Inference of Reference Immutability

Jaime Quinonez, Matthew S. Tschantz, and Michael D. Ernst

MIT Computer Science and Artificial Intelligence Lab
Cambridge, MA, USA
{jaimeq, tschantz, mernst}@csail.mit.edu

Abstract. Javari is an extension of Java that supports reference immutability constraints. Programmers write `readonly` type qualifiers and other constraints, and the Javari typechecker detects mutation errors (incorrect side effects) or verifies their absence. While case studies have demonstrated the practicality and value of Javari, a barrier to usability remains. A Javari program will not typecheck unless all the references in the APIs of libraries it uses are annotated with Javari type qualifiers. Manually converting existing Java libraries to Javari is tedious and error-prone.

We present an algorithm for inferring reference immutability in Javari. The flow-insensitive and context-sensitive algorithm is sound and produces a set of qualifiers that typecheck in Javari. The algorithm is precise in that it infers the most `readonly` qualifiers possible; adding any additional `readonly` qualifiers will cause the program to not typecheck. We have implemented the algorithm in a tool, Javarifier, that infers the Javari type qualifiers over a set of class files.

Javarifier automatically converts Java libraries to Javari. Additionally, Javarifier eases the task of converting legacy programs to Javari by inferring the mutability of every reference in a program. In case studies, Javarifier correctly inferred mutability over Java programs of up to 110 KLOC.

1 Introduction

An immutability reference constraint, such as a `readonly` type qualifier, prevents a reference from being used to modify its referent objects (including their transitive state). Immutability constraints have many benefits: programmers can formally express intended properties of their code; explicit, machine-checked documentation enhances program understanding; static or dynamic checkers can detect errors or guarantee their absence; and analyses and transformations depending on compiler-verified properties are enabled. In practice, immutability constraints have been shown to be practical and to find errors in software.

Writing reference immutability annotations to obtain these benefits can be tedious and error-prone. An even more important motivation for immutability inference is the need to annotate the signatures of all used libraries. Otherwise, a sound reference immutability type checker would be forced to assume that all methods in these libraries modify their arguments. In particular, passing a `readonly` reference to any library method would be a type error.

J. Vitek (Ed.): ECOOP 2008, LNCS 5142, pp. 616–641, 2008.

Java

```
class Event {
  Date date;

  Date getDate() {
    return Date;
  }

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

Javari

```
class Event {
  /*this-mutable*/ Date date;

  polyread Date getDate() polyread {
    return Date;
  }

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

Fig. 1. A Java class (left) and the corresponding Javari class (right) that is automatically produced by Javarifier. Underlines indicate added immutability qualifiers. The figure shows default qualifiers in comments for clarity (Javarifier adds nothing in such cases). The qualifiers are explained in Section 2. A qualifier after the parameter list and before the opening curly brace annotates that method's receiver, similar to annotations on other parameters.

We have created an algorithm that soundly calculates reference immutability. Although our framework can accommodate other notions of reference immutability, for concreteness, this paper uses the reference immutability constraints of Javari [29]. Javari is an extension of Java with reference immutability type qualifiers (see Section 2).

This algorithm computes all the references (including local variables, method parameters, and static and instance fields) that may have Javari's `readonly`, `polyread`, or `? readonly` keywords added to their declarations. Figure 1 shows an example Java class and the corresponding inferred Javari class.

Our algorithm targets a realistic and fully-featured implementation of reference immutability, Javari. The algorithm infers the multiple annotations that are needed for an expressive language, including `readonly`, an extension to wildcards (`? readonly`), non-generics polymorphism (`polyread`), and containing-object context `this-mutable`.[1] Javari provides reference immutability guarantees over the abstract state of an object (see Section 2.1). The algorithm handles the complexities of the Java language, including subtyping, generics, arrays, and unseen code. The algorithm is sound and precise.

Javarifier is a scalable tool that implements this algorithm. Javarifier's input is a Java (or partially annotated Javari) program in classfile format, because programmers may wish to convert library code whose source is unavailable. The Javarifier toolset can insert the inferred qualifiers in source or class files, or present them to a user for inspection. If the user wants to refine the results, the user can insert any number of annotations in the program and run Javarifier in the presence of these annotations (see Section 3.3). All of the tools use the JSR

[1] As a pre-pass, the algorithm heuristically recommends fields to exclude from the abstract state of a class via the `assignable` or `mutable` field annotations; the user may accept some or all of the recommendations. Page limits prohibit an explanation of these heuristics [28, 22], though they are implemented in the Javarifier tool.

Type qualifiers	
`readonly`	The reference cannot be used to modify its referent
`/*mutable*/`	The reference may be used to modify its referent
`polyread`	Polymorphism (for parameters and return types) over mutability
`? readonly`	The type has a `readonly` upper bound and a `mutable` lower bound

Field annotations	
`/*this-mutable*/`	The field inherits its mutability from the reference through which it is reached
`assignable`	The field may be reassigned through a `readonly` reference
`mutable`	The field may be mutated through a `readonly` reference

Fig. 2. Javari keywords: type qualifiers and field annotations. Default keywords that are not written in a program are shown in comments.

308 [10] extension to Java annotations, which is backward-compatible and which is planned for inclusion in Java 7.[2]

The rest of this paper is organized as follows. Section 2 provides an overview of the Javari language for reference immutability. Sections 3–5 describe the algorithm: sound inference of reference immutability for ordinary readonly references (Section 3), arrays and generic types (Section 4), and `polyread` polymorphic references (Section 5). Section 6 reports experience using Javarifier. Section 7 discusses related work. Finally, Section 8 concludes.

2 The Javari Language: Java with Reference Immutability

Javari extends Java's type system to allow programmers to specify and statically enforce reference immutability constraints. This section briefly explains Javari's keywords, as listed in figure 2. The language is fully defined elsewhere [29, 28].

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:

```
readonly Date d = new Date();
d.setHours(9);                 // compile-time error
```

Mutation is any modification to an object's abstract state (see Section 2.1). References that are not `readonly` can be used to modify their referent and are said to be *mutable*. By Java's subtyping rules, a mutable reference can be used anywhere a `readonly` reference is expected, but a `readonly` reference cannot be treated as a mutable reference.

Javari handles generic type parameters in a natural way to account for the fact that every type now specifies its mutability. Below are four declarations of type `List`. The mutability of the parameterized type `List` does not affect the mutability of the type argument.

[2] To avoid explaining JSR 308, this paper uses keywords rather than annotations for the Javari type qualifiers.

```
/*mutable*/ List</*mutable*/ Date> ld1; // List: may add/remove; Date: may mutate
/*mutable*/ List<readonly    Date> ld2; // List: may add/remove
readonly    List</*mutable*/ Date> ld3; // Date: may mutate
readonly    List<readonly    Date> ld4; // (no side effects allowed)
```

As in Java, subtyping is invariant in terms of type arguments. Javari expresses the common supertype of `List</*mutable*/ Date>` and `List<readonly Date>` as `List<? readonly Date>`. The `? readonly` wildcard keyword is an extension to Java's wildcard mechanism. It specifies that `readonly Date` is the type argument's upper bound and `/*mutable*/ Date` is its lower bound. Elements are read from this type of list as `readonly`, but must be written to it as `mutable`. This type would be written as `List<? extends readonly Date super /*mutable*/ Date>`, except Java does not allow the declaration of both a lower and an upper bound on a wildcard.

The mutability wildcard is useful for the same reasons Java wildcards are. For example, a method that prints all the `Date`s in an input `List` can have a `List<? readonly Date>` parameter. If the parameter were declared as `List<readonly Date>`, a `List<mutable Date>` argument could not be passed in.

Javari keywords, including `? readonly`, apply to arrays analogously to parameterized types: each level of an array has its own mutability, and Javari arrays are invariant with respect to mutability.

