points-to escape graphs, and Choi et al. [1999] use connection graphs. Connection graphs are similar to alias graphs and points-to graphs, but the connection graph of a method can be easier summarized and reused at each call site, to avoid recomputing the escape information when the method is called in different escape contexts. These analyses are applied to stack allocation and synchronization elimination. They are more precise than our analysis because they distinguish different fields of objects and are flow sensitive. The integer manipulations of our analysis should be less costly in time and memory than graph manipulations and our analysis appears to be precise enough on our benchmarks. Choi et al. [1999] consider a `new` statement as stack allocatable only when it is so in all calling contexts of the method. Therefore, inlining cannot be used to increase stack allocation opportunities.

Ruf [2000] eliminates synchronizations not only on objects unreachable from static fields, but also on objects reachable from static fields accessed by only one thread. This idea could also be applied to our escape analysis. Even if Ruf's analysis seems reasonably fast in practice, its worst-case complexity is exponential.

1.1.3 *Other ways of reducing the cost of synchronization.* Other techniques have been investigated to reduce the cost of synchronization not by eliminating synchronizations, but by designing a faster synchronization primitive [Bacon et al. 1998; Agesen et al. 1999; Onodera and Kawachiya 1999]. Both approaches could be combined to reduce even more the synchronization cost.

Diniz and Rinard [1998] reduce the cost of synchronization by coarsening the granularity at which the computation acquires and releases locks. First, they allow objects to share locks. Second, when a lock is repeatedly acquired and released in a region of code, they move synchronization constructs outside this region. However, this can increase the contention on locks. They reduce the cost of synchronization for objects shared by several threads, while our goal is to eliminate synchronization on thread local objects.

All synchronizations can be eliminated in single threaded programs by simply detecting that no second thread is created [Muller et al. 1997; Bacon 1997; Fitzgerald et al. 2000]. However, such an approach cannot optimize multithreaded programs, which considerably reduces its scope.

1.2 Overview

After an informal presentation in Section 2, Sections 3 to 7 explain the design of our escape analysis and its correctness proof. The proof handles a representative subset of the Java language, including subtyping and virtual calls. The main originality of our analysis is that it uses a bidirectional propagation (there are two analyses, one forward and one backward), to deal with assignments. Most proofs have been put in the appendix because of their length. Section 8 discusses extensions to other features of Java.

```
class BinTree {
  BinTree left, right; Object elem;

  BinTree(BinTree l, BinTree r,
          Object e) {
    left = l; right = r; elem = e;
  }
  static BinTree leaf(Object o) {
    return new BinTree(null, null, o);
  }
  static Object run() {
    BinTree local = new BinTree(null,
      leaf(new Integer(1)),
      new Integer(2));
    return local.elem;
  }
}
```

Stack allocation

```
  void setData(int l) {
    a = new int[l];
    Random r = new Random();
    for(int i = 0; i < a.length; i++) {
        a[i] = r.next(32);
    }
  }

class Random {
  ...
  synchronized int next(int bits) {
    /* Integer computations */
    ...
    return result;
  }
}
```

Synchronization elimination

Fig. 1. Examples.

Section 9 describes our implementation of this analysis for the full Java language, in the turboJ [Weiss et al. 1998] Java-to-C compiler from the Silicomp Research Institute (see `http://www.ri.silicomp.com`). Section 10 details program transformation features: stack allocation with extensions to reuse the stack allocated space in loops, and inline methods to increase stack allocation opportunities; synchronization elimination, with a global synchronization analysis.

Section 12 gives a summary of our experimental results. For our benchmarks, we get up to 43% runtime decrease (21% on average) and our analysis can be applied to the largest applications thanks to its very good measured efficiency.

1.3 Notations

The set of integers $\{a, ..., b\}$ is denoted by $[a, b]$. The set of lists whose elements are in S is S `list`. The empty list is []. The list l at the head of which p_1 has been concatenated is $p_1 : l$. The set of words on alphabet S is S^*. The function that maps x to $f(x)$ is $\{x \mapsto f(x)\}$. The extension of f that maps x to y is $f[x \mapsto y]$; if f was already defined at x, the new value replaces the old one. In a lattice, the join is $\sqcup$ and the meet is $\sqcap$.

2. INFORMAL PRESENTATION OF ESCAPE ANALYSIS

Our escape analysis has two goals: stack allocation and synchronization elimination. An object o can be stack allocated in method m if and only if it is not reachable after the end of m. This is true in particular if it is not reachable from the parameters and the result of m, nor from static fields.

We will eliminate synchronizations on objects that are reachable from a single thread, that is, objects that are not reachable from static fields or from `Thread` objects.

Example 2.1. Consider for instance the method `BinTree.run()` of Figure 1. It creates a binary tree in the variable `local`, and returns the contents of a node of the tree. The object `local.elem = new Integer(2)` is reachable from the result

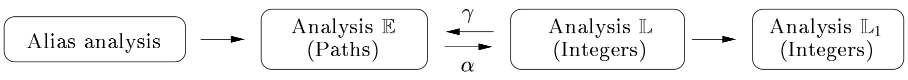


Fig. 2. Analysis hierarchy

of `run()`, so it cannot be stack allocated in `run()`. On the contrary, the `BinTree` node `local` can be stack allocated, because it is not reachable after the end of `run()`.

The method `setData` (simplified from the Symantec benchmark) fills an array with random numbers. In this method, `r` is not reachable from static fields or `Thread` objects (in fact, it can also be stack allocated), `r` is therefore local to each thread that calls `setData`. The call `r.next` is synchronized. This synchronization can therefore be eliminated (we create a copy of `next` without synchronization).

In both cases, what we need is in fact a kind of alias information: is an object aliased with the contents of static fields, or of parameters or result of a method ? It is therefore natural to use an alias analysis for these purposes. However, we are in a particular case: the question is "is object o aliased with a *given* set of objects ?". We can use this fact to avoid performing a full and costly alias analysis: alias analysis will only be an intermediate step in the design of our escape analysis.

We start therefore from an alias analysis (Section 4), then we define a very precise escape analysis $\mathbb{E}$ (Section 5) that represents escaping parts of objects by sets of access paths. Analysis $\mathbb{E}$ introduces an important idea of our escape analysis: it uses a bidirectional propagation to analyze precisely assignments. However, it is still too complex to be implemented directly, so it is abstracted to give analyses $\mathbb{L}$ (Section 6) and $\mathbb{L}_1$ (Section 7) that represent escaping parts of objects by integers (Figure 2). This is the key to obtain a fast analysis. Analysis $\mathbb{L}_1$ is slightly more approximate than $\mathbb{L}$, to have a faster implementation.

3. SYNTAX OF THE ANALYZED LANGUAGE

Our analysis applies to the Java bytecode. However, for clarity, we will represent the bytecodes by instructions listed in Figure 3, which correspond to a small but representative subset of the Java bytecodes. The syntax of these instructions is similar to the Java source code.

We assume that the analyzed program is in Static Single Assignment (SSA) form [Cytron et al. 1991]. In this form, there is a single assignment statement for each local variable of the methods. When the control-flow merges, the values of two different variables may have to be merged into a single one by a ϕ-function. The meaning of $v_1 = \phi(v_2, v_3)$ is equivalent to $v_1 = v_2$ if v_2 has been assigned after v_3 in the execution trace or v_3 has never been assigned. Otherwise, $v_1 = \phi(v_2, v_3)$ is equivalent to $v_1 = v_3$.

We assume that all methods yield a result. For simplicity, we do not consider `jsr`, `ret`, and exceptions in the theoretical study, since they raise specific problems, with no connection with escape analysis. We summarize how our implementation handles these features in Section 8. When there is no ambiguity, fields and methods will simply be designated by their name, without indicating their class and type. The type of arrays whose elements are of type t is denoted by `t[]`.

Address	Address in the method ($[0, 65535]$)	$V = V$	$V = \phi(V, V)$
$v \in V$	Name of a local variable	$V = (\mathit{RefType})V$	
$C \in \mathit{Class}$	Name of a class	$V.\mathit{Field} = V$	$V = V.\mathit{Field}$
Name	Name of a field or a method	$\mathit{Class}.\mathit{Field} = V$	$V = \mathit{Class}.\mathit{Field}$
SimpleType	Simple type (we restrict ourselves to `int`)	$V[V] = V$	$V = V[V]$
RefType	Reference type (*Class* or array of *Type*)	$V =$ `null`	$V =$ `new` *Class*
Type	Type ($\mathit{RefType} \cup \mathit{SimpleType}$)	$V =$ `new` $\mathit{Type}[V]$	
MethodType	Method type ($\mathit{Type}^* \times (\mathit{Type} \cup \{\texttt{void}\})$)	$V = V.\mathit{Method}(V, ..., V)$	
$m \in \mathit{Method}$	Method ($\mathit{Class} \times \mathit{Name} \times \mathit{MethodType}$)	`return` V	
$f \in \mathit{Field}$	Field ($\mathit{Class} \times \mathit{Name} \times \mathit{Type}$)	`if` (...) `goto` *Address*	

Fig. 3. Analyzed language syntax

4. ALIAS ANALYSIS

The correctness proof of our escape analysis could be performed using a wide variety of alias analyses. We have chosen a relatively simple alias analysis that does not remove aliases when objects are overwritten.

In our alias analysis, objects are represented by *object names* $N \in \mathit{OName}$, which are similar to locations, except that different names may represent the same location. An alias between two object names means precisely that they may represent the same location. Parts of objects are represented by access paths:

$$\mathit{Path} = (\mathit{Field} \cup \mathbb{N})^*$$

where $(C, f, t) \in \mathit{Field}$ means that we access the (C, f, t) field of the considered object, and $n \in \mathbb{N}$ means that we access the nth element of the considered array. Let $\pi = N.p \in \mathit{OName} \times \mathit{Path}$ denote the object accessed from object of name N by path p. Let static.$f.p$ denote the object accessed from the static field f by path p. If π denotes object o, then $\pi.(C, f, t)$ where $(C, f, t) \in \mathit{Field}$ denotes the field (C, f, t) of the object o. If π denotes an array o, $\pi.n$ where $n \in \mathbb{N}$ denotes the nth element of o. The empty path is ϵ. An alias is represented by a pair (π_1, π_2), which is written $\pi_1 \sim \pi_2$ for readability. It means that π_1 and π_2 may represent the same object. Alias analysis computes two pieces of information. First, it associates with each computation step the object name contained in each variable. These object names are used to reference objects in paths π. Second, alias analysis also computes the alias relation: a set of pairs $\pi_1 \sim \pi_2$ such that π_1 and π_2 may correspond to the same object. An object can be stack allocated if it has a name N such that the alias relation contains no alias of the form $N.\epsilon \sim \pi$ where π represents any data of scope larger than the current method (that is, the parameters and result of the method and the static fields).

Example 4.1. Consider for example the structure represented in Figure 4. If N is the object name representing the vector `v`, then N.(`Vector`, `count`, `int`) (or simply N.`count`) represents the integer 2, N.`elementData` represents the array of objects, N.`elementData`.1.`value` represents the integer 3 (`value` is the field of the class `Integer`). static.`s` and N.`elementData`.0 both represent the string `"Yes"`, hence there is an alias static.`s` $\sim$ N.`elementData`.0.

In Example 2.1, let *Res* be an object name of the result of `run()`. Informally, *Res* is aliased with the `new Integer(2)`, which can therefore not be stack allocated.

Alias analysis is defined in Figure 5. In this figure, $\mathit{NewName} \in \mathit{OName}$ desig-

```
Vector {
  int count;
  Object[] elementData;
}
```

Let `v` be a `Vector` such that

```
static String s = "Yes"
v.count = 2
v.elementData = new Object[] { s, new Integer(3) }
```

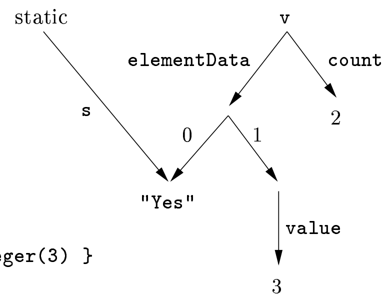


Fig. 4. Data structure for examples of access paths

nates a new object name. Abstract values can either be object names $N \in \mathit{OName}$ for objects and arrays, or $\emptyset$ for variables of simple types such as integers.

We define the date $d \in \mathit{Date}$ as the number of instructions executed in the analyzed program. An execution trace T of the analyzed program is then a function from dates to states: $T(d)$ is the state at date d. Consider one fixed execution trace T of the program. The program counter at date d in this execution trace is $\mathrm{pc}(T(d))$. We compute two pieces of alias information corresponding to the execution trace T:

(1) An *abstract trace* $T^\#$ which gives the abstract state for each date d. The abstract state gives an object name for each variable. It contains two parts: $L \in \mathit{LocVar}^\#$ yields the abstract value of the local variables of the current method being executed m, whereas $J \in \mathit{JavaStack}^\#$ corresponds to the local variables of the callers of m. Notice that we have the abstract state for each date, and not for each program point. When there is a loop in the program, we simply follow the iterations of the execution trace (we come back to the same *pc*, but with a different date d, so we do not have fixpoint equations here).

(2) A *dated alias relation* $R \in \mathit{DateAlias}$ in which each alias is registered with its creation date. R summarizes all aliases created during the runtime of the program. The symmetric element of an alias and its right-regular closure are always added at the same time as the alias itself, i.e. if the instruction executed at date d adds alias $\pi_1 \sim \pi_2$, R contains $\{(\pi_1.p, \pi_2.p, d+1), (\pi_2.p, \pi_1.p, d+1), p \in \mathit{Path}\}$ (the alias only exists *after* the instruction is executed, i.e. at date $d+1$). R may contain several times the same alias with different creation dates, if an instruction that adds an alias is executed when this alias already existed. The right-regular equivalence relation[1] $R(d)$ gives aliases at or before date d:

$$R(d) = \{\pi_1 \sim \pi_p \mid \forall i = 1, ..., p-1, (\pi_i \sim \pi_{i+1}, d_i) \in R, d_i \le d\} \in \mathit{Alias}.$$

These two pieces of information could also be considered as an instrumentation of a small steps semantics of Java.

Remark. Notice that the SSA form is in fact not necessary for this alias analysis. Indeed, we create a new object name at every local variable assignment, thus providing a form of "dynamic single assignment". However, SSA form is useful for

[1] Reflexive, symmetric and transitive relation, such that $\forall(\pi_1, \pi_2) \in R, \forall p \in \mathit{Path}, (\pi_1.p, \pi_2.p) \in R$.

$d \in Date = \mathbb{N}$	Date
$pc \in PC = Method \times Address$	Program counter
$\mathrm{P}(pc)$	Instruction at pc
$p \in Path = (Field \cup \mathbb{N})^*$	Access paths
$N \in OName$ (any countable set)	Object names
$L \in LocVar^{\#} = V \to (OName \cup \{\emptyset\})$	Abstract local variables
$J \in JavaStack^{\#} = (LocVar^{\#})$ `list`	Abstract Java stack
$(L, J) \in State^{\#} = LocVar^{\#} \times JavaStack^{\#}$	Abstract state
$T^{\#} \in Trace^{\#} = Date \to State^{\#}$	Abstract trace
$Alias = \mathcal{P}(((OName \cup \mathrm{static}) \times Path)^2)$	Alias relation
$R \in DateAlias = \mathcal{P}(((OName \cup \mathrm{static}) \times Path)^2 \times Date)$	Dated alias relation
$\mathrm{Val}^{\#} : State^{\#} \times V \to (OName \cup \{\emptyset\})$	Abstract value of a variable
$\mathrm{Val}^{\#}((L, J), v) = L(v)$	

$\mathrm{P}(\mathrm{pc}(T(d)))$	Abstract state transition $d \Rightarrow d+1$	Alias
entry of `main(args)`	$\Rightarrow(\{\texttt{args} \mapsto P'_0\}, []), P'_0 = NewName$	
$w = v_0.m'(v_1, ..., v_n)$	$(L, J) \Rightarrow (\{p_i \mapsto P'_i\}, L : J)$	$L(v_i) \sim P'_i$
The p_i are the names of formal parameters of m', $P'_i = NewName (i \in [0, n])$		
return v	$(L, L' : J) \Rightarrow (L'[w \mapsto L(v)], J)$	
w is the variable in which we store the result of the corresponding method call.		

$\mathrm{P}(\mathrm{pc}(T(d)))$	Local variables transition $d \Rightarrow d+1$	Alias
$v_1 = v_2 / v_1 = (t)v_2$	$L \Rightarrow L[v_1 \mapsto N], N = NewName$	$L(v_2) \sim N$
$v_1 = \phi(v_2, v_3)$	$L \Rightarrow L[v_1 \mapsto N], N = NewName$	$L(v_2)$ or $L(v_3) \sim N^a$
$v_1 = v_2.f$	$L \Rightarrow L[v_1 \mapsto N], N = NewName$	$L(v_2).f \sim N$
$v_1.f = v_2$	$L \Rightarrow L$	$L(v_1).f \sim L(v_2)$
$v_1 = C.f$	$L \Rightarrow L[v_1 \mapsto N], N = NewName$	$N \sim \mathrm{static}.f$
$C.f = v_1$	$L \Rightarrow L$	$L(v_1) \sim \mathrm{static}.f$
$v_1 = v_2[v_3]$	$L \Rightarrow L[v_1 \mapsto N], N = NewName$	$L(v_2).\mathbb{N} \sim N$
$v_1[v_2] = v_3$	$L \Rightarrow L$	$L(v_1).\mathbb{N} \sim L(v_3)$
$v =$ `new` C/`new` $t[w]$/`null`	$L \Rightarrow L[v \mapsto NewName]$	
`if` (...) `goto` a		

[a] The created alias is $L(v_2) \sim N$ if v_2 has been defined after v_3 in the execution trace, or v_3 has never been defined. It is $L(v_3) \sim N$ otherwise.

Fig. 5. Alias analysis. The Java stack J is not modified by instructions from the bottom table. The "Alias" column indicates aliases added to relation R. We assume that all fields and arrays contain objects or arrays, and all parameters of methods are objects or arrays (no simple types). Data of simple types have no alias or escape effect. No alias is created for a value of simple type, and the corresponding abstract value is $\emptyset$ instead of an object name.

the escape analysis we implement. We use it from the beginning to avoid having to introduce it later.

Example 4.2. Consider again the binary tree class of Figure 1. The beginning of the abstract trace is represented in Figure 6. The name `<init>` designates constructor methods. The code is indented: when executing a method call, the code of the called method is two characters on the right with respect to the code of the caller. To make reading easier, we have chosen to use object names with the same index i: N_i, N'_i, N''_i, ... for object names that correspond to the same location. There will be aliases between these objects. The aliases are given in the dated alias relation R at the bottom of Figure 6. The aliases $N_4 \sim N'_4$, $N_5 \sim N'_5$, $N''_2 \sim N'''_2$, $N_6 \sim N'_6$ are created by the call to the `BinTree` constructor in `leaf`,

d	Instruction	Abstract state $T^{\#}(d+1)$
0	entry	$(\{\texttt{args} \mapsto P_0\}, [])$
1	`v0 = BinTree.run()`	$(\{\}, M)$
2	`  v1 = null`	$(\{\texttt{v1} \mapsto N_1\}, M)$
3	`  v2 = new Integer`	$(\{\texttt{v1} \mapsto N_1, \texttt{v2} \mapsto N_2\}, M)$
4	`  v2.<init>(1)`	$(\{\texttt{this} \mapsto N_2', \texttt{x} \mapsto \emptyset\}, J_1)$
5	`    this.value = x`	$(\{\texttt{this} \mapsto N_2', \texttt{x} \mapsto \emptyset\}, J_1)$
6	`    return`	$(\{\texttt{v1} \mapsto N_1, \texttt{v2} \mapsto N_2\}, M)$
7	`  v3 = leaf(v2)`	$(\{\texttt{o} \mapsto N_2''\}, J_1)$
8	`    v4 = new BinTree`	$(\{\texttt{o} \mapsto N_2'', \texttt{v4} \mapsto N_4\}, J_1)$
9	`    v5 = null`	$(\{\texttt{o} \mapsto N_2'', \texttt{v4} \mapsto N_4, \texttt{v5} \mapsto N_5\}, J_1)$
10	`    v6 = null`	$(\{\texttt{o} \mapsto N_2'', \texttt{v4} \mapsto N_4, \texttt{v5} \mapsto N_5, \texttt{v6} \mapsto N_6\}, J_1)$
11	`    v4.<init>(v5,v6,o)`	$(\{\texttt{this} \mapsto N_4', \texttt{l} \mapsto N_5', \texttt{r} \mapsto N_6', \texttt{e} \mapsto N_2'''\}, J_2)$
12	`      this.left = l`	$(\{\texttt{this} \mapsto N_4', \texttt{l} \mapsto N_5', \texttt{r} \mapsto N_6', \texttt{e} \mapsto N_2'''\}, J_2)$
13	`      this.right = r`	$(\{\texttt{this} \mapsto N_4', \texttt{l} \mapsto N_5', \texttt{r} \mapsto N_6', \texttt{e} \mapsto N_2'''\}, J_2)$
14	`      this.elem = e`	$(\{\texttt{this} \mapsto N_4', \texttt{l} \mapsto N_5', \texttt{r} \mapsto N_6', \texttt{e} \mapsto N_2'''\}, J_2)$
15	`      return`	$(\{\texttt{o} \mapsto N_2'', \texttt{v4} \mapsto N_4, \texttt{v5} \mapsto N_5, \texttt{v6} \mapsto N_6\}, J_1)$
16	`    return v4`	$(\{\texttt{v1} \mapsto N_1, \texttt{v2} \mapsto N_2, \texttt{v3} \mapsto N_4\}, M)$
...	...	...

where
$M = [\{\texttt{args} \mapsto P_0\}]$,
$J_1 = \{\texttt{v1} \mapsto N_1, \texttt{v2} \mapsto N_2\} : M$,
$J_2 = \{\texttt{o} \mapsto N_2'', \texttt{v4} \mapsto N_4, \texttt{v5} \mapsto N_5, \texttt{v6} \mapsto N_6\} : J_1$
$R = \mathrm{clo}(\{(N_2 \sim N_2', 5), (N_2 \sim N_2'', 8), (N_4 \sim N_4', 12),$
$(N_5 \sim N_5', 12), (N_6 \sim N_6', 12), (N_2'' \sim N_2''', 12),$
$(N_4'.\texttt{left} \sim N_5', 13), (N_4'.\texttt{right} \sim N_6', 14),$
$(N_4'.\texttt{elem} \sim N_2''', 15), ...\})$
where $\mathrm{clo}(A) = \{(\pi_1.p \sim \pi_2.p, d), (\pi_2.p \sim \pi_1.p, d) \mid (\pi_1 \sim \pi_2, d) \in A, p \in Path\}$.

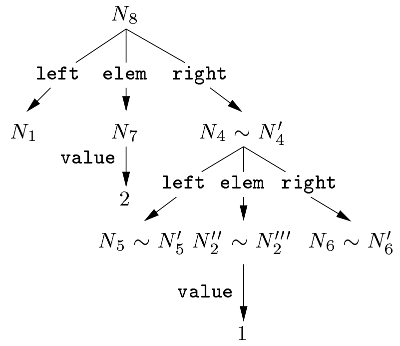


Fig. 6. Example of alias analysis

executed at date 11 (they have date 12). The fact that the `elem` field of the node `v4 = leaf(new Integer(1))` contains `new Integer(1)` is represented by the alias $N_4'.\texttt{elem} \sim N_2'''$, where N_4' is an object name for the node in `v4` and N_2''' an object name for the `new Integer(1)`. This alias is added by instruction `this.elem = e` executed at date 14. It has therefore date 15. Aliases involving N_7 and N_8 are created in the end of the execution trace, that is not shown due to its length.

THEOREM 4.3 (STACK ALLOCATION). *Consider one execution of method m, in execution trace T, and assume that this execution of m terminates at date r.*

Consider a `new` *in method m, allocating object o. Let Res be the object name associated with the result of m, P_l the object names associated with the parameters*[2] *of m (the names of these objects in the caller of m), N the object name of o. Let $Esc = \{\mathrm{static}.p' \mid p' \in Path\} \cup \{Res.p' \mid p' \in Path\} \cup \cup_l \{P_l.p' \mid p' \in Path\}$.*

If $\forall \pi \in Esc, (N.\epsilon \sim \pi) \notin R(r)$, o can be stack allocated in m in the considered execution trace T and in the execution of m that ends at date r.

[2]The receiver `this` is considered as an ordinary parameter.

Intuitively, if $\forall \pi \in Esc, (N.\epsilon \sim \pi) \notin R(r)$, at the end of method m, the object o is not aliased to objects reachable from the parameters, the result of m, or from static fields. So it becomes unreachable at the end of m. A proof sketch of this result can be found in Appendix A.

Example 4.4. Consider again Example 2.1 and its alias analysis (Figure 6). At the end of `BinTree.run`, the only aliases with elements of *Esc* are $N_7.p \sim Res.p$, where p is any path, and others due to aliases of N_7: $\pi' \sim Res.p$ where $\pi' \sim N_7.p$. In particular, since we do not have $N_8.\epsilon \sim N_7.p$, $\forall \pi \in Esc, (N_8.\epsilon \sim \pi) \notin R(r)$, so the `BinTree` node referenced by variable `local` can be stack allocated in `BinTree.run`.

5. ANALYSIS $\mathbb{E}$: ESCAPE ANALYSIS USING ACCESS PATHS

The following analysis is very precise, so it can be the basis for several less precise escape analyses, which can be derived by abstract interpretation. However, it is too complex to be directly implemented. We will therefore define a more approximate analysis in Section 6. Analysis $\mathbb{E}$ does not use types, so we do not have to deal with subtyping. Types will be used in Section 6.

5.1 Abstract lattice for values

An object is said to escape from method m if and only if it is reachable from the parameters or the result of m, or from static fields. If an object escapes from m, it will not be stack allocated in m. Having for each data structure only a binary information, "it may escape" or "it does not escape", would be far too coarse. A part of a data structure may escape, while another part does not escape. For instance, in Example 2.1, `local.elem` escapes from `run`, but not `local` itself. Therefore, we have to represent parts of data structures, to be able to say "this part escapes". In analysis $\mathbb{E}$, we have chosen to use sets of paths to represent parts of objects. This is natural since access paths were already used in our alias analysis. The escaping part of an object is said to be the escape *context* of the object, so escape contexts for this analysis are sets of paths: $Ctx_{\mathbb{E}} = \mathcal{P}(Path)$. They are ordered by inclusion, $Ctx_{\mathbb{E}}$ is then a complete lattice.

Example 5.1.1. If we consider again the structure of Figure 4, the part of `v` corresponding to the array of objects can be represented by paths $\{\texttt{elementData}.p \mid p \in Path\}$. Similarly, the part of `v` corresponding to the integer 3 is represented by $\{\texttt{elementData}.1.\texttt{value}\}$.

Intuitively, analysis $\mathbb{E}$ computes for each variable v its escape context, that is, the part of v that escapes, represented by a set of access paths. A newly allocated object can be stack allocated if the object itself does not escape, that is, if its escape context does not contain the empty path ϵ. While alias analysis computes alias relations indicating whether an access path is aliased to another access path, analysis $\mathbb{E}$ computes the set of access paths that are aliased to data of scope larger than the analyzed method (parameters, result of the method and static fields).

To define analysis $\mathbb{E}$, we first have to define basic operations on the escape contexts. They are of two kinds:

(1) *constructions* $f.c$ where $f \in Field$: if c is the context associated with the f field of an object o, $f.c$ yields the context associated with o. Let $f.c = \{f.p \mid p \in c\}$.

Similarly, $\mathbb{N}.c = \{n.p \mid p \in c, n \in \mathbb{N}\}$ yields the context associated with the array a, where c is the context associated with an element of array a.

(2) Conversely, $f^{-1}.c$ is a *restriction*: if c is the context associated with an object o, $f^{-1}.c$ yields the context of the field $o.f$. Let $f^{-1}.c = \{p \mid f.p \in c\} \cup \{\epsilon \text{ if } \epsilon \in c\}$, $\mathbb{N}^{-1}.c = \{p \mid \exists n \in \mathbb{N}, n.p \in c\} \cup \{\epsilon \text{ if } \epsilon \in c\}$ (we have made an approximation here, since we do not distinguish any more the different elements of the array).

