Javarifier:
inference of reference immutability

Jaime Quinonez
Matthew S. Tschantz
Michael D. Ernst
MIT
Security code in JDK 1.1

class Class {

    private Object[] signers;

    Object[] getSigners() {
        return signers;
    }
}
}
Security code in JDK 1.1

class Class {
    private Object[] signers;
    
    Object[] getSigners() {
        return signers;
    }
}

myClass.getSigners()[0] = "Sun";
Immutability annotations prevent mutation errors

class Class {

    private Object[] signers;
    // Prohibits client from mutating
    @Readonly Object[] getSigners() {
        return signers;
    }
}

myClass.getSigners()[0] = "Sun"; // Error
Immutability annotations prevent mutation errors

class Class {
    // Forces getSigners to return a copy
    private @Readonly Object[] signers;

    Object[] getSigners() {
        return signers; // Error
    }
}

myClass.getSigners()[0] = "Sun";
Reasoning about side effects

• Machine-checked formal documentation
• Error detection
• Verification
• Enables analyses
• Enables transformations and optimizations
• Case studies: expressive, natural, useful
  – 300 KLOC of programmer-written code
Type-checking requires annotations

• Easy for new programs
• Tedious for legacy programs
• Worst for libraries: large, hard to understand
• Library annotations cannot be omitted

// Assume user program is fully annotated

@Readonly MyClass x = ...;
x.toString();       // OK
x.mutate();        // Error
System.out.println(x); // False error!

• Library declares println as:
  void println(Object) { ... }
• But it should be:
  void println(@Readonly Object) { ... }
Immutability inference algorithm

- Sound
- Precise (complete)
- Linear time
- Rich, practical type system: Javari [Tschantz 2005]
- Context-sensitive
  - Type polymorphism (Java generics and wildcards)
  - Mutability polymorphism (type qualifier polymorphism)
  - Containing-object context (deep immutability)
- Infers abstract state
- Handles partially-annotated code, unanalyzable code
Immutability inference implementation

• Implements the full algorithm
• Handles all of Java
• Works on bytecodes
• Inserts results in source or .class file
• Scales to >100 KLOC
• Verified by independent typechecker
• Verified by comparison to other tools
• Publicly available:
  – http://pag.csail.mit.edu/javari/javarifier/
Javari type system

- A given reference cannot be used to modify its referent
  - Other references to the object may modify it
- **Deep**: the transitively reachable state (the abstract state) is protected
- **Generics**: `List<@Readonly Date>`
- **Mutability polymorphism**: 
  ```java
  @Polyread Object id(@Polyread arg) { return arg; }
  ```
Algorithm: propagate mutability

• Syntax-directed constraint generation
• Unconditional constraint: \texttt{x}
  – \texttt{x} is mutable
  – Example code: \texttt{x.field = 22;}
• Conditional constraint: \texttt{y \rightarrow z}
  – if \texttt{y} is mutable, then \texttt{z} is mutable
  – Example code: \texttt{y = z;}
• Constraint solving: graph reachability
  – Unmarked references remain readonly
class Appt {
    private Date d;
    void setDate(Appt this, Date newDate) {
        this.d = newDate;
    }
    Date getDate(Appt this) {
        Date result = this.d;
        return result;
    }
    void reset(Appt this) {
        Date thisDate = this.d;
        setTime(thisDate, 0);
    }
}

class Date {
    private long time;
    ...
    void setTime(Date this, long newTime) {
        this.time = newTime;
    }
}
class Appt {
    private Date d;
    void setDate(Appt this, Date newDate) {
        this.d = newDate;
    }
    Date getDate(Appt this) {
        Date result = this.d;
        return result;
    }
    void reset(Appt this) {
        Date thisDate = this.d;
        setTime(thisDate, 0);
    }
}

class Date {
    private long time;
    ...
    void setTime(Date this, long newTime) {
        this.time = newTime;
    }
}

Sample code:   x.f = y;
Constraints:    { x,
                     f → y }

class Appt {
    private Date d;
    void setDate(Appt this, Date newDate) {
        this.d = newDate;
    }
    Date getDate(Appt this) {
        Date result = this.d;
        return result;
    }
    void reset(Appt this) {
        Date thisDate = this.d;
        setTime(thisDate, 0);
    }
}

class Date {
    private long time;
    ...
    void setTime(Date this, long newTime) {
        this.time = newTime;
    }
}

