Javarifier: Inferring Reference Immutability for Java[*]

6.883: Program Analysis - Course Project

Vineet Sinha   Matthew Tschantz   Chen Xiao

ABSTRACT

Reference immutability constraints restrict certain references in a program from modifying objects. These constraints, such as those provided by the Javari programming language, have been shown to increase the expressiveness of a language and to prevent and detect errors. Unfortunately, annotating existing code, including libraries, with reference immutability constraints is a tedious and error prone task.

We propose an algorithm for inferring the references in a program that can be declared as read-only (or `romaybe`). Additionally, to gain additional precision, we introduce heuristics that suggest assignable and mutable fields.

To evaluate our algorithm, we have implemented a tool, Javarifier, to aid programmers in converting Java programs to Javari. We also demonstrate the practicality of our algorithm by using Javarifier to annotate an existing program.

1. INTRODUCTION

Immutable, or read-only, reference constraints prevent restricted references from being used to modify the objects (including transitive state) to which they refer. Using such immutability constraints can have many benefits: programmers can write more expressive code; program understanding and reasoning is enhanced by providing explicit, machine-checked documentation; errors can be detected and prevented; and analyses and transformations depending on compiler verified properties can be enabled. In practice, immutability constraints have been shown to be practical and to find errors in software [2].

For a programmer to gain the benefits of reference immutability constraints, he must annotate his code with the appropriate immutability constraints. For existing code, this can be a tedious and error prone task. Even worse, a programmer wishing to use reference immutability constraints, must first annotate the libraries he plans to use. Otherwise, a reference immutability type checker would be forced to assume that all methods may modify their arguments. Therefore, the programmer would be unable to invoke any of the library's methods on immutable references even if the method, if fact, does not modify its arguments.

To aid programmers with adding reference immutability constraints to Java programs, we have created an algorithm to soundly infer immutable references from Java code. For concreteness, we use the reference immutability constraints of Javari [13] (described in section 2).

Given a program, our algorithm calculates all the references, including local variables, method parameters, static and instance fields, that may have Javari's `readonly` or `romaybe` keywords added to their declarations. `readonly` references cannot be used to modify the abstract state of the object to which it refers (see section 2). `romaybe` references are used reduce method duplication and cannot be used to modify their referents, but can be passed, via a method's return value, to the client as a mutable reference (see section 2.3). Using this algorithm, programmers can convert Java programs, including libraries, to Javari.

Javari's other annotations, `assignable` and `mutable`, exclude parts of the concrete representation from the abstract state. Thus, they cannot be soundly inferred because they require knowing the intent of the programmer. However, without inferring this annotations, the precision of our algorithm would suffer. This problem can occur when a reference is used to modify a part of an object's concrete state that was not intended to be a part of the object's abstract state. Such a reference would be inferred to be mutable when in reality, it should be read-only. Therefore, to improve the results of our algorithm, we have created heuristics to recommend fields that should be declared `assignable` or `mutable`.

We have implemented our algorithm and heuristics in the form of the Javarifier tool. Given a Java program in classfile format, Javarifier first recommends fields that should be declared `assignable` or `mutable`. After the user has selected the fields to declare `assignable` or `mutable`, Javarifier calculates and returns to the user the references that may be declared `readonly` or `romaybe`.[1] It is important that Javarifier is

[*]Authors in alphabetical order

[1]As future work, Javarifier will be able to automatically insert the correct `readonly` and `romaybe` annotations into source and class files.

```
// Java
class Event {

  Date date;

  Date getDate() {
    return Date;
  }

  void setDate(Date d) {
    this.date = d;
  }

  int hc = 0;
  int hashCode() {
    if (hc == 0) {
      hc = ...;
    }
    return hc;
  }
}


// Javari
class Event {

  /*this-mutable*/ Date date;

  romaybe Date getDate() romaybe {
    return Date;
  }

  void setDate(/*mutable*/ Date d) /*mutable*/ {
    this.date = d;
  }

  assignable int hc = 0;
  int hashCode() readonly {
    if (hc == 0) {
      hc = ...;
    }
    return hc;
  }
}
```

Figure 1: A Java program (above) and the corresponding inferred Javari program (below).

able to operate on classfiles because programmers may wish convert library code that they only have in classfile format.

An example Java program and the corresponding Javari program is shown in figure 1.

Note that `hc` is heuristically recommended to—and in this case approved by—the user to be declared `assignable`. Otherwise, `hashCode` would be inferred, undesirably, to have a mutable receiver.

The rest of this document is organized as follows. Section 2 provides background on the Javari language. Section 3 explains the algorithm which is used to soundly infer read-only references. Section 4 discusses how our algorithm is soundly extended to infer `romaybe` references. Section 5 provides heuristics that can be used to infer `assignable` and `mutable` fields. Section 6 describes the implementation of our algorithm and heuristics within the Javarifier tool. Section 7 provides our experience using Javarifier on a variety of Java programs. Section 8 discusses related work. Finally, section 10 concludes.

2. BACKGROUND

Javari [13] extends Java's type system to allow programmers to specify and statically enforce reference immutability constraints. For every Java type `T`, Javari also has the type `readonly T`, with `T` being a subtype of `readonly T`. A reference declared to have a `readonly` type cannot be used to mutate the object it references.Mutation is any modification to an object's transitively reachable state, which is the state of the object and all state reachable from it by following references.[2] References that are not `readonly`, can be used to modify their referent and are said to be *mutable*. An example of Javari code is shown below

```
/*mutable*/ Date date;
readonly    Date rd;
  date.month = "June"; // OK
rodate.month = "June"; // error
```

A read-only reference cannot be converted to a mutable reference through assignment or casting. Mutable references, being subtypes of read-only references, may be assigned (without casting) or casted to read-only references. This behavior is demonstrated below.

```
/*mutable*/ Date date;
readonly    Date rodate;

rodate = date;                 // OK
rodate = (readonly Date) date; // OK
date = rodate;                 // error
date = (Date) rodate;          // error
```

In addition to local variables, the `readonly` keyword may be applied to fields and method parameters including the implicit `this` parameter. A `readonly this` parameter is declared by writing `readonly` immediately following the parameter list. For example, an appropriate declaration for the `StringBuffer.charAt` method in Javari is:

```
public char charAt(int index) readonly { ... }
```

Such a method is called a read-only method. In the context of the method, `this` is `readonly`. Thus, it is a type error for a read-only method to change the state of the receiver, and it is a type error for a non-read-only method to be called through a read-only reference.

Note that `final` and `readonly` are orthogonal notions in variable declaration: `final` makes the variable not assignable, but the reference may still be used to mutate the object it references, while `readonly` disallows the reference from being used to mutation its referent, but the variable may still be reassigned.

2.1 Fields

By default, a field of an object inherits its assignability and the mutability it is read at from the reference through which the field is reached. This default is called *this-assignability*

[2]Section 2.1.1 discusses overriding this behavior.

and *this-mutability*. If the reference through which the field is accessed is read-only, then the field is unassignable (`final`) and read at read-only. If the reference through which the field is accessed is mutable, then the field is assignable and read at mutable. These defaults ensure that mutability is transitive by default. A this-mutable field is always written to as a mutable type regardless of the mutability of the reference through which it was reached. The requirement that a this-mutable field is never written to with a read-only time is need to prevent a type loop hole which could be used to convert a read-only reference to a mutable reference [13]. The behavior of this-assignable and this-mutable fields is illustrated below.

```
class Cell {
  /*this-assignable this-mutable*/ Date d;
}

/*mutable*/ Cell  c; // mutable
readonly    Cell rc; // read-only

 c.d = new Date();   // OK: c.d is assignable
rc.d = new Date();   // error: rc.d is unassignable (final)

/*mutable*/ Date d1 =  c.d; // OK: c.d is read as mutable
/*mutable*/ Date d2 = rc.d; // error: rc.d is read as read-only
```

Only instance fields may be this-assignable or this-mutable. Other references (such as formal parameters and local variables) do not have a `this` to inherit their assignability from.

2.1.1 *Assignable and mutable fields*

By default, fields are this-assignable and this-mutable. Under these defaults, all the fields are considered to be a part of the object's abstract state and, therefore, can not be modified through a read-only reference. The `assignable` and `mutable` keywords enable a programmer to exclude specific fields from the object's abstract state.

Declaring a field `assignable` specifies that the field may always be reassigned, even through a read-only reference.

The `assignable` keyword specifies that a field may be assigned to even when reached through a read-only reference. Assignable fields can be used for caches as shown below.

```
/** Assignable Cell. */
class ACell {
  assignable /*this-mutable*/ Date d;
}

/** Converts a read-only Date to a mutable date. */
static /*mutable*/ Date
convertReadonlyToMutable(readonly Date roDate) {
  /*mutable*/ ACell mutCell = new ACell();
  readonly ACell roCell = mutCell;
  roCell.d = roDate;                    // error
  /*mutable*/ Date mutDate = mutCell.d;
  return mutDate;
}
```

The `mutable` keyword specifies that a field is mutable even when referenced through a read-only reference. A mutable field's value is not a part of the abstract state of the object (but the field's identity may be). For example, in the code below, `log` is declared `mutable` so that it may be mutated within read-only methods such as `hashCode`.

```
class Foo {
  final mutable List<String> log;

  int hashCode() readonly {
    log.add("entered hashCode()"); // OK: log is mutable
    ...
  }
}
```

Note that `assignable` and `mutable` are orthogonal notations and may be applied in conjunction.

