HAL
open science

Soundly Handling Static Fields: Issues, Semantics and Analysis

Laurent Hubert, David Pichardie

To cite this version:

Laurent Hubert, David Pichardie. Soundly Handling Static Fields: Issues, Semantics and Analysis. Electronic Notes in Theoretical Computer Science, 2009, Proceedings of the Fourth Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE 2009), 253 (5), pp.15 - 30. 10.1016/j.entcs.2009.11.012 . inria-00504028v2

HAL Id: inria-00504028

https://inria.hal.science/inria-00504028v2

Submitted on 20 Jul 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Soundly Handling Static Fields: Issues, Semantics and Analysis

Laurent Hubert[a,1,2] David Pichardie[b,1]

[a] *CNRS, IRISA, Campus Beaulieu , F-35042 Rennes Cedex, France*

[b] *INRIA, Centre Rennes — Bretagne Atlantique*
IRISA, Campus Beaulieu , F-35042 Rennes Cedex, France

Abstract

Although in most cases class initialization works as expected, some static fields may be read before being initialized, despite being initialized in their corresponding class initializer. We propose an analysis which compute, for each program point, the set of static fields that must have been initialized and discuss its soundness. We show that such an analysis can be directly applied to identify the static fields that may be read before being initialized and to improve the precision while preserving the soundness of a null-pointer analysis.

Keywords: static analysis, Java, semantics, class initialization, control flow, verification.

1 Introduction

Program analyses often rely on the data manipulated by programs and can therefore depend on their static fields. Unlike instance fields, static fields are unique to each class and one would like to benefit from this uniqueness to infer precise information about their content.

When reading a variable, be it a local variable or a field, being sure it has been initialized beforehand is a nice property. Although the Java bytecode ensures this property for local variables, it is not ensured for static and instance fields which have default values.

Instance fields and static fields are not initialized the same way: instance fields are usually initialized in a constructor which is explicitly called whereas static fields are initialized in class initializers which are implicitly and lazily invoked. This makes the control flow graph much less intuitive.

The contributions of this work are the followings.

[1] Email:`first.last@irisa.fr`

[2] This work was supported in part by the Région Bretagne

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

- We recall that implicit lazy static field initialization makes the control flow graph hard to compute.
- We identify some code examples that would need to be ruled out and some other examples that would need *not* to be ruled out.
- We propose a language to study the initialization of static fields.
- We propose a formal analysis to infer an under-approximation of the set of static fields that have already been initialized for each program point.
- We propose two possible applications for this analysis: a direct application is to identify potential bugs and another one is to improve the precision while keeping the correctness of a null-pointer analysis.

The rest of this paper is organized as follows. We recall in Sect. 2 that the actual control flow graph which includes the calls the class initializers it not intuitive and give some examples. We present in Sect. 3 the syntax and semantics of the language we have chosen to formalize our analysis. Section 4 then presents the analysis, first giving an informal description and then its formal definition. We then explain in Sect. 5 how the analysis can be extended to handle other features of the Java bytecode language. In Sect. 6, we give two possible applications of this analysis. Finally, we discuss the related work in Sect. 7 and conclude in Sect. 8.

2 Why Static Analysis of Static Fields is Difficult?

The analysis we herein present works at the bytecode level but, for sake of simplicity, code examples are given in Java. As this paper is focused on static fields, all fields are assumed to be static unless otherwise stated.

In Java, a field declaration may include its initial value, such as `A.f` in Fig. 1. A field can also be initialized in a special method called a *class initializer*, which is identified in the Java source code with the `static` keyword followed by no signature and a method body, such as in class `B` in the same figure. If a field is initialized with a compile-time constant expression, the compiler from Java to bytecode may translate the initialization into a *field initializer* (cf. [12], Sect. 4.7.2), which is an attribute of the field. At run time, the field should be set to this value before running the class initializer. In-line initializations that have not been compiled as field initializers are prepended in textual order to the class initializer, named `<clinit>` at the bytecode level. For this analysis, we do not consider field initializers but focus on class initializers as they introduce the main challenges. Although this simplification is sound, it is less precise and we explain how to extend our analysis to handle field initializers in Sect. 5.3.

The class initialization process is not explicitly handled by the user: it is forbidden to explicitly called a `<clinit>` method. Instead, every access (read or write) to a field of a particular class or the creation of an instance of that same class requires that the JVM (Java Virtual Machine) has *invoked* the class initializer of that class. This method can contain arbitrary code and may trigger the initialization of other classes and so on.

The JVM specification [12] requires class initializers to be invoked *lazily*. This implies that the order in which classes are initialized depends on the *execution path*,

```
class A extends Object{static B f = new B();}
class B extends Object{
  static B g;
  static {
    g = A.f;
  }
}
```

Fig. 1. Initial values can depend on foreign code: in this example, the main program should first use `B` for the initialization to start from `B` to avoid `B.g` to be `null`.

```
class A extends Object{
  public static int CST= B.SIZE;}
class B extends Object{
  public static int SIZE = A.CST+5;
}
```

Fig. 2. Integer initial values can also depend on foreign code

it is therefore not decidable in general.

The JVM specification also requires class initializers to be invoked at most once. This avoids infinite recursions in the case of circular dependencies between classes, but it also implies that when reading a field it may not contain yet its "initial" value. For example, in Fig. 1, the class initializer of `A` creates an instance of `B` and therefore requires that the class initializer of `B` has been invoked. The class initializer of `B` reads a field of `A` and therefore requires that the class initializer of `A` has been invoked.