Example 5.1.2. Assume that the escaping part of `v` is represented by context $c = \{\texttt{elementData}.1, \texttt{elementData}.1.\texttt{value}\}$, in the structure of Figure 4. Then the escaping part of the array of objects will be $\texttt{elementData}^{-1}.c = \{1, 1.\texttt{value}\}$. The escaping part of an element of the array of objects will be $\mathbb{N}^{-1}.\texttt{elementData}^{-1}.c = \{\epsilon, \texttt{value}\}$.

If the escaping part of the array of objects is $c = \mathit{Path}$, then the escape context of the vector `v` is at least $\texttt{elementData}.c = \{\texttt{elementData}.p \mid p \in \mathit{Path}\}$.

To compute the escape information, we use a *bidirectional* propagation: $\mathbb{E}$ is a backward analysis, whereas $\mathbb{E}_S$ is a forward analysis. The analyses $\mathbb{E}$ and $\mathbb{E}_S$ depend on each other. First, what is read to build the result of a method escapes (for example, if a method reads a field f with instruction $v_1 = v_2.f$, and returns this field as its result, then the field escapes). This is what would be computed by analysis $\mathbb{E}$ if it were alone, using a backward propagation (from the result to the parameters). The result of the method is first marked as escaping. Then, when a read instruction such as $v_1 = v_2.f$ is analyzed, and v_1 is marked as escaping, the corresponding part of v_2 is also marked as escaping. However, because of assignments, objects may also escape because they are stored in static fields, in parameters, or in the result of the method. The backward propagation cannot take into account that an object escapes because it is stored in another object. For instance, assume that o is stored in a parameter o' of the method. At the point of the assignment, the backward analyzer would not know whether o' is a parameter of the method, therefore it would not know whether this assignment makes o escape. Therefore, we have to introduce a forward analysis $\mathbb{E}_S$ to cope with assignments (S for *store*). For example, in `BinTree.run()`, $\mathbb{E}$ alone is enough to take into account that elements of `local` may be part of the result, but $\mathbb{E}_S$ is necessary to take into account the fact that the new `Integer` may be stored in `local`.

This bidirectional propagation is necessary to model precisely assignments. It introduces mutual dependencies between unknowns, even if there is no loop in the analyzed program. Therefore, the usual fixpoint iteration, which corresponds to iteration over the loops of the analyzed program, is not enough. That is why we have started from an alias analysis, because it makes explicit a second form of iteration: the iteration to compute the transitive closure of the alias relation. This iteration can be mapped to the fixpoint iteration used to compute the escape analysis. This will be used in the correctness proof (see Example 5.2.2).

If we want to know whether we can allocate an object o on the stack in method m, the reference scope is method m. Then the parameters and result of m escape. Therefore, we set to Path the escape context of the parameters and result of m, before computing the escaping part of o. But if we want to perform stack allocation in a method m' which calls m, the reference scope is now m'. Then, the parameters and result of m do not always escape from m' (depending on what m' does). Therefore, we have to analyze m in several calling contexts. To avoid reanalyzing the method m for each calling context, our analysis is a function of the calling context, which is represented by the escape contexts of the parameters and of the result of m. Our escape analysis is therefore context sensitive.

Therefore, abstract values are *context transformers*, i.e. functions from contexts to contexts: knowing the contexts associated with the parameters (by analysis $\mathbb{E}_S$) $c_0, ..., c_j$ and with the result (by analysis $\mathbb{E}$) c_{-1} of the current method, they yield the context associated with the corresponding concrete value $\phi(c_{-1}, ..., c_j)$. Therefore $Val_{\mathbb{E}} = Ctx_{\mathbb{E}}^* \to Ctx_{\mathbb{E}}$. Notice that the receiver `this` is regarded as any other parameter of the methods. The order on abstract values is the pointwise extension of the order on contexts.

We now define for context transformers operations that were previously defined for contexts. If the analyzed method m has $j+1$ parameters, we define $f.E = \{(c_{-1}, ..., c_j) \mapsto f.E(c_{-1}, ..., c_j)\}$, $\mathbb{N}.E$, $f^{-1}.E$ and $\mathbb{N}^{-1}.E$ similarly. The greatest element of the lattice of abstract values is $\top_{\mathbb{E}}[f] = \{(c_{-1}, ..., c_j) \mapsto Path\}$[3]. The abstract value for the result is $first_{\mathbb{E}} = \{(c_{-1}, ..., c_j) \mapsto c_{-1}\}$. If the formal parameters of m are $p_0, ..., p_j$, the abstract values for the parameters are $P_{\mathbb{E}}(p_i) = \{(c_{-1}, ..., c_j) \mapsto c_i\}$.

5.2 The analysis

Analysis $\mathbb{E}$ is defined in Figure 7. The escape analyses $\mathbb{E}$ and $\mathbb{E}_S$ yield for each method m, for each local variable v of m, the corresponding escape abstract value $\mathbb{E}(m)(v)$ or $\mathbb{E}_S(m)(v)$. The environment $\rho \in Env_{\mathbb{E}}$ yields for each method the escape information for its parameters (for analysis $\mathbb{E}$). Each parameter is represented by a parameter index $i \in Ind$. The environment $\rho_S \in Env_{\mathbb{E}S}$ yields for each method the escape information for its result (for analysis $\mathbb{E}_S$).

The analyses $\mathbb{E}$ and $\mathbb{E}_S$ are very similar: they both yield an escape information for each variable. While $\mathbb{E}_S(m)(v)$ takes into account all aliases created in method m, $\mathbb{E}(m)(v)$ takes into account only alias chains whose first alias has been created strictly after the creation of v. (This is discussed more formally in the correctness proof in the next section.)

The form of the equations of the analysis comes almost directly from the directions of each analysis. Analysis $\mathbb{E}_S$ is a forward analysis, so $\mathbb{E}_S(m)(v)$ is defined when v is defined, and used when v is used. That is, for a statement defining v, of the form $v = \ldots$, we have an equation of the form $\mathbb{E}_S(m)(v) = \ldots$ to initialize $\mathbb{E}_S(m)(v)$. For a statement using v, $\mathbb{E}_S(m)(v)$ appears on the right-hand side of the equations. Conversely, analysis $\mathbb{E}$ is backward, so $\mathbb{E}(m)(v)$ is defined (or rather updated) when v is used, and used when v is defined. That is, for a statement of the form $v = \ldots$, $\mathbb{E}(m)(v)$ appears on the right-hand side of the equations, and for a statement using v, we have an equation $\mathbb{E}(m)(v) \geq \ldots$. The value of $\mathbb{E}(m)(v)$ is in essence the maximum of the values of $\mathbb{E}$ at the uses of v. The analysis $\mathbb{E}$ propagates from the uses to the definition of v.

On entry of method m, the abstract values of the parameters for the forward analysis $\mathbb{E}_S$ are initialized. Since c_i represents the escape context of parameter p_i,

[3]The parameter f is useless when defining analysis $\mathbb{E}$, but will be useful when defining $\mathbb{L}$. Mentioning this parameter makes it possible to use the same equations for $\mathbb{E}$ and $\mathbb{L}$.

$i \in Ind = \mathbb{N}$	Parameter indices
$\rho, \rho' \in Env_{\mathbb{E}} = Method \to Ind \to Val_{\mathbb{E}}$	Environment ($\mathbb{E}$ contexts for the parameters)
$\rho_{\mathrm{S}}, \rho'_{\mathrm{S}} \in Env_{\mathbb{E}\mathrm{S}} = Method \to Val_{\mathbb{E}}$	Environment ($\mathbb{E}_{\mathrm{S}}$ context for the result)
$L, L_{\mathrm{S}} \in LocVar_{\mathbb{E}} = V \to Val_{\mathbb{E}}$	Abstract local variables
$\mathbb{E}, \mathbb{E}_{\mathrm{S}} \in Method \to LocVar_{\mathbb{E}}$	Escape analyses

$\mathrm{P}(pc)$	Forward analysis $L_{\mathrm{S}} = \mathbb{E}_{\mathrm{S}}(m)$	Backward analysis $L = \mathbb{E}(m)$
entry of m	$L_{\mathrm{S}} = P_{\mathbb{E}}$	$\forall i \in [0, j], \rho(m)(i) \geq L(p_i)$
$v_1 = v_2 / v_1 = (t)v_2$	$L_{\mathrm{S}}(v_1) = L_{\mathrm{S}}(v_2)$	$L(v_2) \geq L(v_1)$
$v_1 = \phi(v_2, v_3)$	$L_{\mathrm{S}}(v_1) = L_{\mathrm{S}}(v_2) \sqcup L_{\mathrm{S}}(v_3)$	$L(v_2) \geq L(v_1), L(v_3) \geq L(v_1)$
$v_1 = v_2.f$	$L_{\mathrm{S}}(v_1) = f^{-1}.L_{\mathrm{S}}(v_2)$	$L(v_2) \geq f.L(v_1)$
$v_1.f = v_2$		$L(v_2) \geq f^{-1}.L_{\mathrm{S}}(v_1), L(v_1) \geq f.L_{\mathrm{S}}(v_2)$
$v = C.f$	$L_{\mathrm{S}}(v) = \top_{\mathbb{E}}[f]$	
$C.f = v$		$L(v) \geq \top_{\mathbb{E}}[f]$
$v_1 = v_2[v_3]$	$L_{\mathrm{S}}(v_1) = \mathbb{N}^{-1}.L_{\mathrm{S}}(v_2)$	$L(v_2) \geq \mathbb{N}.L(v_1)$
$v_1[v_2] = v_3$		$L(v_3) \geq \mathbb{N}^{-1}.L_{\mathrm{S}}(v_1), L(v_1) \geq \mathbb{N}.L_{\mathrm{S}}(v_3)$
$v =$ `new` $C/$`new` $t[w]$	$L_{\mathrm{S}}(v) = L(v)$	
$v =$ `null`	$L_{\mathrm{S}}(v) = L(v)$	
`if` (...) `goto` a		
$w = v_0.m'(v_1, ..., v_n)$	$L_{\mathrm{S}}(w) = \rho'_{\mathrm{S}}(m') \circ (L(w), L_{\mathrm{S}}(v_0), ..., L_{\mathrm{S}}(v_n))$	
		$L(v_i) \geq \rho'(m')(i) \circ (L(w), L_{\mathrm{S}}(v_0), ..., L_{\mathrm{S}}(v_n)), i \in [0, n]$
`return` v	$\rho_{\mathrm{S}}(m) \geq L_{\mathrm{S}}(v)$	$L(v) \geq first_{\mathbb{E}}$

Fig. 7. Escape analysis. m is the current method, which has $j+1$ parameters. In the above table, we assume that all fields and arrays contain objects or arrays (data of simple types have no escape effect, therefore no equation is generated for field or array accesses or assignments when they contain data of simple types).

it is natural to have $\mathbb{E}_{\mathrm{S}}(m)(p_i)(c_{-1}, ..., c_j) = c_i$, therefore $L_{\mathrm{S}}(p_i) = \{(c_{-1}, ..., c_j) \mapsto c_i\} = P_{\mathbb{E}}(p_i)$. For analysis $\mathbb{E}$, the computation of the backward escape contexts of the parameters is complete when the beginning of the method is reached, and the escape information for the parameters of method m is updated in environment ρ.

For the local variable assignments $v_1 = v_2$ and $v_1 = (t)v_2$, the escape information is transmitted from v_2 to v_1 for analysis $\mathbb{E}_{\mathrm{S}}$ and from v_1 to v_2 for $\mathbb{E}$.

For $v_1 = \phi(v_2, v_3)$, we have to assume the worst case, that is the value in v_1 may be reachable as soon as the values in v_2 or v_3 are reachable, hence the upper bound for analysis $\mathbb{E}_{\mathrm{S}}$. For analysis $\mathbb{E}$, the values in v_2 and v_3 may be reachable as soon as the value in v_1 is reachable.

When reading a field by $v_1 = v_2.f$, the contents of v_1 is reachable as soon as the field f of v_2 is reachable, hence the escape information for v_1 is computed by restricting the information for v_2 to the field f, in analysis $\mathbb{E}_{\mathrm{S}}$. Conversely for analysis $\mathbb{E}$, if a part of v_1 escapes, the same part of the field f of v_2 escapes.

When writing into a field by $v_1.f = v_2$, intuitively, the part of the field f of v_1 that escapes is the part of v_2 that escapes: if the field f of v_1 escapes, v_2 may escape, and if v_2 escapes, the field f of v_1 may escape. The two equations update the escape information for v_2 and v_1 to take this fact into account. One can note that, even if the equations for $v_1.f = v_2$ are equations of analysis $\mathbb{E}$, $\mathbb{E}_{\mathrm{S}}$ is used in these equations. This is why $\mathbb{E}_{\mathrm{S}}$ is necessary to compute the effect of assignments.

The values reachable from static fields are considered to have an unbounded lifetime. Hence when reading or writing static fields, the read or stored values escape entirely. The handling of reads and writes to arrays is similar to the one of objects (we consider that any element of the array may be accessed).

For the virtual method call $w = v_0.m'(v_1, ..., v_n)$ where $m' = (C, m'', t)$, we consider that all methods that have a correct signature (m'', t) and are defined in a subclass of C may be called. This corresponds to class hierarchy analysis [Dean et al. 1995]. Therefore, the escape information used for virtual calls is defined by the following environments: $\rho'(C, m'', t)(i) = \sqcup_{C' \text{ subclass of } C}\rho(C', m'', t)(i)$ and $\rho'_{\mathrm{S}}(C, m'', t) = \sqcup_{C' \text{ subclass of } C}\rho_{\mathrm{S}}(C', m'', t)$.

On the return from method m, the escape information for the result of m is updated in environment ρ_{S}. For analysis $\mathbb{E}$, the escape context of the result is c_{-1}, therefore the abstract value is $\mathit{first}_{\mathbb{E}} = \{(c_{-1}, ..., c_j) \mapsto c_{-1}\}$.

Example 5.2.1. Let m be the `BinTree` constructor from Example 2.1. Let $L_{\mathrm{S}} = \mathbb{E}_{\mathrm{S}}(m)$, $L = \mathbb{E}(m)$. The analysis of m is the following:

Analysis $\mathbb{E}_{\mathrm{S}}$		Analysis $\mathbb{E}$
$L_{\mathrm{S}}(p_i) = P_{\mathbb{E}}(p_i)$	`BinTree(BinTree l,` `BinTree r, Object e) {`	
	`this.left = l;`	$L(\mathtt{l}) \geq \mathtt{left}^{-1}.L_{\mathrm{S}}(\mathtt{this})$, $L(\mathtt{this}) \geq \mathtt{left}.L_{\mathrm{S}}(\mathtt{l})$
	`this.right = r;`	$L(\mathtt{r}) \geq \mathtt{right}^{-1}.L_{\mathrm{S}}(\mathtt{this})$, $L(\mathtt{this}) \geq \mathtt{right}.L_{\mathrm{S}}(\mathtt{r})$
	`this.elem = e;`	$L(\mathtt{e}) \geq \mathtt{elem}^{-1}.L_{\mathrm{S}}(\mathtt{this})$, $L(\mathtt{this}) \geq \mathtt{elem}.L_{\mathrm{S}}(\mathtt{e})$
	`}`	

The middle column is the analyzed instruction. Abstract values for this method have four parameters c_0, c_1, c_2, c_3, which are respectively the $\mathbb{E}_{\mathrm{S}}$ escape contexts of `this`, `l`, `r` and `e` (there is no c_{-1} here since the method has no result). So the abstract values are of the form $\{(c_0, c_1, c_2, c_3) \mapsto \phi(c_0, ..., c_3)\}$. When analyzing another method m' that calls m, we need to know what happens if only a part of the parameters of m escapes from the caller m', that is why c_0, ..., c_3 are parameters and not constants.

The left column corresponds to the forward pass (analysis $\mathbb{E}_{\mathrm{S}}$). We say that an object o store-escapes when, if we store an object o' in o, o' escapes. Analysis $\mathbb{E}_{\mathrm{S}}$ determines a superset of the set of objects that store-escape. At the beginning of the method, the local variables are initialized with the parameters, so the corresponding escape information is $P_{\mathbb{E}}$, with $\mathbb{E}_{\mathrm{S}}(m)(\mathtt{this}) = P_{\mathbb{E}}(\mathtt{this}) = \{(c_0, c_1, c_2, c_3) \mapsto c_0\}$, which means that if the calling context is such that the implicit parameter of m store-escapes, then `this` store-escapes (they have the same escape context c_0). In the same way, $P_{\mathbb{E}}(\mathtt{l}) = \{(c_0, c_1, c_2, c_3) \mapsto c_1\}$ and so on.

The right column corresponds to the backward pass (analysis $\mathbb{E}$). The first inequation $L(\mathtt{l}) \geq \mathtt{left}^{-1}.L_{\mathrm{S}}(\mathtt{this})$ means that if the `left` field of `this` store-escapes, then `l` escapes. We obtain: $\mathbb{E}(m)(\mathtt{l}) = \{(c_0, c_1, c_2, c_3) \mapsto \mathtt{left}^{-1}.c_0\}$. The second inequation $L(\mathtt{this}) \geq \mathtt{left}.L_{\mathrm{S}}(\mathtt{l})$ means that if `l` store-escapes then the `left` field of `this` escapes. Therefore $\mathbb{E}(m)(\mathtt{this}) \geq \{(c_0, c_1, c_2, c_3) \mapsto \mathtt{left}.c_1\}$. We have similar inequations for other instructions.

The analysis yields:

$$\mathbb{E}(m)(\mathtt{this})(c_0, c_1, c_2, c_3) = \mathtt{left}.c_1 \cup \mathtt{right}.c_2 \cup \mathtt{elem}.c_3$$
$$\mathbb{E}(m)(\mathtt{l})(c_0, c_1, c_2, c_3) = \mathtt{left}^{-1}.c_0$$

The context transformers $\mathbb{E}(m)(\mathtt{r})$ and $\mathbb{E}(m)(\mathtt{e})$ are similar to $\mathbb{E}(m)(\mathtt{l})$. So the fields of `this` may escape if the parameters `l`, `r` and `e` escape, and conversely, the parameters `l`, `r` and `e` may escape if the corresponding field of `this` escapes. Analysis $\mathbb{E}$ is therefore able to express relations between escaping parts of different parameters, so it is very precise.

Example 5.2.2. Consider the following code (the `next` field $(\mathtt{List}, \mathtt{next}, \mathtt{List})$ is represented simply by `next`; $\mathbb{E}(m)(\mathtt{a}) = E$; $\mathbb{E}_{\mathrm{S}}(m)(\mathtt{a}) = E_{\mathrm{S}}$):

Analysis $\mathbb{E}_{\mathrm{S}}$
$E_{\mathrm{S}} = E$

```
class List {
  List next; Object elem;

  static List m() {
    a = new List();
    a.next = a;
    return a;
  }
}
```

Analysis $\mathbb{E}$
$E \geq \mathtt{next}^{-1}.E_{\mathrm{S}}, E \geq \mathtt{next}.E_{\mathrm{S}}$
$E \geq \mathit{first}_{\mathbb{E}}$

The left column of the table is the forward analysis $\mathbb{E}_{\mathrm{S}}$ of method m. The right column is the backward analysis $\mathbb{E}$ of m. The equations can be simplified into:

$$E_{\mathrm{S}} \geq \mathtt{next}^{-1}.E_{\mathrm{S}} \sqcup \mathtt{next}.E_{\mathrm{S}} \sqcup \mathit{first}_{\mathbb{E}}.$$

The solution is $E_{\mathrm{S}} = E = \{c \mapsto \mathtt{next}^*.(\mathtt{next}^*)^{-1}.c\}$. This can only be found by iterating (an infinite number of times) the instruction `a.next = a`, which contains no loop. So this iteration cannot be mapped to an iteration in the operational semantics, but corresponds to the iteration over the alias $N \sim N.\mathtt{next}$ created by the instruction `a.next = a` (N is the object name of `a = new List()`). That is why we have introduced aliases in the preceding step.

5.3 Correctness proof

Definition 5.3.1 (Partial closure). Let $R \in \mathit{DateAlias}, r, d, s$ dates, with $s \leq d \leq r$ (in the following, s will be the date of a call to method m, r at the end of the execution of m, and d during the execution of m), $k \in \mathbb{N} \cup \{\infty\}$.

$$R^k(r/d/s) = \{\pi_1 \sim \pi_p \mid \forall i = 1, ..., p-1, (\pi_i \sim \pi_{i+1}, d_i) \in R,$$
$$s < d_i \leq r, d_1 > d, p \leq k, \text{the } \pi_i \text{ are pairwise distinct}\}$$

This relation only considers aliases created during the execution of m, the first alias being created after d. It uses at most $k-1$ aliases transitively. Limiting to $k-1$ aliases will be useful to make explicit the iteration used to compute the transitive closure of the alias relation. $R^k(r/d/s)$ is right-regular, but not transitive nor symmetric. The condition "π_i pairwise distinct" is necessary so that the condition $d_1 > d$ is really meaningful: otherwise, just take any alias $\pi_1 \sim \pi_2$ created after d and before r, and consider the chain $\pi_1 \sim \pi_2 \sim \pi_1 \sim ... \sim \pi$ then $(\pi_1 \sim \pi) \in R^{k+2}(r/d/s)$ for all $(\pi_1 \sim \pi) \in R^k(r/s/s)$. ($R^0(r/d/s) = \emptyset$, $R^1(r/d/s) = \{\pi \sim \pi\}$, $R^\infty(r/d/s) = \{\pi_1 \sim \pi_p \mid \forall i = 1, ..., p-1, (\pi_i \sim \pi_{i+1}, d_i) \in R, s < d_i \leq r, d_1 > d\}$).

Definition 5.3.2 (Correctness). Let d be a date, m be the method that is being executed at date d in trace T (d can be any date during the execution of

m). We consider the particular execution of m that runs at date d in trace T. Let Res be the object name associated with the result of this execution of m, P_l the object names associated with the parameters of m (the names of these objects in the caller of m, not their name in m), p_l the names of the formal parameters of m, r the date of the end of this execution of m, and s the date of the corresponding call to method m (s is just before the beginning of this execution of m; we have $s < d \leq r$). Assume that m has $j+1$ parameters.

$$Esc(c, c_0, ..., c_j) = \{\text{static}.p' \mid p' \in Path\} \cup \{Res.p' \mid p' \in c\} \cup \cup_l \{P_l.p' \mid p' \in c_l\}$$
$$\text{escap}^k(r/d/s)(N)(c, c_0, ..., c_j) = \{p \mid \exists \pi \in Esc(c, c_0, ..., c_j), (N.p \sim \pi) \in R^k(r/d/s)\}$$
$$\mathbb{E}^k(d)(v) = \text{escap}^k(r/d/s)(\text{Val}^\#(T^\#(d), v))$$
$$\mathbb{E}^k_S(d)(v) = \text{escap}^k(r/s/s)(\text{Val}^\#(T^\#(d), v))$$
$$\begin{aligned}&\text{corr}_{\mathbb{E}}(k,d) \Leftrightarrow \forall c, c_0, ..., c_j \in Ctx_{\mathbb{E}}, \text{ such that } \forall l, c_l \supseteq \mathbb{E}(m)(p_l)(c, c_0, ..., c_j),\\ &\quad \forall v \in V \text{ such that } \text{Val}^\#(T^\#(d), v) \in OName,\\ &\quad \mathbb{E}^k(d)(v)(c, c_0, ..., c_j) \subseteq \mathbb{E}(m)(v)(c, c_0, ..., c_j)\\ &\text{corr}_S(k,d) \Leftrightarrow \forall c, c_0, ..., c_j \in Ctx_{\mathbb{E}}, \text{ such that } \forall l, c_l \supseteq \mathbb{E}(m)(p_l)(c, c_0, ..., c_j),\\ &\quad \forall v \in V \text{ such that } \text{Val}^\#(T^\#(d), v) \in OName,\\ &\quad \mathbb{E}^k_S(d)(v)(c, c_0, ..., c_j) \subseteq \mathbb{E}_S(m)(v)(c, c_0, ..., c_j)\end{aligned}$$

Intuitively, $\text{escap}^k(r/d/s)(N)$ corresponds to the paths representing escaping parts of the object of name N, when only aliases contained in the relation $R^k(r/d/s)$ are taken into account. The only considered alias chains are alias chains of length at most k, created during the execution of m, and whose first element dates from after d. The escaping parts are represented by the paths aliased to paths contained in $Esc(c, c_0, ..., c_j)$ (static fields or parts of the parameters or of the result that are considered as escaping).

Notice that if method m does not terminate, r is not defined. But objects stack allocated in m will never be deallocated and therefore any object can be stack allocated in m. The escape analysis can then yield any result, it will always be correct.

The functions $\mathbb{E}^k(d)(v)$ and $\mathbb{E}^k_S(d)(v)$ give the escape information for variables in the same way as $\text{escap}^k(r/d/s)(N)$ gives the escape information for object names. If variable v contains an object of name N at date d, $\mathbb{E}^k(d)(v) = \text{escap}^k(r/d/s)(N)$ and $\mathbb{E}^k_S(d)(v) = \text{escap}^k(r/s/s)(N)$. $\mathbb{E}^k(d)(v)$ corresponds to the backward analysis $\mathbb{E}$, and considers only aliases whose first element of the chain is created after date d. $\mathbb{E}^k_S(d)(v)$ corresponds to the forward analysis $\mathbb{E}_S$ and all aliases created during the execution of method m are considered. Also notice that s and r can be computed from d: they are the dates at the call and return of the method executed at date d in execution trace T.

The escape functions $\text{escap}^k(r/d/s)$, $\mathbb{E}^k(d)$, $\mathbb{E}^k_S(d)$ are in fact abstractions of $(T^\#, R)$ or even of the execution trace T.

A simplified version of $\text{corr}_{\mathbb{E}}(k,d)$ would be $\forall v \in V, \mathbb{E}^k(d)(v) \leq \mathbb{E}(m)(v)$, that is all objects that escape at date d according to $\mathbb{E}^k(d)$ (i.e. we consider only aliases of $R^k(r/d/s)$) are correctly taken into account by analysis $\mathbb{E}$. However, we have to check that the considered variables really contain objects (hence the condition $\mathrm{Val}^{\#}(T^{\#}(d), v) \in \mathit{OName}$). The condition $\forall l, c_l \supseteq \mathbb{E}(m)(p_l)(c, c_0, ..., c_j)$ will be necessary to prove that the escaping parts c_l for analysis $\mathbb{E}_\mathrm{S}$ at the entry point of method m are correct. The intuition for corr_S is similar, but this time, we check that aliases in $R^k(r/s/s)$ are correctly taken into account by $\mathbb{E}_\mathrm{S}$.

Analyses $\mathbb{E}$ and $\mathbb{E}_\mathrm{S}$ are correct with respect to $R \in \mathit{DateAlias}$ if and only if for all d, $\mathbb{E}_\mathrm{S}$ and $\mathbb{E}$ are correct at date d when composing an unbounded number of aliases transitively: $\mathrm{corr}_{\mathbb{E}\mathrm{S}} \Leftrightarrow \forall d, \mathrm{corr}_\mathrm{S}(\infty, d)$ and $\forall d, \mathrm{corr}_\mathbb{E}(\infty, d)$.

Theorem 5.3.3 (Additivity). *The functions* $\phi = \mathrm{escap}^k(r/d/s)(N)$, $\mathbb{E}^k(d)(v)$ *and* $\mathbb{E}^k_\mathrm{S}(d)(v)$ *are additive, i.e.* $\forall c, c_0, ..., c_j, \forall c', c'_0, ..., c'_j \in \mathit{Ctx}_\mathbb{E}, \phi(c, c_0, ..., c_j) \sqcup \phi(c', c'_0, ..., c'_j) = \phi(c \sqcup c', c_0 \sqcup c'_0, ..., c_j \sqcup c'_j)$.

Proof. This comes immediately from the additivity of *Esc*. □

The escape analyses $\mathbb{E}(m)(v)$ and $\mathbb{E}_\mathrm{S}(m)(v)$ and their abstractions that will be defined later are also additive. This will simplify their general form, making the implementation much easier. This theorem shows that the additivity of $\mathbb{E}$ and $\mathbb{E}_\mathrm{S}$ does not come from approximations, but is an intrinsic property of the escape analysis as defined above.

