Inference of Field Initialization

Fausto Spoto and Michael D. Ernst

University of Verona, Italy & University of Washington, USA

Honolulu, May 25, 2011, ICSE
public class MyWindow extends JWindow {
    private final String name; // never null
    private final static Map<String, MyWindow> map = new Hashtable<String, MyWindow>();
    public MyWindow(String name) {
        this.name = name;
        setVisible(true);
    }
    public static void main(String[] args) {
        new MyWindow("first");
        new MyWindow("second");
    }
    @Override
    protected void windowInit() {
        super.windowInit();
        map.put(name, this);
    }
}
Clever tracking of windows by name (from a real story)

```java
public class MyWindow extends JWindow {
    private final String name; // never null
    private final static Map<String, MyWindow> map = new Hashtable<String, MyWindow>();
    public MyWindow(String name) {
        this.name = name;
        setVisible(true);
    }
    public static void main(String[] args) {
        new MyWindow("first");
        new MyWindow("second");
    }
    @Override
    protected void windowInit() {
        super.windowInit();
        map.put(name, this);
    }
}
```
public class MyWindow extends JWindow {
    private final String name; // never null
    private final static Map<String, MyWindow> map = new Hashtable<String, MyWindow>();
    public MyWindow(String name) {
        this.name = name;
        setVisible(true);
    }
    public static void main(String[] args) {
        new MyWindow("first");
        new MyWindow("second");
    }
    @Override
    protected void windowInit() {
        super.windowInit();
        map.put(name, this);
    }
}
public class MyWindow extends JWindow {
    private final String name; // never null
    private final static Map<String, MyWindow> map = new Hashtable<String, MyWindow>();
    public MyWindow(String name) {
        this.name = name;
        setVisible(true);
    }
    public static void main(String[] args) {
        new MyWindow("first");
        new MyWindow("second");
    }
    @Override
    protected void windowInit() {
        super.windowInit();
        map.put(name, this);
    }
}
public class MyWindow extends JWindow {
  private final String name; // never null
  private final static Map<String, MyWindow> map = new Hashtable<String, MyWindow>();
  public MyWindow(String name) {
    this.name = name;
    setVisible(true);
  }
  public static void main(String[] args) {
    new MyWindow("first");
    new MyWindow("second");
  }
  @Override
  protected void windowInit() {
    super.windowInit();
    map.put(name, this);
  }
}
public class MyWindow extends JWindow {
    private final String name; // never null
    private final static Map<String, MyWindow> map = new Hashtable<String, MyWindow> ();
    public MyWindow(String name) {
        this.name = name;
        setVisible(true);
    }
    public static void main(String[] args) {
        new MyWindow("first");
        new MyWindow("second");
    }
    @Override
    protected void windowInit() {
        super.windowInit();
        map.put(name, this);
    }
}
public class MyWindow extends JWindow {
    private final String name; // never null
    private final static Map<String, MyWindow>
        map = new Hashtable<String, MyWindow>();
    public MyWindow(String name) {
        this.name = name;
        setVisible(true);
    }
    public static void main(String[] args) {
        new MyWindow("first");
        new MyWindow("second");
    }
    @Override
    protected void windowInit() {
        super.windowInit();
        map.put(name, this);
    }
}

Execution trace
NullPointerException
at Hashtable.put()
at MyWindow.windowInit()
at JWindow.<init>()
at MyWindow.<init>()
at MyWindow.main()
Clever tracking of windows by name (from a real story)

```java
public class MyWindow extends JWindow {
    private final String name; // never null
    private final static Map<String, MyWindow> map = new Hashtable<String, MyWindow>();
    public MyWindow(String name) {
        this.name = name;
        setVisible(true);
    }
    public static void main(String[] args) {
        new MyWindow("first");
        new MyWindow("second");
    }
    @Override
    protected void windowInit() {
        super.windowInit();
        map.put(name, this);
    }
}
```