The `polyread` keyword (see Figure 3) expresses parametric polymorphism over mutability. (`polyread` was previously named "`romaybe`" [29, 28].) The type checker conceptually duplicates any method containing a `polyread` keyword. In the first version of the method, all instances of `polyread` are replaced by `readonly`. In the second version, all instances of `polyread` are removed, so the references are mutable. Clients may use either version. `polyread` may occur on fields of method-local classes, and Javarifier inferred such annotations in our case studies. `polyread` is critical for precision; in the JDK, `polyread` is needed 70% as often as `readonly` [19]. `polyread` is not expressible in terms of Java generics; neither of `polyread` and `? readonly` subsumes the other [28].

2.1 Abstract State

By default, the abstract state of an object is its transitively reachable state, which is the state of the object and all state reachable from it by following references. Javari's deep reference immutability is achieved by giving each field the default annotation of `this-mutable`, which means the field inherits its mutability from the reference (`this`) through which it is accessed. Since it is the default, `this-mutable` is never written in a program.

The `assignable` and `mutable` keywords enable a programmer to exclude specific fields from an object's abstract state. The `assignable` keyword specifies that the field may always be reassigned, even through a `readonly` reference; Java's `final` keyword plays a related role, specifying that a field may not be reassigned at all through any reference once it has been set. The `mutable` keyword specifies that a field has mutable type (its own fields may be reassigned or mutated) even

```
class Bicycle {
  private Seat seat;
  polyread Seat getSeat() polyread { return seat; }
}

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

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

Fig. 3. The `polyread` keyword expresses polymorphism over mutability without polymorphism over the Java type. `lowerSeat` uses the mutable version of `getSeat` and takes a mutable `Bicycle` parameter. `printSeat` uses the readonly version of `getSeat` and can take a readonly `Bicycle` parameter. Without `polyread`, all the underlined annotations would be `/*mutable*/`. In particular, `printSeat` would take a mutable `Bicycle` parameter, and this imprecision could propagate through the rest of the program.

when referenced through a `readonly` reference. A `mutable` field's abstract value is not a part of the abstract state of the object (but the field's identity may be). Assignability and mutability of fields are orthogonal notions. Both are necessary to express code idioms such as caches, logging, and benevolent side effects, where not every field is part of the object's abstract state. For example, in the following class, the value of the `log` field is excluded from the abstract state of the object:

```
public class NetworkRouter {
  mutable List<String> log;

   // The readonly keyword indicates that the method does not modify its receiver
  public void selectRoute(String destination) readonly {
    log.add("selecting route to: " + destination);
  }
}
```

The (implicit, default) `mutable` type qualifier denotes that a reference may be used to modify its referent. The (explicit) `mutable` field annotation denotes that the field may always be used to modify its referent — it is excluded from the abstract state of the object.

3 Inferring Reference Immutability

Javarifier uses a flow-insensitive and context-sensitive algorithm to infer reference immutability. The algorithm determines which references may be declared

$$\begin{array}{lll}
\mathtt{Q} ::= & \mathtt{class}\ \{\overline{\mathtt{f}}\ \overline{\mathtt{M}}\} & \text{class def} \\
\mathtt{M} ::= & \mathtt{m}(\overline{\mathtt{x}})\{\overline{\mathtt{s;}}\} & \text{method def} \\
\mathtt{s} ::= & \mathtt{x} = \mathtt{x} & \text{statements} \\
\mid & \mathtt{x} = \mathtt{x.m}(\overline{\mathtt{x}}) & \\
\mid & \mathtt{return}\ \mathtt{x} & \\
\mid & \mathtt{x} = \mathtt{x.f} & \\
\mid & \mathtt{x.f} = \mathtt{x} &
\end{array}$$

Fig. 4. Grammar for core language used during constraint generation. $\overline{\mathtt{x}}$ is shorthand for the (possibly empty) sequence $\mathtt{x}_1 \ldots \mathtt{x}_n$. The special variable $\mathtt{this}_\mathtt{m}$ is the receiver of method `m`; it is treated as a normal variable, except that any program that attempts to reassign $\mathtt{this}_\mathtt{m}$ is malformed.

with `readonly` or other Javari keywords; other references are left as the default (`this-mutable` for fields, `mutable` for everything else). The algorithm is sound: Javarifier's recommendations type check under Javari's rules. Furthermore, the algorithm is precise: declaring any references in addition to Javarifier's recommendations as `readonly` — without other modifications to the code — will result in the program not type checking.

Section 3.1 describes the core inference algorithm. The algorithm extends to handle subtyping (Section 3.2); unseen code and pre-existing constraints including assignable and mutable fields (Section 3.3); arrays (Section 4.1); Java generics (Section 4.2); and mutability polymorphism (Section 5).

3.1 Core Algorithm

Given as input a program, Javarifier generates, then solves, a set of mutability constraints. A mutability constraint states when a given reference must be declared `mutable`. The core algorithm uses two types of constraints: unguarded and guarded. (Section 5 introduces a third variety of constraints, *double-guarded constraints.*) An unguarded constraint such as "`x`" states that a reference is unconditionally mutable. `x` is a *constraint variable* that refers to a Java reference or other entity in the code. A guarded constraint such as "$\mathtt{y} \rightarrow \mathtt{x}$" states that if `y` is mutable, then `x` is mutable; again, `x` and `y` are constraint variables.

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 by assignments and field dereferences.

We present constraint generation using a simple three-address core language (Figure 4). Control-flow constructs are not modeled, because the flow-insensitive algorithm is unaffected by such constructs. Java types are not modeled because the core algorithm does not use them. Constructors are modeled as regular methods returning a mutable reference to $\mathtt{this}_\mathtt{m}$. Static members are omitted because they do not illustrate any interesting properties. Without loss

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

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

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

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

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

Fig. 5. Constraint generation rules for the statements of Figure 4. Auxiliary functions this(m) and params(m) return the receiver reference ($\mathtt{this_m}$) and parameters of method m, respectively. retVal(m) returns $\mathtt{ret_m}$, the constraint variable that represents the reference to m's return value. *type*(x) returns the static type of x.

of generality, all references and methods have globally-unique names. (While this paper's formalism is simplified, the Javarifier implementation handles the full Java language.)

Each statement from Figure 4 has a constraint generation rule (Figure 5):

Assign The assignment of variable y to x causes the guarded constraint x → y to be generated because, if x is a mutable reference, y must also be mutable for the assignment to be valid.

Invk The constraints are extensions of the Assign rule when method invocation is viewed as pseudo-assignments or framed in terms of operational semantics: the receiver, y, is assigned to $\mathtt{this_m}$, each actual argument is assigned to the method's corresponding formal parameter, and the return value, $\mathtt{ret_m}$, is assigned to x.

Ret The return statement return x adds the constraint $\mathtt{ret_m} \rightarrow \mathtt{x}$ because, if 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, x → f, is required because, if x is mutable, then the field f cannot be readonly. The second, x → y, is required because, if x is mutable, then y must be mutable to yield a mutable reference to field f. (The core algorithm assumes all fields are this-mutable. Fields that have been manually annotated as mutable can override this behavior, as discussed in Section 3.3.)

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 f → y is added because if f, which is this-mutable, is ever read as mutable from a mutable reference, then a mutable reference must be assigned to it. If f is never mutated, the algorithm infers that it is readonly, in which case y is not constrained to be mutable.