Lemma 5.3.4. *Let* m *be a method,* $v \in V$ *a local variable of* m *whose definition site is reachable (i.e. may be executed). Let* $c, c_0, ..., c_j \in \mathit{Ctx}_\mathbb{E}$ *such that* $\forall i \in [0, j], c_i \supseteq \mathbb{E}(m)(p_i)(c, c_0, ..., c_j)$ *where* $p_0, ..., p_j$ *are the parameters of* m*. Then*

$$\mathbb{E}_\mathrm{S}(m)(v)(c, c_0, ..., c_j) \supseteq \mathbb{E}(m)(v)(c, c_0, ..., c_j).$$

Proof. (sketch) We consider the control-flow graph G of m, that is the graph whose nodes are instructions, and that contains an edge $i_1 \to i_2$ if and only if i_2 may be executed immediately after i_1. For a method call $w = v_0.m'(v_1, ..., v_n)$, we have an edge from the instruction before the call towards the beginning of all methods that may be called, and an edge from the return of these methods to the call instruction $w = v_0.m'(v_1, ..., v_n)$.

We prove the following property by induction on k: for all variables v whose definition site can be reached from the starting point of m by a path of at most k edges in the control-flow graph, $\mathbb{E}_\mathrm{S}(m)(v)(c, c_0, ..., c_j) \supseteq \mathbb{E}(m)(v)(c, c_0, ..., c_j)$.

This is done by cases on the definition of v. □

The condition $\forall i \in [0, j], c_i \supseteq \mathbb{E}(m)(p_i)(c, c_0, ..., c_j)$ is necessary to have the above property. Indeed, $\mathbb{E}_\mathrm{S}(m)(p_i)(c, c_0, ..., c_j) = c_i \supseteq \mathbb{E}(m)(p_i)(c, c_0, ..., c_j)$.

This condition will also always be satisfied. If m is the current scope, $c = c_0 = ... = c_j = \mathit{Path}$, and the condition is satisfied. If the current scope is a caller m' of m: $w = v_0.m(v_1, ..., v_j)$, $c_i = \mathbb{E}_\mathrm{S}(m')(v_i)(\mathit{Path}, ..., \mathit{Path}) \supseteq \mathbb{E}(m')(v_i)(\mathit{Path}, ..., \mathit{Path}) \supseteq \rho(m)(i)(c, c_0, ..., c_j) = \mathbb{E}(m)(p_i)(c, c_0, ..., c_j)$ (using the above lemma to prove $\mathbb{E}_\mathrm{S}(m')(v_i)(\mathit{Path}, ..., \mathit{Path}) \supseteq \mathbb{E}(m')(v_i)(\mathit{Path}, ..., \mathit{Path})$). Otherwise, the current scope is a caller of a caller of m, and so on.

Slightly abusing notations, we could therefore write: $\forall m, \forall v \in V, \mathbb{E}_\mathrm{S}(m)(v) \geq \mathbb{E}(m)(v)$.

Theorem 5.3.5 (Correctness). $\mathbb{E}$ *and* $\mathbb{E}_\mathrm{S}$ *are correct:* $\mathrm{corr}_{\mathbb{E}\mathrm{S}}$.

PROOF. We only give a proof sketch here. More details can be found in Appendix B. We show

(1) $\forall d, \mathrm{corr}_{\mathrm{S}}(0,d), \mathrm{corr}_{\mathbb{E}}(0,d)$. Obvious, since $R^0(r/d/s) = \emptyset$.
(2) $(\forall d, \mathrm{corr}_{\mathrm{S}}(k,d) \wedge \mathrm{corr}_{\mathbb{E}}(k,d)) \Rightarrow (\forall d, \mathrm{corr}_{\mathbb{E}}(k+1,d))$. By backward induction on d.
(3) $(\forall d, \mathrm{corr}_{\mathbb{E}}(k,d)) \Rightarrow (\forall d, \mathrm{corr}_{\mathrm{S}}(k,d))$. By forward induction on d.
(4) $\forall k, \mathrm{corr}_{\mathbb{E}}(k,d) \Rightarrow \mathrm{corr}_{\mathbb{E}}(\infty,d)$, $\forall k, \mathrm{corr}_{\mathrm{S}}(k,d) \Rightarrow \mathrm{corr}_{\mathrm{S}}(\infty,d)$. Obvious since $\cup_{k\in\mathbb{N}} R^k(r/d/s) = R^\infty(r/d/s)$.

The result follows by induction on k. We find here two iterations: iteration over the number k of aliases transitively composed and over the date d. □

THEOREM 5.3.6 (STACK ALLOCATION). *Consider v = `new` C/`new` $t[w]$ in method m, allocating object o. If $\epsilon \notin \mathbb{E}(m)(v)(Path, Path, ..., Path)$, then o can be stack allocated.*

PROOF. Consider one execution of method m in trace T. Assume that this execution of m returns at date r. Let N be the name associated with object o in this execution of m. We show that $\forall \pi \in Esc, (N.\epsilon \sim \pi) \notin R(r)$ thanks to the correctness of $\mathbb{E}$ with $c = c_0 = ... = c_j = Path$: $\{p \mid \exists \pi \in Esc, (N.p \sim \pi) \in R^\infty(r/d/s)\} \subseteq \mathbb{E}(m)(v)(Path, Path, ..., Path)$. Noticing that N is a new name at date d, the aliases of N are the same in $R^\infty(r/d/s)$ and in $R(r)$. Therefore $\epsilon \notin \{p \mid \exists \pi \in Esc, (N.p \sim \pi) \in R(r)\}$. Then Theorem 4.3 proves that o can be stack allocated in the execution of m that returns at date r in trace T. Since this is true for all executions of m in all traces, we have the required result. Intuitively, we assume that all parameters and the result of m escape (their context is *Path*), and object o does not escape (its context does not contain ϵ). □

Example 5.3.7. In Example 2.1, the analysis finds that `local` can be stack allocated:

$$\mathbb{E}(\texttt{run})(\texttt{local})(c) = \texttt{elem}.c, \text{ so } \epsilon \notin \mathbb{E}(\texttt{run})(\texttt{local})(Path).$$

Escape analysis is not control-flow sensitive, or more precisely, it is as flow-sensitive as the SSA form (its result does not depend on the order of assignments to fields of objects, but it depends on the order of assignments to local variables; the control flow is only used to compute the SSA form). Taking the control-flow more precisely into account would require a much more costly analysis such as [Whaley and Rinard 1999; Choi et al. 1999]. Experiments have shown that this does not prevent our analysis from giving precise results.

Example 5.3.8. When there is an assignment `x.f = y` in method m, we have: $\mathbb{E}(m)(\texttt{y}) \geq \texttt{f}^{-1}.\mathbb{E}_{\mathrm{S}}(m)(\texttt{x})$, which expresses that when `x.f` escapes, `y` escapes, and $\mathbb{E}(m)(\texttt{x}) \geq \texttt{f}.\mathbb{E}_{\mathrm{S}}(m)(\texttt{y})$, which expresses that if `y` escapes, `x.f` escapes. The first equation seems fairly natural, but the second one may be surprising at first sight. It is however necessary, as the following example shows:

```
t x = new t(); t' y = new t'(); t'' z = new t''();
C.static_field = y;
x.f = y;
x.f.f' = z;
```

First notice that `y` escapes in the static field `C.static_field`. When executing the assignment `x.f = y`, without this second equation $\mathbb{E}(m)(\mathtt{x}) \geq \mathtt{f}.\mathbb{E}_S(m)(\mathtt{y})$, the $\mathbb{E}$ and $\mathbb{E}_S$-escaping parts of `x` would be empty. Therefore we would think that `z` does not escape, which is wrong. In effect, the above program stores `z` in `y` (it is equivalent to `y.f' = z`) and since `y` escapes, `z` also escapes.

Because of these two equations, as soon as an object `y` is stored in an object `x` by `x.f = y`, the escaping part $\mathbb{E}(m)(\mathtt{y})$ is equal to $\mathbb{E}_S(m)(\mathtt{y})$. Indeed, using notations of the assignment at the beginning of this example, and assuming that $\forall l, c_l \supseteq \mathbb{E}(m)(p_l)(c, c_0, ..., c_j)$,

$$\begin{aligned} \mathbb{E}(m)(\mathtt{y})(c, c_0, ..., c_j) &\supseteq \mathtt{f}^{-1}.\mathbb{E}_S(m)(\mathtt{x})(c, c_0, ..., c_j) \\ &\supseteq \mathtt{f}^{-1}.\mathbb{E}(m)(\mathtt{x})(c, c_0, ..., c_j) && \text{by Lemma 5.3.4.} \\ &\supseteq \mathbb{E}_S(m)(\mathtt{y})(c, c_0, ..., c_j) \\ \text{and } \mathbb{E}_S(m)(\mathtt{y})(c, c_0, ..., c_j) &\supseteq \mathbb{E}(m)(\mathtt{y})(c, c_0, ..., c_j) && \text{again by Lemma 5.3.4.} \end{aligned}$$

Similarly, for the object `x` in which we store, we have

$$\mathtt{f}^{-1}.\mathbb{E}(m)(\mathtt{x})(c, c_0, ..., c_j) = \mathtt{f}^{-1}.\mathbb{E}_S(m)(\mathtt{x})(c, c_0, ..., c_j).$$

This might suggest that we could let $\mathbb{E} = \mathbb{E}_S$ without losing much precision. But contrary to what one could think at first sight, this would not much simplify the analysis. This would divide by two the number of unknowns, and remove the equations $\mathbb{E}_S(m)(v) = \mathbb{E}(m)(v)$ for instructions $v = \mathtt{null}$, $v = \mathtt{new}\ C$, and $v = \mathtt{new}\ t[w]$, and also equations for $v_1 = v_2$, $v_1 = (t)v_2$, $v_1 = \phi(v_2, v_3)$ if the same unknown is used for the escape information of v_1, v_2, v_3. All other equations have to be kept, and the bidirectional propagation of abstract values is still maintained. The following table gives examples of equations that we would obtain in this case (in this table, $L = \mathbb{E}(m) = \mathbb{E}_S(m)$):

P(pc)	Forward propagation	Backward propagation
entry of m	$L \geq P_{\mathbb{E}}$	$\forall i \in [0, j], \rho(m)(i) \geq L(p_i)$
$v_1 = v_2 / v_1 = (t)v_2$	$L(v_1) = L(v_2)$	
$v_1 = \phi(v_2, v_3)$	$L(v_1) = L(v_2) = L(v_3)$	
$v_1 = v_2.f$	$L(v_1) \geq f^{-1}.L(v_2)$	$L(v_2) \geq f.L(v_1)$
...	...	...

Moreover, using two distinct analyses $\mathbb{E}$ and $\mathbb{E}_S$ can substantially improve the precision of the escape information of `x` when no assignment to a field or array element of `x` is done. Consider for example the following code, similar to that of Ruf [2000, Section 3.5.7], in a method m, with $L = \mathbb{E}(m)$, $L_S = \mathbb{E}_S(m)$:

Code	Analyses $\mathbb{E}_S$ and $\mathbb{E}$
`obj1 = ...`	
`if (obj1 == null)`	
`obj2 = C.f;`	$L_S(\mathtt{obj2}) = \top_{\mathbb{E}}[\mathtt{f}]$
`obj = ` ϕ`(obj1,obj2)`	$L_S(\mathtt{obj}) = L_S(\mathtt{obj1}) \sqcup L_S(\mathtt{obj2})$
	$L(\mathtt{obj1}) \geq L(\mathtt{obj}), L(\mathtt{obj2}) \geq L(\mathtt{obj})$
... (no field or array assignment)	

In this example, when `obj1` is `null`, a default value taken from a static field is used, hence this default value escapes, and `obj` may escape: $L_S(\mathtt{obj}) = \top_{\mathbb{E}}[\mathtt{f}]$.

However, when no field or array assignment involving obj or parts of obj is done, the value of $L_{\mathrm{S}}(\texttt{obj})$ is not used, hence the equations $L_{\mathrm{S}}(\texttt{obj2}) = \top_{\mathbb{E}}[\texttt{f}]$, $L_{\mathrm{S}}(\texttt{obj}) = L_{\mathrm{S}}(\texttt{obj1}) \sqcup L_{\mathrm{S}}(\texttt{obj2})$ do not change the value of $L(\texttt{obj1})$. Therefore, we can find that obj1 does not escape if the end of the code does not make obj escape. In contrast, if we set $\mathbb{E}_{\mathrm{S}} = \mathbb{E}$, we meet the same problem as in Ruf's analysis: we think that obj1 always escapes ($L(\texttt{obj1}) \geq L(\texttt{obj}) = L_{\mathrm{S}}(\texttt{obj}) = \top_{\mathbb{E}}[\texttt{f}]$). Distinguishing $\mathbb{E}_{\mathrm{S}}$ and $\mathbb{E}$ in our analysis partially solves this problem, when no field or array assignment involving obj is performed.

6. ANALYSIS $\mathbb{L}$: ESCAPE ANALYSIS USING INTEGER CONTEXTS

In analysis $\mathbb{L}$, the escaping part of an object is represented by an integer. So the corresponding set of contexts is: $Ctx_{\mathbb{L}} = \mathbb{N}$, ordered by the natural ordering. Integers can be manipulated much faster than sets of paths, therefore this will lead to an efficient escape analysis. These integers are defined from types: we first define the height $\top[\tau]$ of a type τ (see below; Figure 8 gives an example), and let the escape context of a value be the height of the type of its escaping part. Analysis $\mathbb{L}$ computes for each variable, its escape context. This is very similar to analysis $\mathbb{E}$, except that sets of access paths are replaced by integers. When the escape context for an object of type τ is strictly smaller than $\top[\tau]$, the object can be stack allocated. (The object cannot escape, since its escaping part has a height smaller than $\top[\tau]$.)

6.1 Abstraction from sets of paths to integers

The height of a type is defined as follows. For each object type $\tau \in \mathit{RefType}$, let $\mathrm{Cont}(\tau)$ be the set of the types that τ contains (types of the fields, or type of elements if τ is an array). The height $\top[\tau]$ of an object type τ is the greatest context used for objects of type τ. Let $\top[\tau]$ be the smallest integer such that:

$$\top[\tau] \geq 1 \tag{1}$$

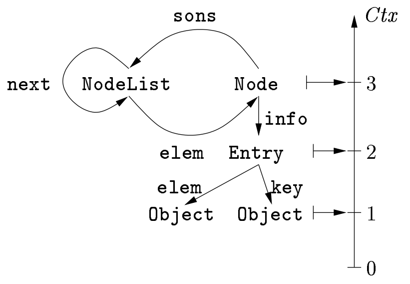
$$\text{if } \tau' \in \mathrm{Cont}(\tau), \top[\tau'] \leq \top[\tau] \tag{2}$$

$$\text{if } \tau' \text{ is a subtype of } \tau, \text{ and } \tau \text{ is not } \texttt{Object}, \top[\tau'] \leq \top[\tau] \tag{3}$$

$$\text{if that does not contradict (2) and (3), if } \tau' \in \mathrm{Cont}(\tau), 1 + \top[\tau'] \leq \top[\tau] \tag{4}$$

Only Rule (2) is necessary for the correctness of the analysis. Rules (3) and (4) are useful to get the best possible precision. Rule (3) avoids losing precision in type conversions (when we have a context represented with respect to one type, and we want to represent it with respect to another type). Indeed, type conversions occur in general between a type τ and one of its subtypes τ'. Thanks to Rule (3), these types have in general the same height, and the type conversion is the identity. Rule (4) distinguishes as many levels as possible. The more levels we distinguish, the more precise the analysis is (but also the more costly).

Example 6.1. Consider the types of Figure 8, and assume that no other subtypes are defined. For Entry, there is no cycle in the Cont relation (i.e. no object may contain an object of the same type), so (4) always applies, and the height of a type is 1 plus the maximum height of the types it contains. For Node, there is a strongly connected component in the graph of the Cont relation, which contains NodeList and Node since $\mathrm{Cont}(\texttt{Node}) = \{\texttt{Entry}, \texttt{NodeList}\}$, and




```
class Entry {
  Object elem; Object key;
}
class Node {
  NodeList sons; Entry info;
}
class NodeList {
  NodeList next; Node elem;
}
```

Fig. 8. Type heights. An arrow $\tau \to \tau'$ means $\tau' \in \mathrm{Cont}(\tau)$ i.e. τ has a field of type τ'.

$\mathrm{Cont}(\texttt{NodeList}) = \{\texttt{Node}, \texttt{NodeList}\}$. According to (2), $\top[\texttt{Node}] \geq \top[\texttt{NodeList}] \geq \top[\texttt{Node}]$, so we cannot apply (4) between `NodeList` and `Node`. (4) yields $\top[\texttt{Node}] \geq \top[\texttt{Entry}] + 1 = 3$, so $\top[\texttt{Node}] = \top[\texttt{NodeList}] = 3$.

The escaping part of an object will be represented by the height of its type (in Example 2.1 for instance, since `local.elem` escapes in `BinTree.run`, and has type `Object`, the context for `local` will be $\top[\texttt{Object}] = 1$). We do not directly compute types, because computing integers will lead to a faster analysis. However, with integers, we cannot in general distinguish different fields of the same object, therefore we lose some precision with this representation. The information for the top of data structures can however be precisely represented. This is very important for stack allocation, since the top of data structures is the part that can most often be stack allocated. The more we go inside complex data structures, the more we lose precision, but also the less stack allocation is probable anyway, even with a precise analysis. This representation is therefore well adapted to stack allocation. The experiments confirm that, even with such an approximation, the analysis is still precise.

We now define an abstraction from sets of paths to integers. The height $\top[\tau.p]$ of a path p in an object of type τ is intuitively the height of the type τ' of the value accessed at the end of the path (that is, the type of the value that escapes). For instance, using the types of Figure 8,

$$\top[\texttt{Node}.(\texttt{Node}, \texttt{info}, \texttt{Entry}).(\texttt{Entry}, \texttt{key}, \texttt{Object})] = \top[\texttt{Object}] = 1$$

since accessing the fields `info` and `key` in a value of type `Node` leads to a value of type `Object`. Similarly, $\top[\tau.\epsilon] = \top[\tau]$, since the access by the empty path to a value of type τ yields a value of type τ. Note that, in this case, the value of τ is important to define the height, since no type is mentioned in the path ϵ. In contrast, τ seemed unnecessary in the first case: The type `Object` was already indicated in the field $(\texttt{Entry}, \texttt{key}, \texttt{Object})$. The type τ is also necessary for the same reason to define the height of paths that contain only array accesses: $\top[\tau.n_1 \dots n_n]$. Also note that we have to define $\top[\tau.p]$ not only when p is a statically valid path for a value of type τ, but for any path. For instance, we have to define $\top[\texttt{Object}.(\texttt{Node}, \texttt{info}, \texttt{Entry})]$, since a value that is believed to be of type `Object`, can in fact be of type `Node` (a subtype of τ can appear when τ is expected). This complicates a bit the following definition.

We denote by $\sqcap$ the minimum of integers, and by $\sqcup$ the maximum. Formally, the definition of $\top[\tau.p]$ is then:

$$\top[t\,\texttt{[]}.n.p] = \top[t.p] \qquad (n \in \mathbb{N}) \tag{5}$$

$$\top[t.n.p] = \top[t.p] \text{ if } t \text{ is not an array } (n \in \mathbb{N}) \tag{6}$$

$$\top[t.(C, f, t').p] = \top[t] \sqcap \top[t'.p] \tag{7}$$

$$\top[t.\epsilon] = \top[t] \tag{8}$$

Equation (6) may seem strange, since we use an object of type t, which is not an array, as if it were one (data accessor $n \in \mathbb{N}$). It may indeed happen if we have type conversions. For example, consider a field `l` of type `Object`, and assume that `this.l` is accessed by the expression `((int[]) this.l)[n]`. The escaping part of `l` is then represented by a path $\texttt{n}.p$. The field `l` is an array at runtime, but its static type is $t = \texttt{Object}$. For (7), a more intuitive definition would be $\top[t.(C, f, t').p] = \top[t'.p]$ but the previous definition is necessary to make sure that $\top[t.p] \leq \top[t]$. Indeed, we have chosen that $\top[\tau.\epsilon] = \top[\tau]$, and the path ϵ corresponds to the maximum escaping part: the whole value escapes. Therefore, the escape context for all other paths should be smaller than $\top[\tau]$.

The abstraction $\alpha^\tau : Ctx_{\mathbb{E}} \to Ctx_{\mathbb{L}}$ is $\alpha^\tau(c) = \sqcup\{\top[\tau.p] \mid p \in c\}$ and the concretization $\gamma^\tau : Ctx_{\mathbb{L}} \to Ctx_{\mathbb{E}}$ is $\gamma^\tau(c') = \{p \mid \top[\tau.p] \leq c'\}$. $Ctx_{\mathbb{L}}^\tau = \alpha^\tau(Ctx_{\mathbb{E}}) = [0, \top[\tau]]$. $Ctx_{\mathbb{E}} \underset{\alpha^\tau}{\overset{\gamma^\tau}{\leftrightarrows}} Ctx_{\mathbb{L}}^\tau$ is a Galois connection.

Example 6.2. If we consider again the types of Figure 8, we have

$$\begin{aligned}&\alpha^{\texttt{Node}}(\{(\texttt{Node}, \texttt{info}, \texttt{Entry}).(\texttt{Entry}, \texttt{key}, \texttt{Object})\})\\ &\quad = \top[\texttt{Node}] \sqcap \top[\texttt{Entry}] \sqcap \top[\texttt{Object}] = \top[\texttt{Object}] = 1.\end{aligned}$$

We also have

$$\alpha^{\texttt{Object}}(\{(\texttt{Node}, \texttt{info}, \texttt{Entry})\}) = \top[\texttt{Object}] \sqcap \top[\texttt{Entry}] = \top[\texttt{Object}] = 1.$$

In this case, the height is not the height of the last element of the path. It is limited to $\top[\texttt{Object}]$, since the context $\{(\texttt{Node}, \texttt{info}, \texttt{Entry})\}$ represents only a part of the object, therefore the height of this context should be smaller than the height of the whole value $\top[\texttt{Object}]$.

Intuitively, if a value escapes, all values contained in it also escape. More formally, if we define closed contexts by $c \in Ctx_{\mathbb{E}}$ is closed if and only if $\forall p \in c, \forall p' \in Path, p.p' \in c$, all contexts interesting for escape analysis are closed. Moreover, all contexts that can be represented by integers are closed: $\gamma^\tau(c')$ is always closed. This way, our analysis does not carry useless information.

6.2 Abstract values and operators

As in analysis $\mathbb{E}$, abstract values are context transformers. But Java supports type casts and subtyping, so the static type of the same object may not be the same during the whole runtime, and we have to remember the assumed type of objects with each context transformer. Therefore, abstract values are now $Val_{\mathbb{L}} = \cup_{n\in\mathbb{N}}((Ctx_{\mathbb{L}}^n \to Ctx_{\mathbb{L}}) \times Type^n \times Type)$. Abstract values are $\mathbb{L}(m)(v) = (\phi, (\tau_{-1}, ..., \tau_j), \tau') \in Val_{\mathbb{L}}$, where $(\tau_{-1}, ..., \tau_j)$ are the statically declared types of the result and the parameters of the method m, and τ' is the type that the analysis assumes for the object contained in v.

An abstract value $v = (\phi, (\tau_{-1}, ..., \tau_j), \tau') \in \mathit{Val}_{\mathbb{L}}$ is correct with respect to $E \in \mathit{Val}_{\mathbb{E}}$ if and only if

$$\mathrm{corr}_{\mathbb{L}}(v, E) \Leftrightarrow \forall c_{-1}, ..., c_j, \phi(c_{-1}, ..., c_j) \geq \alpha^{\tau'}(E(\gamma^{\tau_{-1}}(c_{-1}), ..., \gamma^{\tau_j}(c_j))).$$

We have seen that each context in $\mathit{Ctx}_{\mathbb{L}}$ is associated with a type. We define a conversion operation $\mathrm{convert}(\tau, \tau')$ to convert a context computed for one type τ to another type τ'. In the language, type conversions appear explicitly when using $v_1 = (t)v_2$ to convert types, or implicitly when using a subtype as a supertype (in method calls or field accesses). In the analysis, all type conversions are delayed until necessary. Indeed, we can freely choose the type τ that is used to represent a context: if $c \in \mathit{Ctx}_{\mathbb{E}}$, then $\alpha^{\tau}(c)$ is a correct representation of c associated with type τ. The conversion enables us to change the type as we wish. However, type conversions can lead to a loss of precision: for example, consider an object of type `Node`, for which only the `Entry info` field escapes, hence the escape context is $2 = \top[\texttt{Entry}]$. If we convert the context to `Object`, then we will think that the whole object escapes (context $1 = \top[\texttt{Object}]$). Therefore, we should avoid useless type conversions. Along the same ideas, we can also see that the precision of the escape information depends on the type that is used to represent it: the escape information is less precise when represented using the `Object` type than using the `Node` type for instance (only two levels can be distinguished with `Object`, whereas four levels are distinguished with `Node`). Ideally, the best precision would be achieved by using the dynamic type of the object to represent the escape information, since the dynamic type remains constant for the same object during the whole execution, so no type conversions would be needed. The dynamic type represents exactly the accesses that can actually be done on the object. However, using the dynamic type is impossible in the analysis since the dynamic type cannot be determined statically, and several objects of different dynamic types can appear in the same variable during the execution of the program. Nevertheless, we do not systematically convert to the static type, to reduce the loss of precision coming from type conversions. We will come back to this treatment after the definition of abstract operations.

The conversion $\mathrm{convert}(\tau, \tau') : \mathit{Ctx}_{\mathbb{L}}^{\tau} \to \mathit{Ctx}_{\mathbb{L}}^{\tau'}$ is defined by $\mathrm{convert}(\tau, \tau') = \alpha^{\tau'} \circ \gamma^{\tau}$. We can compute a more explicit formula:

Lemma 6.3. *Let $t\,[]^k$ be the type $t\,[]...[]$ with k times $[]$. Assume that τ and τ' are array types with k and k' dimensions, $\tau = t\,[]^k$, $\tau' = t'\,[]^{k'}$, t and t' are not arrays.*

$$\mathrm{convert}(\tau, \tau')(c) = (c \sqcap \top[\tau']) \sqcup \begin{cases} \top[t'\,[]^{k'-i}] & \textit{if } i \geq 0 \textit{ minimum such that } i \leq k, i \leq k' \textit{ and } c = \top[t\,[]^{k-i}], \\ \top[t'] & \textit{if } k > k' \textit{ and } \top[t] \leq c < \top[t\,[]^{k-k'}], \\ 0 & \textit{if } c < \top[t]. \end{cases}$$

This result is proved in Appendix C. Intuitively, the context c corresponds, for analysis $\mathbb{E}$, to a certain set of access paths $\gamma^{\tau}(c)$. Each part of the formula handles a certain kind of access paths. The first part $c \sqcap \top[\tau']$ deals with access paths that contain at least a field access. Fields (C, f, t'') such that $\top[t'']$ is at

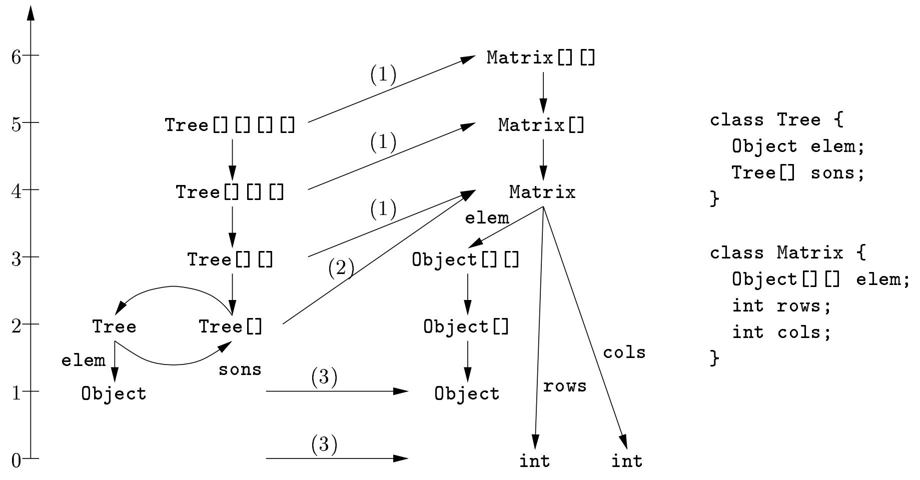


Fig. 9. Example of conversion: convert(`Tree[][][][]`,`Matrix[][]`). The arrows labeled by numbers between parentheses represent the conversion function. The number indicates which of the three cases of the formula of convert applies.

most c can escape, and this is also true after conversion to type τ', except that the escaping height is at most $\top[\tau']$. The second part, containing three cases, deals with access paths containing only array accesses (including the empty path ϵ). The first case corresponds to the situation when the escaping part of the value of type τ is obtained by i array accesses ($c = \top[t\texttt{[]}^{k-i}]$), and the type τ' has at least i dimensions. Then, after the conversion, the escape context is obtained by the same number i of array accesses, hence it is $\top[t'\texttt{[]}^{k'-i}]$. The second case is when the escaping part is obtained by i array accesses, but τ' has less than i dimensions. In this case, the value of type t', obtained after as many array accesses as possible in τ', is considered as escaping. At last, in the third case, no access path containing only array accesses can appear in the context, because c is smaller than $\top[t]$. In this case, array accesses have no contribution to the result of the conversion, hence the value 0.