2.2 Parametric types and arrays

In Java, the client of a generic class controls the type of any reference whose declared type is a type parameter. A client may instantiate a type parameter with any type that is equal to or a subtype of the declared bound. One can think of the type argument being directly substituted into the parameterized class wherever the corresponding type parameter appears.

Javari uses the same rules, extended in the natural way to account for the fact that Javari types include a mutability specification. A use of a type parameter within the generic class body has the exact mutability with which the type parameter was instantiated. Generic classes require no special rules for the mutabilities of type arguments, and the defaults for local variables and fields are unchanged.

As with any local variable's type, type arguments to the type of a local variable may be mutable (by default) or read-only (through use of the `readonly` keyword). Below, four valid local variable, parameterized type declarations of `List` are shown. Note that the mutability of the parameterized type `List` does not affect the mutability of the type argument.

```
/*mutable*/ List</*mutable*/ Date> ld1; // add/rem./mut.
/*mutable*/ List<readonly    Date> ld2; // add/remove
readonly    List</*mutable*/ Date> ld3; // mutate
readonly    List<readonly    Date> ld4; // (neither)
```

As in Java, subtyping is invariant in terms of type arguments. Therefore, `List</*mutable*/ Date>` is not related (a subtype or supertype) to `List<readonly Date>`. Wildcard types can be used to provide a common supertype for `List </*mutable*/ Date>` and `List<readonly Date>`. The common supertype will have `readonly Date` as its type argument's lower bound and `/*mutable*/ Date` as its the upper bound. Using Java-like syntax, this type is written as `List<? extends readonly Date super /*mutable*/ Date>`. Java, however, does not allow the declaration of a lower and upper bound on a wildcard. Therefore, Javari provides the special keyword `? readonly`, to specify a that a wildcard should be bounded above by the read-only version of a type and below, by the mutable version of the same type. For example, using `? readonly`, the common supertype of `List<readonly Date>` and `List</*mutable*/ Date>` would be written as `List<? readonly Date>`. Below, `List<? readonly Date>`'s relationship to the other `List` types is demonstrated.

```
List</*mutable*/ Date> ld1;
List<readonly    Date> ld2;
```

```
List<? readonly  Date> ld3;

ld1 = ld2; // error: ld1 and ld2 are unrelated

ld3 = ld1; // OK: ld3 supertype of ld1
ld3 = ld2; // OK: ld3 supertype of ld2
```

Since `List<readonly Date>`s may be assigned to a reference of type `List<? readonly Date>`, `Date`s taken from a `List` of `? readonly` elements must be read as being read-only. Otherwise, one would be able to place a `readonly Date` in a `List` of `readonly Date`'s and take the `Date` out of a aliasing `List` of `? readonly` dates. Similarly, because a `List` of `mutable Dates` could alias a `List` of `? readonly Dates`, only `mutable Dates` may be written to a `List` of `? readonly Dates`. Below, these type rules are demonstrated.

```
List<? readonly  Date> ld;

/*mutable*/ Date  d;
readonly    Date rd;

rd = ld.get(0); // OK
d  = ld.get(0); // error: elms. read as read-only

ld.add(rd);      // error: elms. written as mutable
ld.add(d);       // OK
```

As with any instance field's type, type arguments to the type of a field default to this-mutable, and this default can be overridden by declaring the type argument to be `readonly`, `mutable` or `? readonly`:

```
  class DateList {
    // 3 readonly lists whose elements have different mutability
    readonly List</*this-mutable*/ Date> lst;
    readonly List<readonly         Date> lst2;
    readonly List<mutable          Date> lst3;
    readonly List<? readonly       Date> lst4;
  }
```

As in any other case, the mutability of a type with this-mutability is determined by the mutability of the object in which it appears (not the mutability of the parameterized class in which it might be a type argument). In the case of `DateList` above, the mutability of `lst`'s elements is determined by the mutability of the reference to `DateList`, not by the mutability of `lst` itself.

Similar to non-parametric typed field, when reached through a mutable reference, the this-mutable elements of a parameterized class are read as mutable, and when reached through a read-only reference, the elements are read as mutable. However, whether reached through a mutable or read-only reference, the elements of a this-mutable parameterized class must always be written to by mutable references. Thus, this-mutable elements of a `List` when reached through a read-only reference are read as read-only and written to as mutable, which is the same behavior as a list of `? readonly` elements. The following example illustrates this behavior.

```
  class EventPlanner {
    /*this-mut*/ List</*this-mut*/ Event> evts;
  }

  /*mutable*/ EventPlanner  p;
  readonly    EventPlanner rp;

  /*mutable*/ List</*mutable*/ Event> e1 =  p.evts; // OK
  readonly    List<? readonly Event>  e2 = rp.evts; // OK
```

2.2.1 *Arrays*

As with generic container classes, a programmer may independently specify the mutability of each level of an array. As with any other local variable's type, each level of an array is mutable by default and may be declared read-only with the `readonly` keyword. Additionally, as with parametric types, the type of an array's elements may be declared as `? readonly`. As with any other field's type, each level may be declared mutable with the `mutable` keyword, read-only with the `readonly` keyword, or this-mutable by default. Again, the type of an array's elements may be declared as `? readonly`. Parentheses may be used to specify to which level of an array a keyword is to be applied. Below, four valid array local variable declarations are shown.

```
                    Date [] ad1; // add/remove, mutate
          (readonly Date)[] ad2; // add/remove
readonly            Date [] ad3; // mutate
readonly (readonly Date)[] ad4; // no add/rem., no mutate
```

The above syntax can be applied to arrays of any dimensionality. For example, the type `(readonly Date[])[]` is a two-dimensional array with a read-only inner-array and that is otherwise mutable.

As in the case of parameterized list, an array may have this-mutable elements. When such an array is reached through a mutable reference, the elements have `mutable` type and when reached through a read-only reference, the elements have `? readonly` type. Below, gives an example of this behavior.

```
  class Wheel
    /*this-mut*/ (/*this-mut*/ Spoke)[] spokes;


  /*mutable*/ Wheel  w;
  readonly    Wheel rw;

  /*mutable*/ (/*mutable*/ Spoke)[] spks1 = w.spokes; // OK
  readonly    (? readonly  Spoke)[] spks2 = w.spokes; // OK
```

Java's arrays are covariant. To maintain type safety, the JVM performs a check when a object is stored to an array. To avoid a run-time representation of immutability, Javari does not allow covariance across the mutability of array element types. For example,

```
            Date [] ad1;
  (readonly Date)[] ad2;

  ad2 = ad1; // error: arrays are not covariant over mutability
```

2.3 `romaybe`

When the mutability of the return type of a method is dependant on the mutability of one (or more) of the method's formal parameters, it is often necessary to write two methods that differ on in their signatures. For example, below, the return type of `getDate` depends on the mutability of the method's receiver.

```
  class Event {
    Date date;
    /*mutable*/ Date getDate() /*mutable*/ {
      return date;
    }

    readonly    Date getDate() readonly {
      return date;
    }
}
```

Thus, as shown above, one is forced to write two versions of `getDate`.

To reduce this code duplication, Javari provides the `romaybe` keyword, which is used to template a method over the mutability over the mutability of formal parameters and return types. If the type modifier `romaybe` is applied to any formal parameter and the return type (including `this`), then the type checker conceptually duplicates the method, creating two versions of it. In the first version of the method, all instances of `romaybe` are replaced by `readonly`[3]. In the second version, all instances of `romaybe` are removed. For example, the two `getDate` methods from `Event` could be replaced by the single method declaration shown below:

```
    romaybe Date getDate() romaybe {
      return date;
    }
```

3. INFERRING READ-ONLY REFERENCES

We use a flow and context insensitive algorithm to infer which references may be declared read-only. A read-only reference may have the `readonly` keyword added to its Javari type declaration. The algorithm is sound: Javarifier's recommendations will type check under Javari's rules. Furthermore, our algorithm is precise: declaring any references in addition to Javarifier's recommendations as read-only—without other modifications to the code—will result in the program not type checking.

The read-only inference algorithm does not determine which fields should be declared `mutable` or `assignable`; however, it is capable of inferring in the presence of fields that have been declared `mutable` or `assignable` (section 2.1.1) by an outside source (see section 5). The read-only inference algorithm assumes that all fields that are not read-only are this-mutable. This choice is made because declaring fields to be `mutable` is state that the field is not a part of the object's abstract state, an exceptional case.

For simplicity, we begin by describing our core algorithm (in section 3.1), i.e., the algorithm in the absence of subtyping, user-provided reference immutability annotations including assignable and mutable fields, arrays, and parametric types. We then extend the algorithm to subtyping (in section 3.2), user-provided constraints (in section 3.3), and arrays and parametric types (in section 3.4).

