class A extends Object {
    static A EMPTY = new A("'");
    static HashMap ALL = new HashMap();
    String name;
    public A(String n){
        this.name = n;
        ALL.add(n,this);
    }
}
class A extends Object {
    static A EMPTY = new A(" ");
    static HashMap ALL = new HashMap();
    String name;
    public A(String n) {
        this.name = n;
        ALL.add(n, this);
    }
}
class A extends Object {
  static A EMPTY;
  static HashMap ALL;
  String name;
  public A(String n) {
    this.name = n;
    ALL.add(n, this);
  }
  static {
    EMPTY = new A(""");
    ALL = new HashMap();
  }
}

Java Source vs. Java Bytecode

class A extends Object {
    static A EMPTY;
    static HashMap ALL;
    String name;
    public A(String n) {
        this.name = n;
        ALL.add(n, this);
    }
    static <clinit> {
        EMPTY = new A(" ");
        ALL = new HashMap();
    }
}
The JVM Spec Says...

Initialization of a class consists of executing its static initializers (§2.13) and the initializers for static fields (§2.13.2) declared in the class. Initialization of an interface consists of executing the initializers for fields declared in the interface (§2.13.2.1).

Before a class or interface is initialized, its direct superclasses must be initialized, but interfaces implemented by the class need not be initialized. Similarly, the superinterfaces of an interface need not be initialized before the interface is initialized.

A class or interface type T will be initialized immediately before one of the following occurs:

- T is a class and an instance of T is created.
- T is a class and a static method of T is invoked.
- A nonconstant static field of T is used or assigned. A constant field is one that is (explicitly or implicitly) both final and static, and that is initialized with the value of a compile-time constant expression. A reference to such a field must be resolved at compile time to a copy of the compile-time constant value, so any of such a field never cause initialization.

Invocation of certain methods in library classes (§5.12) also causes class or interface initialization. See the Java 2 Platform class library specifications (for example, class Class and package java.lang.reflect) for details.

The intent here is that a type have a set of initializers that put it in a consistent state and that this state be the first state that is observed by other classes. The static initializers and class variable initializers are executed in textual order and may not refer to class variables declared in the class whose declarations appear textually after the use, even though these class variables are in scope. This restriction is designed to detect, at compile time, most circular or otherwise malformed initializations.

Before a class or interface is initialized its superclass is initialized, if it has not previously been initialized.

2.17.5 Detailed Initialization Procedure

Initialization of a class or interface requires careful synchronization, since some other thread may be trying to initialize the same class or interface at the same time. There is also the possibility that initialization of a class or interface may be requested recursively as part of the initialization of that class or interface. For example, a variable initializer in class A might invoke a method of an unrelated class B, which might in turn invoke a method of class A. The implementation of the Java virtual machine is responsible for taking care of synchronization and recursive initialization by using the following procedure. It assumes that the class object has already been verified and prepared and that the class object contains state that can indicate one of four situations:

- This class object is verified and prepared but not initialized.
- This class object is being initialized by some particular thread T.
- This class object is fully initialized and ready for use.
- This class object is in an erroneous state, perhaps because the verification step failed or because initialization was attempted and failed.

The procedure for initializing a class or interface is as follows:

1. Synchronize on the class object that represents the class or interface to be initialized. This involves waiting until the current thread that can obtain the lock for that object (§8.13).
2. If initialization by some other thread is in progress for the class or interface, then wait on this class object (which temporarily releases the lock). When the current thread awakens from the wait, repeat this step.
3. If initialization is in progress for the class or interface by the current thread, then this must be a recursive request for initialization. Release the lock on the class object and complete normally.
4. If the class or interface has already been initialized, then no further action is required. Release the lock on the class object and complete normally.
5. If the class object is in an erroneous state, then initialization is not possible. Release the lock on the class object and throw a ClassInitializedError.
6. Otherwise, record the fact that initialization of the class object is now in progress by the current thread and release the lock on the class object.
7. Next, if the class object represents a class rather than an interface, and the direct superclasses of this class have not yet been initialized, then recursively perform this entire procedure for the direct superclass. If the initialization of the direct superclass completes abruptly because of a thrown exception, then lock this class object, label it erroneous, notify all waiting threads, release the lock, and complete abruptly, throwing the same exception that resulted from the initializing the superclass.
8. Next, create either the class variable initializers and static initializers of the class or the field initializers of the interface, in textual order, as though they were a single block, except that final static variables and fields of interfaces whose values are compile-time constants are initialized first.
9. If the execution of the initializers completes normally, then lock this class object, label it fully initialized, notify all waiting threads, release the lock, and complete this procedure normally.
10. Otherwise, the initializers must have completed abruptly by throwing some exception E. If the class object is not a class, then create a new instance of the class ExceptionalInitializationError, with E as the argument, and use this object in place of E in the following step. But if a new instance of ExceptionalInitializationError cannot be created because a NullPointerException occurs, then instead use anNullPointerException in place of E in the following step.
11. Lock the class object, label it erroneous, notify all waiting threads, release the lock, and complete this procedure abruptly with reason E or its replacement as determined in the previous step.

In some early implementations of the Java virtual machine, an exception during class initialization was ignored rather than allowing it to cause an ExceptionalInitializationError as described here.
A class or interface type T will be *initialized* immediately before one of the following occurs:

- T is a class and an instance of T is created.
- T is a class and a static method of T is invoked.

A nonconstant static field of T is used or assigned. A constant field is one that is (explicitly or implicitly) both final and static, and that is initialized with the value of a compile-time constant expression. A reference to such a field must be resolved at compile time to a copy of the compile-time constant value, so uses of such a field never cause initialization.
A class or interface type $T$ will be *initialized* immediately before one of the following occurs:

- $T$ is a class and an instance of $T$ is created.
- $T$ is a class and a static method of $T$ is invoked.

A nonconstant static field of $T$ is used or assigned. A constant field is one that is (explicitly or implicitly) both *final* and *static*, and that is initialized with the value of a compile-time constant expression. A reference to such a field must be resolved at compile time to a copy of the compile-time constant value, so uses of such a field never cause initialization.
The JVM Spec Says...

Class initializers are lazily invoked, triggered by (get|put|invoke)static and new instructions.

Interface type T will be initialized immediately before one of the following:

- T is a class and an instance of T is created.
- T is a class and a static method of T is invoked.

A nonconstant static field of T is used or assigned. A constant field is one that is (explicitly or implicitly) both final and static, and that is initialized with the value of a compile-time constant expression. A reference to such a field must be resolved at compile time to a copy of the compile-time constant value, so uses of such a field never cause initialization.
The JVM Spec Says...

A class is considered to be initialized if and only if...

...triggered by \((\text{get|put|invoke})\) \text{static} and \text{new} instructions

A nonconstant static field of T is used or assigned. A constant field is one that is (explicitly or implicitly) both \text{final} and \text{static}, and that is initialized with the value of a compile-time constant expression. A reference to such a field must be resolved at compile time to a copy of the compile-time constant value, so uses of such a field never cause initialization.

- T is a class and an instance of T is created.
- T is a class and a static method of T is invoked.

4. If the class or interface has already been initialized, then no further action is required. Release the lock on the class object and complete normally.
The JVM Spec Says...

Class initializers are lazily invoked, triggered by \texttt{(get|put|invoke)static} and new instructions, and called only once.
The JVM Spec Says...