- If `B.<clinit>` is invoked before `A.<clinit>`, then the read access to the field `A.f` triggers the invocation of `A.<clinit>`. Then, as `B.<clinit>` has already been invoked, `A.<clinit>` carries on normally and creates an instance of class `B`, store its reference to the field `A.f` and returns. Back in `B.<clinit>`, the field `A.f` is read and the reference to the new object is also affected to `B.g`.
- If `A.<clinit>` is invoked before `B.<clinit>`, then before allocating a new instance of `B`, the JVM has to initialize the class `B` by calling `B.<clinit>`. In `B.<clinit>`, the read access to `A.f` does not trigger the initializer of `A` because `A.<clinit>` has already been started. `B.<clinit>` then reads `A.f`, which has not been initialized yet, `B.g` is therefore set to the default value of `A.f` which is the `null` constant.

This example shows that the order in which classes are initialized modifies the semantics. The issue shown in Fig. 1 is not limited to reference fields. In the example in Fig. 2, depending on the initialization order, `A.CST` will be either 0 or 5, while `B.SIZE` will always be 5.

One could notice that those problems are related to the notion of circular dependencies between classes and may think that circular dependencies should be avoided. Figure 3 shows an example with a single class. In (a), `A.ALL` is read in the constructor before it has been initialized and it leads to a `NullPointerException`.

```
class A extends Object{
  static A EMPTY=new A("");
  static HashMap ALL
            =new HashMap();

  String name;
  public A(String name){
    this.name = name;
    ALL.add(name,this);
  }
}
```

(a) An uninitialized field read leads to a `NullPointerException`

```
class A extends Object{
  static HashMap ALL
            =new HashMap();
  static A EMPTY=new A("");

  String name;
  public A(String name){
    this.name = name;
    ALL.add(name,this);
  }
}
```

(b) No uninitialized field is read

Fig. 3. The issue can arise with a single class

(b) is the correct version, where the initializations of `ALL` and `EMPTY` have been switched. This example is an extract of `java.lang.Character$UnicodeBlock` of Sun's Java Runtime Environment (JRE) that we have simplified: we want the analysis to handle such cases. If we consider that `A` depend on itself, then we forbid way to much programs. If we do not consider that `A` depend on itself and we do not reject (a) then the analysis is incorrect. We therefore cannot rely on circular dependencies between classes.

3 The Language

In this section we present the program model we consider in this work. This model is a high level description of the bytecode program that discards the features that are not relevant to the specific problem of static field initialization.

3.1 Syntax

We assume a set $\mathbb{P}$ of program points, a set $\mathbb{F}$ of field names, a set $\mathbb{C}$ of class names and a set $\mathbb{M}$ of method names. For each method m, we note $m.\mathit{first}$ the first program point of the method m. For convenience, we associate to each method a distinct program point $m.\mathit{last}$ which models the output point of the method. For each class C we note $C.$`<clinit>` the name of the class initializer of C. We only consider four kinds of instructions.

- `put`(f) updates a field $f \in \mathbb{F}$.
- `invoke` calls a method (we do not mention the name of the target method because it would be redundant with the $\mathit{flow}_{\text{inter}}$ information described below).
- `return` returns from a method.
- `any` models any other intra-procedural instruction that does not affect static fields.

As the semantics presented below will demonstrate, any instruction of the standard sequential Java bytecode can be represented by one of these instructions.

The program model we consider is based on control flow relations that must have been computed by some standard control flow analysis.

Definition 3.1 A program is a 5-tuple $(m_0, \mathit{instr}, \mathit{flow}_{\text{intra}}, \mathit{flow}_{\text{inter}}, \mathit{flow}_{\text{clinit}})$ where:

- $m_0 \in \mathbb{M}$ is the method where the program execution starts;
- $\mathit{instr} \in \mathbb{P} \rightharpoonup \{\texttt{put}(f),\ \texttt{invoke}, \texttt{return}, \texttt{any}\}$ is a partial function that associates program points to instructions;
- $\mathit{flow}_{\text{intra}} \subseteq \mathbb{P} \times \mathbb{P}$ is the set of intra-procedural edges;
- $\mathit{flow}_{\text{inter}} \subseteq \mathbb{P} \times \mathbb{M}$ is the set of inter-procedural edges, which can capture dynamic method calls;
- $\mathit{flow}_{\text{clinit}} \in \mathbb{P} \rightharpoonup \mathbb{C}$ is the set of initialization edges which forms a partial function since an instruction may only refer to one class;

and such that instr and $\mathit{flow}_{\text{intra}}$ satisfy the following property:

For any method m, for any program point $l \in \mathbb{P}$ that is reachable from $m.\mathit{first}$ in the intra-procedural graph given by $\mathit{flow}_{\text{intra}}$, and such that $\mathit{instr}(l) = \texttt{return}$, $(l, m.\mathit{last})$ belongs to $\mathit{flow}_{\text{intra}}$.

In practice, $\mathit{flow}_{\text{clinit}}$ will contain all the pairs (l, C) of a bytecode program such that the instruction found at program point l is of the form `new` C, `putstatic` $C.f$, `getstatic` $C.f$ or `invokestatic` $C.m$ (see Section 2.17.4 of [12]). [3]

Figure 7 in Sect. 4.2 presents an example of program with its three control flow relations. In this program, the main method m_0 contains two distinct paths that lead to the call of a method m. In the first one, the class `A` is initialized first and its initializer triggers the initialization of `B`. In the second path, `A` is not initialized but `B` is. `A.<clinit>` is potentially called at exit point 8 but since 8 is only reachable after a first call to `A.<clinit>`, this initialization edge is never taken.

