```
print(Object x) {
  ...
}
```

```
print(@Readonly Object x) {
  ...
}
```

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) { … }
```

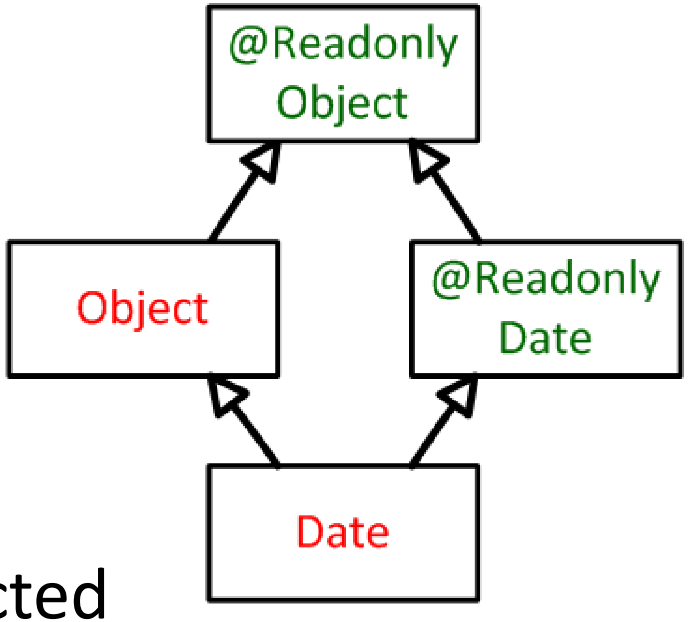
Immutability inference algorithm

- 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:

```
@Polyread Object id(@Polyread arg) { return arg; }
```

Algorithm: propagate mutability

- Syntax-directed constraint generation
- Unconditional constraint: $\underline{\mathbf{x}}$
  - **`x`** is mutable
  - Example code: **`x.field = 22;`**
- Conditional constraint: $\mathbf{y} \rightarrow \mathbf{z}$
  - if **`y`** is mutable, then **`z`** is mutable
  - Example code: **`y = z;`**
- Constraint solving: graph reachability
  - Unmarked references remain readonly

Appointment class

```
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);
  }
}
```

Receivers are written explicitly

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

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);
 }
}
```

Sample code: x.f = y;
Constraints: { $\underline{x}$,
 $f \rightarrow y$ }

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

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);
  }
}
```

Sample code: x.f = y;
Constraints: $\{ \underline{x}, f \rightarrow y \}$

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

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);
  }
}
```

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

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

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);
  }
}
```

Sample code:	x.f = y;
Constraints:	{ <u>x</u>, f → y }

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

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);
 }
}
```

Sample code: $x = y.f;$
Constraints: $\{ x \to f,$
$x \to y \}$

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

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);
 }
}
```

Sample code: x = y.f;
Constraints: $\{ x \rightarrow f,$
$x \rightarrow y \}$

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

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);
  }
}
```

Sample code: x = y.f;
Constraints: { $x \rightarrow f$,
$x \rightarrow y$ }

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

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);
 }
}
```

Sample code: $x = y.f;$

Constraints: $\{ x \rightarrow f,$

$x \rightarrow y \}$

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

Method invocation

```
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 = m(y1);
Constraints: { param1 $\rightarrow$ y1,
x $\rightarrow$ ret }

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

Method invocation

```
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 = m(y1);
Constraints: { param1 $\rightarrow$ y1, x $\rightarrow$ ret }

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

Method invocation

```
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: $x = m(y1);$

Constraints: $\{\ param1 \rightarrow y1,$
$x \rightarrow ret\ \}$

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

All constraints

```
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;
  }
}
```

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;
  }
}
```

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;
  }
}
```

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;
  }
}
```

Results inserted in source code

```
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

```
String toString() @Readonly {
  if (cache == null)
    cache = computeToString();
  return cache;
}
```

Does not modify receiver

Modifies receiver

  - Heuristics for common programming patterns
    - Requires user confirmation

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 : { $\underline{x}$, *type*(y) <: *type*(x[z]) }
  - Dereference rule:
    x = y.f : { *type*(f) <: *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

```
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

```
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

```
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

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

upper bound

`lst` may be of any List type whose element extends Number

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

Mutability wildcards
let a method accept more arguments

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

List<@Readonly Date> is not a supertype of List<Date>

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

Mutability wildcards let a method accept more arguments

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

lst 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>

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

Lower bound	Upper bound	Javari type
mutable	mutable	mutable
readonly	readonly	readonly
mutable	readonly	? readonly

Mutability (qualifier) polymorphism

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

Orthogonal to type polymorphism (generics)

Conceptually, duplicate the method:

```
@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`:

Mutable context	Immutable context	Javari type
mutable	mutable	mutable
readonly	readonly	readonly
mutable	readonly	polyread

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 [Greenfieldboyce 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