Execution trace
```
NullPointerException
at Hashtable.put()
at MyWindow.windowInit()
at JWindow.<init>()
at MyWindow.<init>()
at MyWindow.main()
```
public class MyWindow extends JWindow {
    private final String name; // never null
    private final static Map<String, MyWindow> map = new Hashtable<String, MyWindow>();
    public MyWindow(String name) {
        this.name = name;
        setVisible(true);
    }
    public static void main(String[] args) {
        new MyWindow("first");
        new MyWindow("second");
    }
    @Override
    protected void windowInit() {
        super.windowInit();
        map.put(name, this);
    }
}
Clever tracking of windows by name (from a real story)

```java
public class MyWindow extends JWindow {
    private final String name; // never null
    private final static Map<String, MyWindow> map = new Hashtable<String, MyWindow>();

    public MyWindow(String name) {
        this.name = name;
        setVisible(true);
    }

    public static void main(String[] args) {
        new MyWindow("first");
        new MyWindow("second");
    }

    @Override
    protected void windowInit() {
        super.windowInit();
        map.put(name, this);
    }
}
```

Execution trace
- NullPointerException
  - at Hashtable.put()
  - at MyWindow.windowInit()
  - at JWindow.<init>()
  - at MyWindow.<init>()
  - at MyWindow.main()
The notion of *rawness*

**Definition (Raw object)**

An object is *raw wrt. fields* $F$ iff some field in $F$ is not initialized.

**Example**

Variable this is raw inside windowInit wrt. field name:

```java
@Override @Raw
protected void windowInit() {
    ... map.put(name, this);
}
```

Hence there is no guarantee that name is already initialized *there*.

Note: assigning null into a field makes it initialized
The notion of *rawness*

**Definition (Raw object)**

An object is *raw wrt. fields* $F$ iff some field in $F$ is not initialized.

**Example**

Variable `this` is raw inside `windowInit` wrt. field `name`:

```java
@Override @Raw
protected void windowInit() {
    ... map.put(name, this);
}
```

Hence there is no guarantee that `name` is already initialized *there*.

Note: assigning `null` into a field makes it initialized
Our goal: an automatic inference for initialization

1. define a concrete operational **semantics** of a Java-like language
2. define a constraint-based **abstract interpretation** of that semantics
3. prove them related by a correctness relation
4. use our abstract interpretation as an **inference engine** for initialization
5. measure its **precision** by using nullness analysis
   - but any other analysis could be used instead
Java bytecode as a graph of basic blocks

- a graph for each constructor or method
- explicit, inferred types
- resolved field and method references (through class analysis)
- explicit exception handlers
Bytecodes work over states

A state is a triple $\langle l \parallel s \parallel \mu \rangle$ of local variables, operand stack and heap, that binds locations to objects.

An object $o$ belongs to class $o.\kappa \in \mathbb{K}$ and maps field identifiers $f$ into $o.f$, which can be a value or uninit.

```java
this.name = name;
⇓
load 0 of type MyWindow
load 1 of type String
putfield MyWindow.name
```
Bytecodes work over states

A state is a triple $\langle l \parallel s \parallel \mu \rangle$ of local variables, operand stack and heap, that binds locations to objects.

An object $o$ belongs to class $o.\kappa \in K$ and maps field identifiers $f$ into $o.f$, which can be a value or uninit.

```java
this.name = name;
⇓
load 0 of type MyWindow
load 1 of type String
putfield MyWindow.name
```
Bytecodes work over states

A state is a triple \( \langle l \parallel s \parallel \mu \rangle \) of local variables, operand stack and heap, that binds locations to objects.