3.2 Semantics

The analysis we consider does not take into account the content of heaps, local variables or operand stacks. To simplify the presentation, we hence choose an abstract semantics which is a conservative abstraction of the concrete standard semantics and does not explicitly manipulate these domains.

The abstract domains the semantics manipulates are presented below.

$$
\begin{array}{rll}
v \in \text{Value} & & (\textit{abstract}) \\
s \in \text{Static} & \stackrel{\text{def}}{=} & \mathbb{F} \to \text{Value} + \{\Omega\} \\
h \in \text{History} & \stackrel{\text{def}}{=} & \mathcal{P}(\mathbb{C}) \\
cs \in \text{Callstack} & \stackrel{\text{def}}{=} & (\{0,1\} \times \mathbb{P})^\star \\
<l, cs, s, h> \in \text{State} & \stackrel{\text{def}}{=} & \mathbb{P} \times \text{Callstack} \times \text{Static} \times \text{History}
\end{array}
$$

[3] To be completely correct we also need to add an edge from the beginning of the `static void main` method to the initializer of its class. We can also handle correctly superclass and interface initialization without deep modification of the current formalization.

$$\frac{\mathit{NeedInit}(l, C, h)}{<l, cs, s, h> \rightarrow <C.\texttt{<clinit>}.\mathit{first}, (\!|l|\!) :: cs, s, h \cup \{C\}>}$$

$$\frac{<l, cs, s, h> \rightarrow_1 <l', cs', s', h'> \qquad \forall C,\ \neg \mathit{NeedInit}(l, C, h)}{<l, cs, s, h> \rightarrow <l', cs', s', h'>}$$

$$\frac{\mathit{instr}(l) = \texttt{put}(f) \qquad \mathit{flow}_{\text{intra}}(l, l') \qquad v \in \text{Value}}{<l, cs, s, h> \rightarrow_1 <l', cs, s[f \mapsto v], h>}$$

$$\frac{\mathit{instr}(l) = \texttt{any} \qquad \mathit{flow}_{\text{intra}}(l, l')}{<l, cs, s, h> \rightarrow_1 <l', cs, s, h>} \qquad \frac{\mathit{instr}(l) = \texttt{invoke} \qquad \mathit{flow}_{\text{inter}}(l, m)}{<l, cs, s, h> \rightarrow_1 <m.\mathit{first}, l :: cs, s, h>}$$

$$\frac{\mathit{instr}(l) = \texttt{return} \quad \mathit{flow}_{\text{intra}}(l', l'')}{<l, l' :: cs, s, h> \rightarrow_1 <l'', cs, s, h>} \qquad \frac{\mathit{instr}(l) = \texttt{return} \quad <l', cs, s, h> \rightarrow_1\ st'}{<l, (\!|l'|\!) :: cs, s, h> \rightarrow_1\ st'}$$

Fig. 4. Operational semantics

A field contains either a value or a default value represented by the symbol Ω. Since a class initializer cannot be called twice in a same execution, we need to remember the set of classes whose initialization has been started (but not necessarily ended). This is the purpose of the element $h \in$ History. Our language is given a small-step operational semantics with states of the form $<l, cs, s, h>$, where the label l uniquely identifies the current program point, cs is a call stack that keeps track of the program points where method calls have occurred, s associates to each field its value or Ω and h is the history of class initializer calls. Each program point l of a call stack is tagged with a Boolean which indicates whether the call in l was a call to a class initializer (the element of the stack is then noted $(\!|l|\!)$) or was a standard method call (simply noted l).

The small-step relation $\rightarrow \subseteq$ State $\times$ State is given in Fig. 4 (we left implicit the program $(m_0, \mathit{instr}, \mathit{flow}_{\text{intra}}, \mathit{flow}_{\text{inter}}, \mathit{flow}_{\text{clinit}})$ that we consider). It is based on the relation $\mathit{NeedInit} \subseteq \mathbb{P} \times \mathbb{C} \times$ History defined by

$$\mathit{NeedInit}(l, C, h) \quad \stackrel{\text{def}}{=} \quad \mathit{flow}_{\text{clinit}}(l) = C \ \wedge\ C \notin h$$

which means that the class initializer of class C must be called at program point l if and only if there is a corresponding edge in $\mathit{flow}_{\text{clinit}}$ and $C.\texttt{<clinit>}$ has not been called yet (*i.e.* $C \notin h$).

The relation $\rightarrow$ is defined by two rules. In the first one, the class initializer of class C needs to be called. We hence jump to the first point of $C.\texttt{<clinit>}$, push on the call stack the previous program point (marked with the flag $(\!|\cdot|\!)$) and record C in the history h. In the second rule, there is no need to initialize a class, hence we simply use the standard semantic of the current instruction, given by the relation $\rightarrow_1 \subseteq$ State $\times$ State.

The relation $\rightarrow_1$ is defined by five rules. The first one corresponds to a field update $\texttt{put}(f)$: an arbitrary value v is stored in field f. The second rule illustrates that the instruction `any` does not affect the visible elements of the state. For a method call (third rule), the current point is pushed on the call stack and the

control is transferred to (one of) the target(s) of the inter-procedural edge. At last, the instruction `return` requires two rules. In the first case, a standard method call, the transfer comes back to the intra-procedural successor of the caller. In the second case, a class initializer call, we have finished the initialization and we must now use the standard semantic $\rightarrow_1$ of the pending instruction in program point l.

We end this section with the formal definition of the set of reachable states during a program execution. An execution starts in the main method m_0 with an empty call stack, an empty historic and with all fields associated to the default value Ω.