Example 6.4. An example of conversion is given in Figure 9. (This is an unlikely case, but it illustrates all the details of the conversion function.) Instantiating the formula with τ = `Tree[][][][]`, τ' = `Matrix[][]`, $k = 4$, $k' = 2$, t = `Tree`, t' = `Matrix`, we obtain $\text{convert}(\tau,\tau')(5) = 6$, $\text{convert}(\tau,\tau')(4) = 5$, $\text{convert}(\tau,\tau')(3) = 4$ by the first case with $i = 0, 1, 2$ respectively, $\text{convert}(\tau,\tau')(2) = 4$ by the second case, $\text{convert}(\tau,\tau')(1) = 1$, $\text{convert}(\tau,\tau')(0) = 0$ by the third case.

When τ and τ' are not array types, $k = k' = 0$, so only the first and third cases of the formula of convert can apply, and in the first case, $i = 0$. Then, when τ and τ' are not array types, we have:

$$\text{convert}(\tau,\tau')(c) = (c \sqcap \top[\tau']) \sqcup (\text{if } c = \top[\tau] \text{ then } \top[\tau'] \text{ else } 0).$$

This even reduces to:

$$\begin{cases}\text{If } \top[\tau'] \le \top[\tau], \text{convert}(\tau,\tau')(c) = c \sqcap \top[\tau'].\\ \text{Otherwise, convert}(\tau,\tau')(c) = (\text{if } c = \top[\tau] \text{ then } \top[\tau'] \text{ else } c).\end{cases}$$

The difference between array and object types for the conversion comes from the fact that converting object types means adding or removing fields, whereas converting array types means converting the type of their elements.

Let $v = (\phi, (\tau_{-1}, ..., \tau_j), \tau')$, $v_k = (\phi_k, (\tau_{-1}, ..., \tau_j), \tau'_k)$. We can now define the following abstract ordering and operations:

(1) Abstract values are ordered by the following preorder relation[4]: $(\phi, (\tau_{-1}, ..., \tau_j), \tau) \le (\phi', (\tau'_{-1}, ..., \tau'_j), \tau') \Leftrightarrow \forall c_{-1} \in [0, \top[\tau'_{-1}]], ..., \forall c_j \in [0, \top[\tau'_j]], \text{convert}(\tau, \tau')(\phi(\text{convert}(\tau'_{-1}, \tau_{-1})(c_{-1}), ..., \text{convert}(\tau'_j, \tau_j)(c_j))) \le \phi'(c_{-1}, ..., c_j)$. $v_1 \le v_2$ if and only if v_1 gives a more precise escape information than v_2.

(2) Upper bound: $v_1 \sqcup_{\mathbb{L}} v_2 = ((\text{convert}(\tau'_1, \tau'_2) \circ \phi_1) \sqcup \phi_2, (\tau_{-1}, ..., \tau_j), \tau'_2)$ if $\top[\tau'_2] > \top[\tau'_1]$. Otherwise, we swap the indices 1 and 2.

(3) Construction: $(C, f, t)_{\mathbb{L}}.v = (\text{convert}(\tau', t) \circ \phi, (\tau_{-1}, ..., \tau_j), C)$.
$\mathbb{N}_{\mathbb{L}}.v = (\phi, (\tau_{-1}, ..., \tau_j), \tau'\texttt{[]})$.

(4) Restriction: $(C, f, t)_{\mathbb{L}}^{-1}.v = ((\text{convert}(\tau', C) \circ \phi) \sqcap \top[t], (\tau_{-1}, ..., \tau_j), t)$.
$\mathbb{N}_{\mathbb{L}}^{-1}.v = (\phi \sqcap \top[t], (\tau_{-1}, ..., \tau_j), t)$ if $\tau' = t\texttt{[]}$. Otherwise, $\mathbb{N}_{\mathbb{L}}^{-1}.v = ((\text{convert}(\tau', t\texttt{[]}) \circ \phi) \sqcap \top[t], (\tau_{-1}, ..., \tau_j), t)$ where t can be arbitrarily chosen. (In the implementation, $t = \texttt{Object}$.)

(5) Composition: $(\phi', (\tau''_{-1}, ..., \tau''_i), \tau''') \circ_{\mathbb{L}} (v_{-1}, ..., v_i) = (\phi' \circ (\text{convert}(\tau'_{-1}, \tau''_{-1}) \circ \phi_{-1}, ..., \text{convert}(\tau'_i, \tau''_i) \circ \phi_i), (\tau_{-1}, ..., \tau_j), \tau''')$.

If the analyzed method has j parameters of types $\tau_0, ..., \tau_j$ and a result of type τ_{-1}, $\top_{\mathbb{L}}[(C, f, t)] = (\{(c_{-1}, ..., c_j) \mapsto \top[t]\}, (\tau_{-1}, ..., \tau_j), t)$. $\mathit{first}_{\mathbb{L}} = (\{(c_{-1}, ..., c_j) \mapsto c_{-1}\}, (\tau_{-1}, ..., \tau_j), \tau_{-1})$. $\forall i \in [0, j], P_{\mathbb{L}}(p_i) = (\{(c_{-1}, ..., c_j) \mapsto c_i\}, (\tau_{-1}, ..., \tau_j), \tau_i)$.

Let us explain how these definitions have been built, on the example of the restriction for objects. $(C, f, t)_{\mathbb{L}}^{-1}.v$ is used for example when analyzing the instruction $v_1 = v_2.f$: $\mathbb{L}_{\mathbb{S}}(m)(v_1) = (C, f, t)_{\mathbb{L}}^{-1}.\mathbb{L}_{\mathbb{S}}(m)(v_2)$. v_2 should be of type C (or a subtype of C), and v_1 of type t (or a subtype of t). For a first step, we assume that the types are exactly as expected, that is, v_2 is of type C, $\mathbb{L}_{\mathbb{S}}(m)(v_2) = (\phi, (\tau_{-1}, ..., \tau_j), C)$, v_1 is of type t, $\mathbb{L}_{\mathbb{S}}(m)(v_1) = (\phi', (\tau_{-1}, ..., \tau_j), t)$. Assume that the escape context of v_2 is greater than $\top[t]$, then the whole field f of type t escapes, therefore v_1 escapes entirely, hence the escape context of v_1 must be $\top[t]$. Assume on the contrary, that the escape context c of v_2 is smaller than $\top[t]$. Then a certain part of the field f of type t escapes (this part has height c), and the same part of $v_1 = v_2.f$ escapes, hence the escape context of v_1 is also c. These two cases can be reconciled by taking $\phi' = \phi \sqcap \top[t]$. Therefore, we obtain the formula:

$$(C, f, t)_{\mathbb{L}}^{-1}.(\phi, (\tau_{-1}, ..., \tau_j), C) = (\phi \sqcap \top[t], (\tau_{-1}, ..., \tau_j), t).$$

When the types are not exactly C and t, we use type conversions, hence the general formula is:

$$(C, f, t)_{\mathbb{L}}^{-1}.(\phi, (\tau_{-1}, ..., \tau_j), \tau') = ((\text{convert}(\tau', C) \circ \phi) \sqcap \top[t], (\tau_{-1}, ..., \tau_j), t).$$

[4]reflexive and transitive but not necessarily antisymmetric

```
if (...)
  NodeList a1 = new NodeList();
else
  Node[] a2 = new Node[10];
Object a = φ(a1,a2);
...
if (a instanceof NodeList) {
  Entry x1 = ((NodeList)a).elem.info;

  MyClass.f = x1;
} else {
  Entry x2 = ((Node[])a)[0].info;

  MyClass.f = x2;
}
```

Analyses $\mathbb{L}_{\mathrm{S}}$ and $\mathbb{L}$
$L_{\mathrm{S}}(\mathtt{a1}) = L(\mathtt{a1})$
$L_{\mathrm{S}}(\mathtt{a2}) = L(\mathtt{a2})$
$L_{\mathrm{S}}(\mathtt{a}) = L_{\mathrm{S}}(\mathtt{a1}) \sqcup L_{\mathrm{S}}(\mathtt{a2})$
$L(\mathtt{a1}) \geq L(\mathtt{a}), L(\mathtt{a2}) \geq L(\mathtt{a})$
$L_{\mathrm{S}}(\mathtt{x1}) = \mathtt{info}_{\mathbb{L}}^{-1}.\mathtt{elem}_{\mathbb{L}}^{-1}.L_{\mathrm{S}}(\mathtt{a})$
$L(\mathtt{a}) \geq \mathtt{elem}_{\mathbb{L}}.\mathtt{info}_{\mathbb{L}}.L(\mathtt{x1})$
$L(\mathtt{x1}) \geq \top_{\mathbb{L}}[\mathtt{f}]$
$L_{\mathrm{S}}(\mathtt{x2}) = \mathtt{info}_{\mathbb{L}}^{-1}.\mathbb{N}_{\mathbb{L}}^{-1}.L_{\mathrm{S}}(\mathtt{a})$
$L(\mathtt{a}) \geq \mathbb{N}_{\mathbb{L}}.\mathtt{info}_{\mathbb{L}}.L(\mathtt{x2})$
$L(\mathtt{x2}) \geq \top_{\mathbb{L}}[\mathtt{f}]$

Fig. 10. Example of analysis $\mathbb{L}$

Notice that we do not convert t to the static type of v_2: convert operations are only performed when abstract operations require arguments of a certain type. For instance, the restriction above requires an escape information corresponding to type C, so we convert τ' to C. Its output can be of any type, so we leave t as the type of the result. Also notice that, if τ' is a supertype of C, the conversion convert(τ', C) does not change the result. Indeed, $\top[\tau'] \geq \top[t]$ and $\top[C] \geq \top[t]$, since τ' and C have the field f of type t, hence convert$(\tau', C)(c) \sqcap \top[t] = (c \sqcap \top[C] \sqcap \top[t]) \sqcup (\text{if } c = \top[\tau'] \text{ then } \top[C] \sqcap \top[t] \text{ else } 0) = (c \sqcap \top[t]) \sqcup (\text{if } c = \top[\tau'] \text{ then } \top[t] \text{ else } 0) = c \sqcap \top[t]$.

The formulae for other abstract operations are built in a similar way: first, we establish a formula assuming that the types are as expected, then we use type conversions to handle the general case. The upper bound is naturally:

$$(\phi_1, (\tau_{-1}, ..., \tau_j), \tau') \sqcup_{\mathbb{L}} (\phi_2, (\tau_{-1}, ..., \tau_j), \tau') = (\phi_1 \sqcup \phi_2, (\tau_{-1}, ..., \tau_j), \tau')$$

When both values do not have the same type, we convert one of them to the type of the other one. Converting to the highest type improves the precision (intuitively, the higher the type, the more different escape levels we can distinguish). Similarly, the construction for object types is

$$(C, f, t)_{\mathbb{L}}.(\phi, (\tau_{-1}, ..., \tau_j), t) = (\phi, (\tau_{-1}, ..., \tau_j), C)$$

and we convert the escape information of ϕ to type t when it has a different type.

Example 6.5. Let us consider the program extract of Figure 10 (in the Java syntax, except for the use of the SSA form), where `Entry f` is a static field in the class `MyClass`. We assume that this code is in a method m, without parameters and result, and we define $L_{\mathrm{S}} = \mathbb{L}_{\mathrm{S}}(m)$, $L = \mathbb{L}(m)$. We use the type hierarchy of Figure 8.

After some minor simplifications, we obtain $L(\mathtt{a}) \geq \mathtt{elem}_{\mathbb{L}}.\mathtt{info}_{\mathbb{L}}.\top_{\mathbb{L}}[\mathtt{f}]$ and $L(\mathtt{a}) \geq \mathbb{N}_{\mathbb{L}}.\mathtt{info}_{\mathbb{L}}.\top_{\mathbb{L}}[\mathtt{f}]$. We have

$$\top_{\mathbb{L}}[\mathtt{f}] = (() \mapsto \top[\mathtt{Entry}], (), \mathtt{Entry}),$$
$$\mathtt{info}_{\mathbb{L}}.\top_{\mathbb{L}}[\mathtt{f}] = (() \mapsto \top[\mathtt{Entry}], (), \mathtt{Node}),$$
$$\mathtt{elem}_{\mathbb{L}}.\mathtt{info}_{\mathbb{L}}.\top_{\mathbb{L}}[\mathtt{f}] = (() \mapsto \top[\mathtt{Entry}], (), \mathtt{NodeList}),$$

$$\mathbb{N}_{\mathbb{L}}.\mathtt{info}_{\mathbb{L}}.\top_{\mathbb{L}}[\mathtt{f}] = (() \mapsto \top[\mathtt{Entry}], (), \mathtt{Node[]})$$

so $L(\mathtt{a}) = \mathtt{elem}_{\mathbb{L}}.\mathtt{info}_{\mathbb{L}}.\top_{\mathbb{L}}[\mathtt{f}] \sqcup_{\mathbb{L}} \mathbb{N}_{\mathbb{L}}.\mathtt{info}_{\mathbb{L}}.\top_{\mathbb{L}}[\mathtt{f}] = (() \mapsto \top[\mathtt{Entry}], (), \mathtt{Node[]})$ since $\top[\mathtt{Node[]}] = 4 > \top[\mathtt{NodeList}] = 3$. The other equations imply $L_{\mathrm{S}}(\mathtt{a1}) = L(\mathtt{a1}) = L_{\mathrm{S}}(\mathtt{a2}) = L(\mathtt{a2}) = L_{\mathrm{S}}(\mathtt{a}) = L(\mathtt{a})$ so we find that both objects created by this code do not escape.

Note that, if we converted to the static types, we would have the formula:

$$\begin{aligned}&(\phi_1, (\tau_{-1}, ..., \tau_j), \tau_1') \sqcup_{\mathbb{L}} (\phi_2, (\tau_{-1}, ..., \tau_j), \tau_2')\\ &\quad = (\mathrm{convert}(\tau_1', \tau') \circ \phi_1 \sqcup \mathrm{convert}(\tau_2', \tau') \circ \phi_2, (\tau_{-1}, ..., \tau_j), \tau')\end{aligned}$$

where τ' is the least common supertype of τ_1' and τ_2'. This would lead to a very important loss of precision. If we consider the example again, $L(\mathtt{a})$ would be converted to type `Object`: $L(\mathtt{a}) = (() \mapsto 1, (), \mathtt{Object})$, and we would think that `a` escapes entirely, and therefore also `a1` and `a2`: $L(\mathtt{a1}) = (() \mapsto \top[\mathtt{NodeList}], (), \mathtt{NodeList})$, $L(\mathtt{a2}) = (() \mapsto \top[\mathtt{Node[]}], (), \mathtt{Node[]})$. Our more dynamic choice of types therefore improves precision.

Remark. In most cases, the type height is the only information used in the definition of abstract operations. But notice that for the type conversion to or from type $t\mathtt{[]}^k$, $\top[t\mathtt{[]}^i]$ is needed for all $i \leq k$, and these type heights cannot be inferred from $\top[t\mathtt{[]}^k]$: they also depend on the cycles that can appear in the type, as illustrated by type `Tree[]` of Figure 9. This is the reason why we have kept the exact type together with each context, and not only the type height.

THEOREM 6.6 (CORRECTNESS). *Assume that* $\mathrm{corr}_{\mathbb{L}}(v_1, E_1)$ *and* $\mathrm{corr}_{\mathbb{L}}(v_2, E_2)$. *Then* $\mathrm{corr}_{\mathbb{L}}(v_1 \sqcup_{\mathbb{L}} v_2, E_1 \sqcup E_2)$, $\mathrm{corr}_{\mathbb{L}}((C, f, t)_{\mathbb{L}}^{-1}.v_1, (C, f, t)^{-1}.E_1)$, *and similarly for other primitive operations.*

This result is proved in Appendix C.

All definitions of Section 5.2 are still correct for analysis $\mathbb{L}$, provided that we use the new abstract domain for values, with its associated abstract primitives. We get the correctness theorem from Theorems 5.3.6 and 6.6:

THEOREM 6.7 (STACK ALLOCATION). *Consider* $v = \mathtt{new}\ C/\mathtt{new}\ t[w]$ *in method* m, *allocating object* o. *With* $\mathbb{L}(m)(v) = (\phi, (\tau_{-1}, ..., \tau_j), \tau')$, *if* $\phi(\top[\tau_{-1}], ..., \top[\tau_j]) < \top[\tau']$, *then* o *can be stack allocated.*

THEOREM 6.8 (ADDITIVITY). *Let* $\mathbb{L}(m)(v) = (\phi, (\tau_{-1}, ..., \tau_j), \tau')$, *then* ϕ *is additive.* $\mathbb{L}_{\mathrm{S}}$ *has the same property.*

Example 6.9. Let m be the `BinTree` constructor from Example 2.1. The analyzer yields

$$\begin{aligned}\mathbb{L}(m)(\mathtt{this}) = (\{(c_0, c_1, c_2, c_3) \mapsto c_1 \sqcup c_2 \sqcup c_3\},&\\ (\mathtt{BinTree}, \mathtt{BinTree}, \mathtt{BinTree}, \mathtt{Object}), \mathtt{BinTree})&\end{aligned}$$

$$\mathbb{L}(m)(\mathtt{l}) = (\{(c_0, c_1, c_2, c_3) \mapsto c_0\}, (\mathtt{BinTree}, \mathtt{BinTree}, \mathtt{BinTree}, \mathtt{Object}), \mathtt{BinTree})$$

Analysis $\mathbb{L}$ yields the same relations between escaping parts of parameters as $\mathbb{E}$, so it is also very precise. The analyzer finds that `local` can be stack allocated: it yields $\mathbb{L}(\mathtt{BinTree.run()})(\mathtt{local}) = (\phi = \{c_{-1} \mapsto c_{-1}\}, (\mathtt{Object}), \mathtt{BinTree})$ and $\phi(\top[\mathtt{Object}]) = 1 < \top[\mathtt{BinTree}]$.

Let $v = (\phi, (\tau_{-1}, ..., \tau_j), \tau')$, $v_k = (\phi_k, (\tau_{-1}, ..., \tau_j), \tau'_k)$.

$$v_1 \le v_2 \Leftrightarrow g_{\mathrm{convert}(\tau'_1,\tau'_2)} \circ' \phi_1 \le' \phi_2$$

$$v_1 \sqcup_{\mathbb{L}_1} v_2 = ((g_{\mathrm{convert}(\tau'_1,\tau'_2)} \circ' \phi_1) \sqcup' \phi_2, (\tau_{-1}, ..., \tau_j), \tau'_2) \text{ if } \top[\tau'_2] > \top[\tau'_1].$$

Otherwise, we swap the indices 1 and 2.

$$(C, f, t)_{\mathbb{L}_1}.v = (g_{\mathrm{convert}(\tau',t)} \circ' \phi, (\tau_{-1}, ..., \tau_j), C)$$

$$\mathbb{N}_{\mathbb{L}_1}.v = (\phi, (\tau_{-1}, ..., \tau_j), \tau'[])$$

$$(C, f, t)^{-1}_{\mathbb{L}_1}.v = (g_{\sqcap\top[t]} \circ' g_{\mathrm{convert}(\tau',C)} \circ' \phi, (\tau_{-1}, ..., \tau_j), t)$$

$$\mathbb{N}^{-1}_{\mathbb{L}_1}.v = \begin{cases} (g_{\sqcap\top[t]} \circ' \phi, (\tau_{-1}, ..., \tau_j), t) & \text{if } \tau' = t[]. \\ (g_{\sqcap\top[t]} \circ' g_{\mathrm{convert}(\tau',t[])} \circ' \phi, (\tau_{-1}, ..., \tau_j), t) \text{ with } t \text{ arbitrary}, & \text{otherwise.} \end{cases}$$

$$(\{(c_{-1}, ..., c_i) \mapsto g_{-1}(c_{-1}) \sqcup \ldots \sqcup g_i(c_i) \sqcup u\}, (\tau''_{-1}, ..., \tau''_i), \tau''') \circ_{\mathbb{L}_1} (v_{-1}, ..., v_i) =$$
$$((g_{-1} \circ' g_{\mathrm{convert}(\tau'_{-1},\tau''_{-1})} \circ' \phi_{-1}) \sqcup' ... \sqcup' (g_i \circ' g_{\mathrm{convert}(\tau'_i,\tau''_i)} \circ' \phi_i)$$
$$\sqcup' \{(c_{-1}, ..., c_j) \mapsto u\}, (\tau_{-1}, ..., \tau_j), \tau''').$$

$$\top_{\mathbb{L}_1}[(C, f, t)] = (\{(c_{-1}, ..., c_j) \mapsto \top[t]\}, (\tau_{-1}, ..., \tau_j), t)$$

$$\mathit{first}_{\mathbb{L}_1} = (\{(c_{-1}, ..., c_j) \mapsto g_{\mathrm{id}}(c_{-1}) \sqcup 0\}, (\tau_{-1}, ..., \tau_j), \tau_{-1})$$

$$P_{\mathbb{L}_1}(p_i) = (\{(c_{-1}, ..., c_j) \mapsto g_{\mathrm{id}}(c_i) \sqcup 0\}, (\tau_{-1}, ..., \tau_j), \tau_i)$$

Let $\phi = \{(c_{-1}, ..., c_j) \mapsto g_{-1}(c_{-1}) \sqcup \ldots \sqcup g_j(c_j) \sqcup u\}$
and $\phi' = \{(c_{-1}, ..., c_j) \mapsto g'_{-1}(c_{-1}) \sqcup \ldots \sqcup g'_j(c_j) \sqcup u'\}$.

$$\phi \le' \phi' \Leftrightarrow u \le u' \wedge \forall i \in \{-1, \ldots, j\}, g_i \le'' g'_i$$
$$g \circ' \phi = \{(c_{-1}, ..., c_j) \mapsto (g \circ'' g_{-1})(c_{-1}) \sqcup \ldots \sqcup (g \circ'' g_j)(c_j) \sqcup g(u)\}$$
$$\phi \sqcup' \phi' = \{(c_{-1}, ..., c_j) \mapsto (g_{-1} \sqcup'' g'_{-1})(c_{-1}) \sqcup \ldots \sqcup (g_j \sqcup'' g'_j)(c_j) \sqcup (u \sqcup u')\}$$

Fig. 11. Primitive operators and values of analysis $\mathbb{L}_1$ (first part)

7. REPRESENTATION OF CONTEXT TRANSFORMERS

Now, we wish to find a simple representation of the context transformers of analysis $\mathbb{L}$, to make the implementation easier.

7.1 Using additivity

Context transformers of analysis $\mathbb{L}$ are of the form $((\mathit{Ctx}^n_{\mathbb{L}} \to \mathit{Ctx}_{\mathbb{L}}) \times \mathit{Type}^n \times \mathit{Type})$, $v = (\phi, (\tau_{-1}, ..., \tau_j), \tau')$. Moreover, ϕ is additive (Theorem 6.8), so it can be split into $\phi(c_{-1}, ..., c_j) = g_{-1}(c_{-1}) \sqcup ... \sqcup g_j(c_j) \sqcup u$ where g_{-1}, ..., g_j are monotone functions $\mathbb{N} \to \mathbb{N}$ such that $g_i(0) = 0$, and $u \in \mathbb{N}$. Therefore, we can represent ϕ by a tuple of functions $(g_{-1}, ..., g_j)$ and an integer u. Moreover, we use a sparse representation, where only the useful parameters appear: we represent ϕ by a pair $(u, ((i_{k_1}, g_{i_{k_1}}), \ldots, (i_{k_n}, g_{i_{k_n}})))$ containing the integer u and a tuple of pairs (i_k, g_{i_k}), such that a pair (i_k, g_{i_k}) only appears if ϕ really depends on c_{i_k}, that is if $g_{i_k}(c_{i_k})$ is greater than u for some c_{i_k}. The pairs (i_k, g_{i_k}) are ordered by increasing i_k

$(i_{k_1} < i_{k_2} < \ldots < i_{k_n})$, so that all operations can be done by scanning the tuple representing ϕ from left to right. Then, the complexity is not greater with the sparse representation than with the tuple $(g_{-1}, \ldots, g_j, u)$. This sparse representation saves up both memory and compute time, since in general a context only depends on a few of the parameters of the method. In Figure 11, we give the formulae for the analysis in which ϕ is decomposed into $g_{-1}(c_{-1}) \sqcup \ldots \sqcup g_j(c_j) \sqcup u$ (we keep this notation instead of using tuples to have more intuitive formulae). For the moment, the elementary operations used in this figure have their standard definitions: $\leq''$ is the pointwise ordering $\leq$, $\sqcup''$ is the pointwise upper bound $\sqcup$, $\circ''$ is the composition $\circ$, $g_{\mathrm{id}}(c) = c$, $g_{\mathrm{convert}(\tau,\tau')} = \mathrm{convert}(\tau,\tau')$, $g_{\sqcap n}(c) = c \sqcap n$. These formulae are obvious consequences of the definitions of the previous section, and with the given values of elementary operations, the obtained analysis $\mathbb{L}_1$ is the same as $\mathbb{L}$. Now, we still have to find an efficient representation for the functions g_{-1}, ..., g_j.