class initializer are lazily invoked, ...
triggered by (get|put|invoke) static and new instructions
...and called only once

this makes the control flow hard to predict
class A extends Object {
    static A EMPTY;
    static HashMap ALL;
    String name;
    public A(String n){
        this.name = n;
        ALL.add(n,this);
    }
    static <clinit>{
        EMPTY = new A(""Initializing static methods");
        ALL = new HashMap();
    }
}
class A extends Object {
    static A EMPTY;
    static HashMap ALL;
    String name;
    public A(String n) {
        this.name = n;
        ALL.add(n, this);
    }
    static <clinit> {
        EMPTY = new A(""");
        ALL = new HashMap();
    }
}
class A extends Object {
    static A EMPTY;
    static HashMap ALL;
    String name;
    public A(String n){
        this.name = n;
        ALL.add(n,this);
    }
    static <clinit>{
        EMPTY = new A(" ");
        ALL = new HashMap();
    }
}

class A extends Object {
    static A EMPTY;
    static HashMap ALL;
    String name;
    public A(String n) {
        this.name = n;
        ALL.add(n, this);
    }
    static <clinit> {
        EMPTY = new A(" ");
        ALL = new HashMap();
    }
}
class A extends Object {
    static A EMPTY;
    static HashMap ALL;
    String name;
    public A(String n) {
        this.name = n;
        ALL.add(n, this);
    }
    static <clinit> {
        EMPTY = new A(" ");
        ALL = new HashMap();
    }
}
class A extends Object {
    static A EMPTY;
    static HashMap ALL;
    String name;
    public A(String n) {
        this.name = n;
        ALL.add(n, this);
    }
    static <clinit> {
        ALL = new HashMap();
        EMPTY = new A(" ");
    }
}
class A extends Object {
  static A EMPTY;
  static HashMap ALL;
  String name;
  public A(String n) {
    this.name = n;
    ALL.add(n, this);
  }

  static <clinit> {
    ALL = new HashMap();
    EMPTY = new A(" ");
  }
}
class A extends Object {
    static A EMPTY;
    static HashMap ALL;
    String name;
    public A(String n){
        this.name = n;
        ALL.add(n,this);
    }
    static <clinit>{
        ALL = new HashMap();
        EMPTY = new A(" ");
    }
}
class A extends Object {
    static A EMPTY;
    static HashMap ALL;
    String name;
    public A(String n) {
        this.name = n;
        ALL.add(n, this);
    }
    static <clinit> {
        ALL = new HashMap();
        EMPTY = new A(" ");
    }
}
class A extends Object {
    static A EMPTY;
    static HashMap ALL;
    String name;
    public A(String n){
        this.name = n;
        ALL.add(n,this);
    }
    static <clinit>{
        ALL = new HashMap();
        EMPTY = new A(" ");
    }
}
```java
class A extends Object {
    static A EMPTY;
    static HashMap ALL;
    String name;
    public A(String n){
        this.name = n;
        ALL.add(n, this);
    }
    static <clinit>{
        ALL = new HashMap();
        EMPTY = new A("");}
}
```
class A extends Object {
    static A EMPTY;
    static HashMap ALL;
    String name;
    public A(String n) {
        this.name = n;
        ALL.add(n, this);
    }
    static <clinit> {
        ALL = new HashMap();
        EMPTY = new A(" ");
    }
}
class A extends Object {
    static A EMPTY;
    static HashMap ALL;
    String name;
    public A(String n) {
        this.name = n;
        ALL.add(n, this);
    }
    static <clinit> {
        ALL = new HashMap();
        EMPTY = new A(" ");
    }
}
class A extends Object {
    static A EMPTY;
    static HashMap ALL;
    String name;
    public A(String n) {
        this.name = n;
        ALL.add(n, this);
    }
    static <clinit> {
        ALL = new HashMap();
        EMPTY = new A(" ");
    }
}
Outlines

- Introduction on class initialization
  - The control flow is hard to predict
  - Such examples can be found in SUN's JRE
- Language (syntax and semantics)
  - Focused on class initialization and field writes
- Analysis
  - Proves static fields are initialized
- Conclusion on related and future work
The Language

A program is a 5-tuple
\[ (m_0, \text{instr}, \text{flow}_{\text{intra}}, \text{flow}_{\text{inter}}, \text{flow}_{\text{clinit}}) \]

\[ m_0 \in \mathbb{M} \]

\[ \text{instr} \in \mathbb{P} \rightarrow \{ \text{put}(f), \ \text{invoke}, \ \text{return}, \ \text{any} \} \]

\[ \text{flow}_{\text{intra}} \subseteq \mathbb{P} \times \mathbb{P} \]

\[ \text{flow}_{\text{inter}} \subseteq \mathbb{P} \times \mathbb{M} \]

\[ \text{flow}_{\text{clinit}} \in \mathbb{P} \rightarrow \mathbb{C} \]
A program is a 5-tuple