Definition 3.2 [Reachable States] The set of reachable states of a program $p = (m_0, \mathit{instr}, \mathit{flow}_{\text{intra}}, \mathit{flow}_{\text{inter}}, \mathit{flow}_{\text{clinit}})$ is defined by

$$[\![p]\!] = \{ <i, cs, s, h> \mid <m_0.\mathit{first}, \varepsilon, \lambda f.\Omega, \emptyset> \rightarrow^\star <i, cs, s, h> \}$$

4 A Must-Have-Been-Initialized Data Flow Analysis

In this section we present a sound data flow analysis that allows to prove a static field has already been initialized at a particular program point. We first give an informal presentation of the analysis, then present its formal definition and we finish with the statement of a soundness theorem.

4.1 Informal presentation

For each program point we want to know as precisely as possible, which fields we are sure we have initialized. Since fields are generally initialized in class initializers, we need an inter-procedural analysis that infers the set of fields *Wf* that *must* have been initialized at the end of each method. Hence at each program point l where a method is called, be it a class initializer or another method, we will use this information to improve our knowledge about initialized fields.

However, in the case of a call to a class initializer, we need to be sure the class initializer will be effectively executed if we want to safely use such an information. Indeed, at execution time, when we reach a point l with an initialization edge to a class C, despite $\mathit{flow}_{\text{clinit}}(l) = C$, two exclusive cases may happen:

(i) C.`<clinit>` has not been called yet: C.`<clinit>` is immediately called.

(ii) C.`<clinit>` has already been called (and may still be in progress): C.`<clinit>` will not be called a second time.

Using the initialization information given by C.`<clinit>` is safe only in case (i), *i.e.* the control flow information in $\mathit{flow}_{\text{clinit}}$ is not precise enough. To detect case (i), we keep track in a flow-sensitive manner of the class initializer that *may* have been called during all execution reaching a given program point. We denote by *May* this set. Here, if C is not in *May*, we are sure to be in case (i). *May* is computed by gathering, in a flow sensitive way, all classes that may be initialized starting from the main method. Implicit calls to class initializer need to be taken in account, but the smaller *May* is, the better.

```
1 class A{static int f = 1;}
2 class B{static int g = A.f;}
3 class C{
4   public static void main(String[] args){
5     ... = B.g;
6     ... = A.f;
7     ... = B.g;
8   }
9 }
```

Fig. 5. Motivating the *Must* set

For the simplicity's sake we consider in this work a context-insensitive analysis where for each method, all its calling contexts are merged at its entry point. Consider the program example given in Fig. 5. Before line 5, *May* only contains `C`, the class of the `main` method. There is an implicit flow from line 5 to the class initializer of `B`. At the beginning of the class initializer of `B`, *May* equals to $\{\mathtt{B}, \mathtt{C}\}$. We compute the set of fields initialized by `A.<clinit>`, which is $\{\mathtt{A.f}\}$. As `A` is not in *May* at the beginning of `B.<clinit>`, we can assume the class initializer will be fully executed before the actual read to `A.f` occurs, so it is a safe read. However, when we carry on line 7, the *May* set contains `A`, `B` and `C`. If we flow this information to `B.<clinit>`, then the merged calling context of `B.<clinit>` is now $\{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$, which makes impossible to assume anymore that `A.<clinit>` is called at line 2. To avoid such an imprecision, we try to propagate as few calling context as possible to class initializers by computing in a flow-sensitive manner a second set of class whose initializer *must* have already been called in all execution reaching a given program point. We denote by *Must* this set. Each time we encounter an initialization edge for a class `C`, we add `C` to *Must* since `C.<clinit>` is either called at this point, or has already been called before. If $\mathtt{C} \in \mathit{Must}$ before an initialization edge for `C`, we are sure `C.<clinit>` will not be called at this point and we can avoid to propagate a useless calling context to `C.<clinit>`.

To sum up, our analysis manipulates, in a flow sensitive manner, three sets *May*, *Must* and *Wf*. *May* and *Must* correspond to a control flow analysis that allows to refine the initialization graph given by $\mathit{flow}_{\text{clinit}}$. The more precise control flow graph allows a finer tracking of field initialization and therefore a more precise *Wf*.

4.2 Formal specification

In this part we consider a given program $p = (m_0, \mathit{instr}, \mathit{flow}_{\text{intra}}, \mathit{flow}_{\text{inter}}, \mathit{flow}_{\text{clinit}})$. We note $\mathcal{P}_p(\mathbb{C})$ (resp. $\mathcal{P}_p(\mathbb{F})$) the finite set of classes (reps. fields) that appears in p. For each program point $l \in \mathbb{P}$, we compute before and after the current point three sets of data $(\mathit{May}, \mathit{Must}, \mathit{Wf}) \in \mathcal{P}_p(\mathbb{C}) \times \mathcal{P}_p(\mathbb{C}) \times \mathcal{P}_p(\mathbb{F})$. Since *May* is a *may* information, and *Must* and *Wf* are *must* information, the underlying lattice of the data flow analysis is given by the following definition.

Definition 4.1 [Analysis lattice] The analysis lattice is $(A^\sharp, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ where:

- $A^\sharp = \mathcal{P}_p(\mathbb{C}) \times \mathcal{P}_p(\mathbb{C}) \times \mathcal{P}_p(\mathbb{F})$.