Sample code:  x.f = y;
Constraints:  { x, f → y }
Assignment

class Appt {
    private Date d;
    void setDate(Appt this, Date newDate) {
        this.d = newDate;
    }
    Date getDate(Appt this) {
        Date result = this.d;
        return result;
    }
    void reset(Appt this) {
        Date thisDate = this.d;
        setTime(thisDate, 0);
    }
}

class Date {
    private long time;
    ...
    void setTime(Date this, long newTime) {
        this.time = newTime;
    }
}
class Appt {
    private Date d;
    void setDate(Appt this, Date newDate) {
        this.d = newDate;
    }
    Date getDate(Appt this) {
        Date result = this.d;
        return result;
    }
    void reset(Appt this) {
        Date thisDate = this.d;
        setTime(thisDate, 0);
    }
}

class Date {
    private long time;
    ...
    void setTime(Date this, long newTime) {
        this.time = newTime;
    }
}
class Appt {
    private Date d;
    void setDate(Appt this, Date newDate) {
        this.d = newDate;
    }
    Date getDate(Appt this) {
        Date result = this.d;
        return result;
    }
    void reset(Appt this) {
        Date thisDate = this.d;
        setTime(thisDate, 0);
    }
}

class Date {
    private long time;
    ...
    void setTime(Date this, long newTime) {
        this.time = newTime;
    }
}

Sample code:  \(x = y.f;\)
Constraints:  \(\{ x \rightarrow f, \quad x \rightarrow y \}\)
Dereference

class Appt {
    private Date d;
    void setDate(Appt this, Date newDate) {
        this.d = newDate;
    }
    Date getDate(Appt this) {
        Date result = this.d;
        return result;
    }
    void reset(Appt this) {
        Date thisDate = this.d;
        setTime(thisDate, 0);
    }
}

class Date {
    private long time;
    ...
    void setTime(Date this, long newTime) {
        this.time = newTime;
    }
}

Sample code:  \( x = y.f; \)
Constraints:  \{ x \rightarrow f, \ x \rightarrow y \}
class Appt {
    private Date d;
    void setDate(Appt this, Date newDate) {
        this.d = newDate;
    }
    Date getDate(Appt this) {
        Date result = this.d;
        return result;
    }
    void reset(Appt this) {
        Date thisDate = this.d;
        setTime(thisDate, 0);
    }
}

class Date {
    private long time;
    ...
    void setTime(Date this, long newTime) {
        this.time = newTime;
    }
}
class Appt {
    private Date d;
    void setDate(Appt this, Date newDate) {
        this.d = newDate;
    }
    Date getDate(Appt this) {
        Date result = this.d;
        return result;
    }
    void reset(Appt this) {
        Date thisDate = this.d;
        setTime(thisDate, 0);
    }
}

Sample code: 
\[ x = y.f; \]
Constraints: 
\[ \{ x \rightarrow f, \quad x \rightarrow y \} \]

class Date {
    private long time;
    ... 
    void setTime(Date this, long newTime) {
        this.time = newTime;
    }
}
class Appt {
    private Date d;
    void setDate(Appt this, Date newDate) {
        this.d = newDate;
    }
    Date getDate(Appt this) {
        Date result = this.d;
        return result;
    }
    void reset(Appt this) {
        Date thisDate = this.d;
        setTime(thisDate, 0);
    }
}

class Date {
    private long time;
    ...
    void setTime(Date this, long newTime) {
        this.time = newTime;
    }
}
class Appt {
    private Date d;
    void setDate(Appt this, Date newDate) {
        this.d = newDate;
    }
    Date getDate(Appt this) {
        Date result = this.d;
        return result;
    }
    void reset(Appt this) {
        Date thisDate = this.d;
        setTime(thisDate, 0);
    }
}

class Date {
    private long time;
    ...
    void setTime(Date this, long newTime) {
        this.time = newTime;
    }
class Appt {
    private Date d;
    void setDate(Appt this, Date newDate) {
        this.d = newDate;
    }
    Date getDate(Appt this) {
        Date result = this.d;
        return result;
    }
    void reset(Appt this) {
        Date thisDate = this.d;
        setTime(thisDate, 0);
    }
}

class Date {
    private long time;
    ...
    void setTime(Date this, long newTime) {
        this.time = newTime;
    }
}
class Appt {
  private Date d;
  void setDate(Appt this, Date newDate) {
    this.d = newDate;
  }

  Date getDate(Appt this) {
    Date result = this.d;
    return result;
  }

  void reset(Appt this) {
    Date thisDate = this.d;
    setTime(thisDate, 0);
  }
}

class Date {
  private long time;

  void setTime(Date this, long newTime) {
    this.time = newTime;
  }
}
Propagate mutability via graph reachability

class Appt {
    private Date d;
    void setDate(Appt this, Date newDate) {
        this.d = newDate;
    }
    Date getDate(Appt this) {
        Date result = this.d;
        return result;
    }
    void reset(Appt this) {
        Date thisDate = this.d;
        setTime(thisDate, 0);
    }
}

class Date {
    private long time;
    ...
    void setTime(Date this, long newTime) {
        this.time = newTime;
    }
}
class Appt {
    private Date d;
    void setDate(Appt this, Date newDate) {
        this.d = newDate;
    }
    Date getDate(Appt this) {
        Date result = this.d;
        return result;
    }
    void reset(Appt this) {
        Date thisDate = this.d;
        setTime(thisDate, 0);
    }
}

class Date {
    private long time;
    ... 
    void setTime(Date this, long newTime) {
        this.time = newTime;
    }
}