An object \( o \) belongs to class \( o.\kappa \in \mathbb{K} \) and maps field identifiers \( f \) into \( o.f \), which can be a value or \texttt{uninit}.

\[\text{this.name} = \text{name};\]

\[\downarrow\]

load 0 of type \texttt{MyWindow}

load 1 of type \texttt{String}

putfield \texttt{MyWindow.name}
Formalisation of state transformations

**load i of type t**

\[ \langle l \| s \| \mu \rangle \Rightarrow \langle l \| l[i]::s \| \mu \rangle \]

**putfield \( \kappa.f \)**

\[ \langle l \| top::rec::s \| \mu \rangle \Rightarrow \langle l \| s \| \mu[\mu(rec).f \mapsto top] \rangle \text{ if } \text{rec} \neq \text{null} \]

**new \( \kappa \) (\( \ell \) is fresh, all reference fields in \( o \) contain uninit)**

\[ \lambda \langle l \| s \| \mu \rangle \Rightarrow \langle l \| \ell::s \| \mu[\ell \mapsto o] \rangle \text{ if there is enough memory} \]

We define an operational semantics over an activation record of states (see the paper for details).
From concrete to abstract

Concrete
We have a concrete notion of states and of state transformers
- concrete states store locations, integers, everything
- we have seen an execution of three bytecodes in sequence

Abstract
We are going to define abstract states and state transformers
- abstract states store the sets of uninitialized fields, only
- we will see the same execution over this abstraction

Abstract Interpretation
- we will define this abstraction systematically
- and link concrete and abstract with a correctness result
Our abstraction of the concrete states

Abstraction of $\langle [l_0 \ldots l_{i-1}] \| s_{j-1} : \cdots : s_0 \| \mu \rangle$

- variable-wise: $\langle [l_0^\alpha \ldots l_p^\alpha] \| s_q^\alpha : \cdots : s_0^\alpha \| f_1^\alpha \ldots f_r^\alpha \rangle$

- $l_k^\alpha = \begin{cases} \emptyset & \text{if } l_k \in \mathbb{Z} \cup \{\text{null}\} \\ \{ f \mid \mu(l_k).f = \text{uninit} \} & \text{if } l_k \in \mathbb{L} \end{cases}$

- $s_k^\alpha = \begin{cases} \emptyset & \text{if } s_k \in \mathbb{Z} \cup \{\text{null}\} \\ \{ f \mid \mu(s_k).f = \text{uninit} \} & \text{if } s_k \in \mathbb{L} \end{cases}$

- $f_k^\alpha = \begin{cases} \emptyset & \text{if } f_k \text{ has primitive type} \\ \{ f \mid \text{there exists } l \in \mathbb{L} \text{ s.t. } \mu(\mu(l).f_k).f = \text{uninit} \} & \text{if } f_k \text{ has reference type} \end{cases}$
Example of abstract execution

load 0 of type MyWindow

locals stack
< [this,name], this, μ>

String

MyWindow

char[]

value

offset 0

count 12

uninit

name

⟨[[{name}, ∅] ∥ {name}] ∥ ∅, ∅⟩

locals

stack

name value

fields
Example of abstract execution

load 1 of type String

locals
stack

< [this,name], this ::name, μ >

String

MyWindow

char[]

value

name

uninit

textual representation:

⟨

[[{name}, ∅] ∥ {name}, ∅] ∥ ∅, ∅⟩

locals
stack
name
value
fields
Example of abstract execution

```java
putfield MyWindow.name
```

locals stack

< [this,name], ε, μ >

MyWindow String char[]

String

name value

MyWindow

char[]

offset 0 count 12

⟨ [∅, ∅] || ε || ∅, ∅ ⟩

locals stack name value fields

From program code to an abstract graph

nodes stand for local variables, stack elements, fields...
nodes contain a set of non-initialized fields
From program code to an abstract graph

arcs propagate those sets from source to sink (set inclusion)
Nodes contain fields not yet initialized, for that local variable or stack element

Arcs propagate those fields from source to sink

{point p}
new C
{point q}
Propagation of uninitialized fields

\{\text{point p}\}
\text{const v}
\{\text{point q}\}

Nodes contain fields not yet initialized, for that local variable or stack element

Arcs propagate those fields from source to sink
{point p} 
load k of type t 
{point q}

Nodes contain fields not yet initialized, for that local variable or stack element

Arcs propagate those fields from source to sink
Propagation of uninitialized fields

{point p}
store k of type t
{point q}

Nodes contain fields not yet initialized, for that local variable or stack element

Arcs propagate those fields from source to sink
Propagation of uninitialized fields

Fields are approximated in a context insensitive way.

```
{point p}
getfield f
{point q}
```
Propagation of uninitialized fields

{point p}
putfield f
{point q}

If local \( l_k \) is a definite alias of the stack element \( s_{j-1} \) at \( p \).