```
class C {                                                class C {
  C f;                  field declaration                  readonly C f;
  C empty(P p)                                             readonly C empty(P p)
  {                                                          /*mutable*/ {
    C x = p;            ASSIGN: {x→p}                        /*mutable*/ C x = p;
    C y = x.f;          REF: {y → f, y→x}                    readonly    C y = x.f;
    C z = x.empty(y);     INVK: {this_empty→x,               readonly    C z = x.empty(y);
                          p→y, z→ret_empty}
    this.f = y;         SET: {this_empty, f→y}               this.f = y;
    return y;           RET: {ret_empty→y}                   return y;
  }                                                        }
  void empty(C p)                                          void empty(readonly C p)
  {                     no constraints to generate           readonly {
  }                                                        }
}                                                        }
                        Simplified program constraints:
                         {this_empty, x, p}
```

Fig. 6. Example of constraint generation and solving. The left part of the figure shows the original code. The center shows, for each line of code, the constraint generation rule used, the constraints generated, and the simplified program constraints — the references that may not be declared readonly. All the other references (y, z, $\texttt{ret}_{\texttt{empty}}$, and f) can be declared readonly, as shown in the Javarifier output on the right side of the figure.

The constraint set for a program is the union of the constraints generated for each line of the program. Figure 6 shows constraints for a sample program.

Constraint Solving The second phase of the algorithm solves the constraints by simplifying the constraint set. If any unguarded constraint satisfies (i.e., matches) the guard of a guarded constraint, then the guarded constraint is "fired" by removing it from the constraint set and adding its consequent to the constraint set as an unguarded constraint. Once no more constraints can be fired, constraint simplification terminates. This approach can be implemented with linear time complexity in the number of constraints (see Section 5.3 and [22]), and the Javarifier tool does so. If the guarded constraints are viewed as graph edges, then the core algorithm can be viewed as graph reachability starting at the unguarded constraints.

The unguarded constraints in the simplified constraint set must be declared `mutable` (or `this-mutable` in the case of instance fields). All other references may safely be declared `readonly`, since the algorithm propagated unguarded constraints to every reference that those constraints could reach. Thus, the algorithm excludes the maximum number of constraint variables from the unguarded constraint set when there are no field annotations. (Section 3.3 discusses how the `assignable` and `mutable` field annotations change the constraint generation rules, but they do not change the constraint solving step.) For a fixed set of initial field annotations, constraint solving therefore results in the maximum number of `readonly` references in the program. (Section 4.1 expands this argument to the other Javari qualifiers.) Constraint solving cannot fail, because the algorithm always terminates (see Section 5.3 and [22]) and in the worst case, every reference is mutable when the algorithm terminates.

Figure 6 shows the result of applying the algorithm to an example program.

3.2 Subtyping

Java and Javari allow subtyping polymorphism, which enables multiple implementations of a method to be specified through overriding[3]. Javari requires that overriding methods have covariant return mutability types and contravariant parameter mutability types (including the receiver, the implicit `this` parameter). To enforce these constraints, the algorithm adds the appropriate guarded constraints for every return and parameter of an overriding method. If a parameter is mutable in an overriding method, it must be mutable in the overridden method. If the return type is mutable in an overridden method, it must be mutable in the overriding method. For simplicity, a previous formalism [28] forced the mutabilities of overriding methods to be identical to the overridden method, but that is not required for correctness, and Javarifier does not do so.

3.3 Pre-existing Annotations and Unanalyzed Code

This section extends the inference algorithm to incorporate pre-existing annotations. These are useful for un-analyzable code such as native methods; for missing code, such as clients of a library, which might have arbitrary effects; and to permit users to override inferred annotations, such as when a reference is not currently used for mutation, but its specification permits it to be. Furthermore, user-provided annotations enable the algorithm to recognize which fields should be excluded from the abstract state of a class [28, 22].

This section first discusses pre-existing annotations that specify that a reference is either `readonly` or `mutable`. Then, it discusses annotations that exclude a field from the abstract state of the object.

Mutability Annotations A `readonly` annotation causes the algorithm, upon finishing, to check whether the reference may be declared `readonly`. If not, the algorithm issues an error. (Alternately, the algorithm can recommend code changes that permit the reference to be declared `readonly` [28, 22].)

A `mutable` type qualifier (not field annotation) or a `this-mutable` field annotation causes the algorithm to add an unguarded constraint that the reference is not readonly.

The algorithm has two modes. In *closed-world*, or whole-program, mode, the algorithm may change the type qualifiers of returned/escaped references, such as public method return types and types of public fields. This yields more precise results — that is, more `readonly` references. In *open-world* mode, the algorithm marks as mutable (i.e., adds an unguarded constraint for) every non-private field and non-private method return value. The open-world assumption is required

[3] We use the term *overriding* both for overriding a concrete method, and for implementing an abstract method or a method from an interface. For brevity and to highlight their identical treatment, we refer to both abstract methods and interface methods as *abstract methods*.

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

$$\frac{\texttt{mutable}(\texttt{f})}{\{\texttt{f}\}}\ (\textsc{Mutable})$$

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

Fig. 7. Modified constraint generation rules for assignable and mutable fields. The Set and Ref rules of Figure 5 are replaced by those of this figure. Mutable is new.

when analyzing partial programs or library classes with unknown clients, because an unseen client may mutate a field or return value.

Assignable and Mutable Fields Javarifier handles fields annotated as `mutable` or `assignable` by extending the constraint generation rules to check the assignability and mutability of fields before adding constraints. The auxiliary function `assignable(f)` returns true if and only if `f` is declared to be `assignable`; likewise for `mutable(f)`. The changes to the constraint generation rules are shown in Figure 7 and are described below.

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

Constraint generation rule Mutable adds an unguarded constraint for each `mutable` field.

The Ref rule is divided into two rules 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 the original Ref rule, (1) the consequence of the first constraint, $\texttt{x} \rightarrow \texttt{f}$, has already been added to the constraint set via the Mutable rule, and (2) the second constraint, $\texttt{x} \rightarrow \texttt{y}$, is eliminated because a `mutable` field is mutable regardless of how it is reached.

4 Arrays and Generics

This section discusses how to infer immutability for arrays and generic classes. (Javarifier also handles generic methods [28], but the details are omitted here for brevity.) The key difficulty is inferring the `? readonly` type, which requires

`s` ::= ...	`T, S` ::= `A` | `C`	*types*	`T, S` ::= `C<`$\overline{\texttt{T}}$`>` | `X`	*types*
| `x[x]` = `x`	`A, B` ::= `T[]`	*array types*	`C, D`	*class names*
| `x` = `x[x]`	`C, D`	*class names*	`X, Y`	*type variables*

Fig. 8. Core language grammar (Figure 4) extended for arrays (left). Constraint generation type meta-variables extended for arrays (center) and parametric types (right).

inferring two types (an upper and a lower bound) for each array/generic class. If the bounds are different, then the resulting Javari type is `? readonly`.

4.1 Arrays

This section extends the algorithm to handle arrays. First, we extend the core language grammar to allow storing to and reading from arrays (Figure 8).

A non-array reference has a single immutability annotation; therefore, a single constraint variable per reference suffices. Arrays need more constraint variables, for two reasons. First, an array reference's type may have multiple immutability annotations: the element type can be annotated in addition to the array itself. Second, Javari array elements have two-sided bounded types (Section 2). For example, the type `(? readonly Date)[]` has elements with upper bound `readonly Date` and lower bound `mutable Date`, and `(readonly Date)[]` has elements with identical upper bound and lower bound `readonly Date`.