- $\bot = (\emptyset, \mathcal{P}_p(\mathbb{C}), \mathcal{P}_p(\mathbb{F}))$.
- $\top = (\mathcal{P}_p(\mathbb{C}), \emptyset, \emptyset)$.
- for all $(May_1, Must_1, Wf_1)$ and $(May_2, Must_2, Wf_2)$ in $A^\sharp$,

$$(May_1, Must_1, Wf_1) \sqsubseteq (May_2, Must_2, Wf_2) \text{ iff}$$
$$May_1 \subseteq May_2,\ Must_1 \supseteq Must_2 \text{ and } Wf_1 \supseteq Wf_2$$
$$(May_1, Must_1, Wf_1) \sqcup (May_2, Must_2, Wf_2) =$$
$$(May_1 \cup May_2, Must_1 \cap Must_2, Wf_1 \cap Wf_2)$$
$$(May_1, Must_1, Wf_1) \sqcap (May_2, Must_2, Wf_2) =$$
$$(May_1 \cap May_2, Must_1 \cup Must_2, Wf_1 \cup Wf_2)$$

Each element in $A^\sharp$ expresses properties on fields and on an initialization historic. This is formalized by the following correctness relation.

Definition 4.2 [Correctness relation] $(May, Must, Wf)$ is a correct approximation of $(s, h) \in \text{Static} \times \text{History}$, written $(May, Must, Wf) \sim (s, h)$ iff:

$$Must \subseteq h \subseteq May \quad \text{and} \quad Wf \subseteq \{\ f \in \mathbb{F} \mid s(f) \neq \Omega\ \}$$

This relation expresses that

(i) *May* contains all the classes for which we may have called the `<clinit>` method since the beginning of the program (but it may not be finished yet).

(ii) *Must* contains all the classes for which we must have called the `<clinit>` method since the beginning of the program (but it may not be finished yet neither).

(iii) *Wf* contains all the fields for which we are sure they have been written at least once.

The analysis is then specified as a data flow problem.

Definition 4.3 [Data flow solution] A *Data flow solution* of the Must-Have-Been-Initialized analysis is any couple of maps $A_{\text{in}}, A_{\text{out}} \in \mathbb{P} \to A^\sharp$ that satisfies the set of equations presented in Fig. 6, for all program point l of the program p.

In this equation system, $A_{\text{in}}(l)$ is the abstract union of three kinds of data flow information: (i) $A_0(l)$ gives the abstraction of the initial states if l is the starting point of the main method m_0; (ii) $A_{first}(l)$ is the abstract union of all calling contexts that may be transferred to l if it is the starting point of a method m. We distinguish two cases, depending on whether m is the class initializer of a class C. If it is, incoming calling contexts are transformed with $F^{\text{init}}_{\text{call}}$ which filters unfeasible calling edges with *Must* and adds C to *May* and *Must* otherwise. Otherwise, incoming calling contexts are transformed with F^{init} (described below) which take into account the potential class initialization that may have been performed before the call. (iii) At last, we merge all incoming data flows from predecessors in the intra-procedural graph.

$$A_{\text{in}}(l) = A_0(l) \sqcup A_{\mathit{first}}(l) \sqcup \bigsqcup\{A_{\text{out}}(l') \mid \mathit{flow}_{\text{intra}}(l', l)\}$$

$$A_{\text{out}}(l) = \begin{cases} F_{\text{call}}\left(F^{\text{init}}(l, A_{\text{in}}(l)), \bigsqcup\{A_{\text{in}}(m.\mathit{last}) \mid \mathit{flow}_{\text{inter}}(l, m)\}\right) & \text{if } \mathit{instr}(l) = \texttt{invoke} \\ F_{\mathit{instr}(l)}(F^{\text{init}}(l, A_{\text{in}}(l))) & \text{otherwise} \end{cases}$$

where

$$A_0(l) = \begin{cases} (\emptyset, \emptyset, \emptyset) & \text{if } l = m_0.\mathit{first} \\ \bot & \text{otherwise} \end{cases}$$

$$A_{\mathit{first}}(l) = \begin{cases} \bigsqcup\left\{F^{\text{init}}_{\text{call}}(C, A_{\text{in}}(l')) \mid \mathit{flow}_{\text{clinit}}(l') = C\right\} & \text{if } l = C.\texttt{<clinit>}.\mathit{first} \\ \bigsqcup\left\{F^{\text{init}}(l', A_{\text{in}}(l')) \mid \mathit{flow}_{\text{inter}}(l', m)\right\} & \text{if } l = m.\mathit{first} \\ \bot & \text{otherwise} \end{cases}$$

and $F_{\texttt{return}}$, $F_{\texttt{any}}$, $F_{\texttt{put}(f)} \in A^\sharp \to A^\sharp$, $F_{\text{call}} \in A^\sharp \times A^\sharp \to A^\sharp$, $F^{\text{init}}_{\text{call}} \in \mathbb{C} \times A^\sharp \to A^\sharp$ and $F^{\text{init}} \in \mathbb{P} \times A^\sharp \to A^\sharp$ are transfer functions defined by:

$$F_{\texttt{return}}(\mathit{May}, \mathit{Must}, \mathit{Wf}) = F_{\texttt{any}}(\mathit{May}, \mathit{Must}, \mathit{Wf}) = (\mathit{May}, \mathit{Must}, \mathit{Wf})$$
$$F_{\texttt{put}(f)}(\mathit{May}, \mathit{Must}, \mathit{Wf}) = (\mathit{May}, \mathit{Must}, \mathit{Wf} \cup \{f\})$$

