A COMPARISON OF EAGER AND LAZY CLASS INITIALIZATION IN JAVA

STANISLAW M. DZIOBIAK

Abstract. We prove that under some natural condition eager class initialization of a Java program P, as proposed in [3], does not depend on the choice of a topological sort of the graph of class initialization dependencies of P. We also identify further natural conditions under which the eager and lazy class initializations of P assign the same initial values to the static fields of P. The latter result partially solves a problem raised in [3].

1. Introduction

In [3], a static analysis method is proposed to determine class initialization dependencies of a Java program. The method uses the bytecode of a Java program P and produces a directed graph $G(P)$ whose nodes are the classes of P and the system classes used by P and whose edges are ordered pairs $(\mathtt{A}, \mathtt{B})$ such that there is a sequence of method calls from `A.<clinit>` that ends with `getstatic` or `putstatic` instruction of a static field of `B`. In [3], $G(P)$ is called the graph of class initialization dependencies of P. $G(P)$ captures initialization circularities in case P has any. It is a property of the graph $G(P)$ that P has an initialization circularity if and only if $G(P)$ contains a cycle of length at least 2. It is argued in [3] that such circularities in P are almost surely programming errors. This led the authors of [3] to assume that $G(P)$ does not contain cycles (except for possible self-loops) and propose a way of initializing classes of P, called eager class initialization, that respects a topological sort of $G(P)$. (For benefits of eager class initialization, see [3]). The authors of [3] pose an open problem of formulating conditions under which the standard lazy class initialization (i.e. used by the Java Virtual Machine) and eager class initialization assign the same initial values to the static fields of P.

In the following, we assume that $G(P)$ does not contain cycles (except for possible self-loops) and call $G(P)$ *acyclic*.

Since $G(P)$ may have many topological sorts, the question - under what conditions does eager class initialization of P not depend on the choice of a topological sort of $G(P)$? - emerges naturally in the context of the above open problem.

The aim of this paper is to respond to the above question and the open problem. We prove that if P is `putstatic` - free (for the definition, see the end of this section), then eager class initialization of P does not depend on the choice of a topological sort of $G(P)$. We also prove that under some additional conditions (specified in sections 4 and 5), the standard lazy class initialization and eager class initialization of P assign the same initial values to the static fields of P. We also give examples showing that the additional conditions are necessary.

Let P be a Java program. We do not distinguish between system classes used to run P and application classes of P, denoted as in [3] by the sets $SC(P)$ and $LC(P)$,

respectively. Throughout the paper we work with the set of all classes of P denoted by $Cl(P)$ and defined as $SC(P) \cup LC(P)$. We make this simplification since system classes do not pose any logical threat to class initialization in the sense that "no system class would normally know about application classes and thus would not reference their static fields [3, p.5]."

The graph $G(P)$ is the induced subgraph on $Cl(P)$ of the call graph of P. The latter one is constructed in [3] using an algorithm, which runs on the bytecode of P. For every class `A`, it searches for all possible sequences of method calls that start with `A.<clinit>`. These sequences can end only in either a static or instance method of P from which there are no direct or indirect references to any static field of P, or in a `getstatic` or `putstatic` instruction of a static field of P. We will call such sequences of method calls *chains of intermediate method calls from* `A.<clinit>`.

If a chain of intermediate method calls from `A.<clinit>` ends in a static or instance method of P, then we say that the chain is of *type* 1. If a chain ends in a `getstatic` or `putstatic` instruction of a static field of P, then we say that the chain is of *type* 2.

Let G be the graph whose set of vertices is the set $Cl(P)$ and whose edges are defined as follows:

`A` $\Rightarrow$ `B` is an edge in G if and only if there is a chain of intermediate method calls from `A.<clinit>` (of type 2) that ends in a `getstatic` or `putstatic` instruction of a static field of `B`.

Intuitively, if `A` $\Rightarrow$ `B` is an edge in G then the static fields of `B` must be initialized before those of `A`.

The graph $G(P)$ is the transitive closure of G. We call G *the skeleton of* $G(P)$. Obviously, G is acyclic if and only if $G(P)$ is acyclic.

Let s be a topological sort of $G(P)$ given in the form of a bijection $s : Cl(P) \to \{1, 2, \ldots, n\}$, where $n = |Cl(P)|$, such that $s(\texttt{B}) < s(\texttt{A})$ whenever `A` $\Rightarrow$ `B` is an edge in $G(P)$ with `A` $\neq$ `B`. Assuming that the classes of P have been loaded and prepared eagerly by the JVM, we say that the classes of P are *initialized in the eager way respecting* s if their `<clinit>` methods are executed in the following order: $s^{-1}(1)$, $s^{-1}(2)$, ..., $s^{-1}(n)$. We call this process *eager class initialization of* P *respecting* s. In this context, initializing a class means executing its `<clinit>` method if the method was created during compilation (see [4, p.248]), or doing nothing otherwise. This, in particular, means that the *first active use* rule (see [4, p.251]) that the JVM uses to initialize classes does not apply here, and neither does the JVM requirement saying that before a class is initialized its direct superclass (if any) has to be initialized if it has not been already (see [4, p.246-247]).

Note that s is a topological sort of $G(P)$ if and only if s is a topological sort of G.

Let s be a topological sort of G. Let `A` $\in Cl(P)$ and let `a` be a static field of `A`. Let $val_{P,s}(\texttt{A.a})$ denote the value stored in `A.a` as soon as `A.<clinit>` is fully executed during eager class initialization of P respecting s (see [3]).

Let s_1 and s_2 be two topological sorts of G. We write $s_1 \equiv_P s_2$ if for every class `A` $\in Cl(P)$ and every static field `a` of `A`, $val_{P,s_1}(\texttt{A.a}) = val_{P,s_2}(\texttt{A.a})$.

We define P to be `putstatic` - *free* if for every two distinct classes $\mathtt{A}, \mathtt{B} \in Cl(P)$ with an edge $\mathtt{A} \Rightarrow \mathtt{B}$ in G there is no chain (of type 2) of intermediate method calls from `A.<clinit>` that ends in a `putstatic` instruction of a static field of `B`.

2. First Result and Its Proof

We prove the following:

Theorem 1. *If P is `putstatic` - free, then $s_1 \equiv_P s_2$ for all topological sorts s_1 and s_2 of $G(P)$.*

Proof. (The proof is by induction on the number k of edges in the skeleton of $G(P)$).

Basis Step: $k = 0$.

We clearly have $s_1 \equiv_P s_2$ since there are no edges in the skeleton G of $G(P)$, and so there are no class initialization dependencies. Hence, executions of the `<clinit>` methods of classes of P do not depend on each other. This means that the process of initializing classes of P is insensitive to the order in which classes of P are being initialized.

