The Semantic Gap in Java Programs

Presenter: Devin Coughlin

Evan Chang, Amer Diwan, Jeremy Siek

University of Colorado at Boulder

FRACTAL 2009

A Motivating Example

```
class AddressBook {
  Person people[];
  . . .
  Person findPerson(String name) {
    for (Person person : people) {
      if (person.getName().equals(name))
            return person;
    }
    return null;
  }
}
```

Suppose the VM notices most searches find people near the end of the array. Can it optimize to **search backwards** starting from the end of the array?

2

A Motivating Example

```
class AddressBook {
  Person people[];
  . . .
  Person findPerson(String name) {
    for (Person person : people) {
      if (person.getName().equals(name))
            return person;
    }
    return null;
  }
}
```

Suppose the VM notices most searches find people near the end of the array. Can it optimize to **search backwards** starting from the end of the array?

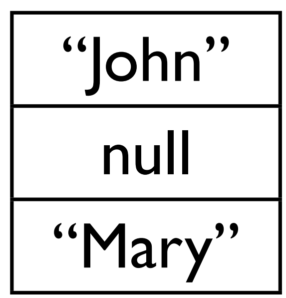
SURPISINGLY, NO

because `people[]` might contain null elements.

Null Pointer Exception

```
class AddressBook {
  Person people[];
  . . .
  Person findPerson(String name) {
    for (Person person : people) {
      if (person.getName().equals(name))
            return person;
    }
    return null;
  }
}
```

`findPerson("John")`



`people[]`

Null Pointer Exception

```
class AddressBook {
  Person people[];
  . . .
  Person findPerson(String name) {
    for (Person person : people) {
      if (person.getName().equals(name))
            return person;
    }
    return null;
  }
}
```

forwards:

`findPerson("John")`

"John"
null
"Mary"

`people[]`

3

Null Pointer Exception

```
class AddressBook {
  Person people[];
  . . .
  Person findPerson(String name) {
    for (Person person : people) {
      if (person.getName().equals(name))
            return person;
    }
    return null;
  }
}
```

forwards:

`findPerson("John")`

"John"
null
"Mary"

`people[]`

3

Null Pointer Exception

```
class AddressBook {
  Person people[];
  . . .
  Person findPerson(String name) {
    for (Person person : people) {
      if (person.getName().equals(name))
            return person;
    }
    return null;
  }
}
```

forwards:

`findPerson("John")`

"John"
null
"Mary"

`people[]`

3

Null Pointer Exception

```
class AddressBook {
  Person people[];
  . . .
  Person findPerson(String name) {
    for (Person person : people) {
      if (person.getName().equals(name))
            return person;
    }
    return null;
  }
}
```

forwards: succeeds

`findPerson("John")`

"John"
null
"Mary"

`people[]`

Null Pointer Exception

```
class AddressBook {
  Person people[];
  . . .
  Person findPerson(String name) {
    for (Person person : people) {
      if (person.getName().equals(name))
            return person;
    }
    return null;
  }
}
```

forwards: succeeds

`findPerson("John")`

"John"
null
"Mary"

`people[]`

backwards:

Null Pointer Exception

```
class AddressBook {
  Person people[];
  . . .
  Person findPerson(String name) {
    for (Person person : people) {
      if (person.getName().equals(name))
            return person;
    }
    return null;
  }
}
```

forwards: succeeds

`findPerson("John")`

"John"
null
"Mary"

`people[]`

backwards:

Null Pointer Exception

```
class AddressBook {
  Person people[];
  . . .
  Person findPerson(String name) {
    for (Person person : people) {
      if (person.getName().equals(name))
            return person;
    }
    return null;
  }
}
```

forwards: succeeds

`findPerson("John")`

"John"
null
"Mary"

`people[]`

backwards:

Null Pointer Exception

```
class AddressBook {
  Person people[];
  . . .
  Person findPerson(String name) {
    for (Person person : people) {
      if (person.getName().equals(name))
            return person;
    }
    return null;
  }
}
```

forwards: succeeds

`findPerson("John")`

"John"
null
"Mary"

`people[]`

backwards:

3

Null Pointer Exception

```
class AddressBook {
  Person people[];
  . . .
  Person findPerson(String name) {
    for (Person person : people) {
      if (person.getName().equals(name))
            return person;
    }
    return null;
  }
}
```

forwards: succeeds

`findPerson("John")`

“John”
null
“Mary”

`people[]`

backwards: **exception!**

Programmer vs. Compiler

```
class AddressBook {
  Person people[];
  . . .
  Person findPerson(String name) {
    for (Person person : people) {
      if (person.getName().equals(name))
            return person;
    }
    return null;
  }
}
```

Programmer: **`person`** is never null.