Javarifier constrains each *part* of a type using a separate constraint variable. An array has parts for the top-level array type and for the upper and lower bounds of the element type. If the elements are themselves arrays, then there are parts for the upper and lower bounds of elements of the elements, and so on. For example, the type `Date[][]` has seven type parts: `Date[][]`, the top-level type; `Date[]`$_{\triangleleft}$, the upper bound of the element type, and `Date[]`$_{\triangleright}$, the lower bound of the element type; and four `Date` types corresponding to the upper/lower bound of the upper/lower bound.[4]

We subscript upper bounds with $_{\triangleleft}$ and lower bounds with $_{\triangleright}$. This matches the conventional ordering: in the declaration `List<? extends readonly Date super /*mutable*/ Date>`, the upper bound is on the left and the lower bound is on the right. We assume that within a program, textually different instances of the same type are distinguishable. The type meta-variables are shown in Figure 8. As usual, `T` and `S` range over types, and `C` and `D` over class names. We add `A` and `B` to range over array types.

The type constraint generation rules use the auxiliary function *type*, which returns the declared type of a reference, similar to the less intuitively named Γ type environment used in other work.

[4] An alternate approach of treating arrays as objects with fields of the same type as the array element type would not allow inferring different mutabilities on the different levels of the array. This alternate approach would not be able to infer the `? readonly` qualifier.

$$\frac{\texttt{S[]} \rightarrow \texttt{T[]} \qquad \texttt{T} \subset: \texttt{S}}{\texttt{T[]} <: \texttt{S[]}} \qquad \frac{\texttt{D} \rightarrow \texttt{C}}{\texttt{C} <: \texttt{D}} \qquad \frac{\texttt{T}_{\triangleleft} <: \texttt{S}_{\triangleleft} \qquad \texttt{S}_{\triangleright} <: \texttt{T}_{\triangleright}}{\texttt{T} \subset: \texttt{S}}$$

Fig. 9. Simplified subtyping (<:) rules for mutability in Javari. These simplified rules only check the mutabilities of the types, because we assume the program being converted type checks under Java. An array element's type, `T`, is said to be contained by another array element's type, `S`, written `T` ⊂: `S`, if the set of types denoted by `T` is a subset of the types denoted by `S`. Each rule states an equivalence between subtyping and guarded constraints on types, so each rule can be replicated with predicates and consequents swapped. Java arrays are covariant. Javari arrays are invariant in respect to mutability (see Section 2); therefore, we use the contains relationship as Java's parametric types do.

$$\texttt{x} = \texttt{y} : \{\textit{type}(\texttt{y}) <: \textit{type}(\texttt{x})\} \ (\textsc{Assign})$$

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

$$\frac{\texttt{retVal(m)} = \texttt{ret}_{\texttt{m}}}{\texttt{return x} : \{\textit{type}(\texttt{x}) <: \textit{type}(\texttt{ret}_{\texttt{m}})\}} \ (\textsc{Ret})$$

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

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

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

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

Fig. 10. Constraint generation rules extended for arrays. These rules replace the constraint generation rules of Figure 5, where the *type*() function was not needed.

Constraint Generation The constraint generation rules are extended to enforce subtyping constraints. 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 9.

The constraint generation rules now use types as constraint variables and enforce the subtyping relationship across assignments including the implicit pseudo-assignments that occur during method invocation. The extended rules are shown in Figure 10.

Type Well-formedness Constraints In addition to the constraints generated for each line of code, the algorithm adds constraints to the constraint set to ensure that every array type is well-formed. Array well-formedness constraints enforce that an array element's lower bound is a subtype of the element's upper bound.

Constraint Solving Before the constraint set can be simplified as before, subtyping ($<:$) and containment ($\subset:$) constraints must be reduced to guarded ($\rightarrow$) constraints. To do so, the algorithm replaces each subtyping or containment constraint by guarded constraints and/or simplified subtyping or contains constraints (see Figure 9). 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` have the types `T[]` and `S[]`, respectively, would generate and reduce constraints as follows:

$$\begin{aligned}
\mathtt{x} = \mathtt{y} &: \{\mathit{type}(\mathtt{y}) <: \mathit{type}(\mathtt{x})\} \\
&: \{\mathtt{S[]} <: \mathtt{T[]}\} \\
&: \{\mathtt{T[]} \rightarrow \mathtt{S[]},\ \mathtt{S} \subset: \mathtt{T}\} \\
&: \{\mathtt{T[]} \rightarrow \mathtt{S[]},\ \mathtt{S}_{\lhd} <: \mathtt{T}_{\lhd},\ \mathtt{T}_{\rhd} <: \mathtt{S}_{\rhd}\} \\
&: \{\mathtt{T[]} \rightarrow \mathtt{S[]},\ \mathtt{T}_{\lhd} \rightarrow \mathtt{S}_{\lhd},\ \mathtt{S}_{\rhd} \rightarrow \mathtt{T}_{\rhd}\}
\end{aligned}$$

In the final result, the first guarded constraint enforces that `y` must be a mutable array if `x` is a mutable array, while the second and third constraints constrain the bounds on the arrays' element types. $\mathtt{T}_{\lhd} \rightarrow \mathtt{S}_{\lhd}$ requires the upper bound of `y`'s elements to be mutable if the upper bound of `x`'s elements is mutable. This rule is due to covariant subtyping between upper bounds. $\mathtt{S}_{\rhd} \rightarrow \mathtt{T}_{\rhd}$ requires the lower bound of `x`'s elements to be mutable if the lower bound of `y`'s elements is mutable. This rule is due to contravariant subtyping between lower bounds.

After reducing all subtyping and containment constraints, the remaining guarded and unguarded constraint set is simplified as before. A subtype or containment constraint on an array type only leads to one guarded constraint for the top-level type and two guarded constraints for the lower and upper bounds. Compared to the non-array algorithm, the total number of constraints only increases by a constant factor that depends on the maximum array nesting. Therefore, the constraint simplification algorithm remains linear-time.

Applying Results Finally, the results must be mapped back to the initial Java program. Top-level types are annotated the same way they were before. However, for element types, the constraints on the type upper bound and type lower bound must map back to a single Javari type. Figure 11 illustrates this mapping.

As in Section 3.1, given a fixed set of field annotations, the algorithm excludes the maximum number of constraint variables from the unguarded constraint set. After the mapping of mutabilities on constraint variables to Javari types, no reference that is `? readonly` could be `readonly` because a mutable lower bound implies the reference cannot be `readonly` (since only `mutable` references can be assigned to it). Therefore, the algorithm infers the maximum number of references that do not need to be `mutable`, and each of these references is either `readonly` or `? readonly`.

Upper bound ($_\triangleleft$)	Lower bound ($_\triangleright$)	Javari type
`mutable`	`mutable`	`mutable`
`readonly`	`readonly`	`readonly`
`readonly`	`mutable`	`? readonly`

Fig. 11. The inferred mutability of the upper and lower bounds on array element types are mapped to a single Javari type. The case that the upper bound is `mutable` and the lower bound is `readonly` cannot occur due to the well-formedness constraints.

$$\mathit{asType}_\Delta(\texttt{C<}\overline{\texttt{T}}\texttt{>}, \texttt{C}) = \texttt{C<}\overline{\texttt{T}}\texttt{>}$$

$$\frac{\texttt{class C<}\overline{\texttt{X}}\ \overline{\texttt{V}}\texttt{>} \triangleleft \texttt{C}'\texttt{<}\overline{\texttt{U}}\texttt{>} \qquad \texttt{S} = \mathit{asType}_\Delta([\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{C}'\texttt{<}\overline{\texttt{U}}\texttt{>}, \texttt{D})}{\mathit{asType}_\Delta(\texttt{C<}\overline{\texttt{T}}\texttt{>}, \texttt{D}) = \texttt{S}}$$

Fig. 12. *asType* returns $\texttt{C<}\overline{\texttt{T}}\texttt{>}$'s supertype of class `D`.

4.2 Parametric Types (Java Generics)

Parametric types (Java generics) are handled similarly to arrays. For a parametric type, constraint variables are created for the upper and lower bound of each type argument to a parametric class. As with arrays, type parts serve as constraint variables.

The following meta-syntax represents parametric types. Figure 8 shows the type meta-variable definitions. As with arrays, $_\triangleleft$ denotes type arguments' upper bounds and $_\triangleright$ denotes their lower bounds.

Auxiliary Functions The subtyping rules use the auxiliary function bound_Δ. $\mathit{bound}_\Delta(\texttt{T})$ returns the declared upper bound of `T` if `T` is a type variable; if `T` is not a type variable, `T` is returned unchanged. In this formulation, there is a global type environment, Δ, that maps type variables to their declared bounds. *bound* ignores any upper bound ($_\triangleleft$) or lower bound ($_\triangleright$) subscripts on the type.

As with arrays, the type constraint generation rules use the auxiliary function *type*, which returns the declared type of a reference.

The subtyping rules use the $\mathit{asType}_\Delta(\texttt{C<}\overline{\texttt{T}}\texttt{>}, \texttt{D})$ function (Figure 12) to return `C`'s supertype of class `D`[5]. *asType* is used when a value is assigned to a reference that is a supertype of the value's type. In such a case, *asType* converts the value's type to have the same class as the reference. For example, consider