Inductive Step: Let $k > 0$. Assume that the statement of the theorem holds for every Java program Q such that the skeleton of $G(Q)$ has $< k$ edges.

Let P be a Java program that is `putstatic` - free. Let s_1 and s_2 be two topological sorts of the skeleton G of $G(P)$. Since $k > 0$ and the skeleton G of $G(P)$ is acyclic, there must be a class $\mathtt{A} \in Cl(P)$ such that

(i) $\mathtt{A} \Rightarrow \mathtt{B}$ is an edge in G for some $\mathtt{B} \in Cl(P)$
(ii) There is no $\mathtt{C} \in Cl(P)$ such that $\mathtt{C} \Rightarrow \mathtt{A}$ is an edge in G.

Pick such a class `A`. Note that (i) and the definition of G imply that `A.<clinit>` is non-empty. Now, form a new class $\mathtt{A}'$ from `A` by deleting all class variable initializers of `A` as well as all of its static initializers (for definitions see [4, p.246]), thus leaving only class variable declarations. Note that $\mathtt{A}'$ will now have no `<clinit>` method ([4], p. 248).

Let P' denote the program obtained from P be replacing class `A` with $\mathtt{A}'$. Note that the new program P' is correct in the sense that by (ii) and the definition of the skeleton G of $G(P)$, for every $\mathtt{C} \in Cl(P)$ there is no chain of intermediate method calls from `C.<clinit>` that ends in a `getstatic` or `putstatic` instruction of a static field of `A`.

Since $\mathtt{A}'$`.<clinit>` is empty, it follows from (i) and the definition of the skeleton G' of $G(P')$ that the number of edges in G' is $< k$. Also, since P is `putstatic` - free, P' is `putstatic` - free. And since the skeleton G' is a subgraph of the skeleton G of $G(P)$ (we have essentially only removed outgoing edges of `A`), it follows that s_1 and s_2 are also topological sorts of G'. Hence, by the inductive hypothesis, it follows that $s_1 \equiv_{P'} s_2$. This means that

(1) For all $\mathtt{C} \in Cl(P')$ and all static fields c of `C`, $val_{P',s_1}(\mathtt{C.c}) = val_{P',s_2}(\mathtt{C.c})$.

Now, assume that we have initialized the classes of P eagerly according to s_1 and s_2. The following two claims follow from the construction of P' and (ii):

(2) For all $\mathtt{C} \in Cl(P)$, $\mathtt{C} \neq \mathtt{A}$, and all static fields `c` of `C`, $val_{P,s_1}(\mathtt{C.c}) = val_{P',s_1}(\mathtt{C.c})$.

(3) For all C $\in Cl(P)$, C $\neq$ A, and all static fields c of C, val_{P,s_2}(C.c) = val_{P',s_2}(C.c).

By (1), (2), and (3), we get:

(4) For all C $\in Cl(P)$, C $\neq$ A, and all static fields c of C, val_{P,s_1}(C.c) = val_{P,s_2}(C.c).

In order to complete the proof, we must show that

(5) For every static field a of A, val_{P,s_1}(A.a) = val_{P,s_2}(A.a).

Let $S = \{$B $\in Cl(P) \mid$ A $\Rightarrow$ B is an edge in the skeleton G of $G(P)\}$. The set S consists of all classes on which A's static fields depend (which "participate" in the evaluation of initial values of the static fields of A). Let a be a static field of A. The values val_{P,s_1}(A.a) and val_{P,s_2}(A.a) are obtained by executing the same chains of intermediate method calls from A.<clinit>. Since P is putstatic - free, any such chain ends in a getstatic instruction of a static field of a class in S, or it ends in a getstatic or putstatic instruction of some static field of A that is in the textual order earlier than A.a, or the chain is of type 1. Since the static fields of the classes of S as well as the static fields of A that are textually earlier than A.a have been fully initialized at the time the value of A.a is being evaluated, and since chains of type 1 do not disturb values, we get (5) from (4). ∎

The following Java program shows that the conclusion of the above theorem is not true without the condition that P is putstatic - free.

```
class A
{
    static int a = 1;
    public static void main(String[] args) {}
}
class B
{
    static int b = A.a++;
}
class C
{
    static int c = A.a++;
}
```

Note that in the above program P, $G(P)$ is acyclic and that there are two possible topological sorts of $G(P)$: s_1 given by A, B, C and s_2 given by A, C, B. P initialized eagerly with respect to s_1 gives val_{P,s_1}(A.a) = 1, val_{P,s_1}(B.b) = 1, and val_{P,s_1}(C.c) = 2, while eager initialization with respect to s_2 gives val_{P,s_2}(A.a) = 1, val_{P,s_2}(B.b) = 2, and val_{P,s_2}(C.c) = 1. Hence, it is not the case that $s_1 \equiv s_2$.

3. Extracts from the Process of Initializing Classes by the JVM

Let `A` be a class of a Java program P and let `a` be a static field of `A`. Let val_{lazy}(`A.a`) denote the value stored in `A.a` as soon as the JVM finishes initializing `A.<clinit>` (i.e. finishes executing `A.<clinit>`). Let s be a fixed topological sort of the skeleton G of $G(P)$ (the graph of class initialization dependencies of P). Our second result says that if P is `putstatic` - free and satisfies three additional and natural conditions (specified in sections 4 and 5), then val_{lazy}(`A.a`) $= val_{P,s}$(`A.a`) for every static field `a` of every class `A` of P.

In order to formulate the additional conditions, we extract from the JVM initialization process the entities most essential to our purpose and put them into a formal format. We extract three functions which we denote by S_P, E_P, and F_P. It is possible to record the values of these three functions once the JVM process of initializing classes of P terminates. However, if we assume that the functions are known, then they together with the concept of a chain of intermediate method calls (see section 1) will allow us to capture the part of the JVM initialization process that is most essential to our purpose. All concepts defined in this and the following sections are defined in terms of the functions S_P, E_P, F_P, the skeleton G of $G(P)$, and/or the concept of a chain of intermediate method calls.