\( (m_0, \text{instr}, \text{flow}_{\text{intra}}, \text{flow}_{\text{inter}}, \text{flow}_{\text{clinit}}) \)

\( m_0 \in M \)

\( \text{instr} \in P \rightarrow \{\text{put}(f), \text{invoke}, \text{return}, \text{any}\} \)

\( \text{flow}_{\text{intra}} \subseteq P \times P \)

\( \text{flow}_{\text{inter}} \subseteq P \times M \)

\( \text{flow}_{\text{clinit}} \in P \rightarrow C \)
The Language

A program is a 5-tuple
\[(m_0, \text{instr}, \text{flow}_{\text{intra}}, \text{flow}_{\text{inter}}, \text{flow}_{\text{clinit}})\]

\[m_0 \in \mathcal{M}\]

\[\text{instr} \in \mathcal{P} \rightarrow \{\text{put}(f), \text{invoke}, \text{return}, \text{any}\}\]

\[\text{flow}_{\text{intra}} \subseteq \mathcal{P} \times \mathcal{P}\]

\[\text{flow}_{\text{inter}} \subseteq \mathcal{P} \times \mathcal{M}\]

\[\text{flow}_{\text{clinit}} \in \mathcal{P} \rightarrow \mathcal{C}\]
The Language

A program is a 5-tuple
\((m_0, instr, flow_{\text{intra}}, flow_{\text{inter}}, flow_{\text{cinit}})\)

\(m_0 \in \mathbb{M}\)

\(instr \in \mathbb{P} \rightarrow \{\text{put}(f), \text{invoke}, \text{return}, \text{any}\}\)

\(flow_{\text{intra}} \subseteq \mathbb{P} \times \mathbb{P}\)

\(flow_{\text{inter}} \subseteq \mathbb{P} \times \mathbb{M}\)

\(flow_{\text{cinit}} \in \mathbb{P} \rightarrow \mathbb{C}\)
The Language
Concrete Semantics

\<l, cs, s, h> \in \text{State} \overset{\text{def}}{=} \mathcal{P} \times \text{Callstack} \times \text{Static} \times \text{History}

\begin{align*}
\text{cs} & \in \text{Callstack} \overset{\text{def}}{=} \mathcal{P}^* \\
\text{s} & \in \text{Static} \overset{\text{def}}{=} \mathcal{F} \rightarrow \text{Value} + \{\text{undef}\} \\
\text{h} & \in \text{History} \overset{\text{def}}{=} 2^\mathcal{C}
\end{align*}

set of initialized classes

static heap
\[ \text{NeedInit}(l, C, h) \overset{\text{def}}{=} \text{flow}_{\text{clinit}}(l) = C \land C \notin h \]
The Language
Concrete Semantics

\[ \text{Def} \quad \text{NeedInit}(l, C, h) \iff \text{flow}_{\text{clinit}}(l) = C \land C \notin h \]

\[ \frac{\text{NeedInit}(l, C, h) \quad l' = C.<\text{clinit}>.\text{first}}{\langle l, cs, s, h \rangle \rightarrow \langle l', l :: cs, s, h \cup \{C\} \rangle} \]
The Language
Concrete Semantics

\[ \text{NeedInit}(l, C, h) \overset{\text{def}}{=} \text{flow}_{\text{clinit}}(l) = C \land C \notin h \]

\[ \text{NeedInit}(l, C, h) \quad l' = C.\langle\text{clinit}\rangle.\text{first} \]

\[ \langle l, cs, s, h \rangle \rightarrow \langle l', l :: cs, s, h \cup \{C\} \rangle \]