Compiler: Sorry, can't prove it.

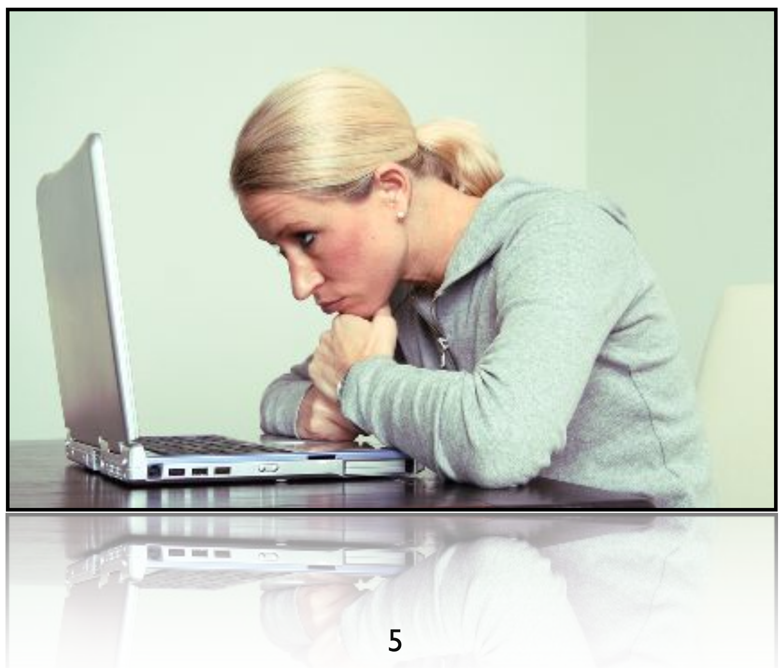
Programmer: No, really. **`person`** is never null.

Compiler: Sorry. And what about **`getName()`** and **`equals()`**? Either could throw an exception even if **`person`** isn't null.

Programmer: Grrr.

The Semantic Gap Defined

We call the mismatch between what the programmer knows/expects and what the language knows/expects the *semantic gap*.



Semantic Gap: Opportunity

The semantic gap *if it exists*, is an opportunity:

- For **language designers**
  - to create more productive languages
- For **compiler writers**
  - to design more effective optimizations

Hypothesis

the semantic gap exists in Java programs

Experimental Methodology

1. Assume **unit tests** and **benchmark** output validation runs are indicative of what **programmer expects**.
2. Identify possibly **over-strict** requirements in language specification.
3. **Observe** and **intervene** in unit test and benchmark runs to see if expectations match specification. If not, then semantic gap exists.

Roadmap

- The rest of this talk examines some potential semantic gaps in:
  - The Java Language
  - The Java Virtual Machine
  - The Java SE class libraries

Argument Evaluation Order

Java Language Specification (3rd Edition)

§15.7.4:

"Each argument expression appears to be fully evaluated before any part of any argument expression to its right."

"If evaluation of an argument expression completes abruptly, no part of any argument expression to its right appears to have been evaluated."

Method arguments must be evaluated left-to-right

Evaluation Order Can Matter

```
public void run() {
  int i = 0;
  print(i = 2, i); //note assignment
}
```

Left-to-Right produces: “2, 2”

Right-to-Left produces: “2, 0”

Behavior *may* depend on method argument evaluation order

Hypothesis

programmers do not actually rely on left-to-right method argument evaluation order

Experiment: Evaluation Order

Intervene: Permute method parameter order

Exhaustively permuted all method parameter orders using the RECODER framework.