3.1. The Functions S_P and E_P.

Let $Cl^+(P)$ denote the set of classes of P that have a `<clinit>` method (created during compilation) (see [4, p.248]). Without loss of generality, we can assume that $Cl^+(P) = Cl(P)$, i.e. every class in $Cl(P)$ has a `<clinit>` method (possibly null). Let n be the number of elements of $Cl^+(P)$. The process of initializing classes of P by the JVM uniquely determines two bijective functions

$$S_P : Cl^+(P) \longrightarrow \{1, ..., n\}$$

and

$$E_P : Cl^+(P) \longrightarrow \{1, ..., n\}$$

The value S_P(`A`) $= k$ means that during the (lazy) JVM initialization, class `A` was the k-th class of P that was invoked for initialization (i.e. `A.<clinit>` was the k-th `<clinit>` method invoked by the JVM). The value E_P(`B`) $= l$ means that during the (lazy) JVM initialization class `B` was the l-th class that was completely initialized by the JVM (i.e. `B.<clinit>` was the l-th `<clinit>` method that was fully executed). The values of these functions depend on the structure of the program P, on the *first active use* rule by which classes are initialized, and on the JVM requirement stating that before a class is initialized, its direct superclass (if any) has to be initialized if it has not been already (see [3, p.246-247]). Of course, S_P(`A`) $= 1$ if and only if `A` contains the `main()` method. The notation S_P and E_P comes from the words *start* and *end*, respectively.

It follows from the given descriptions of S_P and E_P that the values of S_P and E_P are known once the JVM finishes initializing classes of P. One may actually try to determine the values of the functions S_P and E_P in advance by a static analysis of the bytecode of P. But this is not what we want to focus on. The fact that the two functions are uniquely determined and take on the values as described above will suffice for our purpose. We assume that the two functions together with their meanings are determined in advance.

3.2. **The Graph $G^+(P)$.** We now define a directed graph denoted by $G^+(P)$. The vertices of the graph $G^+(P)$ are the elements of $Cl^+(P)$. The edge relation on $Cl^+(P)$ will be denoted by the symbol $\rightarrowtail$. An edge $\mathtt{A} \rightarrowtail \mathtt{B}$ in $G^+(P)$ will mean that either B is the direct superclass of A (which at the moment when the JVM invoked A was not yet initialized) or the JVM while reading A.<clinit> encountered B based on the *first active use* rule (see [4, p.251]), and as a consequence, invoked B.<clinit>. The formal definition of $G^+(P)$ is in terms of the functions S_P and E_P and is as follows.

$G^+(P)$ is a directed graph whose set of vertices is the set $Cl^+(P)$ and whose edges are defined as follows:

$\mathtt{A} \rightarrowtail \mathtt{B}$ is an edge in $G^+(P)$ if and only if

$$S_P(\mathtt{A}) < S_P(\mathtt{B}) \text{ and } E_P(\mathtt{A}) > E_P(\mathtt{B})$$

and the set

$$\{\mathtt{C} \in Cl^+(P) \mid \quad S_P(\mathtt{A}) < S_P(\mathtt{C}) < S_P(\mathtt{B}) \text{ and } E_P(\mathtt{A}) > E_P(\mathtt{C}) > E_P(\mathtt{B})\}$$

is empty.

Lemma 1. *For all* $\mathtt{A}, \mathtt{B}, \mathtt{C} \in Cl^+(P)$, *if* $S_P(\mathtt{A}) < S_P(\mathtt{C}) < S_P(\mathtt{B})$, $E_P(\mathtt{A}) > E_P(\mathtt{B})$, *and* $E_P(\mathtt{C}) > E_P(\mathtt{B})$, *then* $E_P(\mathtt{A}) > E_P(\mathtt{C})$.

Proof. Since $S_P(\mathtt{A}) < S_P(\mathtt{B})$ and $E_P(\mathtt{A}) > E_P(\mathtt{B})$, it follows from the meaning of the functions S_P and E_P and the process of lazy (JVM) initialization that there is a unique sequence $\mathtt{X}_1, ..., \mathtt{X}_k$ of classes in $Cl^+(P)$ such that $\mathtt{A} = \mathtt{X}_1, \mathtt{X}_k = \mathtt{B}$, and for every $1 \leq i < k$ either $\mathtt{X}_{i+1}$ is the direct superclass of $\mathtt{X}_i$ (which at the moment when the JVM invoked $\mathtt{X}_i$ was not yet initialized) or the JVM while reading $\mathtt{X}_i$.<clinit> encountered $\mathtt{X}_{i+1}$based on the *first active use* rule. Since $S_P(\mathtt{C}) < S_P(\mathtt{B})$ and $E_P(\mathtt{C}) > E_P(\mathtt{B})$, there is a similar unique sequence $\mathtt{C} = \mathtt{Y}_1, ..., \mathtt{Y}_l = \mathtt{B}$ of classes in $Cl^+(P)$ as described above but for C and B. Since the JVM initializes classes only once, B is initialized only once. This, by $\mathtt{X}_k = \mathtt{B} = \mathtt{Y}_l$, implies that either $\mathtt{C} = \mathtt{X}_i$ for some $1 \leq i < k$ or $\mathtt{A} = \mathtt{Y}_j$ for some $1 \leq j < l$. But $S_P(\mathtt{A}) < S_P(\mathtt{C})$ and $S_P(\mathtt{C}) \leq S_P(\mathtt{Y}_j)$ for all j, so $\mathtt{A} \neq \mathtt{Y}_j$ for any j. Thus $\mathtt{C} = \mathtt{X}_i$ for some i, which gives $E_P(\mathtt{A}) > E_P(\mathtt{C})$ because $E_P(\mathtt{A}) > E_P(\mathtt{X}_i)$ for all i. ∎

Proposition 1. *For all* $\mathtt{A}, \mathtt{B} \in Cl^+(P)$, *the following conditions are equivalent:*

(i) $S_P(\mathtt{A}) < S_P(\mathtt{B})$ *and* $E_P(\mathtt{A}) > E_P(\mathtt{B})$;
(ii) *There is a unique directed path in* $G^+(P)$ *from* A *to* B.

Proof. (i) $\Longrightarrow$ (ii): Assume (i). If the set $\{\mathtt{C} \in Cl^+(P) : S_P(\mathtt{A}) < S_P(\mathtt{C}) < S_P(\mathtt{B})$ and $E_P(\mathtt{A}) > E_P(\mathtt{C}) > E_P(\mathtt{B})\}$ is empty, then the path from A to B is: A, B. Otherwise, choose the unique class $\mathtt{C}_1$ in the set $\{\mathtt{C} \in Cl^+(P) : S_P(\mathtt{A}) < S_P(\mathtt{C}) < S_P(\mathtt{B})$ and $E_P(\mathtt{A}) > E_P(\mathtt{C}) > E_P(\mathtt{B})\}$ whose value under S_P is the smallest (S_P is one-to-one since it is a bijection). It follows that $\mathtt{A} \rightarrowtail \mathtt{C}_1$ is an edge in $G^+(P)$. Next, consider $\mathtt{C}_1$ and B. Since $\mathtt{C}_1$ and B satisfy $S_P(\mathtt{C}_1) < S_P(\mathtt{B})$ and $E_P(\mathtt{C}_1) > E_P(\mathtt{B})$, repeating the above argument for $\mathtt{C}_1$ and B we finally obtain a path from A to B. We now show that this path is unique. Suppose, on the contrary, that there are