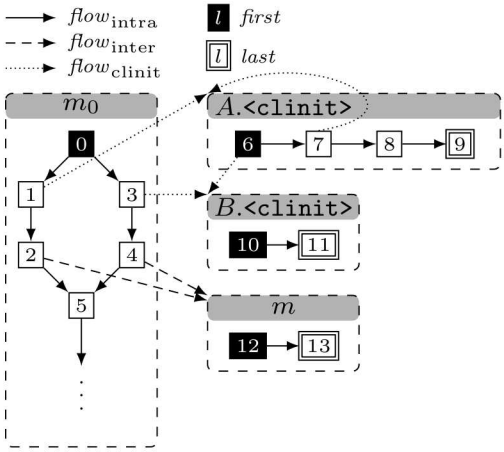
$$F_{\text{call}}((\mathit{May}_1, \mathit{Must}_1, \mathit{Wf}_1), (\mathit{May}_2, \mathit{Must}_2, \mathit{Wf}_2)) = (\mathit{May}_2, \mathit{Must}_1 \cup \mathit{Must}_2, \mathit{Wf}_1 \cup \mathit{Wf}_2)$$

$$F^{\text{init}}_{\text{call}}(C, (\mathit{May}, \mathit{Must}, \mathit{Wf})) = \begin{cases} \bot & \text{if } C \in \mathit{Must} \\ (\mathit{May} \cup \{C\}, \mathit{Must} \cup \{C\}, \mathit{Wf}) & \text{otherwise} \end{cases}$$

$$F^{\text{init}}(l, a) = F^{\text{init}}(l, (\mathit{May}, \mathit{Must}, \mathit{Wf})) = \begin{cases} F_{\text{call}}(a, A_{\text{in}}(C.\texttt{<clinit>}.\mathit{last})) & \text{if } \mathit{flow}_{\text{clinit}}(l) = C \text{ and } C \notin \mathit{May} \\ F^{\text{init}}_{\text{call}}(C, a) \sqcup F_{\text{call}}(a, A_{\text{in}}(C.\texttt{<clinit>}.\mathit{last})) & \text{if } \mathit{flow}_{\text{clinit}}(l) = C \\ & \quad \text{and } C \in \mathit{May} \text{ and } C \notin \mathit{Must} \\ a & \text{otherwise} \end{cases}$$

Fig. 6. Data flow analysis

The equation on $A_{\text{out}}(l)$ distinguishes two cases, depending on $\mathit{instr}(l)$ is a method call or not. If it is, we merge all data flows from the end of the potentially called methods and combine them, using F_{call} described below, with the data flows facts $F^{\text{init}}(l, A_{\text{in}}(l))$ that is true just before the call. Otherwise, we transfer the data flow $A_{\text{in}}(l)$, found at entry of the current instruction with F^{init} and then $F_{\mathit{instr}(l)}$. While F^{init} handles potential class initialization that may have been performed before the instruction, $F_{\mathit{instr}(l)}$ simply handles the effect of the instruction



(a) A program example

l	$instr(l)$	$A_{\text{in}}(l)$			$A_{\text{out}}(l)$		
		May	*Must*	*Wf*	*May*	*Must*	*Wf*
0	any	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1	any	$\emptyset$	$\emptyset$	$\emptyset$	$\{A, B\}$	$\{A, B\}$	$\emptyset$
2	invoke	$\{A, B\}$	$\{A, B\}$	$\{f\}$	$\{A, B\}$	$\{A, B\}$	$\{f\}$
3	any	$\emptyset$	$\emptyset$	$\emptyset$	$\{A, B\}$	$\{B\}$	$\emptyset$
4	invoke	$\{A, B\}$	$\{B\}$	$\emptyset$	$\{A, B\}$	$\{B\}$	$\emptyset$
5	any	$\{A, B\}$	$\{B\}$	$\emptyset$	$\{A, B\}$	$\{B\}$	$\emptyset$
6	put(f)	$\{A\}$	$\{A\}$	$\emptyset$	$\{A, B\}$	$\{A, B\}$	$\{f\}$
7	any	$\{A, B\}$	$\{A, B\}$	$\{f\}$	$\{A, B\}$	$\{A, B\}$	$\{f\}$
8	return	$\{A, B\}$	$\{A, B\}$	$\{f\}$	$\{A, B\}$	$\{A, B\}$	$\{f\}$
9		$\{A, B\}$	$\{A, B\}$	$\{f\}$	$\{A, B\}$	$\{A, B\}$	$\{f\}$
10	return	$\{A, B\}$	$\{B\}$	$\emptyset$	$\{A, B\}$	$\{B\}$	$\emptyset$
11		$\{A, B\}$	$\{B\}$	$\emptyset$	$\{A, B\}$	$\{B\}$	$\emptyset$
12	return	$\{A, B\}$	$\{B\}$	$\emptyset$	$\{A, B\}$	$\{B\}$	$\emptyset$
13		$\{A, B\}$	$\{B\}$	$\emptyset$	$\{A, B\}$	$\{B\}$	$\emptyset$

(b) Least dataflow solution of the program example

Fig. 7. Program and analysis example

$instr(l)$ in a straightforward manner.

The transfer function F^{init} is defined with tree distinct cases. (i) In the first case, we are sure the class initializer C.<clinit> will be called because it has never been called before. We can hence use safely the last data flow of C.<clinit> but we combine it with a using the operator F_{call} described below. (ii) In the second case, we are sure that no class initializer will be called, either because there is no initialization edge at all, or there is one for a class C but we know that C.<clinit> has already been called. (iii) In the last case, the two previous cases may happen so we merge the corresponding data flows.

At last, F_{call} is an operator which combines dataflows about calling contexts and calling returns. It allows to recover some *must* information that may have been discarded during the method call because of spurious calling contexts. It is based on the monotony of *Must* and *Wf*: these sets are under-approximations of initialization history and initialized fields but since such sets only increase during execution a correct under-approximation (*Must*, *Wf*) at a point l is still a correct approximation at every point reachable from l.