7.2 One-step approximation

The needed elementary operations are built by composition and upper bound from identity g_{id}, intersection: $g_{\sqcap n}$, and step: $n \mapsto$ if $n \geq \top[\tau]$ then $\top[\tau']$ else 0 (the conversion between array types is the upper bound of several step functions, other conversions can be represented with at most one step function). This suggests to use the general form

$$c \mapsto (\text{if } c \geq s \text{ then } s^+ \text{ else } 0) \sqcup (c \sqcap l) \qquad (s^+ \in \mathbb{N}, s, l \in \mathbb{N} \cup \{\infty\})$$

(s stands for step, l for lower bound). However, with this general form, we cannot represent all monotone functions. We have to use approximate operations for upper bound $\sqcup''$ and composition $\circ''$. If $g_1(c) = (\text{if } c \geq s_1 \text{ then } s_1^+ \text{ else } 0) \sqcup (c \sqcap l_1)$ and $g_2(c) = (\text{if } c \geq s_2 \text{ then } s_2^+ \text{ else } 0) \sqcup (c \sqcap l_2)$,

$$(g_1 \sqcup'' g_2)(c) = (\text{if } c \geq s_1 \sqcap s_2 \text{ then } s_1^+ \sqcup s_2^+ \text{ else } 0) \sqcup (c \sqcap (l_1 \sqcup l_2))$$

$$\begin{aligned}(g_1 \circ'' g_2)(c) = {} & \text{if } c \geq \underline{(\text{if } l_2 \geq s_1 \text{ then } s_1 \sqcap s_2 \text{ else } s_2)} \text{ then} \\ & \quad \underline{((\text{if } l_2 \sqcup s_2^+ \geq s_1 \text{ then } s_1^+ \text{ else } 0) \sqcup (s_2^+ \sqcap l_1))} \text{ else } 0 \\ & \sqcup (c \sqcap \underline{(l_2 \sqcap l_1)})\end{aligned}$$

Notice that there is not a unique representation for each function. For example, $(\text{if } c \geq 1 \text{ then } 1 \text{ else } 0) \sqcup (c \sqcap 1) = (\text{if } c \geq \infty \text{ then } 0 \text{ else } 0) \sqcup (c \sqcap 1)$. Moreover, the operations $\sqcup''$ and $\circ''$ depend not only on the functions but also on their representations. The choice of the values of s_i, s_i^+, l_i is therefore important. Then, these functions can be better represented by the integers (s, s^+, l) ordered by

$$(s_1, s_1^+, l_1) \leq'' (s_2, s_2^+, l_2) \Leftrightarrow s_1 \geq s_2 \wedge s_1^+ \leq s_2^+ \wedge l_1 \leq l_2$$

and we can define a concretization $\gamma_{\mathbb{L}_1}$, giving the meaning of these integers in terms of functions:

$$\gamma_{\mathbb{L}_1}(s, s^+, l) = \{c \mapsto (\text{if } c \geq s \text{ then } s^+ \text{ else } 0) \sqcup (c \sqcap l)\}$$

We can easily check that $\gamma_{\mathbb{L}_1}$ is monotone, which means that our ordering of integers is coherent with the ordering of functions. The best precision is achieved by choosing the smallest (according to the above ordering) integers representing

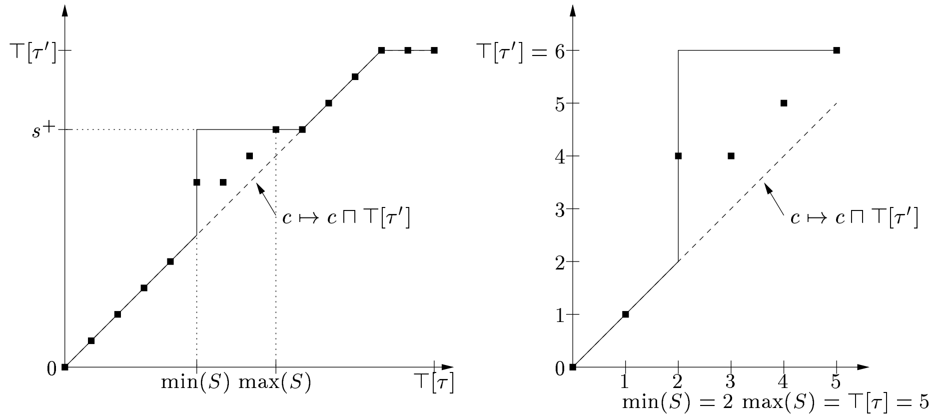


Fig. 12. Approximation of the conversion function. The big dots represent $\mathrm{convert}(\tau, \tau')$, while the solid line is its approximation $g_{\mathrm{convert}(\tau,\tau')} = \{c \mapsto (\text{if } c \geq s \text{ then } s^+ \text{ else } 0) \sqcup (c \sqcap \top[\tau'])\}$ where $S = \{c \mid \mathrm{convert}(\tau, \tau')(c) > c\}$, $s = \min S$, $s^+ = \mathrm{convert}(\tau, \tau')(\max S)$. The graph on the left illustrates the general case, while the one on the right deals with the example of Figure 9: $\tau = \texttt{Tree[][][][]}$, $\tau' = \texttt{Matrix[][]}$. In this example, $\mathrm{convert}(\tau, \tau')$ is approximated by $g_{\mathrm{convert}(\tau,\tau')} = \{c \mapsto \text{if } c \geq 2 \text{ then } 6 \text{ else } 0 \sqcup (c \sqcap 6)\}$.

a given function. Indeed, $\circ''$ and $\sqcup''$ are monotone with respect to the ordering $\leq''$: if $g_1 \leq'' g'_1$ and $g_2 \leq'' g'_2$ then $g_1 \circ'' g_2 \leq'' g'_1 \circ'' g'_2$ and $g_1 \sqcup'' g_2 \leq'' g'_1 \sqcup'' g'_2$. Therefore, if we replace a function g with a smaller one (according to $\leq''$), then the final result of the analysis will also be smaller, hence more precise (if $g \leq'' g'$ then $\gamma_{\mathbb{L}_1}(g)(c) \leq \gamma_{\mathbb{L}_1}(g')(c)$ by monotonicity of $\gamma_{\mathbb{L}_1}$: we find a smaller escaping part with g than with g').

Moreover, we can find examples in which choosing smaller integers for representing a function actually yields strictly more precise results. For instance, take $\{c \mapsto c \sqcap 2\}$ represented by $g_1 = \{c \mapsto (\text{if } c \geq 1 \text{ then } 0 \text{ else } 0) \sqcup (c \sqcap 2)\}$ and by $g'_1 = \{c \mapsto (\text{if } c \geq \infty \text{ then } 0 \text{ else } 0) \sqcup (c \sqcap 2)\}$. We have $g'_1 < g_1$ for the above ordering. Take $g_2 = \{c \mapsto (\text{if } c \geq 3 \text{ then } 2 \text{ else } 0) \sqcup (c \sqcap 0)\}$. We have $g_1 \sqcup'' g_2 = \{c \mapsto (\text{if } c \geq 1 \text{ then } 2 \text{ else } 0) \sqcup (c \sqcap 2)\}$, whereas $g'_1 \sqcup'' g_2 = \{c \mapsto (\text{if } c \geq 3 \text{ then } 2 \text{ else } 0) \sqcup (c \sqcap 2)\}$. Hence $(g_1 \sqcup'' g_2)(1) = 2$ is strictly less precise than $(g'_1 \sqcup'' g_2)(1) = 1$.

Therefore, we choose the following representations for identity: $g_{\mathrm{id}} = \{c \mapsto (\text{if } c \geq \infty \text{ then } 0 \text{ else } 0) \sqcup (c \sqcap \infty)\}$ and intersection: $g_{\sqcap n} = \{c \mapsto (\text{if } c \geq \infty \text{ then } 0 \text{ else } 0) \sqcup (c \sqcap n)\}$. For type conversions from $\tau = t\texttt{[]}^k$ to $\tau' = t'\texttt{[]}^{k'}$, let $S = \{c \mid \mathrm{convert}(\tau, \tau')(c) > c\}$. If $S = \emptyset$, $\mathrm{convert}(\tau, \tau')(c) = c \sqcap \top[\tau']$ (as $\mathrm{convert}(\tau, \tau')(c) \geq c \sqcap \top[\tau']$ and $\mathrm{convert}(\tau, \tau')(c) \leq \top[\tau']$) which is exactly represented by $g_{\mathrm{convert}(\tau,\tau')} = \{c \mapsto (\text{if } c \geq \infty \text{ then } 0 \text{ else } 0) \sqcup (c \sqcap \top[\tau'])\}$. Otherwise let us define $s = \min S$, $s^+ = \mathrm{convert}(\tau, \tau')(\max S)$. Then $\mathrm{convert}(\tau, \tau')(c) \leq (\text{if } c \geq s \text{ then } s^+ \text{ else } 0) \sqcup (c \sqcap \top[\tau'])$, which can be exactly represented. As illustrated by Figure 12, this approximation replaces all parts of the graph of $\mathrm{convert}(\tau, \tau')$ above the graph of $c \mapsto c$ by a single step. Figure 13 summarizes all definitions of elementary operations.

To sum up, we obtain an analysis $\mathbb{L}_1$ with one-step context transformers. This analysis is defined in Figures 7, 11, and 13. We can prove its correctness with

Let $g_1 = \{c \mapsto (\text{if } c \geq s_1 \text{ then } s_1^+ \text{ else } 0) \sqcup (c \sqcap l_1)\}$
and $g_2 = \{c \mapsto (\text{if } c \geq s_2 \text{ then } s_2^+ \text{ else } 0) \sqcup (c \sqcap l_2)\}$.

$$g_1 \leq'' g_2 \Leftrightarrow s_1 \geq s_2 \wedge s_1^+ \leq s_2^+ \wedge l_1 \leq l_2$$

$$g_1 \sqcup'' g_2 = \{c \mapsto (\text{if } c \geq s_1 \sqcap s_2 \text{ then } s_1^+ \sqcup s_2^+ \text{ else } 0) \sqcup (c \sqcap (l_1 \sqcup l_2))\}$$

$$\begin{aligned} g_1 \circ'' g_2 = \{c \mapsto \text{if } c \geq \underline{(\text{if } l_2 \geq s_1 \text{ then } s_1 \sqcap s_2 \text{ else } s_2)} \text{ then} \\ \underline{((\text{if } l_2 \sqcup s_2^+ \geq s_1 \text{ then } s_1^+ \text{ else } 0) \sqcup (s_2^+ \sqcap l_1))} \text{ else } 0 \sqcup (c \sqcap \underline{(l_2 \sqcap l_1)})\} \end{aligned}$$

$$g_{\text{id}} = \{c \mapsto (\text{if } c \geq \infty \text{ then } 0 \text{ else } 0) \sqcup (c \sqcap \infty)\}$$

$$g_{\text{convert}(\tau,\tau')} = \begin{cases} \{c \mapsto (\text{if } c \geq \infty \text{ then } 0 \text{ else } 0) \sqcup (c \sqcap \top[\tau'])\} & \text{if } S = \emptyset \\ \{c \mapsto (\text{if } c \geq \min S \text{ then convert}(\tau,\tau')(\max S) \text{ else } 0) \sqcup (c \sqcap \top[\tau'])\} & \\ & \text{otherwise} \end{cases}$$

where $S = \{c \mid \text{convert}(\tau, \tau')(c) > c\}$.

$$g_{\sqcap n} = \{c \mapsto (\text{if } c \geq \infty \text{ then } 0 \text{ else } 0) \sqcup (c \sqcap n)\}$$

Fig. 13. Primitive operators and values of analysis $\mathbb{L}_1$ (second part)

respect to $\mathbb{L}$ by showing that, if $\mathbb{L}(m)(v) = (\phi, (\tau_{-1}, ..., \tau_j), \tau)$, then $\mathbb{L}_1(m)(v) = (\phi_1, (\tau_{-1}, ..., \tau_j), \tau)$ with $\phi_1 \geq \phi$. Theorem 6.7 immediately extends to $\mathbb{L}_1$.

8. EXTENSIONS

8.1 Exceptions

We consider that all exception objects escape when they are thrown by `throw`. This is not very precise, but in general exception objects cannot be stack allocated anyway. Exception handlers are analyzed by considering that there is a jump from each program point in the `try` block to the exception handler.

8.2 Subroutines

A subroutine is a part of a method that can be called from several points in the method, by a `jsr` instruction. At the end of the subroutine (`ret` instruction), the execution continues just after the call site. To analyze these subroutines, we make sure that, for variables not modified by the subroutine, the escape information flows directly from before the `jsr` to after the `jsr` (and for the backward analysis, from after the `jsr` to before the `jsr`), as if there were no subroutine call. From the point of view of the SSA form, this means that, for variables not modified by the subroutine, we use the same name before and after the `jsr`, even if a new name is introduced in the subroutine to merge the different names of the variable at the different call sites. This technique enables us to reduce the pollution of the escape information at one call site by the escape information coming from other call sites. For example, in the frequent case of a subroutine containing only a lock release instruction, the escape information is exactly the same as if the subroutine did not exist. Of course, for variables modified by the subroutine, we have to use the name introduced in the subroutine, and the escape information coming from different call sites is merged in this case. Moreover, due to the bidirectional nature of escape analysis, there can also be some merging of escape information from different call sites for variables used in field or array assignments in the subroutine. ($\mathbb{E}_S$ flows forward from a call site to the subroutine, is used at the assignment instruction to compute $\mathbb{E}$, and $\mathbb{E}$ flows backward to all call sites.) Therefore, our analysis of subroutines is not fully context-sensitive.

8.3 Native code

For native methods, the escape information cannot be determined automatically. This escape information can be provided by the user, or we assume the worst: we consider that all parameters and the result of native methods escape.

In the benchmarks, the escape information has been given for the native methods `getClass`, `hashCode`, and `clone` in class `java.lang.Object`, `arraycopy` and `identityHashCode` in class `java.lang.System`, `isInstance`, `isAssignableFrom`, `isInterface`, `isArray`, `isPrimitive`, and `getName` in class `java.lang.Class`, and all native methods of `java.io.FileOutputStream`, `java.io.FileInputStream`, `java.io.RandomAccessFile`, `java.util.zip.Inflater` (except `setDictionary`), and `java.util.zip.Deflater` (except `setDictionary`). The worst case is assumed for all other native methods.

8.4 Reflection

Reflection methods can be used to access fields or methods by giving their name dynamically. Essentially, to analyze these calls, we assume the worst when we do not have enough information. Methods reading a field or array element of object type (`Field.get`, `Array.get`) can concern any field, so we assume pessimistically that the whole object may escape. Similarly, methods setting fields or array elements (`Field.set`, `Array.set`) make both the accessed object and the stored object escape. In contrast, methods accessing fields or array elements of simple types (`Field.getInt`, `Field.setInt`, ...) do not make anything escape since they only manipulate data of simple types.

When `Method.invoke` is called, we do not know the called method, so we consider that any method may be called and that all parameters of `Method.invoke` may escape. A call to `Constructor.newInstance` is handled similarly. (In general, we do not exactly know the constructor that will be called: it depends both on the concerned class and on the parameters given to `Constructor.newInstance`; these parameters are to be passed to the called constructor.) However, in the case of `Class.newInstance`, the only constructor that can be called is the constructor without parameters, so we try to determine the concerned class as explained in the dynamic loading section below. If we succeed, we simulate a call to the constructor without parameters. Otherwise, we fall back to the safe assumption that any code can be called. This more precise analysis of `Class.newInstance` is useful in practice since `Class.newInstance` is probably the most frequently called reflection method.

8.5 Dynamic loading

The dynamic loading of classes raises problems for the analysis of virtual calls. Indeed, to analyze a virtual call to the method (C, m'', t), we have to compute the environments $\rho'(C, m'', t)(i) = \sqcup_{C' \text{ subclass of } C}\, \rho(C', m'', t)(i)$ and $\rho'_S(C, m'', t) = \sqcup_{C' \text{ subclass of } C}\, \rho_S(C', m'', t)$, therefore we have to know all subclasses of C. This is difficult in the presence of dynamic loading (except for `final` classes, which cannot have subclasses). We solve this problem by determining automatically as often as possible which classes will actually be loaded. This can be achieved when the name of the loaded class is constant, such as in `Class.forName("name")` or `name.class`. The expression (C)`Class.forName(`s`).newInstance()` loads a class of name s, creates an object of this class, and casts this object to class C. It can only run without error if the loaded class is a subclass of C. Therefore, we consider that all subclasses of C found on the class path may be loaded by such an expression. (In our benchmarks, these constructs are used in particular by Jess, and also by the Java standard library.) In other cases, we rely on information provided by the user, or we consider that any method may be called: the parameters and result of virtual calls may escape.

8.6 Type analysis

The precision of the analysis of virtual calls is also improved by a simple but very useful type analysis. When a parameter of a called method m' has type `Object`, and at the call to m' we have a more precise type information for this parameter (it is of type τ or a subtype of τ), we consider a specialized instance of m' with type τ instead of `Object`, and we reanalyze the code of m' with this more precise information. For example, consider `String.valueOf(Object obj)`

```
class String {
   static String valueOf(Object obj) {
      return (obj == null) ? "null" : obj.toString();
   }
}
```

When we can determine a more precise type for the object `obj`, we can determine more precisely which `toString` method will be called, and improve the precision of the escape information of `String.valueOf`. This way, we can often determine that `obj` does not escape. This is useful for example in string concatenation sequences `"s1" + o1 + "s2" + o2` to prove that objects `o1` and `o2` do not escape (if their `toString` methods do not make them escape, which is most often the case).

This method specialization could be extended to parameters of types other than `Object`, when these types have many subtypes. But in practice, `Object` has many more subtypes than all other types, so the specialization is most useful for the `Object` type. To avoid reanalyzing several times the same method, we could also analyze it only once but narrow the set of virtual call targets to types that can actually appear as parameters of the method. However, this would bring a smaller improvement in precision: for example, `String.valueOf` is in general called with many different types in a program, so many different `toString` methods can be called, and the narrowed analysis is not much more precise than the analysis with `Object`.

In the implementation, for simplicity, we reanalyze the whole method each time a new type of parameter is met. This method specialization is limited to very small methods to avoid an explosion of the analysis time. (In the benchmarks, it is limited to methods containing at most 20 bytes of code.)

9. IMPLEMENTATION

Escape analysis has been implemented in the Java-to-C compiler turboJ [Weiss et al. 1998] produced by the Silicomp Research Institute. The compiler is written in Java and so is our analyzer.

9.1 Computing type levels

The height of type τ, $\top[\tau]$, is computed using the rules (1) to (4) given in Section 6.1. To do this, we split the graph of the relation $\tau' \in \mathrm{Cont}(\tau) \vee (\tau'$ subtype of $\tau \wedge \tau \neq$ `Object`) into strongly connected components. Inside a strongly connected component, all types have the same height by rules (2) and (3). Between strongly connected components, we add one to the height, following rule (4).

Computing strongly connected components is linear in the size of the graph, so the total time to compute type heights is $\mathcal{O}(t)$ where t is the total size of the type declarations of the program.

In the implementation, a type is represented by a data structure that describes its name, fields (or elements for an array), supertype, and also its height to avoid recomputing the height several times for the same type.

9.2 The analysis algorithm

The analysis algorithm works in several passes, as follows:

For each method m, if it has not been analyzed yet, search the graph of methods that may be called from m (and have not been analyzed yet), and compute the strongly connected components of this graph.

For each strongly connected component,

- Build the equations for each method:
  - Transform the code into SSA form.
  - Create unknowns $\mathbb{L}_1(m)(v)$ and $\mathbb{L}_{1S}(m)(v)$ for each variable v. Emit the equations mentioned in Figure 7, for each instruction, in any order.
- Solve the equations, by an iterative fixpoint solver. To speed up the solving, we begin with splitting the dependence graph into strongly connected components, and we solve each component separately, using the algorithm of [Horwitz et al. 1987]. The size of systems is therefore much reduced. To give an idea of the effect of this optimization, with a depth-first ordering of equations, the analysis time is 156 s for jess (an expert system) and 143 s for turboJ itself. The splitting into strongly connected components reduces these times to 129 s and 100 s respectively. (This optimization has less impact on very small programs, since they have simpler equation systems.)
- Prepare the post-transformation, by building the structure giving for each allocation its escape information.

The post-transformation itself is done when generating the C code.

Remark. There are minor differences between our implementation and the algorithm described above. The SSA form of the program is not explicitly generated. However, using the SSA form simplifies the theoretical presentation. (The implementation is in fact equivalent to a very naive way of transforming the program into SSA form, that introduces too many ϕ-functions. Precisely, we introduce ϕ-functions for all variables on meet points that are in loops. The flow sensitivity of the implementation is still exactly the same as with the SSA form.) The implementation generates the equations in two passes, one forward for analysis $\mathbb{L}_{1S}$ and one backward for analysis $\mathbb{L}_1$. In this generation phase, when values of context transformers are already known, the result of abstract operations is computed directly instead of generating an equation. Moreover, in the implementation, for simplicity, the types associated with each context transformer are chosen once for all. This makes a difference with the theoretical presentation of Section 6 in two cases: when creating new objects, the type associated with $\mathbb{L}_{1S}(m)(v)$ is the type of the created object; in loops, the types of the unknowns introduced at back edges are the static types of the objects. The implementation is therefore slightly less precise than the presented theory. The difference can also have an effect on the runtime of the analysis. It does not change the worst case complexity.

10. PROGRAM TRANSFORMATION

10.1 Stack allocation

Theorem 6.7 gives a criterion to decide stack allocation. Stack allocation is implemented by replacing an allocation `new`, `newarray`, `anewarray` or `multianewarray` by a stack allocation by `alloca` when the allocated object does not escape from the method in which it is allocated.

If the size of the stack allocated value is statically known (i.e. it is an object or an array of constant number of elements), and the allocation is not in a loop, the stack allocation is done directly in the method frame, without using `alloca`.

Some extensions to this simple implementation are very important to get satisfying results. They will be detailed in the following sections.

10.2 Reuse allocated space in loops

If a stack allocation takes place in a loop, the allocation is done again at each iteration, which leads to increasing the stack size (this may lead to program failure because of stack overflow [Chase 1988]) whereas the stack size remains constant without stack allocation. Furthermore, data referenced from the stack are considered as always alive by the GC (the GC is conservative for the stack, in the version of the JDK we have used), which may lead to keep important quantities of useless data in the heap.

Example 10.2.1. A typical example of this situation is the following:

```
String s = "";
for (int i = 0; i < 10000; i++)
  s += i + " ";
//i.e. s=new StringBuffer(s).append(i).append(" ").toString();
```

where all intermediate `StringBuffer`s used to compute `s += i + " "` can be stack allocated, and so all intermediate char arrays are kept uselessly by the GC until the end of the method.

To solve this problem, we have extended our analysis so that, if an allocated object is always useless after one iteration, we do not use a new space for each allocation, but we reuse the already allocated space. This way, the stack does not grow. To determine whether the object will still be useful at the next iteration, we use the following criterion:

> Consider that all variables that are live just before the allocation escape. If the allocated object does not escape, then the allocated space can be reused after one iteration.

The intuitive idea behind this criterion is the following: consider for example the `while` loop

```
while (test) { ... x:  new C() ... }
```

and assume that we unroll that loop:

```
    if (!test) goto end;
    ... x1:  new C() ...
    if (!test) goto end;
    ... x2:  new C() ...
    ...
end:
```

If the object allocated at `x1` is not live any more at `x2`, i.e. it is not accessible from a variable live at `x2`, i.e. it does not escape when we assume that all the variables live at `x2` escape, then we can reuse at `x2` the memory space allocated at `x1`.

Technically, when analyzing a `new`, we emit, together with the equations already mentioned in the preceding section, the following equation for each variable v live before the `new`, t being the type of this variable:

$$\phi(p_{-1}, ..., p_n) \geq p_k \sqcap \top[t]$$

where $\mathbb{L}_1(m)(v) = (\phi, (\tau_{-1}, ..., \tau_n), t)$, $n \geq k$, p_k is a new parameter. For the normal escape analysis (the one that determines whether `alloca` should be introduced), p_k is 0. In this case, the above equation becomes $\phi(p_{-1}, ..., p_n) \geq 0$ and it can be removed: we did not change the results of the normal escape analysis. When determining if the allocation at pc can reuse the space allocated at the preceding iteration, $p_k = \infty$, and the above equation becomes $\phi(p_{-1}, ..., p_n) \geq \top[t]$ meaning that variable v escapes. Therefore, when $p_k = \infty$, all variables live before the allocation are regarded as escaping. If the allocated object does not escape with $p_k = \infty$, we can reuse the same stack allocated space for all iterations. A new p_k parameter is introduced for each allocation that occurs in a loop.

Remark. When composing context transformers for analyzing method calls, we have to take into account these additional p_k parameters. We simply give the value 0 to all additional parameters of context transformers of the callee, since we use the normal analysis of the callee.

In the case of objects or constant size arrays, reusing the allocated space is relatively easy. It is more difficult for variable size arrays. In this case, we use the following algorithm:

—If the new size is smaller than the already allocated one, the space is reused;

—Otherwise, if the previously allocated value is still at the top of the stack, the allocated space is enlarged to be able to allocate the new array;

—Otherwise, the new array cannot be allocated in the old space. A new space is then allocated even if this may cause stack overflows if a large size is allocated. This did not cause problems during our experiments.

For very large arrays, even a single allocation could cause a stack overflow. A dynamic test could easily be performed at each stack allocation of arrays whose size is not statically known, to limit the size of the stack allocated arrays (if the allocated array is too large, it is heap allocated).

10.3 Inlining

The algorithm can still be improved to discover more stack allocation opportunities: it may happen that an object d cannot be stack allocated in method m' because it is still live at the end of m', but becomes dead at the end of method m which has called m'. In this case, we inline m' in m and allocate d on the stack in m. Our implementation also applies this technique to longer call chains.

However, we have to determine at the end of which method d will be dead. Assume that we call a method m' with $j+1$ parameters at pc in method m: $r = p_0.m'(p_1, ..., p_j)$. Then, we define

$$\mathbb{L}_{1C}(pc) = (\mathbb{L}_1(m)(r), \mathbb{L}_{1S}(m)(p_0), ..., \mathbb{L}_{1S}(m)(p_j))$$

the $(j+2)$-tuple containing the escaping part of the result and parameters of method m' in method m. $\mathbb{L}_{1C}(pc)$ is the transformer that converts escaping parts computed in the callee m' to escaping parts computed in the caller m, taking into account that the scope of m is larger than the scope of m'.

Theorem 10.3.1 (Inlining). *Assume that method m_0 calls m_1 at pc_1, which calls m_2 at pc_2, ..., which calls m_n at pc_n, and that in method m_n, at pc, an object o of type τ is allocated, and stored in variable v. If $\phi(\top[\tau_{-1}], ..., \top[\tau_j]) < \top[\tau]$ where*

$$\mathbb{L}_1(m_n)(v) \circ \mathbb{L}_{1C}(pc_n) \circ ... \circ \mathbb{L}_{1C}(pc_1) = (\phi, (\tau_{-1}, ..., \tau_j), \tau)$$

then this object can be stack allocated if we inline m_1, ..., m_n in m_0.

Proof. (sketch) We prove the corresponding result for analysis $\mathbb{E}$, and then infer the result for analysis $\mathbb{L}_1$ using the properties of the Galois connection and the concretization.

There are essentially two proofs of this result for analysis $\mathbb{E}$. The first proof is by simulating the analysis of the method m_0 in which m_1, ..., m_n have been inlined, and applying Theorem 5.3.6.

The second proof shows that the object o is not reachable any more at the end of m (even without inlining m_1, ..., m_n). It is then an obvious consequence that o can be stack allocated if the inlining is performed. The advantage of this second proof is that it could also be applied to preallocation, and that most of the proof can be applied to synchronization elimination (in the next section).

This second proof is done by extending the correctness theorem of our alias analysis, and then proving the result for inlining. It is detailed in [Blanchet 2000]. □

Moreover, we wish to inline a call to method m' only if that allows more stack allocations, i.e. if the stack allocation criterion is true for some allocations in m' or in methods that m' calls, when inlining the call to m', but is wrong without the inlining of the call to m'.

A first idea of algorithm would be to store for each method m' the list of all interesting chains of inlining. That is, we would store the call chain to the method in which an allocation is performed, which cannot be stack allocated without inlining. Together with these call chains, we would also store the context transformer corresponding to the stack allocation condition: $\mathbb{L}_1(m_n)(v) \circ \mathbb{L}_{1\mathrm{C}}(pc_n) \circ ... \circ \mathbb{L}_{1\mathrm{C}}(pc_1)$. However, this leads to a memory space quadratic in the size of the program in the worst case (each method contains a copy of all stack allocation opportunities in all the methods it may call). For very large projects, this may lead to memory overflow.

Therefore we had to design a more efficient algorithm, which yields the same results, but keeps only a constant-size summary of the stack allocation opportunities. This algorithm can yield three answers: 1) "we should inline", when a stack allocation possible only with inlining has been found; 2) "we should not inline", when inlining would not yield additional stack allocations; 3) "don't know", when the summary is not precise enough to give the answer to the question "should we inline ?". In the last case, we have to examine all methods that can be called from m, to test whether there exists a stack allocation that can be realized only with inlining. In practice, this case will be rare enough for the system to be very efficient. The technical details of this algorithm can be found in [Blanchet 1999].

10.4 Synchronization elimination

A method only needs to be synchronized if the object on which it is called is accessed by several threads. If an object is accessed by several threads, it must be reachable from static fields or from a `Thread` object. Objects reachable from static fields are considered as escaping by our analysis, and the thread creation method `Thread.start()` makes its parameter escape. Therefore, if our analysis finds that an object does not escape from a method m, it can be accessed only by one thread, the thread in which it is allocated. So there is no need to synchronize methods called on this object.[5] This is an important optimization since synchronization is a costly operation in the JDK. We perform the following transformations.

10.4.1 *Global synchronization analysis.* Assume that a synchronized method m' is called from method m. If for all calls to m, we can prove that the object on which we call m' will be local to one thread, then the synchronization can be removed.

More precisely, we define $\mathbb{L}_{1\mathrm{sync}}(m) \in Ctx_{\mathbb{L}}^{j+2}$ where method m has $j+1$ parameters. $\mathbb{L}_{1\mathrm{sync}}(m)$ indicates for each parameter and for the result of m the part that may be shared by several threads. $\mathbb{L}_{1\mathrm{sync}}$ is defined by the equations:

(1) Assume that method m has $j+1$ parameters of types $\tau_0, ..., \tau_j$ and a result of type τ_{-1}. Also assume that, at pc, m can call method m' which has $j'+1$ parameters of types $\tau'_0, ..., \tau'_{j'}$ and a result of type τ'_{-1}. Let $\mathbb{L}_{1\mathrm{C}}(pc) = (v_{-1}, ..., v_{j'})$ with

[5] On some architectures, there is a memory barrier at each synchronization. We have to leave this barrier even if we remove the locking or unlocking operation.