two different paths in $G^+(P)$ from $\mathtt{A}$ to $\mathtt{B}$. It follows from this assumption that there are three distinct classes $\mathtt{D}_1$, $\mathtt{D}_2$, and $\mathtt{C}$ such that (1) $\mathtt{D}_1 \rightarrowtail \mathtt{C}$ and $\mathtt{D}_2 \rightarrowtail \mathtt{C}$ are edges in $G^+(P)$ and (2) there is neither a directed path in $G^+(P)$ from $\mathtt{D}_1$ to $\mathtt{D}_2$ nor from $\mathtt{D}_2$ to $\mathtt{D}_1$. Since the function S_P is one-to-one and $\mathtt{D}_1 \neq \mathtt{D}_2$, it follows that $S_P(\mathtt{D}_1) \neq S_P(\mathtt{D}_2)$. We assume $S_P(\mathtt{D}_1) < S_P(\mathtt{D}_2)$; the case $S_P(\mathtt{D}_1) > S_P(\mathtt{D}_2)$ is similar. By (1) and the definition of $G^+(P)$, we have $S_P(\mathtt{D}_1) < S_P(\mathtt{D}_2) < S_P(\mathtt{C})$. By (1) again, we have $E_P(\mathtt{D}_1) > E_P(\mathtt{C})$ and $E_P(\mathtt{D}_2) > E_P(\mathtt{C})$. This, by Lemma 1, gives $E_P(\mathtt{D}_1) > E_P(\mathtt{D}_2)$. Hence, by an argument similar to the one in the beginning of this proof, we conclude that there is a directed path in $G^+(P)$ from $\mathtt{D}_1$ to $\mathtt{D}_2$. This contradicts (2).

(ii) $\Longrightarrow$ (i): Obvious from the definition of the edge relation of $G^+(P)$. ∎

3.3. **The Function F_P.** For a class $\mathtt{A} \in Cl(P)$, let $SF(\mathtt{A})$ denote the set of static fields of $\mathtt{A}$. Let m denote the number of elements of $\bigcup_{\mathtt{A}\in Cl(P)} SF(\mathtt{A})$. The process of initializing classes of P by the JVM uniquely determines the following bijective function:

$$F_P : \bigcup_{\mathtt{A}\in Cl(P)} SF(\mathtt{A}) \longrightarrow \{1, ..., m\}$$

The value $F_P(\mathtt{a}) = k$ means that $\mathtt{a}$ was the k-th static field of P to which the JVM assigned an initial value for the first time. If a static field $\mathtt{A.a}$ does not have a class variable initializer nor a static initializer (for definitions see [4, p.246]), we assume that it has an implicit class variable initializer that evaluates to $\mathtt{A.a}$'s default value (the value assigned during preparation - see [4, p.244-245]). Hence, without loss of generality, we assume that all static fields have either class variable initializers or static initializers. As in case of S_P and E_P, the values of F_P are known after the JVM finishes initializing all classes of P. However, we assume that F_P, together with its meaning described above, is known to us.

For a class $\mathtt{C}$ of P with $SF(\mathtt{C}) \neq \varnothing$, we set

$$\min F_P(\mathtt{C}) = \min \{F_P(\mathtt{C.c}) \mid \mathtt{c} \in SF(\mathtt{C})\}$$

and

$$\max F_P(\mathtt{C}) = \max \{F_P(\mathtt{C.c}) \mid \mathtt{c} \in SF(\mathtt{C})\}.$$

4. Condition (C1)

In this section, we formulate the first of three conditions which together with the assumption that a Java program P is `putstatic` - free guarantee that eager and lazy class initialization assign the same initial values to the static fields of P.

A pair $(\mathtt{A}, \mathtt{C})$ of classes of P is said to be *a lazy pair of* P if $\mathtt{A} \rightarrowtail \mathtt{C}$ is an edge in $G^+(P)$. If $(\mathtt{A}, \mathtt{C})$ is a lazy pair, then we will call $\mathtt{C}$ *a child of* $\mathtt{A}$.

In terms of lazy initialization, if $(\mathtt{A}, \mathtt{C})$ is a lazy pair then (see definition of the $\rightarrowtail$ relation) $\mathtt{C}$ is the direct superclass of $\mathtt{A}$ or the JVM while reading $\mathtt{A.<clinit>}$ encountered $\mathtt{C}$ based on the *first active use* rule.

For a lazy pair $(\mathtt{A}, \mathtt{C})$, we define the set $SF^+(\mathtt{A}, \mathtt{C})$ as follows:

If $\mathtt{C}$ is the direct superclass of $\mathtt{A}$, then we set $SF^+(\mathtt{A}, \mathtt{C}) = \varnothing$. Otherwise, we set

$SF^+(\mathtt{A}, \mathtt{C}) = \{\mathtt{a} \in SF(\mathtt{A}) \mid$ the JVM had assigned an initial value to $\mathtt{a}$ before it encountered $\mathtt{C}\}$.

We also set

$$SF^-(\mathtt{A},\mathtt{C}) = SF(\mathtt{A}) \setminus SF^+(\mathtt{A},\mathtt{C})$$

Note that if C is not the direct superclass of A, then the set $SF^+(\mathtt{A},\mathtt{C})$ consists of the static fields of A to which the JVM had assigned initial values before it encountered C and invoked C.<clinit>. A static field whose class variable initializer nor static initializer has not yet been evaluated, and which therefore still holds its default value (assigned during preparation) is called *uninitialized* (we also refer to the values stored by uninitialized static fields as *uninitialized*). The set $SF^-(\mathtt{A},\mathtt{C})$ consists of the static fields of A that were uninitialized right after the JVM invoked C.<clinit>.

The following fact equivalently defines $SF^+(\mathtt{A},\mathtt{C})$ and $SF^-(\mathtt{A},\mathtt{C})$ in the case when $SF(\mathtt{C}) \neq \varnothing$. It follows easily from the meanings of the introduced concepts.

Proposition 2. *If* $(\mathtt{A},\mathtt{C})$ *is a lazy pair of* P *with* $SF(\mathtt{C}) \neq \varnothing$*, then*

(i) $SF^+(\mathtt{A},\mathtt{C}) = \{\mathtt{a} \in SF(\mathtt{A}) \mid F_P(\mathtt{A.a}) < \min F_P(\mathtt{C})\}$;
(ii) $SF^-(\mathtt{A},\mathtt{C}) = \{\mathtt{a} \in SF(\mathtt{A}) \mid F_P(\mathtt{A.a}) > \max F_P(\mathtt{C})\}$.

It is possible to describe the set $SF^+(\mathtt{A},\mathtt{C})$ in terms of class variable initializers and static initializers of A (for definitions see [4, p.246]). Here is such a description.