Propagate mutability via graph reachability
Propagate mutability via graph reachability

class Appt {
    private Date d;
    void setDate(Appt this, Date newDate) {
        this.d = newDate;
    }
    Date getDate(Appt this) {
        Date result = this.d;
        return result;
    }
    void reset(Appt this) {
        Date thisDate = this.d;
        setTime(thisDate, 0);
    }
}

class Date {
    private long time;
    ...
    void setTime(Date this, long newTime) {
        this.time = newTime;
    }
}
class Appt {
    private Date d;

    void setDate(Appt this, Date newDate) {
        this.d = newDate;
    }

    Date getDate(Appt this) {
        Date result = this.d;
        return result;
    }

    void reset(Appt this) {
        Date thisDate = this.d;
        setTime(thisDate, 0);
    }
}

class Date {
    private long time;

    void setTime(Date this, long newTime) {
        this.time = newTime;
    }
}
class Appt {
    private Date d;
    void setDate(Appt this, Date newDate) {
        this.d = newDate;
    }
    @Readonly Date getDate(@Readonly Appt this) {
        @Readonly Date result = this.d;
        return result;
    }
    void reset(Appt this) {
        Date thisDate = this.d;
        setTime(thisDate, 0);
    }
}

class Date {
    private long time;
    ...
    void setTime(Date this, long newTime) {
        this.time = newTime;
    }
}
Subtyping

• Add new constraints for behavioral subtyping:
  – Contravariant parameter mutability
  – Covariant return mutability
• Preserves Java overriding and overloading
• Solve as usual
Annotations for un-analyzed code

• For native methods:
  – **Stub** annotations
  – Also useful for overriding inference results

• For libraries with unknown clients:
  – **Open-world** assumption
  – All public members are mutable (pessimistic)

• If whole program is available:
  – **Closed-world** assumption
  – Use existing clients (optimistic)

• Add constraints stemming from the annotations
• Solve as usual
Abstract state

• **User annotations** of fields not in the abstract state
  – Caches, logging, beneficial side effects
  – Change type rules to ignore permissible mutations

• **Infer** abstract state
  – Inconsistencies with user or library annotations
    ```java
    String toString() @Readonly {
        if (cache == null)
            cache = computeToString();
        return cache;
    }
    ```
  – Heuristics for common programming patterns
    • Requires user confirmation

Does not modify receiver

Modifies receiver
Arrays and generic classes

• Each *part* of the type gets a type constraint variable
  – 2 constraint variables for `List<Date>`
• Generated constraints may use subtyping
  • Array assignment rule:
    \[ x[z] = y : \{ x, \text{type}(y) <: \text{type}(x[z]) \} \]
  • Dereference rule:
    \[ x = y.f : \{ \text{type}(f) <: \text{type}(x), x \rightarrow y \} \]
• Simplify constraints via equivalences between
  – Subtyping
  – Type containment
  – Mutability constraints
• When only mutability constraints remain, solve them
Wildcards let a method accept more arguments

```java
printList(List lst) {
    for (int i=0; i<lst.size(); i++) {
        print(lst.get(i));
    }
}

printList(new List<Integer>());
printList(new List<Long>());
```
Wildcards
let a method accept more arguments

```java
printList(List lst) {
    for (int i=0; i<lst.size(); i++) {
        print(lst.get(i));
    }
}

printList(new List<Integer>());
printList(new List<Long>());
```
Wildcards

let a method accept more arguments

```java
printList(List<Number> lst) {
    for (int i=0; i<lst.size(); i++) {
        print(lst.get(i));
    }
}

printList(new List<Integer>());
printList(new List<Long>());
```

List<Number> is not a supertype of List<Integer>
Wildcards
let a method accept more arguments

```java
printList(List<? extends Number> lst) {
    for (int i=0; i<lst.size(); i++) {
        print(lst.get(i));
    }
}

printList(new List<Integer>());
printList(new List<Long>()));
```

lst may be of any List type whose element extends Number

upper bound
Mutability wildcards let a method accept more arguments

```java
printDates(List<@Readonly Date> lst) {
    for (int i=0; i<lst.size(); i++) {
        print(lst.get(i));
    }
}

printDates(new List<Date>());
printDates(new List<@Readonly Date>());
```