The program example presented is Fig. 7 is given with the least solution of its corresponding dataflow problem. In this example, A.<clinit> has two potential callers in 1 and 7 but we don't propagate the dataflow facts from 7 to 6 because we know that A.<clinit> has already been called at this point, thanks to *Must*. At point 2, the method m is called but we don't propagate in $A_{\text{out}}(2)$ the exact values found in $A_{\text{in}}(m.last)$ because we would lose the fact that $A \in Must$ before the call. That is why we combine $A_{\text{in}}(2)$ and $A_{\text{in}}(m.last)$ with F_{call} in order to refine the must information *Must* and *Wf*.

Theorem 4.4 (Computability) *The least data flow solution for the partial order $\sqsubseteq$ is computable by the standard fixpoint iteration techniques.*

Proof. This is consequence of the facts that each equation is monotone, there is a finite number of program points in p and $(A^\sharp, \sqsubseteq, \sqcup, \sqcap)$ is a finite lattice. $\square$

Theorem 4.5 (Soundness) *If* $(A_{\text{in}}, A_{\text{out}})$ *is a data flow solution then for all reachable states* $<i, cs, s, h> \in [\![p]\!]$, $A_{\text{in}}(i) \sim (s, h)$ *holds.*

Proof sketch. We first define an intermediate semantics $\rightsquigarrow$ which is shown equivalent to the small-step relation $\rightarrow$ but in which method calls are big-steps: for each point l where a method m is called we go in one step to the intra-procedural successor of l using the result of the transitive closure of $\rightsquigarrow$. Such a semi-big-step semantics is easier to reason with method calls. Once $\rightsquigarrow$ is define, we prove a standard subject reduction lemma between $\rightsquigarrow$ and $\sim$ and we conclude. □

5 Handling the Full Bytecode

5.1 Exceptions

From the point of view of this analysis, exceptions only change the control flow graph. As the control flow graph is computed separately, it should not change the analysis herein described.

However, if we really need to be conservative, loads of instructions may throw runtime exceptions (`IndexOutOfBoundException`, `NullPointerException`, etc.) or even errors (`OutOfMemoryError`, etc.) and so there will be edges in the control flow graph from most program points to the exit point of the methods, making the analysis very imprecise.

There are several ways to improve the precision while safely handling exceptions. First, we can prove the absence of exception for some of those (*e.g.* see [9] to remove most `NullPointerExceptions` and [1] to remove the `IndexOutOfBoundExceptions` you need to remove). Then it is cheap to analyse, for each method, the context in which the method is called, *i.e.* the exceptions that may be caught if they are thrown by the method: if there are no handler for some exception in the context of a particular method, then there is no use to take in account this exception in the control flow graph of this method. Indeed, if such an exception were thrown it would mean the termination of the program execution so not taking in account the exception may only add potential behaviours, which is safe.

5.2 Inheritance

In the presence of a class hierarchy, the initialization of a class starts by the initialization of its superclass if it has not been done yet. There is therefore an implicit edge in the control flow graph from each `<clinit>` method to the `<clinit>` method of its superclass (except for `Object.<clinit>`). Although it does not involve any challenging problem, the semantics and the formalization need to be modified to introduce a new label at the beginning of each `<clinit>` method such that, if l is the label we introduce at the beginning of `C.<clinit>`, then $\mathit{flow}_{\text{clinit}}(l) = \mathit{super}(C)$.

Note that it is not required to initialize the interfaces a class implements, nor the super interfaces an interface extends (cf. Sect. 2.17.4 of [12]).

5.3 Field Initializers and Initialization order

Although the official JVM specification states (Sect. 2.17.4) that the initialization of the superclass should be done before the initialization of the current class and that the field initializers are part of the initialization process, in Sun's JVM the field initializers are used to set the corresponding fields before starting the initialization of the superclass. This changes the semantics but removes potential defects as this way it is impossible for some code to read the field before it has been set.

If the analysis is targeted to a such JVM implementation, depending on the application of the analysis, the fields which have a field initializer can be safely ignored when displaying the warnings found, or be added to *Wf* either when the corresponding class is added to *Must* or at the very beginning ($m_o.first$). In order for the analysis to be compatible with the official specification, the analysis needs to simulate the initialization of the fields that have a field initializer at the beginning of the class initializer, just after the implicit call to the class initializer of its superclass.

5.4 Reflection, User-Defined Class-Loader, Class-Path Modification, etc.

There are several contexts in which this analysis can be used: it can be used to find bugs or to prove the correctness of some code, either off-line or in a PCC [15] architecture.

In case it used to find bugs, the user mainly need to be aware that those features are not supported.
If it is used to prove at compile time the correctness of some code, the analysis needs to handle those features. In case of reflection, user-defined class-loaders or modification of the class-path, it is difficult to be sure that all the code that may be executed has been analyzed. The solution would restrict the features of the language in order to be able to infer what code *may* be executed, which is an over-approximation of the code that *will* be executed, and to analyze this code. For example, Livshits *et al.* proposed in [13] an analysis to correctly handle reflection.

In a PCC architecture, the code is annotated at compile time and checked at run time. The issue is no more to find the code that will be executed, because the checking is done at run time when it is a lot easier to know what code will be executed, but to consider *enough* code when annotating. This can be done the same way as with off-line proofs and by asking the programmer when it is not possible to infer a precise enough solution. In this case, if the user gives incorrect data the checker will notice it while checking the proof at run time.

6 Two Possible Applications for the Analysis

6.1 Checking That Fields are Written Before Being Read

The analysis presented in this paper computes a set of fields that have been written for each program point. To check that all fields are written before being read, *i.e.* that when a field `C.f` is read at program point l, we need to check that `C.f` is in the set of written field at this particular program point.