3.1 Core algorithm

Our algorithm generates then solves a set of mutability constraints for a program. A mutability constraint states when a given reference must be declared mutable. The algorithm

[3]If `romaybe` appeared as an type argument to a parameterized class, then it is replaced by `? readonly`.

$$
\begin{array}{lcl}
\texttt{C} & ::= & \texttt{class}\ \{\overline{\texttt{f}}\ \overline{\texttt{M}}\} \\
\texttt{M} & ::= & \texttt{m}(\overline{\texttt{x}})\{\overline{\texttt{s};}\} \\
\texttt{s} & ::= & \texttt{x} = \texttt{x} \\
 & | & \texttt{x} = \texttt{x.m}(\overline{\texttt{x}}) \\
 & | & \texttt{return x} \\
 & | & \texttt{x} = \texttt{x.f} \\
 & | & \texttt{x.f} = \texttt{x}
\end{array}
$$

Figure 2: Grammar for core language used during constraint generation.

uses two types of constraints: unguarded and guarded. Unguarded constraints state that a given reference must be mutable, e.g. "`x` is mutable." Guarded constraints state that a given dependent reference is mutable if the guard reference is mutable, e.g. "if `y` is mutable then `x` is mutable." We use *constraint variables* to name the references in these constraints—`x` and `y` in the previous examples. Unguarded constraints are represented by the constraint variable of the reference it is constraining, e.g. "`x`." Guarded constraints are represented as implications with guard reference's constraint variable as the predicate and dependent reference's constraint variable as the consequence, e.g. "$\texttt{y} \rightarrow \texttt{x}$.

After generating the constraints, the algorithm solves the constraints yielding a simplified constraint set. The simplified constraint set is the set of all unguarded constraints and guarded constraints that are satisfied directly by an unguarded constraint or by indirectly though the consequence of a different satisfied guarded constraint. The simplified constraint set contains the set of references that must be declared mutable; all other references may safely be declared read-only.

3.1.1 Constraint generation

The first phase of the algorithm generates constraints for each statement in a program. Unguarded constraints are generated when a reference is used to modify an object. Guarded constraints are generated when a reference is assigned to another reference or a field is reached through a reference.

We present constraint generation using a core language. The core language is a simple three-address programming language.[4] The grammar of the language is shown in figure 2. We use `C` and `M` to refer to class and method definitions, respectively. `m` ranges over method names, `f` ranges over names fields, and `s` ranges over allowed statements. `p`, `x`, `y`, and `z` range over variables (locals and method parameters). We use $\overline{\texttt{x}}$ as the shorthand for the (possibly empty) sequence $\texttt{x}_1 ... \texttt{x}_n$ We also include the special variable $\texttt{this}_\texttt{m}$, which refers to the receiver of `m`. Furthermore, we assume any program that attempts to reassign $\texttt{this}_\texttt{m}$ is malformed. Otherwise, $\texttt{this}_\texttt{m}$ is treated as a normal variable. Without loss of generality and for ease of presentation, we assume that all references and methods are given globally-unique names.

Control flow constructs are not modeled because the our al-

[4]Java source and classfiles can be converted to such a representation.

gorithm is flow-insensitive and, therefore, unaffected by such constructs. Java types are not modeled because the core algorithm does not use them. Constructors are modelled as regular methods returning a mutable $\texttt{this}_{\texttt{m}}$, and static members are omitted because do not demonstrate any interesting properties.

Each of the statements from figure 2 has a constraint generation rule, as shown in figure 3. The rules make use of the following auxiliary functions. `this(m)` and `params(m)` returns the receiver reference (`this`) and parameters of method `m`, respectively. `retVal(m)` returns the constraint variable, $\texttt{ret}_{\texttt{m}}$, that represents the reference to `m`'s return value. $\texttt{ret}_{\texttt{m}}$ is treated, otherwise, like any other constraint variable.

The constraint generation rules are described as follows:

Assign The assignment of variable `y` to `x` causes the guarded constraint $\texttt{x} \rightarrow \texttt{y}$ to be generated because, if `x` is a mutable reference, `y` must also be mutable for the assignment to succeed.

Invk The assignment of the return value of the invocation of method `m` on `y` with arguments $\overline{\texttt{y}}$ generates three constraints. The guarded constraint $\texttt{this}_{\texttt{m}} \rightarrow \texttt{y}$ is generated because the actual receiver must be mutable if `m` requires a mutable receiver. Similarly, the set of constraints $\overline{\texttt{p}} \rightarrow \overline{\texttt{y}}$ is generated because the i^{th} argument must be mutable if `m` requires the i^{th} parameter to be mutable. Finally, the constraint $\texttt{x} \rightarrow \texttt{ret}_{\texttt{m}}$ is introduced to enforce that `m` must return a mutable reference if its return value is assign to a mutable reference.

These constraints are extensions of the Assign rule when method invocation is framed in terms of operational semantics: the receiver, `y`, is assigned to the $\texttt{this}_{\texttt{m}}$ reference, each actual argument is assigned to the method's formal parameters, and the return value, $\texttt{ret}_{\texttt{m}}$, is assigned to `x`.

Ret The return statement `return x` adds the constraint $\texttt{ret}_{\texttt{m}} \rightarrow \texttt{x}$ because in the case that the return type of the method is found to be mutable, all references returned by the method must be mutable.

Ref The assignment of `y.f` to `x` generates two constraints. The first, $\texttt{x} \rightarrow \texttt{f}$, is required because, if `x` is mutable, then the field `f` cannot be read-only. The second, $\texttt{x} \rightarrow \texttt{y}$, is needed because, if `x` is mutable, then `y` must be mutable to yield a mutable reference to field `f`.

Set The assignment of `y` to `x.f` causes the unguarded constraint `x` to be generated because `x` has just been used to mutate the object to which it refers. The constraint $\texttt{f} \rightarrow \texttt{y}$ is also added because if `f` is not read-only, then a mutable reference must be assigned to it.

The constraints for a program is the union of the constraints generated for each line of the program. Figure 4 shows constraints being generated for a sample program.

Library code: Our algorithm as described above makes the close-world assumption. That is, it assumes that all code

$$\texttt{x} = \texttt{y} : \{\texttt{x} \rightarrow \texttt{y}\} \quad (\textsc{Assign})$$

$$\frac{\texttt{this(m)} = \texttt{this}_{\texttt{m}} \qquad \texttt{params(m)} = \overline{\texttt{p}} \qquad \texttt{retVal(m)} = \texttt{ret}_{\texttt{m}}}{\texttt{x} = \texttt{y.m}(\overline{\texttt{y}}) : \{\texttt{this}_{\texttt{m}} \rightarrow \texttt{y},\ \overline{\texttt{p}} \rightarrow \overline{\texttt{y}},\ \texttt{x} \rightarrow \texttt{ret}_{\texttt{m}}\}} \quad (\textsc{Invk})$$

$$\frac{\texttt{retVal(m)} = \texttt{ret}_{\texttt{m}}}{\texttt{return x} : \{\texttt{ret}_{\texttt{m}} \rightarrow \texttt{x}\}} \quad (\textsc{Ret})$$

$$\texttt{x} = \texttt{y.f} : \{\texttt{x} \rightarrow \texttt{f},\ \texttt{x} \rightarrow \texttt{y}\} \quad (\textsc{Ref})$$

$$\texttt{x.f} = \texttt{y} : \{\texttt{x},\ \texttt{f} \rightarrow \texttt{y}\} \quad (\textsc{Set})$$

Figure 3: Constraint generation rules.

```
class {
  f;
  foo(p) {
    x = p;         // {x -> p}                 Assign
    y = x.f;       // {y -> f, y -> z}         Ref
    z = x.foo(y);  // {this_foo -> x,
                   //  p -> x, z -> ret_foo}   Invk
    this.f = y;    // {this_foo, f -> y}       Set
    return y;      // {ret_foo -> y}           Ret
  }
```

Program constraints:

$$\{\texttt{x} \rightarrow \texttt{p},\ \texttt{this}_{\texttt{foo}} \rightarrow \texttt{x},\ \texttt{p} \rightarrow \texttt{x},\ \texttt{z} \rightarrow \texttt{ret}_{\texttt{foo}},$$
$$\texttt{y} \rightarrow \texttt{f},\ \texttt{y} \rightarrow \texttt{z},\ \texttt{this}_{\texttt{foo}},\ \texttt{f} \rightarrow \texttt{y},\ \texttt{ret}_{\texttt{foo}} \rightarrow \texttt{y}\}$$

Figure 4: Example of constraint generation. The constraints generated and the constraint generation rule used for each line of code is shown after the line as a comment.

is seen and it, therefore, is safe to change public method return types and types of public fields. In the case that a user is running the algorithm on the whole program, the closed world assumption allows the results to be more precise. However, in order to support analyzing library classes which do not have access to their clients, our algorithm needs to be sound even without the presence of all client code. Thus, all non-private[5] fields and the return types of non-private methods must be mutable because an unseen client may rely on a mutability of the field or return value. Our algorithm is easily modified to enforce this restriction: an unguarded constraint for every non-private field and non-private method return value is added to the constraint set. Depending on the needs of the other, the algorithm can be optionally conducted either under the closed-world assumption.

3.1.2 *Simplifying constraints*

The second phase of the algorithm is to simplify the constraint set. Constraint set simplification is done by checking if any of the unguarded constraints satisfies, i.e. matches, the guard of a guarded constraint. If so, the guarded constraint is "fired" by removing it from the constraint set and adding its consequence to the constraint set as an unguarded constraint. The new unguarded constraint can then be used to fire other guarded constraints. Once no more constraints can be fired , constraint simplification terminates. The unguarded constraints in the simplified constraint set is the set of references that are used to mutable their referents.[6]. These references cannot have their declarations annotated with `readonly`. All other references are safe to have their declarations annotated with `readonly`.