Observe: Unit tests

Functional Analyzer

A program written by a member of our group that performs statistical analyses of time series data. 160 classes. ~10,000 LOC. Extensive unit tests.

Experiment: Evaluation Order

Intervene: Permute method parameter order

Exhaustively permuted all method parameter orders using the RECODER framework.

Observe: Unit tests

Functional Analyzer

A program written by a member of our group that performs statistical analyses of time series data. 160 classes. ~10,000 LOC. Extensive unit tests.

Result: Found **0** methods where argument evaluation order mattered.

Precise Exceptions

Java Virtual Machine Specification (2nd Edition)

§2.6.2: Handling an Exception

"All exceptions in the Java programming language are precise: when the transfer of control takes place, all effects of the statements executed and expressions evaluated before the point from which the exception is thrown must appear to have taken place. No expressions, statements, or parts thereof that occur after the point from which the exception is thrown may appear to have been evaluated. If optimized code has speculatively executed some of the expressions or statements which follow the point at which the exception occurs, such code must be prepared to hide this speculative execution from the user-visible state of the program."

No funny business across potentially excepting instructions

Precision Inhibits Optimization

Optimizations must be able to **restore programmer visible state** on exceptions.

Original Code

```
void average(int data[]) {
  double total = 0.0;
  int i = 1;
  try {
    while (1) {
      total += data[i - 1];
      i++;
    }
  } catch (Exception e) {
      //ignored
  }
  return total/i;
}
```

Optimized Code

```
void average(int data[]) {
  double total = 0.0;
  int i = 0;
  try {
    while (1) {
      total += data[i];
      i++;
    }
  } catch (Exception e) {
      i++; //FIXUP
  }
  return total/i;
}
```

Hypothesis

programmers rarely rely on precise exceptions

Experiment: Precise Exceptions

Observe: Instrument benchmark validation runs to see how exceptions are actually used.

Instrument DaCapo benchmarks with BCEL and java.lang.Instrument

We looked at:

- ArrayOutOfBoundsExceptions
- ClassCastExceptions

ArrayOutOfBounds: Precise

Instrumented all of DaCapo (except `fop`).

The only benchmark that threw ArrayOutOfBoundsExceptions was `eclipse`.

> Reads off the end of a character array in org.eclipse.jdt.internal.compiler.parser.Scanner.getNextToken()

`eclipse` uses these exceptions for fairly complex control flow to handle loading more data.

ArrayOutOfBoundsExceptions must sometimes be precise

ClassCastException: Imprecise

Instrumented all **`checkcast`** instructions in DaCapo.

No benchmarks ever threw a ClassCastException.

Perhaps ArrayOutOfBoundsExceptions can be imprecise

Summary of Precision Results

Some exceptions need to be precise.

Some do not.

20

Iteration Order

AbstractList (Java Platform SE 6) Documentation

public Iterator<E> iterator()

“Returns an iterator over the elements in this list in proper sequence.”

```
class AddressBook {
  Person people[];
  . . .
  Person findPerson(String name) {
    for (Person person : people) {
      if (person.getName().equals(name))
            return person;
    }
    return null;
  }
}
```

To be truly safe, iterators must be traversed in their *natural order*.

Hypothesis

programmers frequently do not rely on natural iteration order

22

Experiment: Iteration Order

Intervene: Reverse method iteration order with RECODER framework

Observe: Benchmark validation correctness

`bloat`

The only DaCapo benchmark that uses java.util.Iterator and on which we could run RECODER.

Experiment: Iteration Order

Intervene: Reverse method iteration order with RECODER framework

Observe: Benchmark validation correctness

`bloat`

The only DaCapo benchmark that uses java.util.Iterator and on which we could run RECODER.

Result: 85% of iterator invocations (that are actually used) can be safely reversed.

23

Conclusion

- There *is* a semantic gap in Java for:
  - method argument evaluation order
  - precise exceptions
  - iteration order

24

Open Questions

- Can we exploit the semantic gap for
  - **optimization?**
    - evaluate method arguments speculatively or in parallel?
    - execute for loops speculatively or in parallel?
  - **language design?**
    - opt in to precise exceptions?
    - parallel foreach