6.2 Nullness Analysis of Static Fields

While in our previous work [10] we choose to assume no information about static fields, several tools have targeted the analysis of static fields as part of their analysis but missed the issue of the initialization herein discussed.

To safely handle static fields while improving the precision, we can abstract every field by the abstraction of the values that may be written to it and, if the static field may be read before being initialized, then we add the abstraction of the `null` constant to the abstraction of the field. It is a straightforward extension of the analysis presented in [10].

7 Related work

Kozen and Stillerman studied in [11] eager class initialization for Java bytecode and proposed a static analysis based on fine grained circular dependencies to find an initialization order for classes. If their analysis finds an initialization order, then our analysis will be able to prove all fields are initialized before being read. If their analysis finds a circular dependency, it fails to find an initialization order and issues an error while our analysis considers the initialization order implied by the main program and may prove that all fields are written before being read.

Instance field initialization have been studied for different purposes. Some works are focused on null-ability properties such as Fähndrich and Leino in [5], our work in [10] or Fähndrich and Xia in [6]. Other work have been focused on different properties such as Unkel and Lam [16] who studied stationary fields. Instance field initialization offers different challenges from the one of static fields: the initialization method is explicitly called soon after the object allocation.

Several formalizations of the Java bytecode have been proposed that, among other features, handled class initialization such as the work of Debbabi *et al.* in [4] or Belblidia and Debbabi in [14]. Their work is focused on the dynamic semantics of the Java bytecode while our work is focused on its analysis.

Böerger and Schulte [2] propose another dynamic semantics of Java. They consider a subset of Java including initialization, exceptions and threads. They have exhibited [3] some weaknesses in the initialization process as far as the threads are used. They pointed out that deadlocks could occur in such a situation.

Harrold and Soffa [7] propose an analysis to compute inter-procedural definition-use chains. They have not targeted the Java bytecode language and therefore neither the class initialization problems we have faced but then, our analysis can be seen as a lightweight inter-procedural definition-use analysis where all definitions except the default one are merged.

Hirzel *et al.* [8] propose a pointer analysis that target dynamic class loading and lazy class initialization. There approach is to analyse the program at run time, when the actual classes have been loaded and to update the data when a new class is loaded and initialized. Although it is not practical to statically certify programs, a similar approach could certainly be adapted to implement a checker in PCC architecture such as the one evoked in Sect. 5.4.

8 Conclusion and Future Work

We have shown that class initialization is a complex mechanism and that, although in most cases it works as excepted, in some more complicated examples it can be complex to understand in which order the code will be executed. More specifically, some fields may be read before being initialized, despite being initialized in their corresponding class initialization methods. A sound analysis may need to address this problem to infer precise and correct information about the content of static fields. We have proposed an analysis to identify the static fields that may be read before being initialized and shown how this analysis can be used to infer more precise information about static fields in a sound null-pointer analysis.

We expect the analysis to be very precise if the control flow graph is *accurate enough*, but we would need to implement this analysis to evaluate the precision needed for the control flow graph.

References

[1] Bodik, R., R. Gupta and V. Sarkar, *ABCD: eliminating array bounds checks on demand*, in: *Proc. of PLDI*, 2000, pp. 321–333.

[2] Börger, E. and W. Schulte, *A programmer friendly modular definition of the semantics of java*, in: *Formal Syntax and Semantics of Java*, LNCS (1999), pp. 353–404.

[3] Börger, E. and W. Schulte, *Initialization problems for java*, Software - Concepts and Tools **19** (2000), pp. 175–178.

[4] Debbabi, M., N. Tawbi and H.Yahyaoui, *A formal dynamic semantics of java: An essential ingredient of java security*, Journal of Telecommunications and Information Technology **4** (2002), pp. 81–119.

[5] Fähndrich, M. and K. R. M. Leino, *Declaring and checking non-null types in an object-oriented language*, ACM SIGPLAN Notices **38** (2003), pp. 302–312.

[6] Fähndrich, M. and S. Xia, *Establishing object invariants with delayed types*, in: *Proc. of OOPSLA* (2007), pp. 337–350.

[7] Harrold, M. J. and M. L. Soffa, *Efficient computation of interprocedural definition-use chains*, TOPLAS **16** (1994), pp. 175–204.

[8] Hirzel, M., A. Diwan and M. Hind, *Pointer analysis in the presence of dynamic class loading*, in: *Proc. of ECOOP*, 2004, pp. 96–122.

[9] Hubert, L., *A Non-Null annotation inferencer for Java bytecode*, in: *Proc. of PASTE* (2008), (To Appear).

[10] Hubert, L., T. Jensen and D. Pichardie, *Semantic foundations and inference of non-null annotations*, in: *Proc. of FMOODS*, LNCS **5051** (2008), pp. 132–149.

[11] Kozen, D. and M. Stillerman, *Eager class initialization for java*, in: *Proc. of FTRTFT* (2002), pp. 71–80.

[12] Lindholm, T. and F. Yellin, "The Java^{TM} Virtual Machine Specification, Second Edition," Prentice Hall PTR, 1999.

[13] Livshits, B., J. Whaley and M. S. Lam, *Reflection analysis for java*, in: *Proc of APLAS*, 2005.

[14] N. Belblidia, M. D., *A dynamic operational semantics for JVML*, Journal of Object Technology **6** (2007), pp. 71–100.

[15] Necula, G., *Proof-Carrying Code*, in: *Proc. of POPL* (1997), pp. 106–119.

[16] Unkel, C. and M. S. Lam, *Automatic inference of stationary fields: a generalization of java's final fields*, in: *Proc. of POPL* (2008), pp. 183–195.