The constraints generated from the example program in figure 4 will be simplified as shown below. (For clarity, the unguarded constraints are listed first.)

$$\{\texttt{this}_{\texttt{foo}},\ \texttt{x} \rightarrow \texttt{p},\ \texttt{this}_{\texttt{foo}} \rightarrow \texttt{x},\ \texttt{p} \rightarrow \texttt{x},\ \texttt{z} \rightarrow \texttt{ret}_{\texttt{foo}},$$
$$\texttt{y} \rightarrow \texttt{f},\ \texttt{y} \rightarrow \texttt{z},\ \texttt{f} \rightarrow \texttt{y},\ \texttt{ret}_{\texttt{foo}} \rightarrow \texttt{y}\} \Rightarrow$$
$$\{\texttt{this}_{\texttt{foo}},\ \texttt{x},\ \texttt{p}\}$$

At the end of simplification, the unguarded constraints $\texttt{this}_{\texttt{foo}}$, `x` and `p` are present in the simplified constraint set and are mutable. These references may not have their declarations annotated with `readonly`. All other references, `y`, `z`, $\texttt{ret}_{\texttt{foo}}$, and `f`, are read-only and may have their declarations annotated with `readonly`. Figure 5 shows the result of applying our algorithm's result to the example program.

3.2 Subtyping

Java and Javari allows subtyping polymorphism which enables multiple versions of a method to be specified through overriding[7]. All versions of an overridden method must have

[5]In the case that an entire package is being analyzed, package-protected (default access) methods and fields may be processed as under the closed world assumption.

[6]They must be declared this-mutable for fields, mutable in the case of all other types of references.

[7]We use the term *overriding* to specify when a method implements an interface's method signature, implements an abstract method, or overrides a concrete method in a superclass. Furthermore, for brevity and to highlight their identical treatment, we will refer to both abstract methods and interface method signatures as *abstract methods*.

```
class {
  readonly f;
  foo(p) {
    x = p;
    readonly y = x.f;
    readonly z = x.foo(y);

    this.f = y;
    return y;
  }
}
```

Figure 5: The results of applying our algorithm's results the program.

identical signatures including the mutability of parameters.[8] Therefore, our algorithm must infer the same mutabilities for the parameters of all versions of the overridden methods. If not, the resulting code would not type check under Javari's type rules. For example, in the case of a class's method overriding an abstract method, the class would no longer implement the interface or abstract class that contained the abstract method. On the other hand, in the case of a method overriding a concrete method, the two methods would no longer be in a overriding relationship, but an overloading relationship. This is not allowed, however, because Javari does not allow two overloading methods to differ only in the mutability of parameters.

To ensure that our algorithm does not infer different signatures for polymorphic methods, we must place an additional set of constraints in the constraint set. For every parameter of a overloaded method, we add the constraint that it is equal in mutability to every other version of the method. Equality is represented by two guarded constraints, one indicated if the first parameter is mutable then the second parameter is mutable, the other specifies that if second parameter is mutable then the first is mutable. For example, below, the method `toString` is overloaded.

```
class Event {
  Date d;
  ...
  String toString() {
    return d.toString();
  }
}

class Birthday extends Event {
  ...
  int age;
  String string;
  String toString() {
    if (string == null) {
      string = "I'm" + age + "on" + d + "!";
    }
    return string;
  }
}
```

Thus, the mutability of the `this` parameter of `Event`'s and `Birthday`'s `toString` methods must be the same. This requirement generates the constraints $\texttt{this}_{\texttt{Event.toString}} \rightarrow \texttt{this}_{\texttt{Birthday.toString}}$ and $\texttt{this}_{\texttt{Birthday.toString}} \rightarrow \texttt{this}_{\texttt{Event.toString}}$.

[8]Java signatures do not include the return type, however, such methods are additionally restricted to have covariant return types.

Although, to preserve overloading, both methods are declared to have mutable `this` parameters, only `Birthday`'s `toString` method actually mutates it receiver. One version of the method mutating the receiver while another does not is disconcerting because overloaded methods are suppose to both be implementations of the same specification. Therefore, our algorithm should issue a warning in such cases or make both receivers read-only using the technique of declaring select fields to be mutable or assignable given in section 5.1. For example, the technique would state that `Birthday`'s `string` field should be declared assignable, and doing so would allow `Birthday`'s `toString` along with `Event`'s to be read-only, the expected result.

3.3 User-provided annotations

Our read-only inference algorithm can be extended to incorporate user-provided annotations. This capability is required because the read-only inference algorithm is not capable of directly inferring the `assignable` and `mutable` keywords. Additionally, user-provided annotations are needed for native method invocations, on which our algorithm cannot operate. Finally, a user may, using knowledge of the program, wish to override the annotations inferred by our algorithm. A user may specify that instance fields are this-mutable, read-only or mutable and other references (static fields, local variables and parameters) are read-only or mutable.

A user declaring either a field or non-field reference to be read-only causes the algorithm, upon finishing, to check if it is safe to declare the given reference as `readonly`. If not, the algorithm either issues an error, stating the conflict with the user annotation, or recommend ways to change the code, discussed in section 5.1, such that reference would be safe to declare be `readonly`.

It is expected that this type of declaration will be particularly useful for overridden methods (see section 3.2) because declaring a supertype's method to be read-only would propagate to all the subtypes. For example, a user could annotate `Object`'s `toString`, `hashCode`, and `equals` methods as having read-only `this` parameters. The user would then be notified if any class's implementation of the above methods are not read-only. Such cases may arise from errors or usage of a cache fields that should be declared `assignable`.

A user declaring a non-field reference mutable or a field this-mutable results in the system adding an unguarded constraint that the reference cannot be read-only. This specification is useful when a programmer knows that a certain reference is not currently used to modify its referent, but that it may be used to do so in the future. This case is expected to be particularly relevant for method return types when running the algorithm with the closed world assumption (see section 3.1.1). For example, a user may wish to annotate an program under the closed world assumption but specify that certain public observer methods return mutable references.

Finally, a user may declare fields to be `assignable` or `mutable`. These annotations are particularly important because our algorithm cannot soundly infer fields to be `assignable` or `mutable` as it would require knowing which fields the programmer intended to be a part of an object's abstract state. Our algorithm, however, can be extended to work with fields that are declared, by an outside source, to be `assignable` or `mutable`. This ability is important because fields may be determined to be `mutable` or `assignable` by hand or through heuristics (section 5).

$$\frac{\texttt{assignable(f)}}{\texttt{x.f} = \texttt{y} : \{\texttt{f} \rightarrow \texttt{y}\}}\ (\textsc{Set-A})$$

$$\frac{\neg\texttt{assignable(f)}}{\texttt{x.f} = \texttt{y} : \{\texttt{x},\ \texttt{f} \rightarrow \texttt{y}\}}\ (\textsc{Set-N})$$

$$\texttt{mutable f};: \{\texttt{f}\}\ (\textsc{Mutable})$$

$$\frac{\texttt{mutable(f)}}{\texttt{x} = \texttt{y.f} : \{\}}\ (\textsc{Ref-M})$$

$$\frac{\neg\texttt{mutable(f)}}{\texttt{x} = \texttt{y.f} : \{\texttt{x} \rightarrow \texttt{f},\ \texttt{x} \rightarrow \texttt{y}\}}\ (\textsc{Ref-N})$$

Figure 6: Modified constraint generation rules for handling assignable and mutable fields.

To extend our algorithm to handle `mutable` and `assignable` fields, the constraint generation rules are extended to check the assignability and mutability of fields before adding constraints. The algorithm is given a set of fields that are declared to be `assignable`, AS, and the set of fields that are declared to be `mutable`, MS. The auxiliary functions `assignable(f)` and `mutable(f)` returns true if and only if `f` is contained in AS or MS, respectively. The changes to the constraint generation rules are shown in figure 6 and are described below.

To handle `assignable` fields, the Set rule is divided into two rules, Set-A and Set-N, which depend on the assignability of the field. If the field is `assignable`, then Set-A does not add the unguarded constraint that the reference used to reach the field must be mutable, because an `assignable` field may be assigned through either a read-only or mutable reference. If the field is not `assignable`, then Set-N proceeds as normal.

To handle `mutable` fields, we add the constraint generation rule Mutable, which adds an unguarded constraint for each `mutable` field. The Ref rule is again divided into two rules: Ref-M and Ref-N depending on the mutability of the field. If the field is not `mutable`, then Ref-N proceeds as normal. If the field is `mutable`, then Ref-M does not add any constraints because, when compared to Ref-N, (1) the first constraint, that of the field, has already been added to the constraint set via the Mutable rule, and (2) the second constraint, applied to the reference used to reach the field, is eliminated because `mutable` field will be mutable even when reached through a readonly reference.

3.4 Arrays and parametric types

Our algorithm can be extended to handle arrays and parametric types. We begin by discussing the issues involved for

Grammar:

$$\begin{array}{rcl} \mathtt{s}' & ::= & s \\ & | & \mathtt{x[x]} = \mathtt{x} \\ & | & \mathtt{x} = \mathtt{x[x]} \end{array}$$

Figure 7: Core language grammar extended for arrays.

arrays; parametric types are a simple extension.

We extend our core language grammar to allow for storing and reading from arrays. The extended grammar is shown in figure 7.

3.4.1 Constraint variables

Javari allows array elements to have two-sided bounded types. For example, the array `(? readonly Date)[]` has elements with upper bound `readonly Date` and lower bound `mutable Date`. All array element types can be written in the form of having an upper bound and a lower bound. For example, `(readonly Date)[]` has elements with upper bound `readonly Date` and lower bound `readonly Date`. Therefore, our algorithm will need infer an upper bound and a lower bound for the types of array elements.