For a static field a of A, $\mathtt{a} \in SF^+(\mathtt{A},\mathtt{C})$ if and only if a was "noticed" by the JVM while reading A.<clinit> before C was encountered based on the *first active use* rule and

(i) a does not have a class variable initializer and A does not have a static initializer that initializes a, or

(ii) a has a class variable initializer and the initializer was completely executed before the JVM encountered C, or

(iii) a does not have a class variable initializer but it has a static initializer that which was completely executed before the JVM encountered C.

Now, let $\mathtt{A},\mathtt{B} \in Cl^+(P)$ be such that $S_P(\mathtt{A}) < S_P(\mathtt{B})$ and $E_P(\mathtt{A}) > E_P(\mathtt{B})$. By Proposition 1, there is a unique directed path in $G^+(P)$ from A to B. Clearly, the path must go through a child of A. We will be interested in the triples of the form $(\mathtt{A},\mathtt{C},\mathtt{B})$ such that A and B are classes of P that satisfy $S_P(\mathtt{A}) < S_P(\mathtt{B})$ and $E_P(\mathtt{A}) > E_P(\mathtt{B})$ and C is a child of A on the unique directed path in $G^+(P)$ from A to B. We will call any such triple *a lazy triple of* P. Hence, we have the following 4 equivalent formulations: $(\mathtt{A},\mathtt{C},\mathtt{C})$ is a lazy triple $\iff$ $(\mathtt{A},\mathtt{C})$ is a lazy pair $\iff \mathtt{A} \rightarrowtail \mathtt{C}$ is an edge in $G^+(P) \iff$ C is a child of A.

Note that in any lazy triple $(\mathtt{A},\mathtt{C},\mathtt{B})$, C is uniquely determined by A and B. This is so because C is a child of A that is on a unique directed path in $G^+(P)$ from A to B. Hence instead of using triples $(\mathtt{A},\mathtt{C},\mathtt{B})$ we could use pairs $(\mathtt{A},\mathtt{B})$ such that $S_P(\mathtt{A}) < S_P(\mathtt{B})$ and $E_P(\mathtt{A}) > E_P(\mathtt{B})$. However, we prefer to keep C in $(\mathtt{A},\mathtt{C},\mathtt{B})$ to emphasize that C is the first class through which the JVM process of initializing classes between A and B goes, and through which it goes back to continue initializing A. This emphasis is needed in the formulation of (C1) below.

Recall the assumption that G is acyclic.

(C1) For every lazy triple $(\mathtt{A},\mathtt{C},\mathtt{B})$ of P: If $\mathtt{B} \Rightarrow \mathtt{A}$ is an edge in G, then every chain of intermediate method calls from B.<clinit> that ends in a getstatic

instruction of a static field of A ends in a getstatic instruction of a static field belonging to $SF^+(\mathtt{A},\mathtt{C})$.

Note that (C1) also refers to lazy pairs for if $(\mathtt{A},\mathtt{C})$ is a lazy pair, then $(\mathtt{A},\mathtt{C},\mathtt{C})$ is a lazy triple.

The condition (C1) is equivalent to not admitting access to uninitialized values by a Java program (see Proposition 3 below).

Let A be a class of P. We say that during initialization of A *an access to an uninitialized value is made* if there is a class $\mathtt{D} \in Cl^+(P)$ with $S_P(\mathtt{D}) < S_P(\mathtt{A})$ such that during the JVM initialization of A, the JVM encounters a chain of intermediate method calls from A.<clinit> that ends in a getstatic instruction of a static field of D to which the JVM has not yet assigned an initial value (this static field of D therefore still holds its default value assigned during preparation).

Note that the inequality $S_P(\mathtt{D}) < S_P(\mathtt{A})$ in the above definition is necessary, since otherwise if $S_P(\mathtt{D}) > S_P(\mathtt{A})$, then by the time the JVM encounters a chain of intermediate method calls from A.<clinit> that ends in a getstatic instruction of a static field of D, either D will already be fully initialized, or D.<clinit> will be invoked for the first time (triggered by the getstatic reference). In either case A will not access D's uninitialized values.

We say that P *does not admit access to uninitialized values* if there is no class in P during whose JVM initialization an access to an uninitialized value is made. The following proposition and the example following it provide support for introducing (C1) into our considerations.

Proposition 3. *For a Java program P, the following conditions are equivalent:*

(i) *P satisfies (C1);*
(ii) *P does not admit access to uninitialized values.*

Proof. (i) $\Longrightarrow$ (ii): (by Contrapositive). Assume that A is a class of P during whose JVM initialization an access to an uninitialized value was made. This means that there was a class D with $S_P(\mathtt{D}) < S_P(\mathtt{A})$ such that during the initialization of A, the JVM encountered a chain of intermediate method calls from A.<clinit> that ended in a getstatic of a static field d of D to which the JVM had not yet assigned an initial value. Hence the initialization of D was not complete. This, in turn, means that after completing the initialization of A the JVM went back to D to complete its initialization. This, however, means that $E_P(\mathtt{A}) < E_P(\mathtt{D})$. This, by Proposition 1, means that there is a unique directed path in $G^+(P)$ from D to A. Let C be the child of D on that path. We have a lazy triple $(\mathtt{D},\mathtt{C},\mathtt{A})$. Since the JVM did not assign an initial value to D.d, it follows that $\mathtt{d} \notin SF^+(\mathtt{D},\mathtt{C})$. This means that P does not satisfy (C1), proving that (i) implies (ii)

(ii) $\Longrightarrow$ (i): (by Contrapositive). Assume that P does not satisfy (C1). This means that there is a lazy triple $(\mathtt{A},\mathtt{C},\mathtt{B})$ and a chain of intermediate method calls from B.<clinit> that ends in a getstatic of a static field a of A with $\mathtt{a} \notin SF^+(\mathtt{A},\mathtt{C})$. Since $(\mathtt{A},\mathtt{C},\mathtt{B})$ is a lazy triple, it follows that $S_P(\mathtt{A}) < S_P(\mathtt{B})$. Since $\mathtt{a} \notin SF^+(\mathtt{A},\mathtt{C})$, it follows that while the JVM was initializing B (executing B.<clinit>), a stored its default value instead of its initial value. Thus, during the JVM initialization of B, an access to an uninitialized value of a was made, proving that (ii) implies (i). ∎

For example, consider the following program:

```
class B
{
    static int b1 = A.a1;
    static int b2 = A.a3;
    static int foo() { return 123; }
}
class A
{
    static int a1 = 1;
    static int a2 = B.foo();
    static int a3 = 1;
    public static void main(String[] args)
    {
        System.out.println(‘‘An Example Violating (C1)’’);
    }
}
```

Note that in the above program P, G is acyclic, $(\mathtt{A},\mathtt{B},\mathtt{B})$ is a lazy triple (i.e. $(\mathtt{A},\mathtt{B})$ is a lazy pair), and the edge $\mathtt{B} \Rightarrow \mathtt{A}$ is in G. Also note that there is a chain (of length 1) of intermediate method calls from `B.<clinit>` that ends in `getstatic A.a3` and $\mathtt{A.a3} \notin SF^{+}(\mathtt{A},\mathtt{C})$ (since $SF^{+}(\mathtt{A},\mathtt{C}) = \{\mathtt{a1}\}$), hence (C1) is violated. Now, lazy initialization gives $val_{lazy}(\mathtt{B.b2}) = 0$, while eager initialization respecting the only topological sort s of G (namely $s(\mathtt{A}) = 1$, $s(\mathtt{B}) = 2$) gives $val_{P,s}(\mathtt{B.b2}) = 1$.

