NIT: a non-null Annotation Inferencer for Java Bytecode

Laurent Hubert
CNRS - Rennes, France
laurent.hubert@irisa.fr

PASTE'08 - Atlanta, USA
Examples of annotations

class C extends A{
    @NN O f;

    C(){
        this.f = new O();
    }
}

static @NN O m(@NN C x){
    return x.f;
}
}
Motivations for nullness analysis

- Avoid NullPointerExceptions
  - if used by the developer
- Allow more optimizations
  - if used by the compiler to native code (e.g. JIT)
- Improve the precision of other analyses
  - if the other analyses rely on the control flow graph
Some nullness analyses

Bug-finding oriented analyses
1) Flanagan and Leino. FME’01.
2) Hovemeyer et al.. PASTE’05.
3) Hovemeyer and Pugh. PASTE’07.

Certification oriented analyses
• By checking annotations
  4) Fähndrich and Leino. OOPSLA’03.
  5) Male et al.. CC’08.

• By inference
  6) Hubert et al.. FMOODS’08.
In this talk

- Motivations
- The non-null annotation inference analysis
- Improvements to the former analysis for the bytecode
  - Alias analysis
  - New domain for instanceof
    - NullableInit abstract value
- Results of the implementations
Problem 1: all fields are null by default

class C extends A{
    @NN O f;

    C(){
        this.f = new O();
    }

    static @NN O m(@NN C x){
        return x.f;
    }
}
Problem 1: all fields are null by default

class C extends A{
  @NN O f;

  C()
      this.f = new O();
}

static @NN O m(@NN C x){
  return x.f;
}

Problem 1: all fields are null by default

class C extends A{
  @NN O f;

  C(){
    this.f = new O();
  }

  static @NN O m(@NN C x){
    return x.f;
  }
}
Problem 2: the object under construction can be accessed before the invariant is established.
Problem 2: the object under construction can be accessed before the invariant is established.

Object references are annotated with @Raw(A) if they may be in construction.
Automatic inference

- Annotating a program is not trivial

- Manual annotations
  - cannot guarantee correctness...
    - ...except with a checker
  - can be a burden on the programmer...
    - ...specially for legacy code
How does it work?

- Annotations are given a lattice structure
- The analysis is specified as data flow constraints
- The solution is a fixpoint

For more details, cf. Hubert et al.. FMOODS’08.
Turning theory into practice

From an intermediate language to the bytecode level

- The former analysis
  - Specified on an intermediate language
  - Focused on field annotation inference
  - Correctness proof checked with the Coq proof assistant

- The tool must be at the bytecode level
  - Exceptions, static fields and arrays are conservatively handled
  - How to handle `ifnull? instanceof`?
  - How to recover good information from nullness tests on Nullable values?
Handling ifnonnull instructions

- Problem: Java Bytecode is a stack language
- Solution: a simple must-alias analysis
Handling instanceof instructions

Why is it difficult?

- Java bytecode is a stack language
- The result of the instanceof instructions is an integer (0 if the reference is null, 0 or 1 otherwise)
- instanceof instructions are very often followed by a conditional, not always
Solution

We have added an abstract domain to represent the results of `instanceof` instructions.

We abstract a stack variable by a set of local variables that must be non-null if the stack variable is 1.
Handling instanceof instructions

```java
static int m_io(Object x) {
    if (x instanceof String) {
        return ((String)x).length();
    } else {
        return -1;
    }
}
```

```java
static int m_io(Object)

Code:
0: aload x
1: instanceof java/lang/String
4: ifeq  15
7: aload x
8: checkcast java/lang/String
11: invokevirtual String.length()I
14: ireturn
15: istore 0
16: iretd
```
# Handling instanceof Instructions