In addition to the arrays themselves, we must constraint the upper bound and lower bound of an array's elements. Thus, we create constraint variables for an array's elements' upper and lower bounds in addition to the array's constraint variable. For a reference `x` that is a single dimension array, we denote the array's mutability by the constraint variable `x[]` and the array's elements' upper bound by $\mathtt{x}_{\triangleleft}$ and the lower bound by $\mathtt{x}_{\triangleright}$.

3.4.2 Constraint generation

The constraint generation rules are extended to add constraints between array elements during an assignment between array references. For the assignment, `x = y` where `x` and `y` are arrays, the extension must enforce that `y` is a subtype of `x`. Simplified subtyping rules for Javari are given in figure 8. The simplified rules only check the mutability of the type because we assume the program being converted type checks under Java. For parametric types, the subtyping rules check that the element types are in a *contains* relation.[9] An array element's type (or a parametric class's type argument), $\mathtt{TA}_2$, is said to be contained by another array element's type, $\mathtt{TA}_1$, written $\mathtt{TA}_2 <= \mathtt{TA}_1$, if the set of types denoted by $\mathtt{TA}_2$ is a subset of the types denoted by $\mathtt{TA}_1$. Our subtyping rules use * to denote additional, possibly zero, levels of the array.

We modify the constraint rules to enforce the subtyping relationship across assignments including the implicit assignments that occur during method invocation. The extended rules are shown in figure 9.

3.4.3 Constraint simplification

[9]In Java, array are covariant; however, in Javari, array are invariant in respect to mutability, therefore, we use the contains relationship as Java's parametric types do.

$$\frac{\mathtt{y[]}^*\mathtt{[]} \rightarrow \mathtt{x[]}^*\mathtt{[]} \qquad \mathtt{x[]}^* <= \mathtt{y[]}^*}{\mathtt{x[]}^*\mathtt{[]} <: \mathtt{y[]}^*\mathtt{[]}}$$

$$\frac{\mathtt{y} \rightarrow \mathtt{x}}{\mathtt{x} <: \mathtt{y}}$$

$$\frac{\mathtt{x[]}^*_{\triangleleft} <: \mathtt{y[]}^*_{\triangleleft} \qquad \mathtt{y[]}^*_{\triangleright} <: \mathtt{x[]}^*_{\triangleright}}{\mathtt{x[]}^* <= \mathtt{y[]}^*}$$

Figure 8: Subtyping rules for mutability in Javari.

$$\mathtt{x} = \mathtt{y} : \{\mathtt{y} <: \mathtt{x}\} \ (\textsc{Assign})$$

$$\frac{\begin{array}{c}\mathtt{this(m)} = \mathtt{this_m} \\ \mathtt{params(m)} = \overline{\mathtt{p}} \qquad \mathtt{retVal(m)} = \mathtt{ret_m}\end{array}}{\mathtt{x} = \mathtt{y.m(}\overline{\mathtt{y}}\mathtt{)} : \{\mathtt{y} <: \mathtt{this_m},\ \overline{\mathtt{y}} <: \overline{\mathtt{p}},\ \mathtt{ret_m} <: \mathtt{x}\}} \ (\textsc{Invk})$$

$$\frac{\mathtt{retVal(m)} = \mathtt{ret_m}}{\mathtt{return\ x} : \{\mathtt{x} <: \mathtt{ret_m}\}} \ (\textsc{Ret})$$

$$\mathtt{x} = \mathtt{y.f} : \{\mathtt{f} <: \mathtt{x},\ \mathtt{x} \rightarrow \mathtt{y}\} \ (\textsc{Ref})$$

$$\mathtt{x.f} = \mathtt{y} : \{\mathtt{x},\ \mathtt{y} <: \mathtt{f}\} \ (\textsc{Set})$$

$$\mathtt{x} = \mathtt{y[z]} : \{\mathtt{y[]} <: \mathtt{x}\} \ (\textsc{Array-Ref})$$

$$\mathtt{x[z]} = \mathtt{y} : \{\mathtt{x},\ \mathtt{y} <: \mathtt{x[]}\} \ (\textsc{Array-Set})$$

Figure 9: Constraint generation rules in presence of arrays.

Before the constraint set can be simplified as before, subtyping and contains constraints must be reduced to guarded constraints. To do so, each subtyping or contains constraint is expanded by adding the rule's predicates, as shown in figure 8, to the constraint set. This step is repeated until only guarded and unguarded constraints remain in the constraint set. For example, the statement `x = y` where `x` and `y` are one dimensional arrays would generate then reduce subtyping and contains constraints as follows:

$$\begin{array}{lll} \mathtt{x} = \mathtt{y} & : & \{\mathtt{y[]} <: \mathtt{x[]}\} \\ & : & \{\mathtt{x[]} \rightarrow \mathtt{y[]},\ \mathtt{y} <= \mathtt{x}\} \\ & : & \{\mathtt{x[]} \rightarrow \mathtt{y[]},\ \mathtt{y}_{\triangleleft} <: \mathtt{x}_{\triangleleft},\ \mathtt{x}_{\triangleright} <: \mathtt{y}_{\triangleright}\} \\ & : & \{\mathtt{x[]} \rightarrow \mathtt{y[]},\ \mathtt{x}_{\triangleleft} \rightarrow \mathtt{y}_{\triangleleft},\ \mathtt{y}_{\triangleright} \rightarrow \mathtt{x}_{\triangleright}\} \end{array}$$

The first guarded constraint enforces that `y` must be a mutable array if `x` is a mutable array. The second and third constraints constrain the bounds on the arrays' elements types. $\mathtt{x}_{\triangleleft} \rightarrow \mathtt{y}_{\triangleleft}$ requires the upper bound of `y`'s elements to be mutable if the upper bound `x`'s elements is mutable. This rule is due to covariant subtyping between upper bounds. $\mathtt{y}_{\triangleright} \rightarrow \mathtt{x}_{\triangleright}$ requires the lower bound of `x`'s elements to be mutable if the lower bound `y`'s elements is mutable. This rule is due to contravariant subtyping between lower bounds.

After eliminating all subtyping or contains constraints, the remaining guarded and unguarded constraint set is simplified as before.

Unlike the case without arrays, this approach is not guaranteed to alway provide the most general method signatures[10], or, therefore, the maximum number of read-only annotations. We are unable to always infer the most general method signature because in some cases one does not exist. For example, take the method below:

```
void addDate(Date[] a, Date d) {
   a[0] = d;
}
```

Three type-correct Javari method signatures can be infer (our algorithm as stated above infers the first):

```
addDate( (readonly    Date)[] a, readonly    Date d)
addDate( (? readonly  Date)[] a, /*mutable*/ Date d)
addDate( (/*mutable*/ Date)[] a, /*mutable*/ Date d)
```

In all cases the array `a` is mutable. The last method signature is strictly less general than the second, so we will no longer consider it. The first method signature allows a more general type for `d` because could be applied to `readonly` or `mutable Dates`. The second method signature allows a more general type for `a` because it could applied to arrays with `readonly`, `mutable`, or `? readonly` elements. Thus, neither the first or second method signature is the most general and there is no most general method signature that can be inferred.

Furthermore, this problem cannot be solved with method overriding, including `romaybe` (section 2.3), because one is not allowed to overload methods based only on differing mutabilities of the arguments in Javari and `romaybe` in this case would not expand to the needed types.

[10]The most general method is the method that accepts the most types of arguments. For example `foo(readonly Date)` has a more general method signature than `foo(mutable* Date)`, because it can take read-only `Dates` in addition to mutable ones.

Three approach to deal with this situation are:

1. Duplicate the method into two versions with different names and then infer which should be method should be used at each call site.
2. Rewrite the method to be generic using type parameters.
3. Try both signatures, and see which one allows the most references to be declared `readonly`.

The first and second techniques would guarantee the maximum number of read-only annotations, but make a larger change to the program by adding or parameterizing methods in addition to adding immutability annotations. The third technique could infer less read-only annotations in the case that both versions of the method are needed but makes a smaller change to the code. Unfortunately, the third technique's runtime would grow exponentially in the number of such methods, if it tried every possible combination of signatures. Evaluating these techniques in practice remains future work.

3.4.4 Parametric types

Parametric types are handled in the same fashion as arrays. In their case, a constraint variable must be made for each type argument to a parametric class.

4. INFERRING `ROMAYBE`

Our core read-only inference algorithm can be extended to infer the `romaybe` keyword. Doing so allows more precise immutability annotations to be inferred.

4.1 Motivation

For our algorithm to be practical, it must also be able to infer `romaybe` types. `romaybe` is used to create two versions of a method: a read-only version, which takes a read-only parameter and returns a read-only type, and a mutable version, which takes mutable parameter and returns a mutable type (see section 2). For example, in figure 10, the `getSeat` method could be declared to have a `romaybe this` parameter and return type. Declaring `getSeat` to be `romaybe` allows the mutable version to be used by `lowerSeat`, which mutates the returned `Seat` object, and the read-only version to be used by `printSeat`, which does not mutate the returned `Seat` object. Providing both versions of the method is beneficial, because by using the read-only version of the `getSeat` method, `printSeat`'s parameter `b` may be declared `readonly` as one would expect, since `printSeat` does not modify its argument. The results of the read-only inference when `getSeat` is declared to be `romaybe` are shown in figure 11.