Discrepancies like this example will be common if the JVM accesses uninitialized values. This is because eager class initialization never accesses uninitialized values of any static fields of the program. Hence condition (C1) is necessary for the two initializations (eager and lazy) to give the same initial values for all static fields.

5. Conditions (C2) and (C3)

Recall the assumption that G is acyclic. The second condition is:

(C2) For every lazy triple $(\mathtt{A},\mathtt{C},\mathtt{B})$ of P and every static field `a` of `A`: if there is a chain of intermediate method calls from `B.<clinit>` that ends in `getstatic A.a`, then from the part of `A.<clinit>` that corresponds to $SF^{-}(\mathtt{A},\mathtt{C})$ there is no chain of intermediate method calls that ends in `putstatic A.a`.

We now want to explain the reason of introducing (C2). Let $(\mathtt{A},\mathtt{C},\mathtt{B})$ be a lazy triple. Let $f_1, ..., f_l$ be a chain of intermediate method calls from `B.<clinit>` and let $f_l =$ `getstatic A.a`. Consider the two ways of initializing classes: eager initialization respecting a topological sort s of G and lazy initialization (done by the JVM). In eager initialization, the chain $f_1, ..., f_l$ accesses the value of `A.a` when the initialization of `A` has completed (since in eager initialization `A` is initialized before `B` because of the dependency $\mathtt{B} \Rightarrow \mathtt{A}$ in G). In lazy initialization, however, the chain accesses the value of `A.a` when the initialization of `A` has not completed. Even though it may be the case that $\mathtt{A.a} \in SF^{+}(\mathtt{A},\mathtt{C})$ (thus satisfying (C1)), there is still a threat that the remaining part of `A.<clinit>` to be executed may change the value of `A.a` with a `putstatic A.a`, thus causing the eager and lazy initializations to give different initial values for some static field of `B`. Hence (C2) is necessary to eliminate this threat.

For example, consider the following program:

```
class C
{
    static int c = B.b;
}
class B
{
    static int b = A.a;
}
class A
{
    static int a = 1;
    static { C obj = new C(); }
    static aa = a++;
    public static void main(String[] args)
    {
        System.out.println(``An Example Violating (C2)'');
    }
}
```

Note that in the above program P, G is acyclic, and $(\texttt{A}, \texttt{C}, \texttt{B})$ is a lazy triple. Lazy initialization gives $val_{lazy}(\texttt{B.b}) = 1$ and $val_{lazy}(\texttt{C.c}) = 1$, while eager initialization respecting the only topological sort s of G (namely $s(\texttt{A}) = 1$, $s(\texttt{B}) = 2$, $s(\texttt{C}) = 3$) gives $val_{P,s}(\texttt{B.b}) = 2$ and $val_{P,s}(\texttt{C.c}) = 2$. Also note that in the above example (C1) is satisfied, since $SF^+(\texttt{A}, \texttt{C}) = \{\texttt{a}\}$, while (C2) is violated, since the value of `A.a` is changed after `B.<clinit>` and `C.<clinit>` are done executing.

The third condition is:

(C3) For all `A`, `B` $\in Cl^+(P)$, if there is a chain of intermediate method calls from `A.<clinit>` that ends in `getstatic B.b` for some `b` $\in SF(\texttt{B})$, then from the bytecode of the `main()` method of P there is no chain of intermediate method calls that ends in `putstatic B.b` and originates before (in textual order) the instruction that triggers the initialization of `A`.

The condition (C3) seems to be necessary since eager class initialization does not see the `main()` method. Consider, for instance, the following program:

```
class A
{
    static int a = B.b;
}
class B
{
    static int b = 1;
}
class C
{
    public static void main(String[] args)
    {
```

```
        new B();
        B.b = B.b + 1;
        new A();
    }
}
```

Note that in the above program P, G is acyclic. Also note that there is a chain (of length 1) of intermediate method calls from `A.<clinit>` that ends in `getstatic B.b`, and that from the bytecode of the `main()` method of P there is a chain (of length 1) of intermediate method calls that ends in `putstatic B.b` and originates before (in textual order) the instruction that triggers the initialization of `A` (namely the bytecode instruction corresponding to `new A();`), hence (C3) is violated. Lazy initialization gives $val_{lazy}(\texttt{A.a}) = 2$ while eager initialization respecting the sort s of G (defined by $s(\texttt{C}) = 1$, $s(\texttt{B}) = 2$, $s(\texttt{A}) = 3$) gives $val_{P,s}(\texttt{A.a}) = 1$. The reason for this discrepancy is that during lazy initialization the initial value of a static field (namely `B.b`) on which `A.a` depends is ultimately changed from the `main()` method before `A.a` is initialized, while eager initialization will not see this change since it doesn't see the `main()` method. Hence (C3) ensures that no changes to a static field, on which a class depends, are made before that class is initialized.

6. Main Theorem: Lazy = Eager

Our second result is:

Theorem 2. *Let P be a Java program such that the graph G is acyclic. Let s be a topological sort of G. If P is `putstatic` - free and satisfies (C1), (C2), and (C3), then $val_{lazy}(\texttt{A.a}) = val_{P,s}(\texttt{A.a})$ for all $\texttt{A} \in Cl(P)$ and $\texttt{a} \in SF(\texttt{A})$.*

In the following we assume that P is a Java program such that G is acyclic and that s is a topological sort of G. We also assume that P is `putstatic` - free and satisfies (C1), (C2), and (C3).