$v_k = (\phi_k, (\tau_{-1}, ..., \tau_j), \tau'_k)$. In this case, $\mathbb{L}_{1\text{sync}}(m') \geq (\phi_{-1}, ..., \phi_{j'})(\mathbb{L}_{1\text{sync}}(m))$.

(2) On the other hand, if $\rho_S(m) = (\phi_{-1}, (\tau_{-1}, ..., \tau_j), \tau_{-1})$ and $\rho(m)(k) = (\phi_k, (\tau_{-1}, ..., \tau_j), \tau_k)$, $\mathbb{L}_{1\text{sync}}(m) \geq (\phi_{-1}, ..., \phi_j)(\mathbb{L}_{1\text{sync}}(m))$.

(3) $\mathbb{L}_{1\text{sync}}(\texttt{void Thread.start}()) = (0, \top[\texttt{Thread}])$.

The value of $\mathbb{L}_{1\text{sync}}$ is computed by fixpoint iteration of these equations on the whole program. Notice that we need to have enough information on the call graph (which methods may be called from which program point) to be able to perform this analysis. Therefore, we assume the worst when we are not able to determine which classes will be loaded at runtime: in this case, $\mathbb{L}_{1\text{sync}}(m) = (\top[\tau_{-1}], ..., \top[\tau_j])$ where the parameters of m have type $\tau_0, ..., \tau_j$ and its result τ_{-1}.

THEOREM 10.4.1. *Assume that in method m, a synchronized method m' is called by $w = v_0.m'(v_1, ..., v_j)$. Let $\mathbb{L}_{1\text{S}}(m)(v_0) = (\phi, (\tau_{-1}, ..., \tau_j), \tau'_0)$. If $\phi(\mathbb{L}_{1\text{sync}}(m)) < \top[\tau'_0]$, then the object in v_0, on which m' is called, is thread local, and the synchronization can be eliminated.*

PROOF. (sketch) If an object is never reachable from a static field or from a thread object passed as a parameter to the `Thread.start` method, then it will be local to the thread in which it has been created.

Using a technique similar to the proof of correctness of inlining, we show that the object on which m' is called is never aliased to a path static.p. Hence the result. □

When the conditions of the above theorem are satisfied, we call a copy of m' without synchronization.

10.4.2 *No synchronization on stack allocated objects.* Before acquiring a lock on an object o, we test at runtime whether o is on the stack, and if it is, we skip the synchronization.

This technique could be extended by marking at allocation time objects that are detected to be local to one thread (by the global analysis above), and testing this mark before acquiring a lock, as in [Choi et al. 1999]. However this requires at least one free bit in the representation of each object.

Another way of performing this is by defining a subclass C_{nosync} for each class C that has at least one synchronized method, as in [Bogda and Hölzle 1999]. The class C_{nosync} defines the same methods as C but without synchronization. When allocating an object of class C detected as thread local, we in fact allocate an object of class C_{nosync}. This way, all synchronizations on the object are also eliminated. This replaces the runtime check of the preceding methods with a dynamic dispatch (which may be more costly, when there was no dynamic dispatch before synchronization elimination).

The idea behind these three techniques is that sometimes, at the call site, the object may be thread local or not depending on what happens at runtime. Therefore, the synchronizations cannot be optimized by the transformation of Section 10.4.1. But it may happen that at the allocation site, the allocated object is always thread local. The synchronizations may then be eliminated by one of these transformations. Combining both transformations of Sections 10.4.1 and 10.4.2 is useful to get the best results.

Size of the SSA form	n
Maximum number of parameters of a method + 1	p
Maximum number of parameters of a context	p'
Maximum type height	H

Equations building	$\mathcal{O}(npp')$
Number of equations	$n_e = \mathcal{O}(n)$
Number of unknowns	$n_u = \mathcal{O}(n)$
Number of iterations	$n_i = \mathcal{O}(n_u p' H) = \mathcal{O}(np'H)$
Equations solving	$\mathcal{O}(n_e pp' n_i) = \mathcal{O}(npp' n_i)$
Posttransformation	$\mathcal{O}(np)$

Fig. 14. Notations and results on the complexity of escape analysis

11. COMPLEXITY

Notations are summarized in Figure 14. The size of the SSA form is the sum of the sizes of each instruction in the SSA form of the program, where the size of an instruction is 1, except for $v_0.m(v_1, ..., v_j)$ which has size $s_j = j + 2$. When out-of-loop stack allocation is activated, context transformers may have more parameters than methods, hence the distinction between p' and p. If out-of-loop stack allocation is disabled, $p' = p$. Also note that the method specialization of Section 8.6 increases the analyzed size n, due to reanalysis of some methods. The increase in size is however small, since the reanalysis is limited to very small methods (at most 20 bytes of code) and is only done when the method takes a parameter of type `Object`.

The maximum time required to generate the equations for an instruction depends on the considered instruction: a method call $(v_0.m(v_1, ..., v_j))$ takes $\mathcal{O}(s_j^2 p')$, all other instructions take $\mathcal{O}(p')$. (This complexity comes from the fact that, when the context transformers are constant, we directly compute the result of the operations. A composition of a context transformer having s_j parameters with s_j context transformers having at most p' parameters takes $\mathcal{O}(s_j p')$. An upper bound, conversion, restriction or construction on contexts takes $\mathcal{O}(p')$.) If s is the size of the instruction, the maximum time required to generate the equations for this instruction is then $\mathcal{O}(spp')$. The total complexity of equations building is therefore $\mathcal{O}(npp')$. The number of generated equations or unknowns is $\mathcal{O}(n)$ (each instruction generates a constant number of equations, except the method call, which generates as many equations as there are parameters).

If the system contains n_u unknowns, the number of iterations n_i is less than the height of the lattice of n_u-tuples of context transformers, so $n_i = \mathcal{O}(n_u p' H)$. In practice, the number of iterations is very small (It is at most 17 in our benchmarks. 44.8% of the equations are iterated at most twice, 91.3% at most 7 times, 98.4% at most 10 times. On average, each equation is iterated 3.9 times). This is possible thanks to the splitting of equations systems into strongly connected components. On the other hand, one iteration costs the compute time of equations, which is at most $\mathcal{O}(n_e pp')$. The complexity of the solving is therefore $\mathcal{O}(n_e pp' n_i)$. This complexity, which sounds large, is much reduced in practice by using the sparse representation for context transformers described in Section 7, which ensures that the number of parameters of a context is in fact small compared to p in most cases.

The post-treatment of a method call (when inlining is activated) or of an allocation is in $\mathcal{O}(p)$ if we can decide to inline or not with our fast algorithm. In this

Table I. Benchmark programs. The size is the total size of bytecode of the methods that may be called, including methods of the Java standard library.

Benchmark programs		Size(kb)
dhry	Dhrystone	73
Symantec	Set of small benchmarks	76
javac	Java compiler (jdk 1.1.5) compiling jBYTEmark	242
turboJ	Java to C compiler from Silicomp RI compiling jBYTEmark	311
JLex	Lexer generator (v. 1.2.3) running `sample.lex`	97
jess	Expert system (v. 4.1), solving `fullmab.clp`	363
javacc	Java parser generator (v. 0.8pre2) generating a Java parser	254

case, the post-treatment is dominated by the generation of equations. If our fast algorithm does not give the answer, the post-treatment of a call may take time up to $\mathcal{O}(n'p)$ where n' is the number of allocations in all methods called from the program point, following any call chain. This case is very rare and can be neglected (it happens 21 times only in all the tested benchmarks, which represent more than 2 Mb of classes).

The complexity is therefore $\mathcal{O}(npp'n_i) = \mathcal{O}(n^2pp'^2H)$, with n_i small in practice.

12. BENCHMARKS

Benchmarks have been conducted on a 233 MHz Pentium MMX, 128 Mb RAM, primary cache 16 kb instructions and 16 kb data, secondary cache 512 kb instructions and data, under Linux. We used the JDK 1.1.5, with no just-in-time compiler, but the Java-to-C compiler turboJ [Weiss et al. 1998]. Benchmark programs are listed in Table I.

12.1 Results

Each program has been tested in two stack allocation configurations: in the configuration "No loops", all stack allocations done in a loop must reuse the same memory space. In the configuration "All", `alloca` is allowed even in a loop. The last configuration may stack allocate more objects, but may lead to stack overflows. In our benchmarks, inlining is allowed for methods of less than 150 bytes.

Table II gives the percentage of stack allocated data and thread-local objects in each program. The top table deals with the size of objects. Its last column represents the total size of allocated data in the program. Its first two columns indicate the percentage of the total allocated size that is stack allocated. Its third column mentions the percentage of the total allocated size that consists of objects detected as thread local by our analysis. Synchronizations can be eliminated on thread-local objects, and these objects could also be allocated in a thread-local heap (although we do not perform this optimization). The bottom table is similar but deals with numbers of objects instead of size. One can note that our analysis does not detect many more objects as thread local than it can stack allocate. One way to improve the detection of thread-locality would be to take into account that some objects reachable from static fields are in fact accessed by only one thread. This is done for example by Ruf [2000], and our analysis could be adapted to do it (at the cost of an increased complexity). Essentially, one would add an integer context for each static field, indicating which part of the field is shared between several threads. Table III gives the percentage of eliminated synchronizations at

Table II. Stack allocation and thread-local objects

`alloca`	Size of stack allocated objects (%) No loops	All	Size of thread-local objects (%)	Total allocated size (Mb)
dhry	95	95	95	5.1
Symantec	84	84	84	40.2
javac	13	13	13	6.5
turboJ	27	29	29	23.2
JLex	25	26	57	1.7
jess	21	21	21	1.9
javacc	20	43	45	7.5

`alloca`	Number of stack allocated objects (%) No loops	All	Number of thread-local objects (%)	Total number of objects (K)
dhry	99	99	99	600
Symantec	99	99	99	1199
javac	19	19	19	169
turboJ	36	36	36	468
JLex	29	30	74	39
jess	27	27	27	43
javacc	14	18	23	187

runtime. The left column represents the part of eliminated synchronizations that we would get without testing at runtime whether objects are in the stack or not. The middle column gives the part of eliminated synchronizations with this dynamic test. The right column is the total number of synchronizations (without elimination) at runtime. All other results, in particular runtime decrease measures, were obtained with the dynamic "is in stack" test.

In Dhrystone, we mainly stack allocate arrays of one integer, which are used as references to integers. Symantec is a set of small benchmarks which contains particularly a version of Dhrystone, in which we do the same optimizations. It also contains a function that uselessly allocates many nodes of a tree, which can be stack allocated. We also manage to eliminate all synchronizations corresponding to synchronized calls to a random number generator.

In turboJ and javacc, we mainly stack allocate strings and arrays of characters[6]. Our inlining algorithm is useful to allocate such data on the stack, since they are often allocated in `StringBuffer.ensureCapacity` and `StringBuffer.toString`, and of course still live at the end of these methods (char arrays stack allocated thanks to inlining represent 4% of data for javac, 20% for turboJ, 29% for javacc). The difference between the "No loops" and the "All" configurations is only important for javacc.

In Table IV, we give the average type height and the average cardinality of a strongly connected component of the graph of types in our benchmarks. Types

[6]TurboJ detects string concatenation sequences, and replaces them by an optimized code that computes the length of the final string and directly allocates an array of characters of the required size [Weiss et al. 1998]. This is much more efficient than the code generated by javac which uses `StringBuffer.append`. This also reduces the amount of stack allocated data in our benchmarks because the `StringBuffer`s and the intermediate arrays of characters would be stack allocated. Our analyzer takes the bytecode with `StringBuffer`s as input, before transformation by TurboJ.

Table III. Eliminated synchronizations

	Eliminated synchro. (%)		Thousands
	(1)	(2)	of synchro.
dhry	–	–	0
Symantec	99	99	2520
javac	5	31	430
turboJ	21	46	587
JLex	78	94	1989
jess	13	21	158
javacc	2	3	1075

(1) Eliminated synchronizations without dynamic "is in stack" test.
(2) Eliminated synchronizations with dynamic "is in stack" test.

Table IV. Statistic information on heights of types

	$\top[\tau]$	#scc
dhry	3.3	2.3
Symantec	3.4	2.3
javac	7.3	3.3
turboJ	5.1	3.7
JLex	3.5	2.2
jess	6.5	2.5
javacc	4.4	2.4
All benchmarks	5.7	2.7

$\top[\tau]$: average type height.
#scc: average cardinality of a strongly connected component.

Table V. Runtime decrease

Runtime decrease (%)			Time
`alloca`	No loops	All	(ms)
dhry	43	43	2710
Symantec	26	26	19327
javac	10	11	4109
turboJ	10	13	19678
JLex	42	40	3367
jess	-4	-1	3643
javacc	6	9	7695
mean	21	22	

GC time decrease (%)			GC time
`alloca`	No loops	All	(ms)
dhry	100	100	567
Symantec	97	97	1543
javac	100	100	197
turboJ	20	17	8511
JLex	-1	1	55
jess	28	29	203
javacc	31	37	1852

that are in the same strongly connected component cannot be distinguished by the analysis because they have the same height. So this information can help understanding the precision of the analysis. However, it is difficult to see a real correlation between these figures and the precision of the analysis, because other factors, such as the actual points-to relation between objects, can have a greater impact on the results.

Table V indicates the percentage of runtime decrease, computed by the formula $100 \times (T_{old} - T_{new})/T_{old}$ where T_{old} is the runtime without optimization, and T_{new} the runtime with optimization. The last column is the runtime without any optimization, for reference. The mean in the bottom row is the geometric mean. Figures 15 and 16 also give the percentage of runtime decrease (Total bar), and they detail the reasons of the speedup. The GC bar gives the part of the runtime decrease that comes from a decrease of GC time. The Inline bar gives the part of

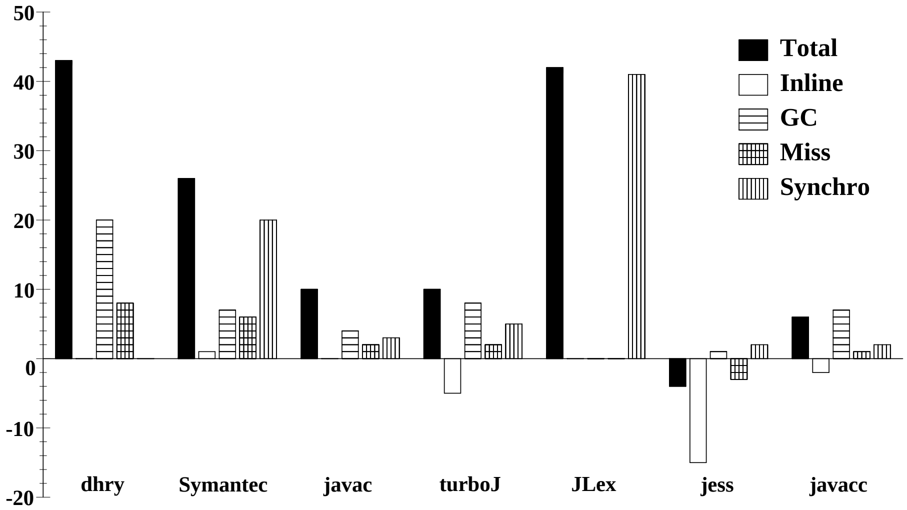


Fig. 15. Runtime decrease without `alloca` in loops.

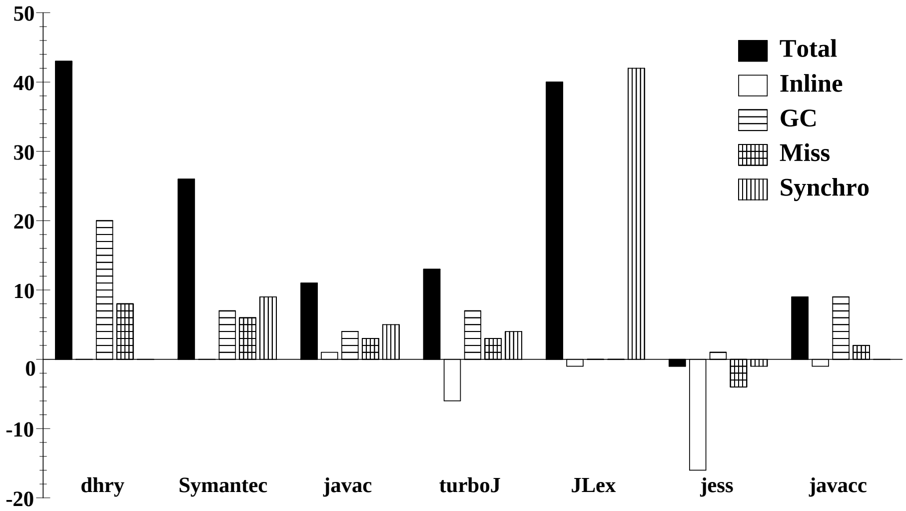


Fig. 16. Runtime decrease with all possible stack allocations, even in loops

Table VI. Effects on the cache.

	Read miss			Write miss		
	Decrease (%)		Thousands	Decrease (%)		Thousands
`alloca`	No loops	All	of misses	No loops	All	of misses
dhry	38	40	2905	22	20	3228
Symantec	33	33	15011	51	50	11241
javac	6	8	6448	5	6	8139
turboJ	10	15	27212	0	5	13284
JLex	-2	-3	2161	1	-7	2485
jess	-9	-8	6931	-9	-21	4886
javacc	6	13	10856	-5	-14	5706

the runtime decrease that comes from inlining of methods (inlining is done as if there were stack allocation, but all allocations are done in the heap). The Miss bar gives the contribution of the improvement on data locality to the speedup. This is based on the number of cycles during which the processor is stalled waiting for a read from memory, or because the write buffer is full, measured experimentally. The Synchro bar gives the part of the runtime decrease that comes from synchronization elimination. A synchronization costs about 790 ns when the monitor is in the monitor cache, and 1450 ns when it is not.

The speedups correspond to what could be expected knowing the percentage of stack allocated data and of synchronization elimination. We get high speedups for dhry (thanks to stack allocation), for JLex (thanks to synchronization elimination), and for Symantec. The runtime decreases by about 10% for javac, turboJ and javacc. Inlining has negative effects for turboJ and jess. This may come from an increase of the code size, which leads to more traffic for loading the code from memory. Inlining also leads to changes in register allocation which may affect performance. This negative effect of inlining explains the slowdown that we obtain for jess. For JLex, stack allocating data gives negligible speedups, because the GC time was very small anyway.

The speedups that come from stack allocation mainly have three causes: reduced allocation time, better data locality, decrease of the GC workload. The graphs show that the part of the speedup coming from the GC is important (about half of the speedup not coming from synchronization elimination). This is logical since the JDK uses a mark and sweep garbage collector, which is not efficient for short lived data, which can precisely be stack allocated. The better data locality [Carr et al. 1994] is responsible for about a quarter of the speedup. The rest, i.e. about a quarter comes from allocation time. Stack allocating an object just requires moving the stack pointer, whereas heap allocating it requires scanning the list of free blocks, and acquiring a lock, which is much less efficient.

For javacc, the speedup is more important in the "All" configuration, in which more allocations can be done in the stack. In the "All" configuration, the stack size increases by a factor 10 for the javacc program, whereas in the "No loops" configuration the increase is limited to only 77%. This shows that it may be useful to limit ourselves to the "No loops" configuration to avoid stack overflows (even if it does not happen in our benchmarks). The stack size increase rate is 129% for JLex in the "All" configuration. It is at most 68% in all other cases.

The right part of Table V gives the percentage of GC time decrease, and in the last column the GC time for reference. The decrease of the GC time is similar to the percentage of stack allocated data. The exceptions are javac, which does a single GC without stack allocation, and none with stack allocation, jess whose GC time is more reduced, and turboJ and JLex for which the GC time is less reduced than what we could expect. This comes from the fact that stack allocated data are mainly short lived data which are scanned only once by the GC, and so have a smaller cost for the GC than data that cannot be stack allocated (they may be scanned in several GC cycles if they remain alive). Data allocated in turboJ and JLex may live longer than in the other benchmarks.

Table VI deals with cache misses. It shows that stack allocation reduces the number of read misses (except for jess and JLex). Stack allocation improves data locality because the top of the stack, which is the most useful part, is very often in the cache. However, in some cases, stack allocation may also have a negative effect on data locality. For example, in javacc, when `alloca` is allowed even in a loop, arrays of characters are stack allocated inside loops. These arrays of characters become in fact unreachable at the next iteration of the loop, but our analysis cannot detect this fact. Therefore, there are gaps in the stack that will not be reused until the end of the method. On the contrary, when these arrays of characters are allocated in the heap, they may be reclaimed earlier by a GC and reused. This negative effect on data locality explains the increase of the number of misses that can be noticed for several programs.

The cost of read misses can be determined by measuring the number of cycles during which the processor is stalled, waiting for a read from memory to complete. Similarly, the cost of write misses is determined by measuring the time during which the pipeline is stalled, because the write buffers are full. The decrease of the number of write misses has a negligible effect on the execution time, because the cost of a write miss is very small: in most cases, data can be written directly to the write buffer, and the pipeline is not stalled. On the contrary, the decrease of the number of read misses has an important effect, as reported in the graphs of Figures 15 and 16. The cost of a read miss is important (about 60 cycles) because it leads to a complex process. First, flush the write buffer if it is not empty. Second, write the cache line that will be replaced by the newly loaded line, if it has been modified in the cache and not yet written to memory. Third, load a cache line from memory.

12.2 Analysis speed

We have measured the analysis time and compilation overhead, as a percentage of the total compilation time without stack allocation (Table VII). The compilation overhead comes from inlining and synchronization elimination, which lead to more code generation. In these tests, the whole program and the useful part of the standard library have been analyzed and compiled (in the preceding benchmarks, the library has been analyzed with each benchmark program, but it has been compiled only once for all benchmark programs. However, it does not make sense to compare the analysis time of a large amount of code with the compilation time of a small program). These tests have been run with TurboJ bootstrapped.

These results show that our analysis takes about 10% of the C compilation time, and the total compilation overhead generated by our optimizations is 29% (arithmetic mean weighted by the size of the code).

Table VII. Analysis speed

	An (%)	*Gen* (%)	*Comp* (%)	*Time* (s)
dhry	9	6	2	83.0
Symantec	8	5	3	106.2
javac	11	19	6	671.5
turboJ	11	27	12	907.8
JLex	6	8	9	143.1
jess	9	21	10	1429.2
javacc	9	23	9	538.2

Time is the total compilation time without stack allocation.
An is the escape analysis time, as a percentage of *Time*.
Gen is the C code generation overhead, as a percentage of *Time*. *Gen* includes *An*.
Comp is the C compilation overhead, as a percentage of *Time*.

13. CONCLUSION

We have given a correctness proof of an escape analysis, extended from [Park and Goldberg 1992; Deutsch 1997; Blanchet 1998] to allow a precise treatment of assignments, and to support subtyping and inheritance. The main originality of this analysis with respect to those previous analyses is that it uses a bidirectional propagation. This is handled in the proof by using an alias analysis. This way we can map the iteration used to compute the transitive closure of the alias relation to the fixpoint iteration between the forward and backward analyses.

We have demonstrated the importance of escape analysis for Java, thanks to our implementation which analyses the full Java language. It can be applied both to stack allocation and synchronization elimination. Our tests show that we get high speedups (geometric mean 21%) at a reasonable analysis cost (the analysis takes about 10% of the compilation time, the total compilation overhead is 29%).

ACKNOWLEDGMENTS

I would like to thank Peeter Laud for discussions on the proof of our escape analysis, and the Java team of the Silicomp Research Institute for providing TurboJ. I am very grateful to the anonymous referees for their detailed comments that were very helpful to improve the presentation of the paper. I also thank Laurent Mauborgne, David Monniaux, and Xavier Rival for their comments on a draft of this paper.

REFERENCES

Agesen, O., Detlefs, D., Garthwaite, A., Knippel, R., Ramakrishna, Y. S., and White, D. 1999. An Efficient Meta-lock for Implementing Ubiquitous Synchronization. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*. ACM Press, Denver, CO, 207–222.

Aiken, A., Fähndrich, M., and Levien, R. 1995. Better Static Memory Management: Improving Region-Based Analysis of Higher-Order Languages. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*. ACM Press, La Jolla, CA, 174–185.

Aldrich, J., Chambers, C., Sirer, E. G., and Eggers, S. 1999. Static Analyses for Eliminating Unnecessary Synchronization from Java Programs. In *Static Analysis Symposium (SAS'99)*. Lecture Notes on Computer Science, vol. 1694. Springer Verlag, Venice, Italy, 19–38.

Bacon, D. F. 1997. Fast and efficient optimization of statically typed object-oriented languages. Ph.D. thesis, University of California, Berkeley.

BACON, D. F., KONURU, R., MURTHY, C., AND SERRANO, M. 1998. Thin Locks: Featherweight Synchronization for Java. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI'98)*. ACM Press, Montreal, Canada, 258–268.

BIRKEDAL, L., TOFTE, M., AND VEJLSTRUP, M. 1996. From Region Inference to von Neumann Machines via Region Representation Inference. In *23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*. ACM Press, St. Petersburg Beach, FL, 171–183.

BLANCHET, B. 1998. Escape Analysis: Correctness Proof, Implementation and Experimental Results. In *25th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'98)*. ACM Press, San Diego, CA, 25–37.

BLANCHET, B. 1999. Escape Analysis for Object Oriented Languages. Application to Java^{TM}. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*. ACM Press, Denver, CO, 20–34.

BLANCHET, B. 2000. Escape Analysis. Applications to ML and Java^{TM}. Ph.D. thesis, École Polytechnique.

BOGDA, J. AND HÖLZLE, U. 1999. Removing Unnecessary Synchronization in Java. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*. ACM Press, Denver, CO, 35–46.

CARR, S., MCKINLEY, K. S., AND TSENG, C.-W. 1994. Compiler Optimizations for Improving Data Locality. In *6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'94)*. ACM Press, 252–262.

CHASE, D. R. 1988. Safety considerations for storage allocation optimizations. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*. ACM Press, Atlanta, GA, 1–10.

CHOI, J.-D., GUPTA, M., SERRANO, M., SREEDHAR, V. C., AND MIDKIFF, S. 1999. Escape Analysis for Java. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*. ACM Press, Denver, CO, 1–19.

COUSOT, P. AND COUSOT, R. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th Annual ACM Symposium on Principles of Programming Languages*. ACM Press, Los Angeles, CA, 238–252.

COUSOT, P. AND COUSOT, R. 1979. Systematic Design of Program Analysis Frameworks. In *Conference Record of the 6th Annual ACM Symposium on Principles of Programming Languages*. ACM Press, San Antonio, TX, 269–282.

CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependance Graph. *ACM Trans. Program. Lang. Syst. 13,* 4 (Oct.), 451–490.

DEAN, J., GROVE, D., AND CHAMBERS, C. 1995. Optimization of Object-Oriented Programs using Static Class Hierarchy Analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*. Lecture Notes on Computer Science, vol. 952. Springer Verlag, 77–101.

DEUTSCH, A. 1990. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *17th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM Press, San Francisco, CA, 157–168.

DEUTSCH, A. 1994. Interprocedural May-Alias Analysis for Pointers: Beyond k-limiting. In *Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation*. ACM Press, Orlando, FL, 230–241.

DEUTSCH, A. 1997. On the Complexity of Escape Analysis. In *24th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'97)*. ACM Press, Paris, France, 358–371.

DINIZ, P. C. AND RINARD, M. C. 1998. Lock Coarsening: Eliminating Lock Overhead in Automatically Parallelized Object-Based Programs. *J. Parallel Distrib. Comput. 49,* 2 (Mar.), 218–244.

FITZGERALD, R., KNOBLOCK, T. B., RUF, E., STEENSGAARD, B., AND TARDITI, D. 2000. Marmot: An Optimizing Compiler for Java. *Softw. Pract. Exper. 30,* 3 (Mar.), 199–232.

Gay, D. and Steensgaard, B. 2000. Fast Escape Analysis and Stack Allocation for Object-Based Programs. In *Compiler Construction, 9th International Conference, CC'2000*, D. A. Watt, Ed. Lecture Notes on Computer Science, vol. 1781. Springer Verlag, Berlin, Germany, 82–93.

Harrison, W. 1989. The interprocedural analysis and automatic parallelisation of Scheme programs. *J. Lisp Symb. Comput. 2*, 176–396.

Hederman, L. 1988. Compile Time Garbage Collection Using Reference Count Analysis. Tech. Rep. Rice COMP TR88-75, Rice University, Houston, TX. Aug.