The complete semantics is in the paper
class A { static A f = new A(); }
class B { static A g = A.f; }
class C {
    static void main(String args[]){
        A a = B.f;
        A a = B.f;}}
class A { static A f = new A(); }
class B { static A g = A.f; }
class C {
    static void main(String args[]){
        A a = B.f;
        A a = B.f;
    }
}
Purpose: proving a static field has already been initialized at a particular program point

Concrete state: \(<l, cs, s, h>\)

Program points are approximated by \((Must, May, Wf)\)

- \(Wf\) under-approximates \(s\)
- \(Must\) and \(May\) under- and over-approximates \(h\)
Concrete state: \(<l, cs, s, h>\)

To each program point we attached

- \(Wf\): set of static fields that \textbf{must} have been set
  \[ Wf \subseteq \{ f \in F \mid s(f) \neq \text{undef} \} \]
Concrete state: \(<l, cs, s, h>\)

To each program point we attached

- \(Wf\): set of static fields that **must** have been set
  \[ Wf \subseteq \{ f \in F \mid s(f) \neq \text{undef} \} \]

- Two sets of classes (abstraction of \(h\))
  **Must**: we must have call their initializer
  **May**: we may have call their initializer
  \[ \text{Must} \subseteq h \subseteq \text{May} \]
Whole-program data-flow analysis
Abstract Domains

\[ A^\# = 2^C \times 2^C \times 2^F \]
\[ A_{\text{in}} \in \mathbb{P} \rightarrow A^\# \]
\[ A_{\text{out}} \in \mathbb{P} \rightarrow A^\# \]
Whole-program data-flow analysis

Abstract Domains

\[
\begin{align*}
A^# &= 2^C \times 2^C \times 2^F \\
A_{\text{in}} \in \mathcal{P} &\rightarrow A^# \\
A_{\text{out}} \in \mathcal{P} &\rightarrow A^#
\end{align*}
\]
Data-flow Analysis

Example

- Call to class initializer
- Usefulness of the May set
main

0:

1: any (a=B.f)

12: any (a=B.f)

13: return

14:

B.<clinit>

2:

3: any (get A.f)

9: put B.g

10: return

11:

A.<clinit>

4:

5: any (new A)

6: put A.f

7: return

8:

\[
\begin{array}{|c|c|c|c|c|c|c|c|}
\hline
& \mathbf{A_{in}(I)} & & \mathbf{A_{out}(I)} & \\
& \text{Must} & \text{May} & \text{Wf} & \text{Must} & \text{May} & \text{Wf} \\
\hline
0 & \{\} & \{\} & \{\} & \{\} & \{\} & \{\} \\
1 & \{\} & \{\} & \{\} & \{\} & \{\} & \{\} \\
2 & \{B\} & \{B\} & \{\} & \{B\} & \{B\} & \{\} \\
3 & \{B\} & \{B\} & \{\} & \{B\} & \{B\} & \{\} \\
4 & \{A,B\} & \{A,B\} & \{\} & \{A,B\} & \{A,B\} & \{A,B\} \\
\hline
\end{array}
\]
main

0:

1: any (a=B.f)

12: any (a=B.f)

13: return

14:

B.<clinit>

2:

3: any (get A.f)

9: put B.g

10: return

11:

A.<clinit>

4:

5: any (new A)

6: put A.f

7: return

8:

\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
 & \multicolumn{3}{c|}{A_{in}(l)} & \multicolumn{3}{c|}{A_{out}(l)} \\
 & Must & May & Wf & Must & May & Wf \\
\hline
0 & {} & {} & {} & {} & {} & {} \\
1 & {} & {} & {} & {} & {} & {} \\
2 & \{B\} & \{B\} & {} & \{B\} & \{B\} & {} \\
3 & \{B\} & \{B\} & {} & {} & {} & {} \\
4 & \{A,B\} & \{A,B\} & {} & \{A,B\} & \{A,B\} & \{A,B\} \\
\hline
\end{tabular}
main

0:
1: any (a=B.f)
12: any (a=B.f)
13: return
14:

B.<clinit>

2:
3: any (get A.f)
9: put B.g
10: return
11:

A.<clinit>

4:
5: any (new A)
6: put A.f
7: return
8:

\[
\begin{array}{|c|c|c|c|c|c|c|}
\hline
\text{A}_{in}(l) & \text{A}_{out}(l) \\
\text{Must} & \text{May} & \text{Wf} & \text{Must} & \text{May} & \text{Wf} \\
\hline
0 & \{\} & \{\} & \{\} & \{\} & \{\} & \{\} \\
1 & \{\} & \{\} & \{\} & \{\} & \{\} & \{\} \\
2 & \{B\} & \{B\} & \{\} & \{B\} & \{B\} & \{\} \\
\hline
\end{array}
\]
main
0:
1: any (a=B.f)
12: any (a=B.f)
13: return
14:

B.<clinit>
2:
3: any (get A.f)
9: put B.g
10: return
11:

A.<clinit>
4:
5: any (new A)
6: put A.f
7: return
8:

<table>
<thead>
<tr>
<th></th>
<th>$A_{in}(l)$</th>
<th></th>
<th>$A_{out}(l)$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Must</td>
<td>May</td>
<td>Wf</td>
</tr>
<tr>
<td>0</td>
<td>{}</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>1</td>
<td>{}</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>2</td>
<td>{B}</td>
<td>{B}</td>
<td>{}</td>
</tr>
<tr>
<td>3</td>
<td>{B}</td>
<td>{B}</td>
<td>{}</td>
</tr>
</tbody>
</table>
main

0:

1: any (a=B.f)

12: any (a=B.f)

13: return

14:

B.<clinit>

2:

3: any (get A.f)

9: put B.g

10: return

11:

A.<clinit>

4:

5: any (new A)

6: put A.f

7: return

8:

\begin{table}
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
 & \(A_{in}(l)\) & & \(A_{out}(l)\) & \\
 & \text{Must} & \text{May} & \text{Wf} & \text{Must} & \text{May} & \text{Wf} \\
\hline
0 & \{\} & \{\} & \{\} & \{\} & \{\} & \{\} \\
1 & \{\} & \{\} & \{\} & \{\} & \{\} & \{\} \\
2 & \{B\} & \{B\} & \{\} & \{B\} & \{B\} & \{\} \\
3 & \{B\} & \{B\} & \{\} & \{B\} & \{B\} & \{\} \\
4 & \{A,B\} & \{A,B\} & \{\} & \{A,B\} & \{A,B\} & \{A,B\} \\
\hline
\end{tabular}
\end{table}
### `main`

- **0:**
- **1:** any \(a = B.f\)
- **12:** any \(a = B.f\)
- **13:** return
- **14:**

### `B.<clinit>`

- **2:**
- **3:** any (get \(A.f\))
- **9:** put \(B.g\)
- **10:** return
- **11:**

### `A.<clinit>`

- **4:**
- **5:** any (new \(A\))
- **6:** put \(A.f\)
- **7:** return
- **8:**

### Tables

<table>
<thead>
<tr>
<th></th>
<th><code>Ain(l)</code></th>
<th><code>Aout(l)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Must</td>
<td>May</td>
</tr>
<tr>
<td>5</td>
<td>{A,B}</td>
<td>{A,B}</td>
</tr>
</tbody>
</table>
main
0:
1: any (a=B.f)
12: any (a=B.f)
13: return
14:

B.<clinit>
2:
3: any (get A.f)
9: put B.g
10: return
11:

A.<clinit>
4:
5: any (new A)
6: put A.f
7: return
8:

<table>
<thead>
<tr>
<th></th>
<th>Ain(l)</th>
<th></th>
<th>Aout(l)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Must</td>
<td>May</td>
<td>Wf</td>
</tr>
<tr>
<td>5</td>
<td>{A,B}</td>
<td>{A,B}</td>
<td>{}</td>
</tr>
<tr>
<td>6</td>
<td>{A,B}</td>
<td>{A,B}</td>
<td>{}</td>
</tr>
</tbody>
</table>
since A is not in May, we are sure that A.<clinit> is called
Formalization