There might be more definite aliases: all are considered.
Interprocedural analysis: return

\{\text{point p}\}
\text{return type}
\{\text{point q}\}

A simpler rule applies when there is no returned value.
Interprocedural analysis: \textbf{call}

\{point p\}
\textbf{call} m
\{point q\}

- \textbf{caller's parameters}
- \textbf{callee's parameters}

- program point p
- program point q

- fields uninitialized in the returned values of m

- returned value
- \texttt{ret@m}

- stack:
  - li
  - s0
  - s(j-k)

- locals:
  - l0
  - l0
  - l(k-1)

- stack locals:
  - li
  - s0
  - s(j-k)

- program point p
- program point q

- fields uninitialized in the returned values of m

- returned value
- \texttt{ret@m}

- stack:
  - li
  - s0
  - s(j-k)

- locals:
  - l0
  - l0
  - l(k-1)
The previous graph construction rules are applied for any $p$ and for every intraprocedural successor $q$ of $p$

A single $p$ may have zero, one or more successors $q$
Putting everything together

- The previous graph construction rules are applied for any $p$ and for every intraprocedural successor $q$ of $p$
- A single $p$ may have zero, one or more successors $q$

```java
load 0 of type MyWindow

call javax.swing.JWindow.<init>()::void [public javax.swing.JWindow.<init>()::void]

load 0 of type MyWindow
load 1 of type java.lang.String

putfield MyWindow.name:java.lang.String [private final MyWindow.name:java.lang.String]
load 0 of type MyWindow
const 1

call MyWindow.setVisible(boolean)::void [public java.awt.Window.setVisible(boolean)::void]
return void

catch
throw java.lang.Throwable

return void
```
The previous graph construction rules are applied for any $p$ and for every intraprocedural successor $q$ of $p$.

A single $p$ may have zero, one or more successors $q$. 
The actual graph construction is more complex
- Exceptions
- Propagation of side-effects
- Optimizations: most nodes are collapsed when they definitely have the same approximation
Correctness of the analysis

Solution of the graph
- A solution is a set of non-initialized fields for each node.
- Arcs stand for set inclusion.
- Arcs labeled with $\neg f$ stand for inclusion of everything but $f$.
- A minimal solution can be computed through a fixpoint engine.

Correctness
For each program point $p$, every time the operational semantics reaches $p$ in a state $\langle [l_0 \ldots l_{i-1}] \parallel s_{j-1} :: \cdots :: s_0 \parallel \mu \rangle$, we have that
- Each $l_i^\alpha$ is included in the solution of node $l_i$ at $p$.
- Each $s_j^\alpha$ is included in the solution of node $s_j$ at $p$.
- Each $f_k^\alpha$ is included in the solution of node $f_k$. 
Definition (Raw object, reminder)

An object is raw wrt. fields \( F \) iff some field in \( F \) is not initialized.

We can use our analysis to annotate each program variable \( v \) that might hold raw objects (w.r.t. \( F \)):
- build the graph
- find its minimal solution
- consider the approximation of the node for \( v \)
- if it intersects \( F \), then it gets annotated as \(@\text{Raw}@\)
- by correctness of the approximation, this annotation is correct
Experiments: integration Julia/Checker Framework

Julia

An inference engine of Java program properties based on abstract interpretation

- nullness and rawness analysis are distinct analyses
  - Julia performs nullness analysis and infers a set of non-null fields $F$
  - then it performs initialization analysis and builds the @Raw annotations wrt. $F$
Experiments: integration Julia/Checker Framework

Julia

An inference engine of Java program properties based on abstract interpretation

jaif file

The jaif file contains nullness (@Nullable, @NonNull, @PolyNull) and initialization (@Raw) annotations of the program under analysis.