```
class Foo<T> extends List<Date> { ... }

Foo<Integer> f;
List<Date> lst = f;
lst.get(0).setMonth(JUNE);
```

[5] We call $\texttt{C<}\overline{\texttt{T}}\texttt{>}$ a type because its type arguments are present. We call `D` a class because type arguments are not provided.

$$\texttt{x} = \texttt{y} : \{\mathit{type}(\texttt{y}) <: \mathit{type}(\texttt{x})\} \text{ (A}\textsc{ssign}\text{)}$$

$$\frac{\texttt{this(m)} = \texttt{this}_{\texttt{m}} \qquad \texttt{params(m)} = \overline{\texttt{p}} \qquad \texttt{retVal(m)} = \texttt{ret}_{\texttt{m}}}{\texttt{x} = \texttt{y.m}(\overline{\texttt{y}}) : \{\mathit{type}(\texttt{y}) <: \mathit{type}(\texttt{this}_{\texttt{m}}),\ \mathit{type}(\overline{\texttt{y}}) <: \mathit{type}(\overline{\texttt{p}}),\ \mathit{type}(\texttt{ret}_{\texttt{m}}) <: \mathit{type}(\texttt{x})\}} \text{ (I}\textsc{nvk}\text{)}$$

$$\frac{\texttt{retVal(m)} = \texttt{ret}_{\texttt{m}}}{\texttt{return x} : \{\mathit{type}(\texttt{x}) <: \mathit{type}(\texttt{ret}_{\texttt{m}})\}} \text{ (R}\textsc{et}\text{)}$$

$$\texttt{x} = \texttt{y.f} : \{\mathit{type}(\texttt{f}) <: \mathit{type}(\texttt{x}),\ \mathit{type}(\texttt{x}) \rightarrow \mathit{type}(\texttt{y})\} \text{ (R}\textsc{ef}\text{)}$$

$$\texttt{x.f} = \texttt{y} : \{\mathit{type}(\texttt{x}),\ \mathit{type}(\texttt{y}) <: \mathit{type}(\texttt{f})\} \text{ (S}\textsc{et}\text{)}$$

Fig. 13. Constraint generation rules in the presence of parametric types.

$$\frac{\texttt{D} \rightarrow \texttt{C} \qquad \overline{\texttt{T}''} \subset: \overline{\texttt{S}'}}{\begin{array}{c}\texttt{T} <: \texttt{S} \textbf{ where } \mathit{bound}_{\triangle}(\texttt{T}) = \texttt{C<}\overline{\texttt{T}'}\texttt{>} \textbf{ and } \mathit{bound}_{\triangle}(\texttt{S}) = \texttt{D<}\overline{\texttt{S}'}\texttt{>} \textbf{ and}\\ \mathit{asType}_{\triangle}(\texttt{C<}\overline{\texttt{T}'}\texttt{>}, \texttt{D}) = \texttt{D<}\overline{\texttt{T}''}\texttt{>}\end{array}}$$

$$\frac{\texttt{T}_{\triangleleft} <: \texttt{S}_{\triangleleft} \qquad \texttt{S}_{\triangleright} <: \texttt{T}_{\triangleright}}{\texttt{T} \subset: \texttt{S}}$$

Fig. 14. Simplified subtyping rules for mutability in the presence of parametric types.

On the assignment of `f` to `lst`, *asType* converts `f`'s type from `Foo<Integer>` to `List<Date>` with the call: $\mathit{asType}_{\triangle}(\texttt{Foo<Integer>}, \texttt{List})$. This conversion ensures that constraints placed on the type of `lst` elements affect `f` indirectly through the type of `lst` rather than the type of `f`, so the final inference result is `class Foo<T> extends List</*mutable*/ Date>` rather than the incorrect `Foo</*mutable*/ Integer> f`.

Constraint Generation As with arrays, the constraint generation rules (shown in Figure 13) use subtyping constraints. However, the subtyping rules (shown in Figure 14) are extended to handle type variables. In Javari, a type variable is not allowed to be annotated as `mutable`; therefore, type variables cannot occur in the constraint set. In the case of a type variable appearing in a subtyping constraint, *bound* is used to calculate the upper bound of the type variable, and the mutability constraints are applied to the type variable's bound. Therefore, mutation of a reference whose type is a type variable results in the type variable's bound being constrained to be mutable. An example of this behavior is shown in Figure 15.

Type Well-formedness Constraints As with arrays, in addition to the constraints from the constraint generation rules, well-formedness constraints are added to the constraint set. As before, a constraint is added that a type argument's lower bound must be a subtype of the type argument's upper bound. Parametric types, additionally, introduce the well-formedness constraint that a

```
class Week<X extends /*mutable*/ Date> {
  X f;
  void startWeek() {
    f.setDay(Day.SUNDAY);
  }
}
```

Fig. 15. The result of applying type inference to a program containing a mutable type variable bound. Since the field `f` is mutated, `X`'s upper bound is inferred to be `/*mutable*/ Date`. The mutable annotation may not be applied directly to `f`'s type because in Javari, a type parameter cannot be annotated as `mutable`.

type argument's upper bound (and, therefore, by transitivity, lower bound) is a subtype of the corresponding type variable's declared upper bound.

Constraint Simplification and Applying Results As with arrays, subtyping (and containment) constraints are simplified into guarded constraints by removing the subtyping constraint from the constraint set and replacing it with the subtyping rule's predicate. The results of the solved constraint set are applied in the same manner as with arrays. Javari does not allow raw types, and this analysis is incapable of operating on code that contains raw types. In particular, this algorithm does not account for the required casts when using raw types.

5 Inferring Mutability Polymorphism

This section extends the inference algorithm to infer the `polyread` keyword (previously named "`romaybe`" [29]). As described in Section 2 and illustrated in Figure 3, `polyread` enables more precise and useful immutability annotations to be expressed than if methods could not be polymorphic over mutability.

5.1 Approach

Methods that have at least one `polyread` parameter or return type have two contexts. In the first context, all `polyread` references are `mutable`. In the second context, all `polyread` references are `readonly`. Javarifier creates both contexts for every method. If a parameter/return type has an identical mutability in both contexts, then that parameter/return type should have that mutability. If a parameter/return type is `mutable` in the `mutable` context and `readonly` in the `readonly` context, then that parameter/return type should be `polyread`.

To create two contexts for a method, Javarifier creates two constraint variables for every method-local reference (local variables, return value, and parameters, including the implicit `this` parameter). To distinguish each context's constraint variables, we superscript the constraint variables from the readonly context with `ro` and those from the mutable context with `mut`. Constraint variables for fields are not duplicated: `polyread` may not be applied to fields and, thus, only a single context exists.

Section 5.3 demonstrates that inferring `polyread` only requires increasing the number of constraints (and the time complexity of the algorithm) by a constant factor.

5.2 Constraint Generation Rules

With the exception of Invk, all the constraint generation rules are the same as before, except now they generate (identical) constraints for constraint variables from both the readonly and mutable versions of the methods. For example, `x = y` now generates the constraints $\{\mathtt{x}^{\mathtt{ro}} \to \mathtt{y}^{\mathtt{ro}}, \mathtt{x}^{\mathtt{mut}} \to \mathtt{y}^{\mathtt{mut}}\}$.

Thus, there are now two constraint variables for every reference, one for when it is in a mutable context and one for when it is in a readonly context. For shorthand, we write constraints that are identical with the exception of constraint variables' contexts by superscripting the constraint variables with "$^{?}$". For example, the constraints generated by `x = y` can be written as $\{\mathtt{x}^{?} \to \mathtt{y}^{?}\}$.

The method invocation rule (shown in Figure 16) must be modified to invoke the `mutable` version of a method when a `mutable` return type is needed, and to invoke the `readonly` version otherwise. This restriction can be represented using double-guarded constraints. For example, consider the code in Figure 3, in which the `Bicycle.getSeat()` method has a `polyread` return type and a `polyread` parameter. In the `lowerSeat()` method, the returned reference is mutated, so the `mutable` version of `getSeat()` must be used. In the `printSeat()` method, the returned reference is indeed `readonly`, so the `readonly` version of `getSeat()` can be used.

The first constraint in the invocation rule of Figure 16 thus states that if the returned reference `s` is `mutable`, then the reference `b` on which (the `mutable` version of) `getSeat()` is called must be `mutable` if the receiver of `getSeat()` is `mutable` inside the `mutable` version of `getSeat()`. (Recall that the receiver inside a `readonly` method is `readonly` in both the `mutable` and `readonly` versions of that method, whereas the receiver of a `polyread` method is `mutable` only in the `mutable` version of the method.)

In matching Figure 3 to the invocation rule of Figure 16, note that the $^{?}$ superscripts would be on the references `s` and `b` local to `lowerSeat()` (or `printSeat()`), whereas the explicit $^{\mathtt{mut}}$ superscript would only occur on references local to `getSeat()`. In particular, since the `lowerSeat()` and `printSeat()` methods are static, they only have one context so the different versions of duplicated constraint variables will always be the same. The $^{?}$ superscripts demonstrate that after fixing the explicit $^{\mathtt{mut}}$ contexts, these constraints are generalized with $^{?}$ in the same fashion all other constraints are generalized.

The last constraint in the invocation rule states that if the reference `s` is later mutated, then the return type of `getSeat()` must be `mutable` in the `mutable` version of `getSeat()`. The Ret rule for return types and Ref rule for field references in Figure 5 together generate the constraint that if the return type of `getSeat()` is mutable (in whichever version of the method is called), then the receiver of `getSeat()` is mutable (in that version of the method). Since the method invocation rule in Figure 16 only generates the constraint that the return

$$\frac{\texttt{this(m)} = \texttt{this}_{\texttt{m}} \qquad \texttt{params(m)} = \overline{\texttt{p}} \qquad \texttt{retVal(m)} = \texttt{ret}_{\texttt{m}}}{\texttt{x} = \texttt{y.m(}\overline{\texttt{y}}\texttt{)} : \{\texttt{x}^{?} \rightarrow \texttt{this}_{\texttt{m}}^{\texttt{mut}} \rightarrow \texttt{y}^{?},\ \texttt{x}^{?} \rightarrow \overline{\texttt{p}^{\texttt{mut}}} \rightarrow \overline{\texttt{y}^{?}},\ \texttt{x}^{?} \rightarrow \texttt{ret}_{\texttt{m}}^{\texttt{mut}}\}} \text{ (INVK-POLYREAD)}$$

Fig. 16. The core algorithm's INVK rule (Figure 5) is replaced by INVK-POLYREAD, which is used for method invocation in the presence of `polyread` references. Each superscript denotes the contexts of the method in which the variable is declared. All of the $^{?}$ contexts refer to the method containing the references `x` and `y`, whereas the explicit $^{\texttt{mut}}$ contexts refer to context inside method `m`.

type of `getSeat()` is `mutable` in the `mutable` version of `getSeat()`, the return type and receiver of `getSeat()` are `mutable` only in the `mutable` version of the method, and thus they are both inferred to be `polyread`.

5.3 Constraint Solving

The algorithm for solving the constraint set extends the algorithm of Section 3.1 to account for double-guarded constraints. We now provide the full algorithm and demonstrate that it has linear time complexity.

There are three constraint sets: the unguarded constraint set ($\mathcal{U}$) which contains constraints of the form a, the guarded constraint set ($\mathcal{G}$) which contains constraints of the form $a \rightarrow b$, and the double-guarded constraint set ($\mathcal{D}$) which contains constraints of the form $a \rightarrow b \rightarrow c$. The following pseudocode illustrates how the algorithm processes constraints using a work-list ($\mathcal{W}$) of unguarded constraints:

```
initialize $\mathcal{W}$ with all the constraints from $\mathcal{U}$
while $\mathcal{W}$ is not empty
  pop a constraint $a$ from $\mathcal{W}$
  for each constraint $g$ in $\mathcal{G}$ that has $a$ as its guard
    let $c$ be the consequent of $g$
    if $c$ is not in $\mathcal{U}$, add $c$ to $\mathcal{W}$ and to $\mathcal{U}$

  for each double-guarded constraint $d$ in $\mathcal{D}$ that has $a$ as its first guard
    let $b \rightarrow c$ be the consequent of $d$
    if $b$ is in $\mathcal{U}$
      if $c$ is not in $\mathcal{U}$, add $c$ to $\mathcal{W}$ and to $\mathcal{U}$
    else, add $b \rightarrow c$ to $\mathcal{G}$
```

The algorithm maintains linear time complexity if the sets $\mathcal{G}$ and $\mathcal{D}$ are implemented as hash tables. For $\mathcal{G}$, the table maps a guard to the constraint variable it guards. For $\mathcal{D}$, the table maps the first guard to a set of the consequents (which are single-guarded constraints) that it guards. That is, given constraints $a \rightarrow b_1 \rightarrow c_1$ and $a \rightarrow b_2 \rightarrow c_2$, the hash table maps a to the set $\{b_1 \rightarrow c_1, b_2 \rightarrow c_2\}$. This allows looking up all single-guarded constraints that are guarded by the same guard in a double-guarded constraint to take constant time, in expectation. Since every constraint is read from either $\mathcal{G}$ or $\mathcal{D}$ at most once, and each double-guarded constraint only adds one single-guarded constraint to

$\mathcal{G}$, the constraint-solving algorithm has linear time complexity in the total number of constraints. The number of constraints is linear in the size of the program under analysis as measured in the three-address core language of Figure 4.

5.4 Interpreting the Simplified Constraint Set

Once the constraint set is solved, the results are applied to the program. For method-local references, the two constraint variables from the readonly and mutable method contexts must be mapped to a single method-local Javari type: readonly, mutable, or polyread.

A reference is declared mutable if both the mutable and readonly contexts of the reference's constraint variable are in the simplified, unguarded constraint set. A reference is declared readonly if both mutable and readonly contexts of the reference's constraint variable are absent from the constraint set. Finally, a reference is declared polyread if the mutable context's constraint variable is in the constraint set but the readonly constraint variable is not in the constraint set, because the mutability of the reference depends on which version of the method is called.[6] Thus, in the example of Figure 3, after the constraints have been solved, the receiver of getSeat() is known to be mutable in a mutable context but not known to be mutable in a readonly context, so it is annotated as polyread. The reference returned by getSeat() is similarly known to be mutable in a mutable context but not known to be mutable in a readonly context, so it is also annotated as polyread.

It is possible for a method to contain polyread references but no polyread parameters. For example, below, x and the return value of getNewDate could be declared polyread.

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

However, polyread references are only useful if the method has a polyread parameter. Thus, if none of a method's parameters (including the receiver) are polyread, all the method's polyread references are converted to mutable references.

6 Evaluation

We have implemented the inference algorithm described in Sections 3–5 as a tool, Javarifier, that reads a set of classfiles, determines the mutability of every reference in those classfiles, and inserts the inferred Javari annotations in either

[6] The case that the readonly constraint variable is found in the constraint set, but the mutable context's constraint variable is not, cannot occur by the design of the Invk-Polyread constraint generation rule.

	Size			Annotatable references					
Program	lines	classes	Time	Total	`readonly`	mutable	this-mut.	`polyread`	`? readonly`
JOlden	6223	57	9	1580	927	553	52	48	0
tinySQL	30691	119	47	5606	2227	2964	175	240	0
htmlparser	63780	238	45	4596	1623	2740	72	144	17
ejc	110822	320	1410	24899	8887	14774	690	548	0

Fig. 17. Subject programs used in our case studies. Inference time is in seconds on a Pentium 4 3.6GHz machine with 3GB RAM. The right portion tabulates the number of annotatable references for each inference result (in Javarifier's closed-world mode). When counting annotatable references, each type argument counts separately; for example, `List<Date>` is counted as two references.

class files or Java source files. Javarifier is publicly available for download at `http://pag.csail.mit.edu/javari/javarifier/`.

To verify that Javarifier infers correct and maximally precise Javari qualifiers we performed two types of case studies. The first variety (Section 6.1) compared Javarifier's output to manually written Javari code that had been type-checked by the Javari type-checker. The second variety (Section 6.2) compared Javarifier to another tool for inferring immutability. For both varieties of case study, we examined every difference among the annotations. The case studies revealed no errors in Javarifier. It is possible that errors in Javarifier were masked by identical errors in the other tools and the manual annotations, but we consider this unlikely.

Figure 17 gives statistics for the subject programs used in our case studies:

- JOlden benchmark suite (`http://osl-www.cs.umass.edu/DaCapo/benchmarks.html`)
- tinySQL database engine (`http://www.jepstone.net/tinySQL/`)
- htmlparser library for parsing HTML (`http://htmlparser.sourceforge.net/`)
- ejc compiler for the Eclipse IDE (`http://www.eclipse.org/`)

The JOlden benchmark suite is written using raw types, so we first converted the source code to use generics. We also renamed some identically named but distinct classes in the different benchmarks within JOlden.

6.1 Comparison to Manual Annotations

Before the Javarifier implementation was complete, a developer (not one of the authors of this paper) manually annotated the JOlden benchmark suite and verified the correctness of the annotations by running the Javari type-checker. We compared the manually-written, automatically-verified annotations with Javarifier's inference results.

There were 74 differences between the manual annotations and Javarifier's output. 58 are human errors, and 16 disappear when using Javarifier's inference of `assignable` fields.

Program	inheritance	`polyread`	`this-mutable`	arrays
tinySQL	0	3	6	0
htmlparser	12	6	0	2
ejc	1	0	17	31

Fig. 18. Reasons for differences between Javarifier and Pidasa inference results. None of the differences indicates an error in Javarifier.

The programmer omitted 22 `readonly` qualifiers, such as on the receiver of `toString()`. Tool support while the programmer was annotating the program would have both eased the annotation task and prevented these errors.

Javarifier inferred 36 private fields to be `readonly`, while the developer accepted the default of `this-mutable`, meaning that the fields are part of the abstract state of the object. However, all 36 of these fields are either never read or are only used to store intermediate values that do not need to be mutated. Thus, Javarifier pointed out that these fields can be excluded from the abstract state, or even removed altogether, without affecting the rest of the program.

The remaining 16 annotations that differed between the manual annotations and Javarifier's results do not represent any conceptual errors, and when we enabled heuristics for inferring `assignable` fields [22], Javarifier's results were identical to the manual annotations. The developer had marked 4 fields as `assignable`. Each of these fields is a placeholder for the current element in an `Enumeration` class. The `assignable` annotation allowed the `nextElement()` method, which reassigns the field, to have a `polyread` receiver and return type. In other words, the manual annotations differentiate the abstract state from the concrete state of an object. When run without inference of assignable fields, Javarifier inferred that the return type is `readonly` and the receiver is `mutable`, and this mutability propagated to other methods, for a total of 16 differences in annotations.

6.2 Comparison to Another Mutation Inference Tool

Pidasa [3] is a combined static and dynamic immutability inference tool for parameters and receivers. Pidasa uses a different but closely related definition of reference immutability. We compared Javarifier's results to Pidasa's results on four randomly-selected classes from each of tinySQL, htmlparser, and ejc (for more details, see Artzi et al. [4]). We manually analyzed each difference to verify the correctness of Javarifier's results.

All of the differences can be attributed to four causes, as tabulated in Figure 18. The first three causes are conservatism in the Javari type system which makes it impossible to express that a particular reference is not mutated. The last cause is inflexibility in Pidasa that prevents it from expressing different mutabilities on arrays and their elements.

Inheritance: In 13 cases, Javarifier inferred a method receiver to be mutable due to contravariant receiver mutability in Javari, even though Pidasa was able to recognize contexts in which the receiver could not be mutated. Figure 19 gives an example.