Horwitz, S., Demers, A., and Teitelbaum, T. 1987. An Efficient General Iterative Algorithm for Dataflow Analysis. *Acta Inf. 24,* 6, 679–694.

Hudak, P. 1986. A Semantic Model of Reference Counting and its Abstraction (Detailed Summary). In *Proceedings of the 1986 ACM Conference on LISP and functional programming.* ACM Press, Cambridge, MA, 351–363.

Hughes, S. 1992. Compile-Time Garbage Collection for Higher-Order Functional Languages. *J. Logic and Comput. 2,* 4, 483–509.

Inoue, K., Seki, H., and Yagi, H. 1988. Analysis of Functional Programs to Detect Run-Time Garbage Cells. *ACM Trans. Program. Lang. Syst. 10,* 4 (Oct.), 555–578.

Jones, N. D. and Muchnick, S. 1982. A Flexible Approach to Interprocedural Data Flow Analysis and Programs with Recursive Data Structures. In *9th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages.* ACM Press, Albuquerque, NM, 66–74.

McDowell, C. E. 1998. Reducing garbage in Java. *ACM Sigplan Notices 33,* 9 (Sept.), 84–86.

Mohnen, M. 1995a. Efficient Closure Utilisation by Higher-Order Inheritance Analysis. In *Static Analysis Symposium (SAS'95).* Lecture Notes on Computer Science, vol. 983. Springer Verlag, Glasgow, UK, 261–278.

Mohnen, M. 1995b. Efficient Compile-Time Garbage Collection for Arbitrary Data Structure. In *Symposium on Programming Language Implementation and Logic Programming (PLILP'95).* Lecture Notes on Computer Science, vol. 982. Springer Verlag, Utrecht, The Netherlands, 241–258.

Muller, G., Moura, B., Bellard, F., and Consel, C. 1997. Harissa: a Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code. In *3rd Usenix Conference on Object-Oriented Technologies and Systems (COOTS'97).* Portland, OR, 1–20.

Onodera, T. and Kawachiya, K. 1999. A Study of Locking Objects with Bimodal Fields. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99).* ACM Press, Denver, CO, 223–237.

Park, Y. G. and Goldberg, B. 1992. Escape Analysis on Lists. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation.* ACM Press, San Francisco, CA, 116–127.

Qian, Z. 1998. A Formal Specification of Java^{tm} Virtual Machine Instructions for Objects, Methods and Subroutines. In *Formal Syntax and Semantics of Java(tm)*, J. Alves-Foss, Ed. Lecture Notes on Computer Science, vol. 1523. Springer Verlag.

Ruf, E. 2000. Effective Synchronization Removal for Java. In *ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI'00).* ACM Press, Vancouver, Canada, 208–218.

Ruggieri, C. and Murtagh, T. P. 1988. Lifetime Analysis of Dynamically Allocated Objects. In *Proceedings of the 15th Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'88).* ACM Press, San Diego, CA, 285–293.

Serrano, M. and Feeley, M. 1996. Storage Use Analysis and its Applications. In *1996 ACM SIGPLAN International Conference on Functional Programming (ICFP).* ACM Press, Philadelphia, PA, 50–61.

Shivers, O. 1988. Control flow analysis in Scheme. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI'88).* ACM Press, Atlanta, GA, 164–174.

Tofte, M. and Talpin, J.-P. 1993. A theory of Stack Allocation in Polymorphically Typed Languages. Tech. Rep. 93/15, Departement of Computer Science, Copenhagen University. July.

$\ell \in Loc$	Set of locations (countable set)
$o \in Val$	$Loc \cup [-2^{31}, 2^{31}-1] \cup \{\texttt{null}\}$
$H \in Heap$	$(Loc \cup Class) \to (Field \cup \mathbb{N}) \to Val$
$L \in LocVar$	$V \to Val$
$J \in JavaStack$	$(LocVar \times PC)$ `list`
$TY \in OTypes$	$Loc \to RefType$
$\Xi \in State$	$Heap \times LocVar \times PC \times JavaStack \times OTypes$
$T \in Trace$	$Date \to State$

Fig. 17. Domains for the small step semantics

WEISS, M. ET AL. 1998. TurboJ: a Java Bytecode-to-Native Compiler. In *Proceedings of Workshop on Languages, Compilers and Tools for Embedded Systems*. Lecture Notes on Computer Science, vol. 1474. Springer Verlag, Montreal, Canada, 119–130.

WHALEY, J. AND RINARD, M. 1999. Compositional Pointer and Escape Analysis for Java Programs. In *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'99)*. ACM Press, Denver, CO, 187–206.

APPENDIX

A. CORRECTNESS OF ALIAS ANALYSIS

We prove the correctness of our alias analysis with respect to a small step semantics similar to that of [Qian 1998]. The main differences are that we do not separate stack and local variables, and we do not consider subroutines, uninitialized objects. Our semantics is described in Figures 17 and 18. Locations $\ell \in Loc$ represent objects allocated in the heap (a new location is created for each object). Values $o \in Val$ are either objects represented by their location or integers or the special value `null`. The heap associates with each object (or class for static fields) and each field $f \in Field$ (or array index $n \in \mathbb{N}$) its contents. $L \in LocVar$ yields for each local variable of the current method its contents. The Java stack J represents the stack of the callers of the current method (for each caller, J contains the value of the local variables and of the program counter just after the call). $TY \in OTypes$ represents the type of each object allocated on the heap. This type information is necessary to define the semantics of virtual method calls. All this information is summarized in the state $\Xi \in State$, and finally the execution trace T indicates the state at each step of the program execution.

The method really called by a virtual call is computed by $\mathrm{getMeth}((C, m', t), C') = (C', m', t)$ if the method (C', m', t) exists, otherwise $\mathrm{getMeth}((C, m', t), C') = \mathrm{getMeth}((C, m', t), C'')$ where C'' is the superclass of C'. The class C' must be a subclass of C. The program counter of the following instruction is: $\mathrm{next}(pc) = \mathrm{next}((m, a)) = (m, a+1)$. $\Xi[H]$ is the state Ξ where the heap has been replaced by H. We have similar notations for other parts of the state.

Definition A.1 (LOCATIONS REPRESENTED BY A PATH). We define $\mathrm{loc} : (Val \cup \{\mathrm{static}\}) \times State \times Path \to \mathcal{P}(Loc)$ such that $\mathrm{loc}(o, \Xi, p)$ is the set of locations of the part of value o represented by path p in state Ξ (this set contains at most one element).

Let H be the heap in state Ξ.

—$\mathrm{loc}(o, \Xi, p) = \emptyset$ if $o \notin Loc \cup \{\mathrm{static}\}$.

P(pc)	Transition
entry	$\Rightarrow \Xi[pc = (\mathtt{main}, 0), L, J = []]$
$v_1 = v_2$	$\Xi[pc, L] \Rightarrow \Xi[\mathrm{next}(pc), L[v_1 \mapsto L(v_2)]]$
$v_1 = (t)v_2$	$\Xi[pc, L] \Rightarrow \Xi[\mathrm{next}(pc), L[v_1 \mapsto L(v_2)]]$
$v_1 = \phi(v_2, v_3)$	$\Xi[pc, L] \Rightarrow \Xi[\mathrm{next}(pc), L[v_1 \mapsto L(v_2)$ or $L(v_3)^a]]$
$v_1.f = v_2$	$\Xi[pc, L, H] \Rightarrow \Xi[\mathrm{next}(pc), H[L(v_1) \mapsto H(L(v_1))[f \mapsto L(v_2)]]$
$v_2 = v_1.f$	$\Xi[pc, L, H] \Rightarrow \Xi[\mathrm{next}(pc), L[v_2 \mapsto H(L(v_1))(f)]]$
$C.f = v$	$\Xi[pc, L, H] \Rightarrow \Xi[\mathrm{next}(pc), H[C \mapsto H(C)[f \mapsto L(v)]]$
$v = C.f$	$\Xi[pc, L, H] \Rightarrow \Xi[\mathrm{next}(pc), L[v \mapsto H(C)(f)]]$
$v_1[v_2] = v_3$	$\Xi[pc, L, H] \Rightarrow \Xi[\mathrm{next}(pc), H[L(v_1) \mapsto H(L(v_1))[L(v_2) \mapsto L(v_3)]]$
$v_1 = v_2[v_3]$	$\Xi[pc, L, H] \Rightarrow \Xi[\mathrm{next}(pc), L[v_1 \mapsto H(L(v_2))(L(v_3))]]$
$v = \mathtt{null}$	$\Xi[pc, L] \Rightarrow \Xi[\mathrm{next}(pc), L[v \mapsto \mathtt{null}]]$
$v = \mathtt{new}\ C$	$\Xi[pc, L, H, TY] \Rightarrow \Xi[\mathrm{next}(pc), L[v \mapsto o], H[o \mapsto \{f \mapsto \mathtt{null}^b\}], TY[o \mapsto C]]$
	o is a new location, f is any field of the C class or a superclass of C.
$v = \mathtt{new}\ t[w]$	$\Xi[pc, L, H, TY] \Rightarrow \Xi[\mathrm{next}(pc), L[v \mapsto o], H[o \mapsto \{n \mapsto \mathtt{null}^c\}], TY[o \mapsto t\mathtt{[]}]]$
	o is a new location, $n \in [0, L(w) - 1]$
$w = v_0.m'(v_1, ..., v_j)$	$\Xi[pc, J, L, TY] \Rightarrow \Xi[(\mathrm{getMeth}(m', TY(o)), 0), (\mathrm{next}(pc), L) : J, \{p_i \mapsto v_i\}]$
	The p_i are the names of formal parameters of m'
$\mathtt{return}\ v$	$\Xi[pc, (pc', L') : J, L] \Rightarrow \Xi[pc', J, L'[w \mapsto L(v)]]$
	w is the variable in which we store the result of the corresponding method call.
$\mathtt{if}\ (...)\ \mathtt{goto}\ a$	$\Xi[pc = (m, x)] \Rightarrow \Xi[(m, a)]$ or $\Xi[\mathrm{next}(pc)]$

[a] Take $L(v_2)$ if v_2 has been defined after v_3 in the execution trace, or v_3 has never been defined. Take $L(v_3)$ otherwise.
[b] This assumes that field f contains objects or arrays. The initial value is 0 for integers.
[c] This assumes that t is an object or array type. The initial value is 0 for integers.

Fig. 18. Small step semantics

—$\mathrm{loc}(\ell, \Xi, \epsilon) = \{\ell\}$ if $\ell \in Loc$.

—$\mathrm{loc}(\ell, \Xi, f.p) = \mathrm{loc}(H(\ell)(f), \Xi, p)$ if $\ell \in Loc$ and $f \in Field$. If $H(\ell)(f)$ is not defined (the f field does not exist in the object), then $\mathrm{loc}(\ell, \Xi, f.p) = \emptyset$.

—$\mathrm{loc}(\ell, \Xi, n.p) = \mathrm{loc}(H(\ell)(n), \Xi, p)$ if $\ell \in Loc$, $n \in \mathbb{N}$. If $H(\ell)(n)$ is not defined, $\mathrm{loc}(\ell, \Xi, n.p) = \emptyset$.

—$\mathrm{loc}(\mathrm{static}, \Xi, (C, f, t).p) = \mathrm{loc}(H(C)(C, f, t), \Xi, p)$. $\mathrm{loc}(\mathrm{static}, \Xi, \epsilon) = \mathrm{loc}(\mathrm{static}, \Xi, n.p) = \emptyset$ where $n \in \mathbb{N}$.

Definition A.2 (Index). We define indices by $Idx = V \cup \{\mathrm{Old}(n, v) \mid n \in \mathbb{N}, v \in V\} \cup \{\mathrm{static}\}$. Indices will be used to designate local variables in a state or an abstract state. The local variable v in the nth element of the Java stack will be represented by $\mathrm{Old}(n, v)$. The local variable $v \in V$ in the method currently executed will the represented by v itself. static is used for static fields.

The value associated with an index is defined by: $\mathrm{Val} : State \times Idx \to (Val \cup \{\mathrm{static}\})$ is $\mathrm{Val}(\Xi[L], v) = L(v)$, $\mathrm{Val}(\Xi[J], \mathrm{Old}(l, v)) = L_l(v)$ with $J(l) = (L_l, pc_l)$, $\mathrm{Val}(\Xi, \mathrm{static}) = \mathrm{static}$.

The abstract value associated with an index $\mathrm{Val}^{\#} : State^{\#} \times Idx \to (OName \cup \{\emptyset, \mathrm{static}\})$ is defined by $\mathrm{Val}^{\#}((L, J), v) = L(v)$ and $\mathrm{Val}^{\#}((L, J), \mathrm{Old}(l, v)) = J(l)(v)$, $\mathrm{Val}^{\#}((L, J), \mathrm{static}) = \mathrm{static}$.

Definition A.3 (Correctness). Let $T \in Trace$ be an execution trace, $T^{\#} \in Trace^{\#}$ an abstract trace, $R \in DateAlias$ the alias relation, and $d \in Date$ a date. The relation $\mathrm{corr}(T, T^{\#}, R, d)$ means that the alias relation R and the abstract trace

$T^{\#}$ computed in Section 4 are correct with respect to the corresponding execution trace T at date d. It is defined as follows.

Let $\Xi = T(d)$ be the concrete state at date d. Let $\Xi_1 = T(d_1)$ and $\Xi_2 = T(d_2)$ be two states (with $d_1 \le d$, $d_2 \le d$), i_1 and $i_2 \in Idx$ two indices. Let $o_1 = \mathrm{Val}(T(d_1), i_1)$ and $o_2 = \mathrm{Val}(T(d_2), i_2)$ the two corresponding values, $N_1 = \mathrm{Val}^{\#}(T^{\#}(d_1), i_1)$ and $N_2 = \mathrm{Val}^{\#}(T^{\#}(d_2), i_2)$ the corresponding abstract values. Then $\mathrm{corr}(T, T^{\#}, R, d)$ if and only if for all $i_1, i_2 \in Idx$, $p_1, p_2 \in Path$, $d_1 \le d$, $d_2 \le d$,

$$\mathrm{loc}(o_1, \Xi, p_1) \cap \mathrm{loc}(o_2, \Xi, p_2) \neq \emptyset \Rightarrow (N_1.p_1 \sim N_2.p_2) \in R(d). \tag{9}$$

Essentially, if two paths $N_1.p_1$ and $N_2.p_2$ may represent the same location, then they are aliased.

Alias analysis has been defined in Figure 5.

Theorem A.4 (Correctness). $\forall d, \mathrm{corr}(T, T^{\#}, R, d)$.

Proof. By induction on d. Let us consider the $v_3.f = v_4$ case to show how the proof is performed. Let $\Xi = T(d)$, $\Xi' = T(d+1)$, $o_3 = \mathrm{Val}(\Xi, v_3)$, $o_4 = \mathrm{Val}(\Xi, v_4)$, $N_3 = \mathrm{Val}^{\#}(T^{\#}(d), v_3)$, $N_4 = \mathrm{Val}^{\#}(T^{\#}(d), v_4)$.

Assume that $\mathrm{loc}(o_1, \Xi', p_1) \cap \mathrm{loc}(o_2, \Xi', p_2) \neq \emptyset$. If $\mathrm{loc}(o_1, \Xi, p_1) \cap \mathrm{loc}(o_2, \Xi, p_2) \neq \emptyset$, the correctness at date d implies correctness at date $d+1$. Otherwise, at least one of $\mathrm{loc}(o_1, \Xi, p_1)$ or $\mathrm{loc}(o_2, \Xi, p_2)$ is changed when we replace Ξ by Ξ'. For example, assume that $\mathrm{loc}(o_1, \Xi, p_1) \neq \mathrm{loc}(o_1, \Xi', p_1)$ and $\mathrm{loc}(o_2, \Xi, p_2) = \mathrm{loc}(o_2, \Xi', p_2)$ (the other cases when $\mathrm{loc}(o_2, \Xi, p_2) \neq \mathrm{loc}(o_2, \Xi', p_2)$ or both are different are similar). In this case, the computation of $\mathrm{loc}(o_1, \Xi', p_1)$ uses the value of $H(o_3)(f)$, therefore $p_1 = q_1.f.q_2.f....f.q_n$, with $o_3 \in \mathrm{loc}(o_1, \Xi', q_1.f....q_i)$ $(i < n)$, and all uses of $H(o_3)(f)$ have been marked above by an f in the path, that is $\mathrm{loc}(o_4, \Xi', q_i) = \mathrm{loc}(o_4, \Xi, q_i)$ $(i > 1)$, $\mathrm{loc}(o_1, \Xi', q_1) = \mathrm{loc}(o_1, \Xi, q_1)$. Since $o_3 \in \mathrm{loc}(o_1, \Xi', q_1) = \mathrm{loc}(o_1, \Xi, q_1)$, by correctness at date d, $(N_1.q_1 \sim N_3) \in R(d)$. If $1 < i < n$, $o_3 \in \mathrm{loc}(o_4, \Xi', q_i) = \mathrm{loc}(o_4, \Xi, q_i)$, therefore $(N_4.q_i \sim N_3) \in R(d)$. $\mathrm{loc}(o_4, \Xi, q_n) \cap \mathrm{loc}(o_2, \Xi, p_2) \neq \emptyset$ therefore $(N_4.q_n \sim N_2.p_2) \in R(d)$. Moreover, we have added $N_3.f \sim N_4$, therefore $(N_3.f \sim N_4) \in R(d+1)$. By transitivity and right regularity of $R(d+1)$, $(N_1.q_1.f.q_2.f....f.q_n \sim N_2.p_2) \in R(d+1)$ hence the result. □

Theorem A.5 (Stack allocation). *Consider one execution of method m, in execution trace T, and assume that this execution of m terminates at date r.*

Consider a `new` *in method m, allocating object o. Let Res be the object name associated with the result of m, P_l the object names associated with the parameters of m (the names of these objects in the caller of m), N the object name of o. Let* $Esc = \{\mathrm{static}.p' \mid p' \in Path\} \cup \{Res.p' \mid p' \in Path\} \cup \cup_l \{P_l.p' \mid p' \in Path\}$.

If $\forall \pi \in Esc, (N.\epsilon \sim \pi) \notin R(r)$, o can be stack allocated in m in the considered execution trace T and in the execution of m that ends at date r.

Proof. The above correctness theorem, combined with $\forall \pi \in Esc, (N.\epsilon \sim \pi) \notin R(r)$, proves that, at the end of method m, the object o is not reachable from the parameters, the result of m or from static fields. So it is in fact not reachable, and can be stack allocated. □

B. CORRECTNESS PROOF FOR ANALYSIS $\mathbb{E}$

THEOREM B.1 (CORRECTNESS). *$\mathbb{E}$ and $\mathbb{E}_S$ are correct:* $\mathrm{corr}_{\mathbb{E}S}$.

PROOF. We show

(1) $\forall d, \mathrm{corr}_S(0,d), \mathrm{corr}_{\mathbb{E}}(0,d)$. Obvious, since $R^0(r/d/s) = \emptyset$.

(2) $(\forall d, \mathrm{corr}_S(k,d) \wedge \mathrm{corr}_{\mathbb{E}}(k,d)) \Rightarrow (\forall d, \mathrm{corr}_{\mathbb{E}}(k+1,d))$. By backward induction on d.

Let us consider the following interesting cases: $v_1 = v_2.f$, $v_1.f = v_2$, method call, and return. A full proof can be found in [Blanchet 2000]. Let $d' = d+1$.

(a) Case $v_1 = v_2.f$, where the f field contains an object. Assume that $N_1 = \mathrm{Val}^\#(T^\#(d'), v_1)$, $N_2 = \mathrm{Val}^\#(T^\#(d'), v_2)$. The created alias is $N_1 \sim N_2.f$. For $N \neq N_1, N_2$, the name N has the same aliases in $R^{k+1}(r/d/s)$ and in $R^{k+1}(r/d'/s)$, therefore

$$\mathrm{escap}^{k+1}(r/d/s)(N) = \mathrm{escap}^{k+1}(r/d'/s)(N). \tag{10}$$

For $N = N_2$, let

$$\begin{aligned} X &= \mathrm{escap}^{k+1}(r/d/s)(N_2)(c, c_0, ..., c_j) \\ &= \{p \mid \exists \pi \in Esc(c, c_0, ..., c_j), (N_2.p \sim \pi) \in R^{k+1}(r/d/s)\}. \end{aligned}$$

We consider the first alias of the chain $N_2.p \sim ... \sim \pi$ used to prove that $(N_2.p \sim \pi) \in R^{k+1}(r/d/s)$. Since this alias dates from after d, either it is $N_2.f.p' \sim N_1.p'$ or it dates from after d'. Also notice that N_1 is created at d'. Therefore all aliases of N_1 date from after d' except $N_1 \sim N_2.f$ but this alias cannot be used as second alias in this chain since this chain would then be $N_2.f.p' \sim N_1.p' \sim N_2.f.p' \sim ...$ contradicting the condition that the π_i must be pairwise distinct in the definition of $R^{k+1}(r/d/s)$. Therefore, if the first alias of the chain is $N_2.f.p' \sim N_1.p'$, then the second alias dates from after d'. Therefore

$$\begin{aligned} X = &\{f.p' \mid \exists \pi \in Esc(c, c_0, ..., c_j), (N_1.p' \sim \pi) \in R^k(r/d'/s)\} \\ &\cup \{p \mid \exists \pi \in Esc(c, c_0, ..., c_j), (N_2.p \sim \pi) \in R^{k+1}(r/d'/s)\} \\ X = &\, f.\mathrm{escap}^k(r/d'/s)(N_1)(c, c_0, ..., c_j) \cup \mathrm{escap}^{k+1}(r/d'/s)(N_2)(c, c_0, ..., c_j). \end{aligned}$$

Then

$$\mathrm{escap}^{k+1}(r/d/s)(N_2) = f.\mathrm{escap}^k(r/d'/s)(N_1) \sqcup \mathrm{escap}^{k+1}(r/d'/s)(N_2). \tag{11}$$

Let us assume that $(c, c_0, ..., c_j)$ satisfy the property:

$$\forall l, c_l \supseteq \mathbb{E}(m)(p_l)(c, c_0, ..., c_j). \tag{12}$$

For $v \neq v_1, v_2$, $\mathrm{Val}^\#(T^\#(d), v) = \mathrm{Val}^\#(T^\#(d'), v)$ and by (10), $\mathbb{E}^{k+1}(d)(v) = \mathbb{E}^{k+1}(d')(v)$. The correctness at date d' then implies the correctness at date d.
For $v = v_2$, by (11), $\mathbb{E}^{k+1}(d)(v_2) = f.\mathbb{E}^k(d')(v_1) \sqcup \mathbb{E}^{k+1}(d')(v_2)$.

$$\begin{aligned} \mathbb{E}^{k+1}(d)(v_2)(c, c_0, ..., c_j) &= f.\mathbb{E}^k(d')(v_1)(c, c_0, ..., c_j) \sqcup \mathbb{E}^{k+1}(d')(v_2)(c, c_0, ..., c_j) \\ &\subseteq f.\mathbb{E}(m)(v_1)(c, c_0, ..., c_j) \sqcup \mathbb{E}(m)(v_2)(c, c_0, ..., c_j) \end{aligned}$$

by correctness at date d'. We have the equation $\mathbb{E}(m)(v_2) \geq f.\mathbb{E}(m)(v_1)$, hence $\mathbb{E}^{k+1}(d)(v_2)(c, c_0, ..., c_j) \subseteq \mathbb{E}(m)(v_2)(c, c_0, ..., c_j)$, which is the expected correctness.

For $v = v_1$, $N = \mathrm{Val}^{\#}(T^{\#}(d), v_1)$ disappears at date d' (since each object name is only present in one variable at a time, and N is overwritten by the new name N_1 at d'). Then $\mathbb{E}^{k+1}(d)(v_1) = \mathrm{escap}^{k+1}(r/d/s)(N) = \mathrm{escap}^{k+1}(r/d'/s)(N) = \emptyset$ since N does not exist any more at d', no alias of N can therefore be created after d'. Then any value of $\mathbb{E}(m)(v_1)$ is correct at date d.

(b) Case $v_1.f = v_2$ where the f field contains an object. Let $N_1 = \mathrm{Val}^{\#}(T^{\#}(d), v_1)$ and $N_2 = \mathrm{Val}^{\#}(T^{\#}(d), v_2)$. The created alias is $N_1.f \sim N_2$. For $N \neq N_1, N_2$, the name N has the same aliases in $R^{k+1}(r/d'/s)$ as in $R^{k+1}(r/d/s)$, then

$$\mathrm{escap}^{k+1}(r/d/s)(N) = \mathrm{escap}^{k+1}(r/d'/s)(N). \tag{13}$$

For $N = N_1$, let

$$\begin{aligned} X &= \mathrm{escap}^{k+1}(r/d/s)(N_1)(c, c_0, ..., c_j) \\ &= \{p \mid \exists \pi \in \mathit{Esc}(c, c_0, ..., c_j), (N_1.p \sim \pi) \in R^{k+1}(r/d/s)\}. \end{aligned}$$

Since the first alias of the chain $N_1.p \sim ... \sim \pi$ used to prove that $(N_1.p \sim \pi) \in R^{k+1}(r/d/s)$ dates from after d, either it is $N_1.f \sim N_2$, or it dates from after d'.

$$\begin{aligned} X &\subseteq \{f.p' \mid \exists \pi \in \mathit{Esc}(c, c_0, ..., c_j), (N_2.p' \sim \pi) \in R^k(r/s/s)\} \\ &\quad \cup \{p \mid \exists \pi \in \mathit{Esc}(c, c_0, ..., c_j), (N_1.p \sim \pi) \in R^{k+1}(r/d'/s)\} \\ X &\subseteq f.\mathrm{escap}^k(r/s/s)(N_2)(c, c_0, ..., c_j) \cup \mathrm{escap}^{k+1}(r/d'/s)(N_1)(c, c_0, ..., c_j) \end{aligned}$$

Hence

$$\mathrm{escap}^{k+1}(r/d/s)(N_1) \leq f.\mathrm{escap}^k(r/s/s)(N_2) \sqcup \mathrm{escap}^{k+1}(r/d'/s)(N_1). \tag{14}$$

We now assume (12).

For $v \neq v_1, v_2$, $\mathrm{Val}^{\#}(T^{\#}(d), v) = \mathrm{Val}^{\#}(T^{\#}(d'), v)$ and by (13), $\mathbb{E}^{k+1}(d)(v) = \mathbb{E}^{k+1}(d')(v)$. The correctness at date d' then implies the correctness at date d.

For $v = v_1$, by (14), $\mathbb{E}^{k+1}(d)(v_1) \leq f.\mathbb{E}_{\mathrm{S}}^k(d)(v_2) \sqcup \mathbb{E}^{k+1}(d')(v_1)$. By correctness of $\mathbb{E}_{\mathrm{S}}$ at date d and iteration k and by correctness of $\mathbb{E}$ at date d' and iteration $k+1$,

$$\begin{aligned} \mathbb{E}^{k+1}(d)(v_1)(c, c_0, ..., c_j) &\subseteq f.\mathbb{E}_{\mathrm{S}}(m)(v_2)(c, c_0, ..., c_j) \cup \mathbb{E}(m)(v_1)(c, c_0, ..., c_j) \\ &\subseteq \mathbb{E}(m)(v_1)(c, c_0, ..., c_j) \end{aligned}$$

since we have the equation $\mathbb{E}(m)(v_1) \geq f.\mathbb{E}_{\mathrm{S}}(m)(v_2)$. This is the correctness.

The case $v = v_2$, $N = N_2$ is similar to $v = v_1$, $N = N_1$ (just swap N_1 and N_2, v_1 and v_2 and replace f with f^{-1}).

(c) Case $w = v_0.m'(v_1, ..., v_n)$. Let d be the date just before the method call, $d' = d+1$ just after, and d'' after the corresponding `return`. Let $P'_l = \mathrm{Val}^{\#}(T^{\#}(d), v_l)$ be the object names of the parameters of m', $P''_l = \mathrm{Val}^{\#}(T^{\#}(d'), p'_l)$ be the object names of the parameters of m' inside m'. The method call creates the aliases $P'_l \sim P''_l$. Let Res' be the name of the result of m'. Let $\mathit{Esc}'(c', c'_0, ..., c'_n) = \{\mathrm{static}.p' \mid p' \in \mathit{Path}\} \cup \{\mathit{Res}'.p' \mid p' \in c'\} \cup \cup_l \{P'_l.p' \mid p' \in c'_l\}$ be the Esc associated with m'.