Let $\texttt{A} \in Cl(P)$ and $\texttt{a} \in SF(\texttt{A})$. No matter whether the initial value of `A.a` is assigned during lazy initialization (done by the JVM) or during eager initialization respecting s, it is assigned based on the execution of the same chains of intermediate method calls that participate in the computation of the initial value of `A.a`. Hence the question whether $val_{lazy}(\texttt{A.a}) = val_{P,s}(\texttt{A.a})$ is the question whether those chains of intermediate method calls *access* (for definition of the term *access* see the proof of Theorem 2 below) the same values during lazy initialization as during eager initialization respecting s. The chains that participate in the computation of the initial value of `A.a` are either of type 2, ending with a `getstatic` instruction, or of type 1 (to recall the definition of the two types see section 1: Introduction). We denote the set of all such call chains by $MetCall(\texttt{A.a})$. Thus the set $MetCall(\texttt{A.a})$ consists of chains of type 2 ending with a `getstatic` instruction, and chains of type 1 both of which participate in the computation of the initial value of `A.a`.

We define the following sets with respect to a given class $\texttt{A} \in Cl(P)$:

$$S_{1\texttt{A}} = \{\texttt{B} \mid S_P(\texttt{B}) < S_P(\texttt{A}) \text{ and } E_P(\texttt{B}) < E_P(\texttt{A})\}$$
$$S_{2\texttt{A}} = \{\texttt{B} \mid S_P(\texttt{B}) > S_P(\texttt{A}) \text{ and } E_P(\texttt{B}) < E_P(\texttt{A})\}$$
$$S_{3\texttt{A}} = \{\texttt{B} \mid S_P(\texttt{B}) < S_P(\texttt{A}) \text{ and } E_P(\texttt{B}) > E_P(\texttt{A})\}$$
$$S_{4\texttt{A}} = \{\texttt{B} \mid S_P(\texttt{B}) > S_P(\texttt{A}) \text{ and } E_P(\texttt{B}) > E_P(\texttt{A})\}$$

Note that the above four sets partition the set $Cl(P)\setminus\{\mathtt{A}\}$. We call the elements of $S_{2\mathtt{A}}$ *the descendants of* $\mathtt{A}$ *in* $G^+(P)$, and the elements of $S_{3\mathtt{A}}$ *the ancestors of* $\mathtt{A}$ *in* $G^+(P)$. The set $S_{4\mathtt{A}}$ is not used in the proof of Theorem 2 given below. It has been included in the above list for the sake of completeness.

Lemma 2. $\mathtt{A}\Rightarrow\mathtt{B}$ *is an edge in* G *and* $\mathtt{A}\neq\mathtt{B}$ *imply that* $\mathtt{B}\in S_{1\mathtt{A}}\cup S_{2\mathtt{A}}\cup S_{3\mathtt{A}}$.

Proof. Let $\mathtt{A}\Rightarrow\mathtt{B}$ be an edge in G and $\mathtt{A}\neq\mathtt{B}$. If $S_P(\mathtt{A})>S_P(\mathtt{B})$, then obviously $\mathtt{B}\in S_{1\mathtt{A}}\cup S_{3\mathtt{A}}$. So let $S_P(\mathtt{A})<S_P(\mathtt{B})$. This inequality and $\mathtt{A}\neq\mathtt{B}$ say that at the moment when the JVM starts initializing A, the class B has not yet been invoked for initialization. Since $\mathtt{A}\Rightarrow\mathtt{B}$ is an edge in G, there is a chain of type 2 of intermediate method calls from A.<clinit> that ends with the instruction getstatic or putstatic of a static field of B. The JVM while initializing A sees the chain and invokes B for initialization, since $S_P(\mathtt{A})<S_P(\mathtt{B})$ (up to this point B has not yet been invoked for initialization). Before the JVM completes initializing A, it completes initializing B. Thus $E_P(\mathtt{A})>E_P(\mathtt{B})$ and so $\mathtt{B}\in S_{2\mathtt{A}}$. ∎

Proof of Theorem 2: Recall that G is acyclic, P is putstatic - free and satisfies (C1), (C2), and (C3), and that s is a topological sort of G.

We prove the conclusion of the theorem by induction on the set $\{1,\ldots,n\}$ with respect to the values of s. Recall that $s:Cl(P)\to\{1,\ldots,n\}$ is a bijection such that $s(\mathtt{B})<s(\mathtt{A})$ whenever $\mathtt{A}\Rightarrow\mathtt{B}$ is an edge in G with $\mathtt{A}\neq\mathtt{B}$.

Let $k\in\{1,...,n\}$ and let $\mathtt{A}\in Cl(P)$ be such that $s(\mathtt{A})=k$.

Basic Step: $k=1$.

In this case, there are no outgoing edges from A in G. This means that $MetCall(\mathtt{A.a})$ is empty, or contains chains of type 1 and/or chains of type 2 that end with getstatic instructions of static fields of A that are in the textual order earlier than a in A.<clinit> (since the JVM does not permit forward references). In either case, we have $val_{lazy}(\mathtt{A.a})=val_{P,s}(\mathtt{A.a})$. This is so because the chains of $MetCall(\mathtt{A.a})$ (if there are any) do not engage any static fields other than the ones of A in the process of computing the initial value of A.a.

Inductive Step: Let $k\geq 2$ and assume that the conclusion of the theorem holds for every class B such that $s(\mathtt{B})<k$.

Let $\mathtt{a}\in SF(\mathtt{A})$. We will show $val_{lazy}(\mathtt{A.a})=val_{P,s}(\mathtt{A.a})$.

Let $f_1,...,f_l=$ getstatic B.b be a chain of type 2 in $MetCall(\mathtt{A.a})$. We say that *during lazy class initialization (eager class initialization respecting* s, respectively) $f_1,...,f_l$ *accesses the value* v if at the moment of execution of the chain $f_1,...,f_l$ during lazy initialization (eager initialization respecting s, respectively) the value stored in B.b is v. If B is A, then since the JVM does not permit forward references, it follows that b is in A earlier than a (in the textual order).

We prove

(1) If B is A and b is in A earlier than a (in the textual order), then the chain $f_1,...,f_l=$ getstatic B.b accesses the same value during lazy initialization as during eager initialization respecting s.

In this case, during eager initialization respecting s, $f_1,...,f_l$ accesses the value stored in a static field of A (namely A.b). Because P is putstatic - free, other classes whose initialization may be triggered by the JVM as a result of executing A.<clinit> cannot change the value of A.b, hence the chain $f_1,...,f_l$ accesses the same value during lazy initialization as during eager initialization respecting s.

Now assume that $\texttt{B} \neq \texttt{A}$. Since $f_1, ..., f_l =$ `getstatic B.b` is a chain from `A.<clinit>`, it follows by the definition of G that $\texttt{A} \Rightarrow \texttt{B}$ is an edge in G. This, by Lemma 2, gives $\texttt{B} \in S_{1\texttt{A}} \cup S_{2\texttt{A}} \cup S_{3\texttt{A}}$.