```
1  class TagNode {
2    private List<Attribute> mAttributes;
3    public /*mutable*/ List<Attribute> getAttributes() /*mutable*/ {
4      return mAttributes;
5    }
6    public String toHtml() /*mutable*/ {
7      String s = "";
8      for(Attribute attr : getAttributes()) {
9        s += attr.toHtml();
10     }
11     return s;
12   }
13 }
14
15 class LazyTagNode extends TagNode {
16   public /*mutable*/ List<Attribute> getAttributes() /*mutable*/ {
17     // Actually mutates the abstract state of the object,
18     // in accordance to the specification for this class.
19   }
20 }
```

Fig. 19. Inheritance conservatism in the Javari type system, as observed in simplified code from the htmlparser program. The method `LazyTagNode.getAttributes()` is inferred to have a `mutable` receiver (line 16) because it may change the state of its receiver. The method subtyping rule thus forces `TagNode.getAttributes()` to have a `mutable` receiver (line 3). Since `TagNode.toHtml()` calls `getAttributes()` (line 8), it must also have a `mutable` receiver (line 6), even though not every call to `toHtml()` can cause a mutation.

`polyread`: In 9 cases, Javarifier inferred a parameter to be `mutable` due to the type rules of the `polyread` qualifier, but Pidasa inferred the parameter to be readonly. A method such as `filter(polyread Date)` cannot mutate its `polyread` parameter because the method would not typecheck when all `polyread` qualifiers are replaced with `readonly`. However, when `filter` is called from another method (from the same class) that has a `mutable` receiver, the type of `this` is `mutable` and thus Javari requires that the program typecheck as if the `filter` method took a `mutable` parameter.

`this-mutable`: In 23 cases, Javarifier inferred a `mutable` parameter due to Javari's type rule that `this-mutable` fields are always written to as `mutable`, but Pidasa inferred the parameter to be readonly. For example, if a method stores a parameter into a `this-mutable` field, that parameter must be declared `mutable`, even if no mutations occur to it.

Arrays: In 33 cases, Javarifier correctly inferred an array type to be partly immutable, but Pidasa was conservative and marked the whole array as mutable. For example, htmlparser used two readonly arrays of mutable objects. Javarifier correctly inferred the outer level of the arrays to be `readonly` and the inner level to be `mutable`. Pidasa infers a single mutability for all levels of the array. Ejc contained examples of mutable arrays of readonly objects.

In conclusion, we found differences among the tools' definitions, but in every case Javarifier inferred correct Javari annotations, even where the results are not immediately obvious — another advantage of a machine-checked immutability definition such as that of Javari.

7 Related Work

Our full inference algorithm, and experience with a preliminary Javarifier implementation, first appeared as part of Tschantz's thesis [28]. This paper builds upon that work with an extensive experimental evaluation.

In subsequent work, JQual [14] cites Tschantz's thesis and adopts our approach. JQual's core rules are essentially identical to Javarifier's. Like Javarifier, JQual uses syntax-directed constraint generation, then solves the constraints using graph reachability, and reports limited experimental results. However, there are some differences in the approaches. (1) Polymorphism: JQual discards our support for Java generics, and with it any hope for compatibility with the Java language. Instead, JQual generalizes our mutability polymorphism. Whereas `polyread` introduces exactly one mutability parameter into a method definition, JQual supports an arbitrary number. Given support for Java generics, we have not yet found a need for multiple mutability parameters. (2) Expressiveness: JQual generalizes Javarifier by being able in theory to infer any type qualifier, not just ones for reference immutability. This generality comes with a cost. JQual is tuned to simple "negative" and "positive" qualifiers that induce subtypes and supertypes of the unqualified type; it appears too inexpressive for richer type systems. JQual was used to create an inference tool for a @`ReadOnly` qualifier, but it lacks support for every other Javarifier keyword, for qualifiers on different levels of an array, for immutable classes, and for various other features of Javari. Additionally, it has a limitation on inheritance that ignores qualifiers in determining method overriding: it does not enforce the constraint, required for backward compatibility with Java, that mutability qualifiers do not affect overriding. (3) Scalability: Context- and flow-sensitive variants of the JQual algorithm exist, but the authors report that they are unscalable, so in their experiments they hand-tuned the application of these features. Even so, JQual has not been run on substantial codebases, and, except for JOlden, crashed on all of our subject programs. By contrast, both Javarifier's algorithm and its implementation are scalable. (4) Evaluation: JQual's output and input languages differ (e.g., it has no surface syntax for its parametric polymorphism), so its analysis results do not type check even in JQual. Artzi et al. [4] report that JQual's recall (fraction of truly immutable parameters that were inferred to be immutable) was 67%, compared to 94% recall for a version of Javarifier without inference of `assignable` or `mutable` fields. JQual misclassifies a receiver as mutable in method `m` if `m` reads a field `f` that is mutated by any other method. JQual also suffered a few errors in which it misclassified a mutable reference as immutable.

Javarifier and JQual can be viewed as extensions of the successful CQual [12, 13] type inference framework for C to the object-oriented context.

Constraint-based type inference has also been used for inferring atomicity annotations to detect races [7, 11], inferring non-local aliasing [1], and supporting type qualifiers dependent on flow-sensitivity (like `read`, `write`, and `open`) [13].

Pidasa [3] is a combined static and dynamic analysis for inferring parameter reference immutability. Pidasa uses a pipeline of (intra- and interprocedural) stages, each of which improves the results of the previous stage, and which can leave a parameter as "unknown" for a future stage to classify. This results in a system that is both more scalable and precise than previous work. Pidasa has both a sound mode and also unsound heuristics for applications that require higher precision and can tolerate unsoundness. By contrast, our work is purely static, making it sound but potentially less precise. Another contrast is that our definition is more expressive: our inference determines reference immutability for fields and for Java generics/arrays. Artzi et al. [4] compare both the definitions and the implementations of several tools including Javarifier, Pidasa, and JQual.

JPPA [27] is a previous reference immutability inference implementation. (Sălcianu also provides a formal definition of parameter safety, but JPPA implements reference immutability rather than parameter safety.) JPPA uses a whole-program pointer analysis, limiting scalability. Earlier work by Rountev [24] takes a similar approach but computes a coarser notion of side-effect-free methods rather than per-parameter mutability.