On the other hand, if the algorithm is unable to infer `romaybe`, it will be able to only provide the mutable version of the method as a mutable return type is required for the `lowerSeat` method. Unfortunately, inferring only the mutable version of `getSeat` requires contexts that do not need a mutable `Seat` object to still invoke the mutable version of a method and,

```
  class Bicycle {
    private Seat seat;

    Seat getSeat() {
      return seat;
    }
  }

 static void lowerSeat(Bicycle b) {
    Seat s = b.getSeat();
    seat.height = 0;
  }

  static void printSeat(Bicycle b) {
    Seat s = b.getSeat();
    System.out.prinln(s);
  }
```

Figure 10: Java code containing `getSeat` method which should be declared to be `romaybe`.

```
  class Bicycle {
    private Seat seat;

    romaybe Seat getSeat() romaybe {
      return seat;
    }
  }

 static void lowerSeat(/*mutable*/ Bicycle b) {
    /*mutable*/ Seat s = b.getSeat();
    seat.height = 0;
  }

  static void printSeat(readonly Bicycle b) {
    readonly Seat s = b.getSeat();
    System.out.prinln(s);
  }
```

Figure 11: Sound and precise Javari code which gives `getSeat` `romaybe` type.

```
  class Bicycle {
    private Seat seat;

    /*mutable*/ Seat getSeat() /*mutable*/ {
      return seat;
    }
  }

 static void lowerSeat(/*mutable*/ Bicycle b) {
    /*mutable*/ Seat s = b.getSeat();
    seat.height = 0;
  }

  static void printSeat(/*mutable*/ Bicycle b) {
    /*mutable*/ Seat s = b.getSeat();
    System.out.println(s);
  }
```

Figure 12: Sound but imprecise Javari code which not using `romaybe`. `getSeat` is given the imprecise mutable type instead of the precise `romaybe` type.

therefore, provide a mutable receiver to `getSeat`. Thus, the `printSeat` method which does not require a mutable reference to the `Seat` object returned by `getSeat`, would still need to provide mutable reference as the receiver to the `getSeat` method as shown in figure 12. This situation can cause many references, such as `printSeat`'s parameter `b`, to be declared `mutable` when they are not used to mutate the object to which they refer. While these annotations are sound they suffer from a loss of precision.

4.2 Approach

To extend our read-only inference algorithm, we create two versions of every method and invoke one version where a mutable return type is needed and the other when a mutable return type is not needed. If the type of any of the parameters of the method is different between the two versions, those parameters and the return type should be declared `romaybe`. Otherwise, the two contexts are redundant, and only a single method should be created.

4.2.1 Method contexts

We extend our read-only inference algorithm to infer `romaybe` by recognizing that `romaybe` methods. In the first context, the return type and `romaybe` parameters are mutable. In the second context, the return type and `romaybe` parameters are read-only. Since we do not know which methods are `romaybe` before hand, we create both contexts for every method.

In the case of methods that should not have `romaybe` parameters, the two contexts are redundant, that is, mutability of parameters from each context will be identical. However, in the case of methods should `romaybe` parameters, the parameters will be mutable in the mutable context and read-only in the read-only context.

To create two contexts for a method, we create two constraint variables for every method local reference (parameters, local variables, and return value). To distinguish constraint variables from each context, we superscript the constraint variables from the read-only context with `ro` and those from the mutable context with `mut`. Constraint variables for fields are not duplicated as `romaybe` may not be

$$\frac{\mathtt{this(m)} = \mathtt{this_m} \qquad \mathtt{params(m)} = \overline{\mathtt{p}} \qquad \mathtt{retVal(m)} = \mathtt{ret_m}}{\mathtt{x} = \mathtt{y.m}(\overline{\mathtt{y}}) : \{\mathtt{x}^? \rightarrow \mathtt{this}_\mathtt{m}^\mathtt{mut} \rightarrow \mathtt{y}^?,\ \mathtt{x}^? \rightarrow \overline{\mathtt{p}^\mathtt{mut}} \rightarrow \overline{\mathtt{y}^?},\ \mathtt{x} \rightarrow \mathtt{ret_m}\}}$$

Figure 13: Constraint generation rule, INVK-ROMAYBE **for method invocation in the presence of `romaybe` references.**

applied to fields and, thus, only single context exists.

4.2.2 Constraint generation rules

With the exception of INVK, all the constraint generation rules are changed to generate identical constraints for both the read-only and the mutable version of constraint variables. Otherwise, the constraints are the same. For example, `x = y` generates the constraints:

$$\{\mathtt{x}^{\mathtt{ro}} \rightarrow \mathtt{y}^{\mathtt{ro}}, \mathtt{x}^{\mathtt{mut}} \rightarrow \mathtt{y}^{\mathtt{mut}}\}$$

For short hand, we write constraints that are identical with the exception of constraint variables' contexts by superscripting the constraint variables with $^?$. For example, the results of the `x = y` constraint generation rule can be written as:

$$\{\mathtt{x}^? \rightarrow \mathtt{y}^?\}$$

The method invocation rule must be modified to invoke the mutable version of a method when a mutable return type is needed and to invoke the read-only version, otherwise. For the method invocation, $\mathtt{x} = \mathtt{y.m}(\overline{\mathtt{y}})$, a mutable return type is needed when `x` is mutable. Otherwise, the read-only version of the method is used. When the mutable version of a method is used, the actual receiver and arguments to the method must be mutable if the `this` parameter and formal parameters of the mutable version of the method are mutable. When the read-only version of a method is used, the actual receiver and arguments to the method must be mutable if the `this` parameter and formal parameters of the read-only version of the method are mutable. The new INVK rule are shown in figure 13.

The invocation rule checks whether `x` is mutable, and in the case that it is, adds constraints that the actual arguments must be mutable of the formal parameters of the mutable context of the method are mutable. Rule also adds the constraints that the the actual arguments must be mutable of the formal parameters if the read-only context of the method are mutable without checking if `x` is mutable. The check can be skipped because the parameters of the read-only version of the method's constraints are guaranteed to be subsumed by the mutable version of the method. Thus, adding the read-only version of the invoked method's constraints in addition to the mutable version's constraints in the case of the mutable method being invoked will add nothing new to the constraint set.

The constraints are solved as before.

4.2.3 Interpreting results of solved constraint set

If both the mutable and read-only version of a constraint variable is found to be in in the simplified, unguarded constraint set, then the corresponding reference is declared mutable. If both versions of the constraint variable is absent from the constraint set, then the corresponding reference is declared `readonly`. Finally, if the mutable context's constraint variable is in the constraint set but the read-only constraint variable is not in the constraint set, the corresponding reference is declared `romaybe` because the mutability of the reference depends on which version of the method is called.[11]

It could be the case that a method contains `romaybe` references but no `romaybe` parameters. For example, below, `x` and the return value of `getNewDate` could be `romaybe`.

```
Date getNewDate() {
  Date x = new Date();
  return x;
}
```

However, `romaybe` references are only allowed, or useful, if the method has a `romaybe` parameter. Thus, if none of a method's parameters are `romaybe`, all the method's `romaybe` references are converted to be `mutable` references.

5. INFERRING `MUTABLE` AND `ASSIGNABLE`

The core algorithm infers references to objects that are not modified as `readonly`. While it is able to leverage user-provided `assignable` and `mutable` annotations for inferring `readonly`, it is not able to infer the `assignable` and `mutable` annotations as these represent overriding of the default transitive immutability guarantees of the Javari language. We present a technique that infers and provides recommendations when the default transitive immutability guarantees should be overridden by leveraging references that have been marked `readonly` but are being modified. Since these references are annotated `readonly` but are being modified, the read-only inferencer will not have recommended such annotations and they are likely to be provided by users or via other techniques such as via heuristics. We present two such heuristic to detect fields used as caches and mark method references as `readonly`.

Since the provided recommendations represent overriding of the immutability guarantees, these techniques are disabled by default and need to be explicitly triggered by the user via a Javarifier input option. The recommendations provided by the sections are presented to the user on as ranked lists of fields that should be `assignable` or `mutable`.

5.1 Leveraging `readonly` annotations

In order to detect the overriding of default transitive immutability guarantees, we need to detect references that are being modified or assigned but have been marked `readonly`. Since these references are annotated `readonly` but are being modified, the read-only inferencer will not have recommended such annotations and they are likely to be provided by users or via other techniques (as described in the next section). Consider the `balance` method below:

```
class BankAccount {
```

[11]The case that read-only constraint variable is not found in the constraint set but the mutable context's constraint variable is not cannot occur.

```
    int balance;

    List<String> securityLog;

    int balance() readonly {
      methodLog.add("balance checked");
      return balance;
    }

}
```

In this example, the read-only inference algorithm finds the method `balance` (the parameter $\texttt{this}_{\texttt{balance}}$) to be mutable because it is used to mutate `this.securityLog`. However, since the user has specified that $\texttt{this}_{\texttt{balance}}$ should be read-only, our technique would need to recommend `securityLog` to be declared `mutable`.

When there are multiple field references leading to an assignment, there may be multiple ways of declaring fields to be `mutable` or `assignable` that would resolve the conflict. It is always more favorable to modify a field of the class containing the method in question than modifying a field of another class. For example, consider the code below:

```
class Cell {
   Object val;
}

class Event {
   Cell current;

   beginWeek() readonly {
      ((Date) current.val).day = Date.Monday;
   }

}
```

The technique will need suggest that either `current` or `val` is declared `mutable`; or `day` is declared `assignable`. However, changing the declarations of `val` or `day` is a non-local change; thus, it is preferred to modify `current`.