List<@Readonly Date> is not a supertype of List<Date>
Mutability wildcards let a method accept more arguments

```java
printDates(List<?@Readonly Date> lst) {
    for (int i=0; i<lst.size(); i++) {
        print(lst.get(i));
    }
}

printDates(new List<Date>());
printDates(new List<@Readonly Date>());
```

1st may be List<Date> or List<@Readonly Date>

Can be expressed as a wildcard with upper and lower bounds:
List<? extends @Readonly Date super Date>
Constraint variables for upper and lower bound

• A generic type List<Date> gives rise to 3 constraint variables:
  – mutability of List
  – mutability of upper bound of Date
  – mutability of lower bound of Date

• Small increase in the number of constraints

• Solve as usual

• Translate to \texttt{@Readonly}:

<table>
<thead>
<tr>
<th>Lower bound</th>
<th>Upper bound</th>
<th>Javari type</th>
</tr>
</thead>
<tbody>
<tr>
<td>mutable</td>
<td>mutable</td>
<td>mutable</td>
</tr>
<tr>
<td>readonly</td>
<td>readonly</td>
<td>readonly</td>
</tr>
<tr>
<td>mutable</td>
<td>readonly</td>
<td>? readonly</td>
</tr>
</tbody>
</table>
Mutability (qualifier) polymorphism

```java
@Polyread Date getDate(@Polyread Appt this) {
    return this.d;
}
```

Orthogonal to type polymorphism (generics)

Conceptually, duplicate the method:

```java
@Readonly Date getDate(@Readonly Appt this){...}
Date getDate(Appt this){...}
```
Duplicate constraint variables: mutable and readonly context

• For each method, duplicate all its constraint variables
  – Constant factor overhead (no need for >1 variable)
• Only the method invocation rule changes
• Solve as usual
• Translate to @Polyread:

<table>
<thead>
<tr>
<th>Mutable context</th>
<th>Immutable context</th>
<th>Javari type</th>
</tr>
</thead>
<tbody>
<tr>
<td>mutable</td>
<td>mutable</td>
<td>mutable</td>
</tr>
<tr>
<td>readonly</td>
<td>readonly</td>
<td>readonly</td>
</tr>
<tr>
<td>mutable</td>
<td>readonly</td>
<td>polyread</td>
</tr>
</tbody>
</table>
Experimental methodology

• Compare to
  – Human-written Javari code
  – Pidasa [Artzi 2007]
  – JPPA [Salcianu 2005]
  – JQual [Greenfieldboyce 2007]
  – Javari type-checker [Correa 2007]

• Subject programs:
  – JOlden (6 KLOC)
  – tinySQL (31 KLOC)
  – htmlparser (64 KLOC)
  – Eclipse compiler (111 KLOC)

• We inspected 8,694 references
Soundness

- Goal: No readonly reference can be modified
  - Result obeys the Javari language specification
- Compared to:
  - Human
  - Other inference tools
  - Javari type-checker
- Javarifier is sound
  - Misidentified no references as readonly
Precision (completeness)

• Goal: Maximal qualifiers (readonly, etc...)  
  – Example imprecise algorithm: every reference is mutable

• Javarifier is precise
  – Proof sketch that Javarifier is precise
  – Javarifier identified every readonly reference
    • That any other inference tool, or the human, did
  – Javarifier identified all fields to exclude from the abstract state
    • According to the human
Type-system-based inference

• Javarifier algorithm & initial experiments: [Tschantz 2006]
• JQual [Greenfieldboyle 2007]:
  – Same basic rules as [Tschantz 2006]
  – Polymorphism: multiple qualifiers, but no generics
  – Generality: any qualifier, but less expressive
  – Scalability: flow- and context-sensitive, unscalable
  – Accuracy: less precise, not sound
Non type-system-based inference

• JPPA [Salcianu 2005]
  – Whole-program static pointer and escape analysis
  – Sophisticated method summaries

• Pidasa [Artzi 2007]
  – Pipeline of static and dynamic stages
  – Sound and unsound variants

• JDynPur [Dallmeier 2008]
  – Lightweight dynamic analysis
Contributions

• Javarifier: inference of reference immutability
  – Sound
  – Precise (complete)
  – Scalable
• Rich, practical type system: Javari
  – Abstract state, type & mutability polymorphism, ...
• Download: http://pag.csail.mit.edu/javari/javarifier/
  – Handles all of Java
  – Works on bytecodes, inserts in source or .class
Contributions

• Javarifier: inference of reference immutability
  – Sound
  – Precise (complete)
  – Scalable

• Rich, practical type system: Javari
  – Abstract state, type & mutability polymorphism, ...

• Download: http://pag.csail.mit.edu/javari/javarifier/
  – Handles all of Java
  – Works on bytecodes, inserts in source or .class