If N is not one of the P'_l, N is not accessible during the evaluation of m', hence no alias of N is created between d and d'', and

$$\mathrm{escap}^{k+1}(r/d/s)(N) = \mathrm{escap}^{k+1}(r/d''/s)(N). \tag{15}$$

Then the correctness at date d'' implies the correctness at date d for the variables v different from v_l.
For the variables v_l, the general idea is to split the alias chain, to separate the aliases created by m' from those created outside m'. Let

$$\begin{aligned} X &= \text{escap}^{k+1}(r/d/s)(P'_l)(c, c_0, ..., c_j) \\ &= \{p \mid \exists \pi \in \mathit{Esc}(c, c_0, ..., c_j), (P'_l.p \sim \pi) \in R^{k+1}(r/d/s)\}. \end{aligned}$$

Either the first alias of the chain is $P'_l \sim P''_l$ or it has been created after d'' (since P'_l is not accessible during the runtime of m'). In the first case, the alias chain can be decomposed into $P'_l.p \sim P''_l.p \sim ... \sim \pi' \sim ... \sim \pi$, where all aliases of the $P''_l.p \sim \pi'$ part have been created during the runtime of m', and the first alias of the chain $\pi' \sim \pi$ has been created outside m' or $\pi' = \pi$. Then $(P''_l.p \sim \pi') \in R^k(d'' - 1/d'/d)$. Let $c' = \text{escap}^k(r/d''/s)(\mathit{Res}')(c, c_0, ..., c_j)$, $c'_i = \text{escap}^k(r/s/s)(P'_i)(c, c_0, ..., c_j)$. The path π' has to be accessible both from inside m' and outside m' (because either the first alias of the chain $\pi' \sim \pi$ has been created outside m', or $\pi' = \pi \in \mathit{Esc}(c, c_0, ..., c_j)$). So we can distinguish three cases for $\pi' \sim \pi$:

—If $\pi' = \mathit{Res}'.p'$, the first alias of the chain $\pi' \sim \pi$ cannot have been created before the call to m' since Res' did not exist yet. Therefore, it has been created after this call, and $(\pi' \sim \pi) \in R^k(r/d''/s)$. Then $p' \in \text{escap}^k(r/d''/s)(\mathit{Res}')(c, c_0, ..., c_j) = c'$, and $\pi' \in \mathit{Esc}'(c', c'_0, ..., c'_n)$.
—If $\pi' = P'_i.p'$, $(\pi' \sim \pi) \in R^k(r/s/s)$. Then $p' \in \text{escap}^k(r/s/s)(P'_i)(c, c_0, ..., c_j) = c'_i$, and $\pi' \in \mathit{Esc}'(c', c'_0, ..., c'_n)$.
—If $\pi' = \text{static}.p'$, $\pi' \in \mathit{Esc}'(c', c'_0, ..., c'_n)$.

In all cases, $\pi' \in \mathit{Esc}'(c', c'_0, ..., c'_n)$ and $(P''_l.p \sim \pi') \in R^k(d'' - 1/d'/d)$ therefore $p \in \text{escap}^k(d'' - 1/d'/d)(P''_l)(c', c'_0, ..., c'_n)$ and

$$\begin{aligned} \text{escap}^{k+1}(r/d/s)(P'_l) \leq\ & \text{escap}^k(d'' - 1/d'/d)(P''_l) \circ \\ & (\text{escap}^k(r/d''/s)(\mathit{Res}'), \text{escap}^k(r/s/s)(P'_0), ..., \text{escap}^k(r/s/s)(P'_n)) \\ \sqcup\ & \text{escap}^{k+1}(r/d''/s)(P'_l). \end{aligned} \tag{16}$$

Then $\mathbb{E}^{k+1}(d)(v_l) \leq \mathbb{E}^k(d')(p'_l) \circ (\mathbb{E}^k(d'')(w), \mathbb{E}^k_{\mathrm{S}}(d)(v_0), ..., \mathbb{E}^k_{\mathrm{S}}(d)(v_n)) \sqcup \mathbb{E}^{k+1}(d'')(v_l)$. Let $c'' = \mathbb{E}(m)(w)(c, c_0, ..., c_j)$, $c''_i = \mathbb{E}_{\mathrm{S}}(m)(v_i)(c, c_0, ..., c_j)$, and assume (12). The correctness at iteration k yields $\mathbb{E}^k(d'')(w)(c, c_0, ..., c_j) \subseteq \mathbb{E}(m)(w)(c, c_0, ..., c_j) = c''$, $\mathbb{E}^k_{\mathrm{S}}(d)(v_i)(c, c_0, ..., c_j) \subseteq \mathbb{E}_{\mathrm{S}}(m)(v_i)(c, c_0, ..., c_j) = c''_i$. Then

$$\mathbb{E}^{k+1}(d)(v_l)(c, c_0, ..., c_j) \subseteq \mathbb{E}^k(d')(p'_l)(c'', c''_0, ..., c''_n) \cup \mathbb{E}^{k+1}(d'')(v_l)(c, c_0, ..., c_j).$$

By correctness at iteration k, $\mathbb{E}^k(d')(p'_l)(c'', c''_0, ..., c''_n) \subseteq \mathbb{E}(m')(p'_l)(c'', c''_0, ..., c''_n)$ (since by Lemma 5.3.4, $c''_i = \mathbb{E}_{\mathrm{S}}(m)(v_i)(c, c_0, ..., c_j) \supseteq \mathbb{E}(m)(v_i)(c, c_0, ..., c_j) \supseteq \mathbb{E}(m')(p'_i)(c'', c''_0, ..., c''_n)$). Therefore, using also the correctness of $\mathbb{E}$ at date d'',

$$\mathbb{E}^{k+1}(d)(v_l)(c, c_0, ..., c_j) \subseteq \mathbb{E}(m')(p'_l)(c'', c''_0, ..., c''_n) \cup \mathbb{E}(m)(v_l)(c, c_0, ..., c_j).$$

Then using equations $\mathbb{E}(m)(v_l) \geq \rho(m')(l) \circ (\mathbb{E}(m)(w), \mathbb{E}_{\mathrm{S}}(m)(v_0), ..., \mathbb{E}_{\mathrm{S}}(m)(v_n))$ and $\rho(m')(l) \geq \mathbb{E}(m')(p'_l)$, we get $\mathbb{E}^{k+1}(d)(v_l)(c, c_0, ..., c_j) \subseteq \mathbb{E}(m)(v_l)(c, c_0, ..., c_j)$, which is the required correctness.
(d) Case `return` v. Let $\mathit{Res} = \mathrm{Val}^\#(T^\#(d), v)$. We have $\text{escap}^{k+1}(r/d/s)(\mathit{Res})(c, c_0, ..., c_j) = c$. For names $N \neq \mathit{Res}$, $\text{escap}^{k+1}(r/d/s)(N)(c, c_0, ..., c_j) = \emptyset$ (the first alias of the chain $N.p \sim ... \sim \pi$ considered in the definition of $R^{k+1}(r/d/s)$

should be after d and before r, and this is impossible since $d = r$. Therefore, only the empty alias chain $\pi \sim \pi$ has to be considered. Since N is different from the parameter names P_l and from *Res*, it has no alias with elements of *Esc*).
Hence $\mathbb{E}^{k+1}(d)(v)(c, c_0, ..., c_j) = c = \mathit{first}_{\mathbb{E}}(c, c_0, ..., c_j) \subseteq \mathbb{E}(m)(v)(c, c_0, ..., c_j)$, and for $v' \neq v$, $\mathbb{E}^{k+1}(d)(v')(c, c_0, ..., c_j) = \emptyset \subseteq \mathbb{E}(m)(v')(c, c_0, ..., c_j)$, which is the expected correctness.

(3) $(\forall d, \mathrm{corr}_{\mathbb{E}}(k, d)) \Rightarrow (\forall d, \mathrm{corr}_{\mathrm{S}}(k, d))$. By forward induction on d.
We consider again the same cases as above. Let $d' = d + 1$.
(a) Case $v_1 = v_2.f$ where the field f contains an object. For $v \neq v_1$, the correctness comes immediately from the induction hypothesis on d, since $\mathbb{E}_{\mathrm{S}}^k(d')(v) = \mathbb{E}_{\mathrm{S}}^k(d)(v)$. For $v = v_1$, let $N_1 = \mathrm{Val}^{\#}(T^{\#}(d'), v_1)$ be the new name, let $N_2 = \mathrm{Val}^{\#}(T^{\#}(d), v_2)$.

$$\begin{aligned}
&\mathrm{escap}^k(r/s/s)(N_1)(c, c_0, ..., c_j) \\
&\quad = \{p \mid \exists \pi \in \mathit{Esc}(c, c_0, ..., c_j), (N_1.p \sim \pi) \in R^k(r/s/s)\} \\
&\quad \subseteq \{p \mid \exists \pi \in \mathit{Esc}(c, c_0, ..., c_j), (N_2.f.p \sim \pi) \in R^{k-1}(r/s/s)\} \\
&\quad\quad \cup \{p \mid \exists \pi \in \mathit{Esc}(c, c_0, ..., c_j), (N_1.p \sim \pi) \in R^k(r/d'/s)\}
\end{aligned}$$

since if the first alias used to show that $(N_1.p \sim \pi) \in R^k(r/d'/s)$ dates from before d', then we use alias $N_1 \sim N_2.f$ followed by $N_2.f.p \sim \pi$. Then

$$\mathrm{escap}^k(r/s/s)(N_1) \leq f^{-1}.\mathrm{escap}^{k-1}(r/s/s)(N_2) \sqcup \mathrm{escap}^k(r/d'/s)(N_1). \tag{17}$$

Therefore

$$\mathbb{E}_{\mathrm{S}}^k(d')(v_1) \leq f^{-1}.\mathbb{E}_{\mathrm{S}}^{k-1}(d)(v_2) \sqcup \mathbb{E}^k(d')(v_1).$$

Therefore, assuming (12), by correctness of $\mathbb{E}_{\mathrm{S}}$ at date d and iteration $k-1$ and correctness of $\mathbb{E}$ at date d' and iteration k,

$$\begin{aligned}
\mathbb{E}_{\mathrm{S}}^k(d)(v_1)(c, c_0, ..., c_j) &\subseteq f^{-1}.\mathbb{E}_{\mathrm{S}}(m)(v_2)(c, c_0, ..., c_j) \cup \mathbb{E}(m)(v_1)(c, c_0, ..., c_j) \\
&\subseteq \mathbb{E}_{\mathrm{S}}(m)(v_1)(c, c_0, ..., c_j) \cup \mathbb{E}(m)(v_1)(c, c_0, ..., c_j)
\end{aligned}$$

since $\mathbb{E}_{\mathrm{S}}(m)(v_1) = f^{-1}.\mathbb{E}_{\mathrm{S}}(m)(v_2)$. By Lemma 5.3.4, $\mathbb{E}_{\mathrm{S}}(m)(v_1)(c, c_0, ..., c_j) \supseteq \mathbb{E}(m)(v_1)(c, c_0, ..., c_j)$. This gives the correctness.
(b) Case $v_1.f = v_2$: obvious by induction hypothesis on d ($\mathrm{Val}^{\#}(T^{\#}(d'), v) = \mathrm{Val}^{\#}(T^{\#}(d), v)$ and $R^k(r/s/s)$ does not depend on d).
(c) Entry of m (method call). Let P_l be the names of the parameters of m in the caller, and $P'_l = \mathrm{Val}^{\#}(T^{\#}(s+1), p_l)$ the names of the parameters of m, in m. The aliases created by the method call are $P_l \sim P'_l$. Let

$$\begin{aligned}
X &= \mathrm{escap}^k(r/s/s)(P'_l)(c, c_0, ..., c_j) \\
&= \{p \mid \exists \pi \in \mathit{Esc}(c, c_0, ..., c_j), (P'_l.p \sim \pi) \in R^k(r/s/s)\}.
\end{aligned}$$

Either the first alias of the chain used to prove $(P'_l.p \sim \pi) \in R^k(r/s/s)$ is $P'_l \sim P_l$, or it dates from after $s+1$. Moreover the only alias of P_l created between s and r is $P_l \sim P'_l$, and it cannot be used again in this chain since the π_i have to be pairwise distinct in the definition of $R^k(r/s/s)$. Then, if the first alias is $P'_l \sim P_l$, $\pi = P_l.p$. Hence,

$$\begin{aligned}
X = &\{p \mid P_l.p \in \mathit{Esc}(c, c_0, ..., c_j)\} \\
&\cup \{p \mid \exists \pi \in \mathit{Esc}(c, c_0, ..., c_j), (P'_l.p \sim \pi) \in R^k(r/s+1/s)\}
\end{aligned}$$

$$X = c_l \cup \text{escap}^k(r/s+1/s)(P'_l)(c, c_0, ..., c_j).$$

Hence

$$\text{escap}^k(r/s/s)(P'_l) = P_{\mathbb{E}}(p_l) \sqcup \text{escap}^k(r/s+1/s)(P'_l). \tag{18}$$

Then $\mathbb{E}^k_{\mathrm{S}}(s+1)(p_l) = P_{\mathbb{E}}(p_l) \sqcup \mathbb{E}^k(s+1)(p_l)$. We have by (12) and by correctness of $\mathbb{E}$, $c_l \supseteq \mathbb{E}(m)(p_l)(c, c_0, ..., c_j) \supseteq \mathbb{E}^k(s+1)(p_l)(c, c_0, ..., c_j)$. Therefore we obtain the correctness: $\mathbb{E}^k_{\mathrm{S}}(s+1)(p_l)(c, c_0, ..., c_j) \subseteq c_l = \mathbb{E}_{\mathrm{S}}(m)(p_l)(c, c_0, ..., c_j)$.
(d) Case `return` v. Assume that the corresponding call is $w = v_0.m'(v_1, ..., v_n)$. Let d be the date just before the method call, $d' = d+1$ just after, and d'' after the corresponding `return`. Let $P'_l = \mathrm{Val}^{\#}(T^{\#}(d), v_l)$ be the names of the parameters of m'. Let Res' be the name of the result of m'. Let $Esc'(c', c'_0, ..., c'_n) = \{\text{static}.p' \mid p' \in Path\} \cup \{Res'.p' \mid p' \in c'\} \cup \cup_l \{P'_l.p' \mid p' \in c'_l\}$ be the Esc associated with m'. For variables v' different from w, $\mathrm{Val}^{\#}(T^{\#}(d''), v') = \mathrm{Val}^{\#}(T^{\#}(d), v')$ and the correctness at date d'' comes immediately from the correctness at date d.
For w, let

$$\begin{aligned} X &= \text{escap}^k(r/s/s)(Res')(c, c_0, ..., c_j) \\ &= \{p \mid \exists \pi \in Esc(c, c_0, ..., c_j), (Res'.p \sim \pi) \in R^{k+1}(r/s/s)\}. \end{aligned}$$

The first alias of the chain used to prove $(Res'.p \sim \pi) \in R^{k+1}(r/s/s)$ has either been created after d'' or between d and d''. In the latter case, the alias chain is of the form $Res'.p \sim ... \sim \pi' \sim ... \sim \pi$ where all aliases of the $Res'.p \sim \pi'$ part have been created during the runtime of m', and the first alias of the chain $\pi' \sim \pi$ has been created outside m' or $\pi' = \pi$. Then $(Res'.p \sim \pi') \in R^k(d''-1/d/d)$. Let $c' = \text{escap}^{k-1}(r/d''/s)(Res')(c, c_0, ..., c_j)$, $c'_i = \text{escap}^{k-1}(r/s/s)(P'_i)(c, c_0, ..., c_j)$. We prove like in the case of a method call in the backward part that $\pi' \in Esc'(c', c'_0, ..., c'_n)$. Therefore $p \in \text{escap}^k(d''-1/d/d)(Res')(c', c'_0, ..., c'_n)$ and

$$\begin{aligned} &\text{escap}^k(r/s/s)(Res') \le \text{escap}^k(d''-1/d/d)(Res') \circ \\ &\quad (\text{escap}^k(r/d''/s)(Res'), \text{escap}^k(r/s/s)(P'_0), ..., \text{escap}^k(r/s/s)(P'_n)) \\ &\sqcup \text{escap}^k(r/d''/s)(Res'). \end{aligned} \tag{19}$$

Therefore,

$$\mathbb{E}^k_{\mathrm{S}}(d'')(w) \le \mathbb{E}^k_{\mathrm{S}}(d''-1)(v) \circ (\mathbb{E}^k(d'')(w), \mathbb{E}^k_{\mathrm{S}}(d)(v_0), ..., \mathbb{E}^k_{\mathrm{S}}(d)(v_n)) \sqcup \mathbb{E}^k(d'')(w).$$

Let $c'' = \mathbb{E}(m)(w)(c, c_0, ..., c_j)$, $c''_i = \mathbb{E}_{\mathrm{S}}(m)(v_i)(c, c_0, ..., c_j)$ and assume (12). Using the correctness of $\mathbb{E}_{\mathrm{S}}$ at date d and of $\mathbb{E}$, we have $\mathbb{E}^k(d'')(w)(c, c_0, ..., c_j) \subseteq \mathbb{E}(m)(w)(c, c_0, ..., c_j) = c''$, $\mathbb{E}^k_{\mathrm{S}}(d)(v_i)(c, c_0, ..., c_j) \subseteq \mathbb{E}_{\mathrm{S}}(m)(v_i)(c, c_0, ..., c_j) = c''_i$. Therefore

$$\mathbb{E}^k_{\mathrm{S}}(d'')(w)(c, c_0, ..., c_j) \subseteq \mathbb{E}^k_{\mathrm{S}}(d''-1)(v)(c'', c''_0, ..., c''_n) \cup \mathbb{E}^k(d'')(w)(c, c_0, ..., c_j).$$

By correctness of $\mathbb{E}_{\mathrm{S}}$ at date $d''-1$, $\mathbb{E}^k_{\mathrm{S}}(d''-1)(v)(c'', c''_0, ..., c''_n) \subseteq \mathbb{E}_{\mathrm{S}}(m')(v)(c'', c''_0, ..., c''_n)$ (using Lemma 5.3.4 as in the method call case of the backward part). Then, using also the correctness of $\mathbb{E}$ at date d'',

$$\mathbb{E}^k_{\mathrm{S}}(d'')(w)(c, c_0, ..., c_j) \subseteq \mathbb{E}_{\mathrm{S}}(m')(v)(c'', c''_0, ..., c''_n) \cup \mathbb{E}(m)(w)(c, c_0, ..., c_j).$$

Then, using equations $\mathbb{E}_S(m)(w) = \rho'_S(m') \circ (\mathbb{E}(m)(w), \mathbb{E}_S(m)(v_0), ..., \mathbb{E}_S(m)(v_n))$ and $\rho'_S(m') \geq \mathbb{E}_S(m')(v)$, we have $\mathbb{E}^k_S(d'')(w)(c, c_0, ..., c_j) \subseteq \mathbb{E}_S(m)(w)(c, c_0, ..., c_j)$, which is the correctness.

(4) $\forall k, \mathrm{corr}_{\mathbb{E}}(k, d) \Rightarrow \mathrm{corr}_{\mathbb{E}}(\infty, d)$, $\forall k, \mathrm{corr}_S(k, d) \Rightarrow \mathrm{corr}_S(\infty, d)$. Obvious since $\cup_{k\in\mathbb{N}} R^k(r/d/s) = R^\infty(r/d/s)$.

The result follows by induction on k. $\square$

C. CORRECTNESS PROOF FOR ANALYSIS $\mathbb{L}$

LEMMA C.1. *Let $t\texttt{[]}^k$ be the type $t\texttt{[]}...\texttt{[]}$ with k times* `[]`. *Assume that τ and τ' are array types with k and k' dimensions, $\tau = t\texttt{[]}^k$, $\tau' = t'\texttt{[]}^{k'}$, t and t' are not arrays.*

$$\mathrm{convert}(\tau, \tau')(n) = (n \sqcap \top[\tau'])\sqcup \begin{cases} \top[t'\texttt{[]}^{k'-i}] & \textit{if } i \geq 0 \textit{ minimum such that } i \leq k, i \leq k' \textit{ and } n = \top[t\texttt{[]}^{k-i}], \\ \top[t'] & \textit{if } k > k' \textit{ and } \top[t] \leq n < \top[t\texttt{[]}^{k-k'}], \\ 0 & \textit{if } n < \top[t]. \end{cases}$$

PROOF. If $n = \top[\tau]$, $\gamma^\tau(n) = \mathit{Path}$, therefore $\alpha^{\tau'} \circ \gamma^\tau(n) = \top[\tau']$.
If $n = 0$, $\alpha^{\tau'} \circ \gamma^\tau(n) = 0$.
Otherwise, if τ is not an array and $0 < n < \top[\tau]$,

$$\begin{aligned} \alpha^{\tau'} \circ \gamma^\tau(n) &= \sqcup\{\top[\tau'.n_1...n_j.(C, f, t).c'] \mid \top[\tau.n_1...n_j.(C, f, t).c'] \leq n, n_i \in \mathbb{N}\} \\ &= \sqcup\{\top[\tau'.n_1...n_j.(C, f, t).c'] \mid \top[\tau.(C, f, t).c'] \leq n, n_i \in \mathbb{N}\} \\ &= n \sqcap \top[\tau'] \end{aligned}$$

since $\top[\tau'.n_1...n_j.(C, f, t).c'] \leq \top[\tau'] \sqcap \top[t.c']$, this value being reached when there is no n_i ($j = 0$).
If τ is an array and $0 < n < \top[\tau]$,

$$\begin{aligned} \alpha^{\tau'} \circ \gamma^\tau(n) = &\sqcup\{\top[\tau'.(C, f, t).c'] \mid \top[\tau.(C, f, t).c'] \leq n\} \\ &\sqcup \sqcup\{\top[\tau'.n'.c'] \mid \top[\tau.n'.c'] \leq n, n' \in \mathbb{N}\} \\ = &(n \sqcap \top[\tau']) \sqcup \alpha^{t'} \circ \gamma^t(n) \end{aligned}$$

where $t\texttt{[]} = \tau$ and if τ' is an array $t'\texttt{[]} = \tau'$, otherwise $t' = \tau'$.
To sum up the array case, we have proved that: If $\tau = t\texttt{[]}$ is an array,

$$\begin{cases} \text{If } n = \top[\tau], & \mathrm{convert}(\tau, \tau')(n) = \top[\tau'] \\ \text{If } 0 < n < \top[\tau], & \mathrm{convert}(\tau, \tau')(n) = (n \sqcap \top[\tau']) \sqcup \mathrm{convert}(t, t')(n) \\ & \text{where } t' = \tau' \text{ if } \tau' \text{ is not an array, and } t'\texttt{[]} = \tau' \text{ otherwise} \\ \text{If } n = 0, & \mathrm{convert}(\tau, \tau')(n) = 0 \end{cases}$$

Using this result, an induction over the number of dimensions k of τ yields the expected result. $\square$

THEOREM C.2 (CORRECTNESS). *Assume that* $\mathrm{corr}_{\mathbb{L}}(v_1, E_1)$ *and* $\mathrm{corr}_{\mathbb{L}}(v_2, E_2)$. *Then* $\mathrm{corr}_{\mathbb{L}}(v_1 \sqcup_{\mathbb{L}} v_2, E_1 \sqcup E_2)$, $\mathrm{corr}_{\mathbb{L}}((C, f, t)^{-1}_{\mathbb{L}}.v_1, (C, f, t)^{-1}.E_1)$, *and similarly for other primitive operations.*

PROOF. We will only handle the conversion, composition, construction and restriction cases.

(1) We have $\mathrm{corr}_{\mathbb{L}}((\alpha^{\tau'} \circ \gamma^{\tau}, (\tau), \tau'), \mathrm{Id})$ hence the correctness of the conversion: $\mathrm{corr}_{\mathbb{L}}((\mathrm{convert}(\tau, \tau'), (\tau), \tau'), \mathrm{Id})$.

(2) Composition: Let $v_k = (\phi_k, (\tau_{-1}, ..., \tau_i), \tau'_k)$, $v' = (\phi', (\tau'_{-1}, ..., \tau'_j), \tau'')$. $v' \circ_{\mathbb{L}} (v_{-1}, ..., v_j) = (\phi' \circ (\phi_{-1}, ..., \phi_j), (\tau_{-1}, ..., \tau_i), \tau'')$. Assume that $\mathrm{corr}_{\mathbb{L}}(v_k, E_k)$ and $\mathrm{corr}_{\mathbb{L}}(v', E')$. Then $\mathrm{corr}_{\mathbb{L}}(v' \circ_{\mathbb{L}} (v_{-1}, ..., v_j), E' \circ (E_{-1}, ..., E_j))$. Indeed

$$\begin{aligned}
&\alpha^{\tau''}((E' \circ (E_{-1}, ..., E_j))(\gamma^{\tau_{-1}}(n_{-1}), ..., \gamma^{\tau_i}(n_i))) \\
&\quad \leq \alpha^{\tau''}(E'(\gamma^{\tau'_{-1}}(\alpha^{\tau'_{-1}}(E_{-1}(\gamma^{\tau_{-1}}(n_{-1}), ..., \gamma^{\tau_i}(n_i)))), ...)) \\
&\quad \leq (\phi' \circ (\phi_{-1}, ..., \phi_j))(n_{-1}, ..., n_i)
\end{aligned}$$

by correctness of v', $v_{-1}, ..., v_j$.

(3) Construction for objects: Assume $t \in \mathrm{Cont}(t')$ and t' is not an array.

$$\begin{aligned}
\alpha^{t'}((C, f, t).\gamma^t(n)) &= \alpha^{t'}(\{(C, f, t).c \mid \top[t.c] \leq n\}) \\
&= \sqcup\{\top[t'.(C, f, t).c] \mid \top[t.c] \leq n\} \\
&= \sqcup\{\top[t'] \sqcap \top[t.c] \mid \top[t.c] \leq n\} \\
&\leq n
\end{aligned}$$

(and since $n \leq \top[t]$ and $\top[t] \leq \top[t']$ by the second rule of computation of $\top$, it is useless to write $n \sqcap \top[t']$). Hence, $\mathrm{corr}_{\mathbb{L}}((\mathrm{Id}, (t), t'), \{c \mapsto (C, f, t).c\})$.

(4) Construction for arrays:

$$\alpha^{t[]}(\mathbb{N}.\gamma^t(n)) = \sqcup\{\top[t[].n'.c] \mid \top[t.c] \leq n, n' \in \mathbb{N}\} \leq n$$

Hence $\mathrm{corr}_{\mathbb{L}}((\mathrm{Id}, (t), t[]), \{c \mapsto \mathbb{N}.c\})$.

(5) Restriction for objects: Assume $t \in \mathrm{Cont}(\tau)$ and τ is not an array. If $n \geq \top[\tau]$,

$$\alpha^t((C, f, t)^{-1}.\gamma^{\tau}(n)) = \alpha^t((C, f, t)^{-1}.Path) = \top[t] = n \sqcap \top[t]$$

(here, we use the second rule of computation of type heights, to make sure that $\top[\tau] \geq \top[t]$, and therefore if $n \geq \top[\tau]$, then $n \geq \top[t]$). Otherwise,

$$\alpha^t((C, f, t)^{-1}.\gamma^{\tau}(n)) = \sqcup\{\top[t.c] \mid \top[\tau.(C, f, t).c] \leq n\} \leq n \sqcap \top[t]$$

(since $\top[\tau.(C, f, t).c] = \top[\tau] \sqcap \top[t.c]$ and since $n < \top[\tau]$, $\top[\tau.(C, f, t).c] \leq n$ if and only if $\top[t.c] \leq n$. Moreover, $\forall t, c, \top[t.c] \leq \top[t]$). Hence $\mathrm{corr}_{\mathbb{L}}((\{n \mapsto n \sqcap \top[t]\}, (\tau), t), \{c \mapsto (C, f, t)^{-1}.c\})$.

(6) Restriction for arrays:

$$\alpha^t(\mathbb{N}^{-1}.\gamma^{t[]}(n)) = \sqcup\{\top[t.c] \mid \top[t[].\mathbb{N}.c] \leq n\} \leq n \sqcap \top[t]$$

since $\forall t, c, \top[t.c] \leq \top[t]$. Therefore $\mathrm{corr}_{\mathbb{L}}((\{n \mapsto n \sqcap \top[t]\}, (t[]), t), \{c \mapsto \mathbb{N}^{-1}.c\})$.

Using the correctness of composition and the elementary correctness proofs for conversion, constructions and restrictions above, we get immediately the required result. □