Approach: This technique needs to find all references being modified, find their cause, and suggest them to be annotated as `mutable` or `assignable`. The constraint-set provided by the read-only inferencer contains most of the information needed by the technique. As can be seen in in figures 3 and 6, whenever an object can be modified guarded constraints are added to the constraint-set. Thus, following the guarded constraints backwards from a modified reference annotated as read-only allows finding all references that need to be considered as being annotated `mutable`.

The backwards traversal would end in the actual modification, represented as an unguarded constraint. This happens when the object is being directly modified by assigning a field (the SET rule), unguarded constraints are added to the constraint-set. In such cases, we use an `objectMutatorFieldSet` to track the field which is being used to mutate the object, each unguarded constraint therefore has its causing field added to this set. Mutator fields found in this set indicates the fields that need to be considered as being annotated `assignable`.

Algorithm: Implementing support for this technique thus has two requirements. The read-only inferencer is modified to add the cause for every unguarded constraint, i.e. the direct mutation done by field assignments should result in a mapping being added to the the `objectMutatorFieldSet`. Then after the read-only inference is run, and constraints are solved, four simple steps are performed:

1. It checks if a user-annotated `readonly` reference is marked as `mutable`.
2. Goes backwards through constraint-set, from the reference to collect references which are modified causing the current reference to be mutable.
3. Maintain the order of collected fields — a breadth-first backwards traversal through the constraint-set lists the fields from the most recent and local causes to indirect causes.
4. Recommend the collected fields to the user for annotations, either as `mutable` or as `assignable`. If a field is added as the cause in `objectMutatorFieldSet` then it needs to be annotated as `assignable`, otherwise the field needs to be annotated as `mutable`.

Since user-provided annotations are by default inferred to be inherited by subclasses, this technique allows one to annotate a few common methods like `Object.hashCode` and `Object.toString` as `readonly` and the parameters of `PrintStream.print` as `readonly`, thereby resulting in all subclasses converting their overriding methods to also include the `readonly` annotation.

5.2 Heuristics and Cache Detection

The main cases where the `mutable` and `assignable` keywords are used is in annotating caches. There are two common traits of caches and are provided as heuristics:

1. Fields which are private and only referenced or assigned in a single method.
2. Fields using the Java `transient` keyword (designed to mark fields that should not be saved during object serialization).

The above heuristics use the technique in the previous section to determine if making the fields `mutable` or `assignable` will result in the corresponding methods becoming `readonly`. Only field annotations that will result in adding the `readonly` on the method are recommended to the user (in order to minimize the number of annotations the user has to consider, especially if providing such annotations do not result in improved annotations of other parts of the code).

The heuristics run on the results of the read-only inferencer. A search for all occurrences of cached fields and their respective methods are performed as suggested by the heuristics. The previous technique is then invoked, to traverse the guarded constraints are traversed in a backward manner from the searched method. If one of the potential output fields is a cache variable, then the field is recommended for the `mutable` or `readonly` annotation.

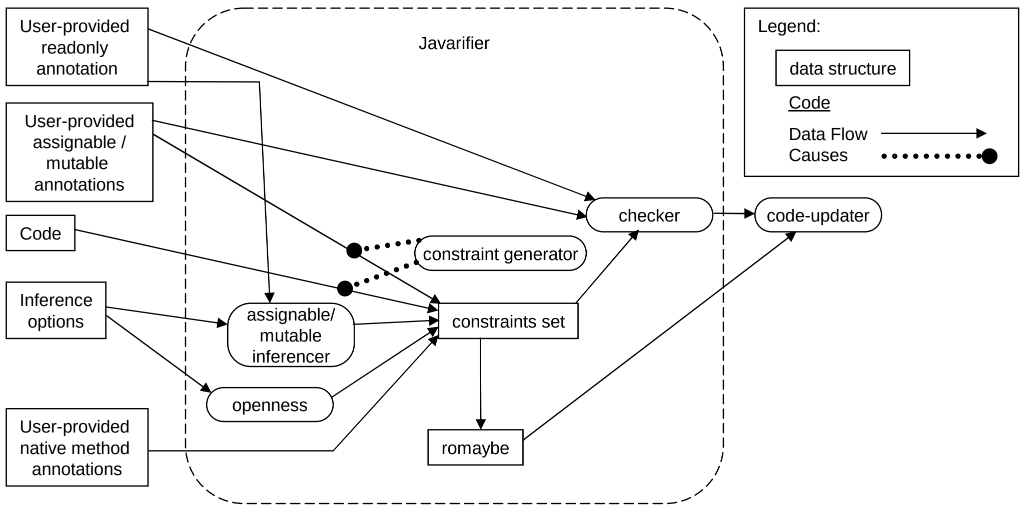
User-provided readonly annotation
User-provided assignable / mutable annotations
Code
Inference options
User-provided native method annotations
Javarifier
constraint generator
assignable/ mutable inferencer
openness
constraints set
romaybe
checker
code-updater
Legend:
data structure
Code
Data Flow
Causes

Figure 14: Data flow diagram of Javarifier.

6. IMPLEMENTATION

We have implemented the read-only and romaybe inference algorithm in the absence of arrays and parametric types. As mentioned in this paper, the algorithm can be extended to handle such constructs. Javarifier leverages user-provided annotations in its inference, and uses readonly annotations references towards inferring mutable and assignable keywords. Heuristics for cache detection are expected to be simple extensions but are currently not implemented.

To enable the annotation of library class-files in the absence of source code, Javarifier takes classfiles as input. Users can specify a number of command-line options to enable/disable various components of the inferencer, including running using the closed-world assumption and running the various mutable and readonly inferencers. Options also specify files containing external annotations, as well as files for annotations of library API's and native methods. Javarifier, processes these inputs and returns a list consisting of the mutability of all references (parameters, fields and locals). In the future work we discuss extending Javarifier to directly apply its results to class and source files.

As shown in Figure 14, Javarifier processes the various inputs and build a constraint set. We use the Soot library [**?**], to convert Java classfiles into a three-address programming language, for the constraint generator to build the constraints. The open-world mode is then checked and constraints are added to the constraint set. The constraint generator then leverages user-provided annotations during its inference, and beyond `readonly` inference also includes constraint information needed for `romaybe` inference in the constraint set. The constraint generator also also builds a set of fields causing the objects to be mutated, in order to support `assignable` and `mutable` inferences the constraint generator .

The constraints solver processes the constraints from the constraint generator. These solved constraints are then examined, by the assignable/mutable inferencer (if enabled) to detect if any fields are to be recommended for being annotated. A checker then ensures that user-provided readonly annotations do not conflict with the mutability recommendations as generated by the read-only inferencer, and reports conflicts to the user as errors.

7. EVALUATION

Our results show that our read-only inference algorithm is sound and precise with the exception of immutable classes.

7.1 Methodology

We evaluate Javarifier's read-only inferencing algorithm by comparing its results against hand-produced annotations. We have hand-annotated the key classes in Gizmoball, an implementation of a pinball game done for a MIT software engineering class. Over approximately 8 hours, we annotated 33 classes in the Gizmoball project, consisting of about 8000 lines of code. Due to time constraints, we eliminated classes that dealt with the graphical user interface. Those classes are not as interesting because they did not make use of abstract state. Instead, we chose classes that modeled the state of the pinball game, its elements and the physics behind the pinball game.

We annotated all non-primitive references under the open world assumption, that is, we annotated all public methods to return mutable references. This choice led to an interesting result for immutable objects. In the case of references to immutable objects used as a return type, we labelled the references as mutable although this label is counterintuitive: in the case of references to immutable objects, we know that these objects can never be mutated, either through a mutable or read-only reference. However, we cannot label these references as read-only, to ensure that any other client code that happens to use the method will type check. Furthermore, the specification of the Javari language [13], does not allow the `readonly` keyword to be applied to references to classes that are declared immutable.

However, our implementation does not actually infer immutable classes, and, thus, we can suffer a loss of precision when a field refers to object of an immutable class. In this case, our analysis would require any reference to the enclosing class to be mutable if accessing the field as a mutable type. This situation is imprecise, because the field's type should be treated the same whether reached through a mutable or read-only reference. To work around this imprecision, we declared all such fields `mutable`.

Another possible source of imprecision is method overloading, particularly in the case of interfaces. All overloaded methods must have the same method signatures. That means if one subtype mutates itself in a method, that mutating method annotation will be propagated up to the annotation of the interface and back down to all subtypes of the interface, even subtypes that do not actually mutate the receiver. Again, we not a loss of precision but not soundness. Additionally, this behavior is required by the Javari language.

To avoid annotating library code through Javarifier, we simulated library annotations by providing external library annotations to the Javarifier. For example, List.clear() has the signature `ro List.clear() mut`, meaning that the method will mutate its receiver and return a read-only object. For simplicity, we represented void returns and primitives as read-only. However, these library annotation simulations are not sound. Namely, the method annotations were provided based on our understanding of the library classes and whether we believed the parameters and fields of a library class would be mutated. The more sound approach is to actually look at the source code for these library classes before making the annotations.