Reference immutability is distinct from the related notions of object immutability and of parameter "safety" [27]; none of them subsumes the others. They are useful for different purposes; for example, reference immutability is effective for specifying interfaces that should not modify their parameters (even though the caller may do so), and for a variety of other purposes [29]. A method parameter is safe if the method never modifies the object passed to the parameter during method invocation. Effect analyses [8, 26, 23, 25, 17, 16] can be used to compute safety or object immutability, often with the assistance of a heavyweight context-sensitive pointer analysis to achieve reasonable precision. (Like type qualifier inference, points-to analysis aims to determine the flow of objects or values through the program.) Our algorithm is much more scalable — the algorithm is flow-insensitive, and the base algorithm is context-insensitive — but is tuned to take advantage of the parametric polymorphism offered by both Java and Javari.

Porat et al. [21] and Liu and Milanova [15] propose immutability inference for fields in Java, the latter in the context of UML, but their definitions differ from ours.

Our focus in this paper is on inference of reference immutability. For reasons of space, we cannot review the extensive literature proposing different variants of immutability. We briefly mention type checkers for closely related notions of reference immutability. Birka built a type-checker for an earlier dialect of Javari that lacked support for Java generics, and wrote 160,000 lines of code in Javari [6]. Correa later wrote a complete Javari implementation using the Checker Framework [20] and did case studies involving 13,000 lines of Javari [19]. The JQual inference system [14] (discussed above) can be treated as a type checker. JavaCOP [2] is a framework for writing pluggable type systems for Java. Like JQual, JavaCOP aims for generality rather than practicality. Also like JQual, JavaCOP has been used to write a type checker for a small subset of Javari. The checker handles only one keyword (`readonly`) and cannot verify even that one in the presence of method overriding. Neither the checker nor any example output is publicly available, so it is difficult to compare to our work. Other frameworks that could be used for writing pluggable type systems include JastAdd [9], JACK [5], and Polyglot [18].

8 Conclusion

This paper presents an algorithm for statically inferring the reference immutability qualifiers of the Javari language. Javari extends the full Java language (including generics, wildcards, and arrays) in a rich and practical way: for example, it includes parametric polymorphism over mutability and permits excluding fields from an object's abstract state. To the best of our knowledge, ours is the first inference algorithm for a practical definition of reference immutability.

The algorithm is both sound and precise. Its correctness has been experimentally confirmed. The experiments also show that, like any conservative static type system, the Javari language's definition sometimes requires a reference to be declared mutable even when no mutation can occur at run time.

The Javarifier tool infers immutability constraints and inserts them in either Java source files or class files. Javarifier solves two important problems for programmers who wish to confirm that their programs are free of (a large class of) mutation errors. First, it can annotate existing programs, freeing programmers of that burden or revealing errors. Second, it can annotate libraries; because the Javari checker conservatively assumes any unannotated reference is mutable, use of any unannotated library makes checking of a program that uses it essentially impossible. Together, these capabilities permit programmers to obtain the many benefits of reference immutability at low cost.

Javarifier is publicly available for download at `http://pag.csail.mit.edu/javari/javarifier/`.

References

1. A. Aiken, J. S. Foster, J. Kodumal, and T. Terauchi. Checking and inferring local non-aliasing. In *PLDI*, pages 129–140, June 2003.
2. C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. In *OOPSLA*, pages 57–74, Oct. 2006.
3. S. Artzi, A. Kieżun, D. Glasser, and M. D. Ernst. Combined static and dynamic mutability analysis. In *ASE*, pages 104–113, Nov. 2007.
4. S. Artzi, J. Quinonez, A. Kieżun, and M. D. Ernst. A formal definition and evaluation of parameter immutability. *ASE*, 2009.
5. G. Barthe, L. Burdy, J. Charles, B. Grégoire, M. Huisman, J.-L. Lanet, M. Pavlova, and A. Requet. JACK: A tool for validation of security and behaviour of Java applications. In *FMCO*, Oct. 2006.
6. A. Birka and M. D. Ernst. A practical type system and language for reference immutability. In *OOPSLA*, pages 35–49, Oct. 2004.

7. K. D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *PLDI*, pages 57–66, June 1988.
8. K. D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *PLDI*, pages 57–66, June 1988.
9. T. Ekman and G. Hedin. The JastAdd extensible Java compiler. In *OOPSLA*, pages 1–18, Oct. 2007.
10. M. D. Ernst. Annotations on Java types: JSR 308 working document. `http://pag.csail.mit.edu/jsr308/`, Nov. 12, 2007.
11. C. Flanagan and S. N. Freund. Type inference against races. In *Static Analysis Symposium*, pages 116–132, 2004.
12. J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *PLDI*, pages 192–203, June 1999.
13. J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *PLDI*, pages 1–12, June 2002.
14. D. Greenfieldboyce and J. S. Foster. Type qualifier inference for Java. In *OOPSLA*, pages 321–336, Oct. 2007.
15. Y. Liu and A. Milanova. Ownership and immutability inference for UML-based object access control. In *ICSE*, pages 323–332, May 2007.
16. A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *ISSTA*, pages 1–11, July 2002.
17. P. H. Nguyen and J. Xue. Interprocedural side-effect analysis and optimisation in the presence of dynamic class loading. In *ACSC*, pages 9–18, Feb. 2005.
18. N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *CC*, pages 138–152, Apr. 2003.
19. M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst. Pluggable type-checking for custom type qualifiers in Java. Technical Report MIT-CSAIL-TR-2007-047, MIT CSAIL, Sep. 17, 2007.
20. M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *ISSTA*, pages 201–212, July 2008.
21. S. Porat, M. Biberstein, L. Koved, and B. Mendelson. Automatic detection of immutable fields in Java. In *CASCON*, Nov. 2000.
22. J. Quinonez. Inference of reference immutability in Java. Master's thesis, MIT Dept. of EECS, May 2008.
23. C. Razafimahefa. A study of side-effect analyses for Java. Master's thesis, School of Computer Science, McGill University, Montreal, Canada, Dec. 1999.
24. A. Rountev. Precise identification of side-effect-free methods in Java. In *ICSM*, pages 82–91, Sep. 2004.
25. A. Rountev and B. G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *CC*, pages 20–36, Apr. 2001.
26. B. G. Ryder, W. A. Landi, P. A. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM TOPLAS*, 23(2):105–186, Mar. 2001.
27. A. Sălcianu and M. C. Rinard. Purity and side-effect analysis for Java programs. In *VMCAI*, pages 199–215, Jan. 2005.
28. M. S. Tschantz. Javari: Adding reference immutability to Java. Master's thesis, MIT Dept. of EECS, Aug. 2006.
29. M. S. Tschantz and M. D. Ernst. Javari: Adding reference immutability to Java. In *OOPSLA*, pages 211–230, Oct. 2005.