Stack variable ∈ Alias# x InstanceOf# x Nullness#

Alias# = Locals²
InstanceOf# = Locals²

```java
static int m_io(Object)
{
    Code:
    0:  aload x
    1:  instanceof java/lang/String
    4:  ifeq 15
    7:  aload x
    8:  checkcast java/lang/String
   11: invokevirtual String.length()I
   14:  ireturn
   15:  iconst_m1
   16:  ireturn
}
```

<table>
<thead>
<tr>
<th>Locals</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nullable</td>
<td>{x} Top Nullable</td>
</tr>
<tr>
<td>Nullable</td>
<td>Top {x} Top</td>
</tr>
<tr>
<td>NonNull</td>
<td>{x} Top NonNull</td>
</tr>
<tr>
<td>NonNull</td>
<td>{x} Top NonNull</td>
</tr>
<tr>
<td>NonNull</td>
<td>Top Top Top</td>
</tr>
<tr>
<td>Nullable</td>
<td>Top Top Top</td>
</tr>
</tbody>
</table>

Stack variable ∈ Alias# x InstanceOf# x Nullness#

Alias# = Locals²
InstanceOf# = Locals²
Efficiently using nullness tests
Efficiently using nullness tests

Former lattice structure

Improved lattice structure
NIT: Nullness Inference Tool

- Free and open source (GPL)
  http://nit.gforge.inria.fr

- Implements in OCaml the basic analysis and the improvements discussed herein while
  - lazily parsing the methods
  - carefully managing memory

- Usual restrictions apply to native methods and reflection is not handled
## Results

<table>
<thead>
<tr>
<th>Program size (methods)</th>
<th>Space (MB)</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Jess</strong></td>
<td>25 686</td>
<td>878</td>
</tr>
<tr>
<td><strong>Soot</strong></td>
<td>17 895</td>
<td>662</td>
</tr>
<tr>
<td><strong>ESC/Java</strong></td>
<td>10 845</td>
<td>428</td>
</tr>
<tr>
<td><strong>Julia</strong></td>
<td>11 248</td>
<td>414</td>
</tr>
<tr>
<td><strong>JavaCC</strong></td>
<td>8 098</td>
<td>319</td>
</tr>
<tr>
<td><strong>JDTCore</strong></td>
<td>4 246</td>
<td>341</td>
</tr>
<tr>
<td><strong>Jasmin</strong></td>
<td>4 316</td>
<td>118</td>
</tr>
<tr>
<td><strong>TightVNC</strong></td>
<td>3 445</td>
<td>89</td>
</tr>
<tr>
<td><strong>SPEC JVM98</strong></td>
<td><strong>sum</strong></td>
<td>33 503</td>
</tr>
<tr>
<td></td>
<td><strong>max</strong></td>
<td>3 818</td>
</tr>
</tbody>
</table>

Average of 200 methods per second
Results

% of non-null annotations out of the total number of annotations

- Basic: 41.2%
- +Ifnull: 48.3%
- +InstanceOf: 48.9%
- +NullableInit: 50.2%
Results

% of non-null annotations out of the total number of annotations

Chalin and James. ECOOP'07. “in Java programs, at least 2/3 of declarations of reference types are meant to be non-null, by design.”
Results

% of safe dereferences out of the total number of dereferences

- Basic: 71.6%
- +Ifnull: 79.3%
- +InstanceOf: 79.8%
- +NullableInit: 81.1%

*e.g.* in Soot, the final analysis removed 9,500 unsafe dereferences more than the basic analysis.
The improvements we propose allow to reduce by a third the number of unsafe dereferences.
Conclusion and Future work

- We enriched a provably sound analysis to handle Java bytecode: alias analysis, abstract domain for instanceof, NullableInit.
- We provide NIT, a scalable implementation (NIT analyses Soot (17 k.meth) within 2 min).
- Future work
  - Make the domain relational
    - cf. Spoto, SEFM’08. Tomorrow, Cape Town
  - Certify a checker in a proof assistant
  - Integrate a checker in the bytecode verifier?
Thank you

NIT can be downloaded at
http://nit.gforge.inria.fr

laurent.hubert@irisa.fr