The current implementation of Javarifier can not handle arrays and parametric types. Therefore, we converted all uses of arrays to `List`s and did not add any parameterized declarations (Gizmoball was written in Java 1.4), including mutabilities. Because of this limitation, for a collection class like `List` we can only have annotate for the actual list, but not for the parameterized type of the list. Therefore, the same mutability would need to be inferred for all `List` elements, a major source of imprecision. This limitation forced us to create two copies of the `List` class and other Collections classes. We created a `ListOfMutable` and a `ListOfReadonly` class in order to make the analysis of the Javarifier more precise.

The results of our hand-annotations are summarized in figure 15.

To evaluate the read-only inferencing algorithm, we provided the algorithm with the set of fields hand-annotated to be `assignable` or `mutable`. In addition, we provided a set of annotations for library method calls. An evaluation of Javarifier's performance is given in the next section.

7.2 Results

We ran Javarifier's read-only inference algorithm on Gizmoball under three-different set of conditions: without hand-annotated `assignable` and `mutable` fields, with hand-annotated fields, and under the closed world assumption with hand-annotated fields. The results of running Javarifier's read-only inference algorithm without providing it the hand-annotated `assignable` and `mutable` fields on are given in figure 16. In addition, the results of running Javarifier's inference algorithm along with providing a set of assignable and mutable fields, shown in figure 17. We also ran Javarifier on the sample project under the close-world assumption with the same set of assignable and mutable fields; these results are shown in figure 18.

Although we only hand-annotated Gizmoball under the second assumption— with `assignable` and `mutable` fields under the closed world assumption—in the other cases, we inspected the changes in Javarifier's output and found it to be correct.

Figure 19 shows a comparison of the hand annotations with the Javarifier results under the three conditions mentioned above. Using the hand annotations as the control, under the first condition (no hand-annotated `assignable`and `mutable`fields), the only difference is that there are more `mutable`annotations in the Javarifier results than the hand annotations because all methods that mutate the fields are labelled as mutable even if the field is not be part of the abstract state of the object. Under the second condition (user provided annotation of assignable and mutable fields), the hand annotations and Javarifier annotations match exactly, as expected. Under condition 3 (closed-world assumption), there are more `readonly`annotations in the Javarifier results as expected because public method are no longer to require return mutable references in all cases.

8. RELATED WORK

First we discuss effect analysis which is a related but orthogonal analysis to our read-only inference algorithm. Then we discuss two other techniques, type inference and the Houdini approach, that can be used to infer read-only references but have greater complexity.

8.1 Effect Analysis

Effect analyses for Java and in general [3, 11, 9, 10, 7, 6] have been widely studied. Similar to our algorithm's readonly parameters, effect analysis can be used to determine the set of a method's "safe" parameters [12]. A method parameter is safe if the method never modifies the object passed to the parameter during method invocation. Unlike read-only references, a parameter is not safe if the method never uses the parameter but instead mutates the object through a different aliasing reference. Additionally, if safe parameter is reassigned with a different object, the safe parameter may be used to mutate that object. A read-only parameter, on the other hand, may not be used to mutate any object. Safety and read-only-ness are orthogonal: safety is a property over objects while read-only-ness is a property over references.

Additionally, our algorithm has a much lower complexity than effect-analyses, which must be context-sensitive to achieve reasonable precision .

8.2 Type inference

Most approaches for type inference, use a constraint-based approach. A basic approach detailed in [8], defines a type inference algorithm for basic object-oriented languages. The

	ro	mutable	this-mut	romaybe	assignable
references	515	373	28	60	7
params	511	356	-	60	-
fields (non-primitives)	4	17	28	-	7
static fields (non-primitives)	1	16	-	-	0
instance fields (non-primitives)	3	1	28	-	7
method receivers	223	158	-	18	-
method params	277	85	-	12	-
method returns	11	113	-	30	-

Figure 15: Results of hand-annotating a software project.

	ro	mutable	this-mut	romaybe	assignable
references	221	690	31	34	0
params	219	674	-	34	-
fields (non-primitives)	2	16	31	-	0
static fields (non-primitives)	1	16	-	-	0
instance fields (non-primitives)	1	0	31	-	0
method receivers	151	231	-	17	-
method params	59	315	-	0	-
method returns	9	128	-	17	-

Figure 16: Results of Javarifier's read-only inference algorithm with no `assignable` and `mutable` external annotations.

	ro	mutable	this-mut	romaybe	assignable
references	515	373	28	60	7
params	511	356	-	60	-
fields (non-primitives)	4	17	28	-	7
static fields (non-primitives)	1	16	-	-	0
instance fields (non-primitives)	3	1	28	-	7
method receivers	223	158	-	18	-
method params	277	85	-	12	-
method returns	11	113	-	30	-

Figure 17: Results of Javarifier's read-only inference algorithm with `assignable` and `mutable` external annotations.

	ro	mutable	this-mut	romaybe	assignable
references	774	189	11	2	7
params	736	189	-	2	-
fields (non-primitives)	38	0	11	-	7
static fields (non-primitives)	17	0	-	-	0
instance fields (non-primitives)	21	0	11	-	7
method receivers	251	147	-	1	-
method params	350	24	-	0	-
method returns	135	18	-	1	-

Figure 18: Results of Javarifier's read-only inference algorithm under the closed world assumption and with `assignable` and `mutable` external annotations.

	params ro	params romaybe	params mutable	fields ro	fields this-mut	fields mut
Hand annotations	511	60	356	4	28	17
Javarifier without help	219	34	674	2	31	16
Javarifier with help	511	60	356	4	38	17
Javarifier closed world	736	2	189	38	11	0

Figure 19: Evaluation of Javarifier's read-only inference algorithm.

approach involves defining a set of type constraints on the program and propagating these constraints to a fixed point. In [1], the introduction of the Cartesian Product Algorithm builds upon basic type inference algorithm to handle parametric polymorphism. By using the Cartesian product to decompose the analysis of the core algorithm, CPA improves the precision, efficiency, generality and simplicity over the other algorithms while addressing the issue of parametric polymorphism. The Data Polymorphic CPA algorithm [14] expands on the CPA algorithm to address the issue of data polymorphism.

8.3 Houdini approach

An approach that has been used successfully for inferring annotations is the Houdini approach [5]. This approach could work by declaring all references `readonly`, then iteratively running a checker to find which annotations causes the checker to fail, and removing such annotations. Using such an approach for providing ESC/Java and rccjava (race conditions) [4] annotations have been found useful. For Javarifier, this approach would have poor performance as it would need to invoke the type-checker multiple times. Additionally, it is unclear if the approach could be used to infer the `romaybe` keyword.

9. FUTURE WORK

We are working on implementing inferencing in the presence of arrays and parametric types. We further plan on using Javarifier on real projects to determine the real-world usefulness of Javarifier, and determine any additional heuristics needed for making the tool practical.

Since many developers currently use the Eclipse IDE and the Ant build systems we plan on integrating Javarifier with these environments. We expect the benefits of the Javari language to be apparent even when the annotations are not provided in the source file and are in separate external files, at least in the few cases where library APIs indicate return types to be readonly. Future integration would involve adding the Javari keywords as part of the source file as either escaped comments such as `/*=readonly*/` or ideally by modifying the various parsers to provide transparent integration. We also plan on adding support for Javarifier to apply its results directly to class, so that separate files do not need to be provided by developers.

10. CONCLUSION

In this paper, we have created an algorithm for inferring read-only and romaybe types. We have implemented a useful subset of our algorithm within Javarifier and verified its effectiveness. We have also designed an heuristic to suggest assignable or mutable fields.

11. REFERENCES

[1] Ole Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming, Volume 952 of Lecture notes in Computer Science*, pages 2–26, London, UK, 1995. Springer-Verlag.

[2] Adrian Birka and Michael D. Ernst. A practical type system and language for reference immutability. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 35–49, New York, NY, USA, 2004. ACM Press.

[3] K. D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 57–66, New York, NY, USA, 1988. ACM Press.

[4] Cormac Flanagan and Stephen N. Freund. Detecting race conditions in large programs. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 90–96, New York, NY, USA, 2001. ACM Press.

[5] Cormac Flanagan, Rajeev Joshi, and K. Rustan M. Leino. Annotation inference for modular checkers. *Information Processing Letters*, 77(2-4):97–108, 2001.

[6] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for java. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 1–11, New York, NY, USA, 2002. ACM Press.

[7] Phung Hua Nguyen and Jingling Xue. Interprocedural side-effect analysis and optimisation in the presence of dynamic class loading. In *CRPIT '38: Proceedings of the Twenty-eighth Australasian conference on Computer Science*, pages 9–18, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc.

[8] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 146–161, New York, NY, USA, October 1991. ACM Press.

[9] C. Razafimahefa. A study of side-effect analyses for Java. Master's thesis, McGill University, December 1999.

[10] A. Rountev and B. Ryder. Practical points-to analysis for programs built with libraries. Technical Report DCSTR-410, Rutgers University, February 2000.

[11] Barbara G. Ryder, William A. Landi, Philip A. Stocks, Sean Zhang, and Rita Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Trans. Program. Lang. Syst.*, 23(2):105–186, 2001.

[12] Alexandru Salcianu and Martin Rinard. Purity and side effect analysis for java programs. In *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation*, 2005.

[13] Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2005)*, San Diego, CA, USA, October 16–20, 2005.

[14] Tiejun Wang and Scott F. Smith. Precise constraint-based type inference for Java. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming, Lecture notes in Computer Science 2072*, pages 99–117, London, UK, 2001. Springer-Verlag.