Data-flow equations

\[ F^{\text{init}} \in \mathbb{P} \times A^{\#} \rightarrow A^{\&} \]
Formalization

Data-flow equations

$$A_{in}(l) \xrightarrow{F_{init}(l)} A^\text{init}$$

$$A_{out}(l) = \left[\left[instr(l)\right]\right]^\sharp(F_{init}(l, A_{in}(l)))$$

if $instr(l) \neq \text{invoke}$ then
Formalization

\[ F^{\text{init}}(l, A_{\text{in}}(l)) \]

- if \( \text{flow}_{\text{clinit}}(l) = C \) and \( C \not\in \text{May} \)
  \[ [A_{\text{in}}(C.<\text{clinit}>.\text{last})]^\#(A_{\text{in}}(l)) \]
Formalization

\[ F^{\text{init}}(l, A_{\text{in}}(l)) \]

- if \( \text{flow}_{\text{clinit}}(l) = C \) and \( C \not\in \text{May} \)
  \[ [A_{\text{in}}(C.<\text{clinit}>.\text{last})]^\#(A_{\text{in}}(l)) \]

- \[ \text{C.<clinit> will be called} \]
Formalization

\[ F^{\text{init}}(\ell, A_{\text{in}}(\ell)) \]

- if \( \text{flow}_{\text{clinit}}(\ell) = C \) and \( C \not\in \text{May} \)
  \[ \llbracket A_{\text{in}}(C.\text{<clinit>}) \rrbracket \]
  \( C.\text{<clinit>} \) may be called

- if \( \text{flow}_{\text{clinit}}(\ell) = C \) and \( C \in \text{May} \setminus \text{Must} \)
  \[ \llbracket A_{\text{in}}(C.\text{<clinit>}.\text{last}) \rrbracket \uparrow (A_{\text{in}}(\ell)) \sqcup A_{\text{in}}(\ell) \]
Formalization

\[ F^{\text{init}}(I, A_{\text{in}}(I)) \]

- if \( \text{flow}_{\text{clinit}}(I) = C \) and \( C \notin \text{May} \)
  \[ [A_{\text{in}}(C.<\text{clinit}>.last)]^2(A_{\text{in}}(I)) \]

- if \( \text{flow}_{\text{clinit}}(I) = C \) and \( C \in \text{May} \setminus \text{Must} \)
  \[ [A_{\text{in}}(C.<\text{clinit}>)]^2(A_{\text{in}}(I)) \]

- otherwise
  \[ A_{\text{in}}(I) \]

*c.<clinit> will not be called*
Correctness

Soundness Theorem

If \((A_{in}, A_{out})\) is a data flow solution then for all reachable states \(<i, cs, s, h> \in \lbrack p\rbrack\), \(A_{in}(i) \sim (s, h)\) holds.

Correctness relation

\[(Must, May, Wf) \sim (s, h)\]

\[\iff\]

\[Must \subseteq h \subseteq May\] and \(Wf \subseteq \{f \in F | s(f) \neq \text{undef}\}\)
Our Work

- Issue about class initialization
- Data-flow analysis
  - to improve the precision of the CFG (when it includes calls to class initializers)
  - to infer a set of definitely initialized static fields for each program point
- Soundness theorem w.r.t. a formal semantics
- In the paper: applications to
  - bug detection
  - precision improvement of a null-pointer analysis
Conclusion

Related Work

- Eager Class Initialization [KOZEN02]
- Instance Initialization [FÄHNDRICHT03],[QI09],[UNKEL07]
- Dynamic Semantics [BORGER98], [DEBBAB102]
- Dynamic Analysis [HIRZEL04]
- Def-use [HARROLD94]
Conclusion

Future Work

* Limites of the formalization
  * Hierarchy not taken in account
  * Indeterminism with interfaces
* Implementation is on-going work
* Memory issues
* Standard RTA [BACON96]
  * not precise enough (flow\text{inter})
* Exceptions introduce loads of potential flow egdes
Thank you