We prove

(2) If $\texttt{B} \in S_{1\texttt{A}} \cup S_{2\texttt{A}}$, then the chain $f_1, ..., f_l =$ `getstatic B.b` accesses the same value during lazy initialization as during eager initialization respecting s.

Since $\texttt{B} \in S_{1\texttt{A}} \cup S_{2\texttt{A}}$, we have $E_P(\texttt{B}) < E_P(\texttt{A})$. Since $f_1, ..., f_l =$ `getstatic B.b` is a chain (of type 2) in $MetCall(\texttt{A.a})$, before the JVM assigns an initial value to `A.a` for the first time it must finish initializing all of `B` first. Hence, $F_P(\texttt{A.a}) > \max F_P(\texttt{B})$. This, by the assumption that P is `putstatic` - free and satisfies (C3), means that during lazy initialization the chain $f_1, ..., f_l$ accesses the value $val_{lazy}(\texttt{B.b})$. Since $\texttt{A} \neq \texttt{B}$ and $\texttt{A} \Rightarrow \texttt{B}$ is an edge in G, it follows that $s(\texttt{B}) < s(\texttt{A})$. This, by the Inductive Hypothesis, gives $val_{lazy}(\texttt{B.b}) = val_{P,s}(\texttt{B.b})$. Thus, during lazy initialization $f_1, ..., f_l$ accesses the value $val_{P,s}(\texttt{B.b})$. Since P is `putstatic` - free, during eager initialization respecting s, $f_1, ..., f_l$ accesses the value $val_{P,s}(\texttt{B.b})$. Thus, during both lazy initialization and eager initialization respecting s, the chain $f_1, ..., f_l$ accesses the same value.

We prove

(3) If $\texttt{B} \in S_{3\texttt{A}}$, then the chain $f_1, ..., f_l =$ `getstatic B.b` accesses the same value during lazy initialization as during eager initialization respecting s.

In this case, `B` is an ancestor of `A` in $G^+(P)$. So we have a lazy triple $(\texttt{B}, \texttt{C}, \texttt{A})$ where `C` is a child of `B` in $G^+(P)$ that is on the unique path from `B` to `A`. By (C1) $\texttt{B.b} \in SF^+(\texttt{B}, \texttt{C})$, and by (C2) there is no chain of intermediate method calls that ends in `putstatic B.b` from the part of `B.<clinit>` corresponding to $SF^-(\texttt{B}, \texttt{C})$. This and the assumption that P is `putstatic` - free imply that after the JVM finishes executing `A.<clinit>` and later `C.<clinit>` and returns to finish executing `B.<clinit>`, the value of `B.b` (assigned before the JVM moved to initialize `C` from `A`) will not change. This, in turn, means that during lazy initialization the chain $f_1, ..., f_l =$ `getstatic B.b` is accessing the value $val_{lazy}(\texttt{B.b})$. Since $\texttt{A} \Rightarrow \texttt{B}$ is an edge in G and $\texttt{A} \neq \texttt{B}$, we have $s(\texttt{B}) < s(\texttt{A})$. This, by the Inductive Hypothesis, gives $val_{lazy}(\texttt{B.b}) = val_{P,s}(\texttt{B.b})$. So during lazy initialization $f_1, ..., f_l$ accesses $val_{P,s}(\texttt{B.b})$. Since P is `putstatic` - free, during eager initialization respecting s, $f_1, ..., f_l$ accesses the value $val_{P,s}(\texttt{B.b})$. Thus, during both lazy initialization and eager initialization respecting s, the chain $f_1, ..., f_l$ accesses the same value.

Since $f_1, ..., f_l =$ `getstatic B.b` is an arbitrary chain of type 2 in $MetCall(\texttt{A.a})$, it follows from Lemma 2 and (1), (2), and (3) that every chain of type 2 from $MetCall(\texttt{A.a})$ accesses the same value during both lazy initialization and eager initialization respecting s. This, in particular, guarantees that every chain of type 1 from $MetCall(\texttt{A.a})$ accesses the same value during both lazy initialization and eager initialization respecting s. Thus all chains from $MetCall(\texttt{A.a})$ access the same value during both lazy initialization and eager initialization respecting s. This gives $val_{lazy}(\texttt{A.a}) = val_{P,s}(\texttt{A.a})$ and completes the proof of the theorem.

7. Concluding Remarks

As mentioned in section 4, eager class initialization never accesses uninitialized values of any static fields of a Java program unless those fields intentionally do not have initializers (class variable initializers or static initializers). This is not the case in the lazy class initialization where uninitialized values may be accessed. If they are accessed, then it is almost surely a programming error unless the static fields that store the uninitialized values intentionally do not have initializers. No JVM that we are aware of reports accesses to uninitialized values. With a little extra effort such a JVM could be designed. Proposition 3 shows one way in which this could possibly be done. Although we assumed that $G(P)$ is acyclic, the assumption is not needed for our proof of Proposition 3. The only significant extra thing a JVM would have to do after invoking every class `A` for initialization is to remember the sets $SF^+(\mathtt{A},\mathtt{C})$ where `C` is a child of `A` in $G^+(P)$.

In [3], the authors write "An interesting open problem is to formulate conditions under which, in the absence of reported bad circularities, the eager and lazy strategies would be guaranteed to give tha same initial values. A formal statement and proof of this result might be based on a bytecode or source-level type system in the style of [2]". Although in our solution to the problem we did not follow the suggestion contained in the quote, we feel that if realized the suggestion will lead to an even more formally rigorous solution.

8. Acknowledgements

I wish to thank my MEng project advisor Professor Dexter Kozen for giving me a chance to work on a theoretically interesting open problem and for sharing with me valuable ideas and discussions. This work was supported in part by NSF grant CCR-0105586 and by ONR Grant N00014-01-1-0968. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the US Government.

References

[1] Börger, Egon and Schulte, Wolfram. Initialization Problems for Java. *Software-Concepts and Tools*, volume 20(4), 1999.

[2] Freud, Stephen N. and Mitchell, John C. A type system for object initialization in the Java bytecode language. *Trans. Programming Languages and Systems,* volume 21(6):1196-1250, 1999.

[3] Kozen, Dexter and Stillerman, Matt. Eager Class Initialization for Java. In W. Damm and E.R. Olderog, editors, *Proc. 7th Int. Symp. Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT'02),* volume 2469 of Lecture Notes in Computer Science, pages 71-80. IFIP, Springer-Verlag, Sept. 2002.

[4] Venners, Bill. Inside the Java 2 Virtual Machine, 2^{nd} ed. McGraw-Hill, 1999.

Computer Science Department, Cornell University, Ithaca, NY 14853-7501, USA
E-mail address: smd34@cornell.edu