The Checker Framework

A generic type-checker for Java program properties based on annotation types
## Experiments: a cheap analysis

<table>
<thead>
<tr>
<th>program</th>
<th>size (lines)</th>
<th>time (sec.)</th>
<th>dereferences safe / all (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>total</td>
<td>init.</td>
<td></td>
</tr>
<tr>
<td>AFU</td>
<td>13892</td>
<td>209</td>
<td>2</td>
</tr>
<tr>
<td>JFlex</td>
<td>14987</td>
<td>118</td>
<td>2</td>
</tr>
<tr>
<td>plume</td>
<td>19652</td>
<td>321</td>
<td>2</td>
</tr>
<tr>
<td>Daikon</td>
<td>112077</td>
<td>2151</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>program</th>
<th>inferred annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>@NonNull/all (%)</td>
</tr>
<tr>
<td>AFU</td>
<td>649 / 854 (76.0)</td>
</tr>
<tr>
<td>JFlex</td>
<td>591 / 741 (79.8)</td>
</tr>
<tr>
<td>plume</td>
<td>675 / 912 (74.0)</td>
</tr>
<tr>
<td>Daikon</td>
<td>7145/10435 (68.5)</td>
</tr>
</tbody>
</table>
Experiments: comparison to Nit

A tool inferring nullness and initialization (one abstract domain)


- sound theory
- crashes on all tests
- we could run it on a subset of AFU
- no @Raw annotations for receivers, return, inner types
- output contained errors

<table>
<thead>
<tr>
<th>AFU</th>
<th>time (s.)</th>
<th>dereferences</th>
<th>inferred annotations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>tot.</td>
<td>init.</td>
<td>safe/all (%)</td>
</tr>
<tr>
<td>Julia</td>
<td>86</td>
<td>1</td>
<td>2683/2725 (98.5)</td>
</tr>
<tr>
<td>Nit</td>
<td>10</td>
<td>?</td>
<td>3145/3887 (80.9)</td>
</tr>
</tbody>
</table>
Experiments: comparison to JastAdd

A tool for type inference and checking

- no sound theory
- crashes on all tests but for JFlex
- does not deal with static fields
- imprecise: the receiver of a constructor is @Raw, always, also in helper functions

<table>
<thead>
<tr>
<th>JFlex</th>
<th>time (s.)</th>
<th>dereferences safe/all (%)</th>
<th>inferred annotations @NonNull/all (%)</th>
<th>inferred annotations @Raw/all (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Julia</td>
<td>118</td>
<td>8624/8753 (98.5)</td>
<td>591/741 (79.8)</td>
<td>3/1109 (0.3)</td>
</tr>
<tr>
<td>JastAdd</td>
<td>3</td>
<td>?/? (?)</td>
<td>389/? (?)</td>
<td>14/? (?)</td>
</tr>
</tbody>
</table>
Experiments: comparison to human-written annotations

- the plume library has a full manual annotation wrt. `@Nullable` and `@Raw`
  - 7 `@Raw` annotations, 3 `@Raw` warning suppressions
- the jaif file generated by Julia is different
  - 1 `@Raw` annotation only
  - the 6 extra are human errors: the developers removed them
  - the 3 warning suppressions are weaknesses in the type-checker
- main difference: rawness is binary for the type-checker, but not for Julia
- similar results for Daikon
an inference technique for field initialization
useful whenever a property of a field holds after its initialization
fully implemented and effective
  its results improve manual annotations
  or can be used as a starting point for manual annotation
proved correct through a graph-based abstract interpretation, not limited to initialization analysis:
  class analysis
  aliasing analysis
  full arrays/collections analysis

Julia: http://julia.scienze.univr.it
The Checker Framework:
http://types.cs.washington.edu/checker-framework