Javarifier: Inference of reference immutability in Java

by

Jaime Quinonez

Submitted to the Department of Electrical Engineering and Computer
Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2008

Author . . . .
Department of Electrical Engineering and Computer Science
May 23, 2008

Certified by . . . .
Michael D. Ernst
Associate Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Javarifier: Inference of reference immutability in Java

by

Jaime Quinonez

Submitted to the Department of Electrical Engineering and Computer Science
on May 23, 2008, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

Javari is an extension of Java that supports reference immutability constraints. Programmers write Javari type qualifiers, such as the `readonly` type qualifier, in their programs, 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 in the development process. 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 both tedious and error-prone; the development process requires an automated solution.

This thesis presents an algorithm for statically 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. A tool, Javarifier, implements this algorithm in order to infer the Javari type qualifiers over a set of class files. Javarifier can also insert these qualifiers into the corresponding source code, if the source code is available.

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.

Thesis Supervisor: Michael D. Ernst
Title: Associate Professor

Acknowledgments

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Javari is an extension of Java with reference immutability type qualifiers [30]. It allows programmers to specify whether references may be used to modify their referents. A `readonly` reference cannot be used to modify its referent, while a `mutable` reference may be used to modify its referent. The Javari type system increases the expressiveness of Java by making certain mutability contracts explicit in a machine-checked format. For example, a library method with a `readonly List` parameter is guaranteed to not add or remove elements from that list. This guarantee allows users of that library to pass their internal data directly—without going through the costly process of making a copy and passing the copy—without fear of the library modifying the data. As another example, a method with a `readonly` return type can safely return a reference to the internal state of the enclosing object without risking the user modifying that state, since the returned reference cannot be used for mutation.

Javari provides reference immutability guarantees over the transitive, abstract state of an object. The abstract state of an object is the value of all its primitive fields and, transitively, the abstract state of all object fields. However, Javari allows explicitly excluding certain fields from the abstract state of an object. Furthermore, the user can exclude either the value or identity (or both) of a field from the abstract state. In our experience with Javari, these features allow enough flexibility to specify the exact abstract state of an object to match the developer's intent.

Figure 1-1 shows an example Java class and the corresponding Javari class.

The Javari typechecker offers the reference immutability guarantee that references are used correctly with respect to mutability. Thus, it detects, or guarantees the absence of, various types of mutability errors. It aids developers both in finding unintended mutability errors in existing Java code and in preventing mutability errors in new Javari code.

In order to port existing Java code to Javari, this thesis presents an algorithm for automatically inferring the reference immutability of Java classes with respect to Javari. The algorithm infers the multiple annotations that are needed for an expressive language like Javari, including `readonly`, specifying containing-object context (`this-mutable`), an extension to wildcards (`? readonly`), and non-generics polymorphism (`polyread`), The algorithm handles the complexities of the Java language, including subtyping, generics, arrays, and unseen code. Furthermore, the algorithm

```
                ___ Java ___                                    ___ Javari ___
class Event {                              class Event {
  Date date;                                 /*this-mutable*/ Date date;
  int hc;                                    int hc;

  Date getDate() {                           polyread Date getDate() polyread {
    return Date;                               return Date;
  }                                          }

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

  static void printDate(Date d) {            static void printDate(readonly Date d) {
    System.out.println(d);                     System.out.println(d);
  }                                          }

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

Figure 1-1: A Java class (left) and the corresponding Javari class (right). Underlines indicate immutability qualifiers. The figure shows default qualifiers in comments for clarity (Javarifier adds nothing in such cases). A qualifier after the parameter list and before the opening curly brace annotates (as for `getDate()` and `hashCode()`) that method's receiver, similar to annotations on other parameters. The qualifiers are explained in Chapter 3.

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. Javarifier, the tool that implements the inference algorithm over class files, has successfully converted programs up to 110 KLOC from Java to Javari. The algorithm is sound and precise. Its results typecheck under the Javari type-checker, and changing any `mutable` qualifiers to `readonly`, without further modifying the qualifiers or the program, causes the results to not typecheck under the Javari typechecker.

All of the tools use the JSR 308 [11] extension to Java annotations, which is backward-compatible and which is planned for inclusion in Java 7[1].

The rest of this thesis is organized as follows. Chapter 2 outlines the need for a tool to automatically convert Java programs to Javari. Chapter 3 provides an overview of the Javari language for reference immutability. Chapter 4 describes the reference

---

[1]To focus on the language and algorithmic details rather than the implementation details, this thesis uses keywords rather than annotations for the Javari type qualifiers. The actual Javarifier tool uses annotations that can be inserted in either the source code or class files of a program.

immutability algorithm in detail, including support for arrays, generics, mutability polymorphism and heuristics for excluding fields from the abstract state of an object. Chapter 5 reports experience using Javarifier. Chapter 6 discusses related work. Finally, Chapter 7 concludes with a summary of contributions.

Chapter 2

Motivation

The reference immutability constraints of Javari 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.

By default, a Java program is a valid Javari program where every reference is `mutable` (or `this-mutable` for fields). Thus, running the Javari typechecker on Java code cannot reveal any mutability errors; the Java code must first be converted to use Javari qualifiers. Writing reference immutability qualifiers to obtain these benefits can be tedious and error-prone; an automatic conversion is necessary. An even more important motivation for immutability inference is the need to annotate the signatures of all used libraries. The Javari typechecker, in order to remain sound, assumes that all methods in an unannotated library may modify their arguments. In particular, passing a `readonly` reference to any library method would be a type error.

Tschantz developed an initial algorithm that soundly calculates reference immutability [29]. 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. Given the Java class on the left side of Figure 1-1, the algorithm produces the Javari class on the right side of Figure 1-1. Note that `hc` is heuristically recommended (see Section 4.5) to be `assignable`. Otherwise, `hashCode` would be inferred, undesirably, to have a `mutable` receiver. This thesis improves upon the algorithm by improving its treatment of mutability polymorphism, method overriding, heuristics for inferring which fields should be excluded from the abstract state of an object, and the algorithm for efficiently solving the algorithm's constraint set.

Javarifier is a scalable tool that implements this algorithm. Javarifier's input is a Java (or partially annotated Javari) program in class file format. Javarifier operates on class files rather than source code 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. Sometimes, the user may want to modify a few qualifiers from the results. For example, a pa-

rameter that is not modified inside a method will be inferred to be `readonly`, but the specification of the method may indicate that the parameter is in fact `mutable`. Also, the user may want to exclude certain fields from the abstract state of an object. To allow this fine-grain tuning of Javarifier results, the user can insert any number of qualifiers in the program and run Javarifier in the presence of these qualifiers (see Section 4.2).

Chapter 3

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 chapter explains Javari's keywords, as listed in Figure 3-1. The Javari language, along with formal type rules, is fully defined elsewhere [30, 29].

3.1   Definition of immutable references

For every Java type `T`, Javari also has the type `readonly T`, where `T` is a subtype of `readonly T`. Figure 3-2 demonstrates how Javari augments the type hierarchy to include both readonly and mutable versions of every Java reference type. References that are not `readonly` can be used to modify their referent and are said to be `mutable`. The default mutability for references with unspecified mutability is `mutable`. This default ensures backwards compatibility, as all Java code is legal Javari code with all references assumed 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.

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 3.6). By default, an object's abstract state is the value of all its primitive fields, and the abstract state of all its object fields. Thus, the abstract state captures all state transitively reachable from a reference.

19

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 reference has a `readonly` upper bound and `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

Figure 3-1: Javari keywords: type qualifiers and field annotations. Default keywords that are not written in a program are shown in comments.
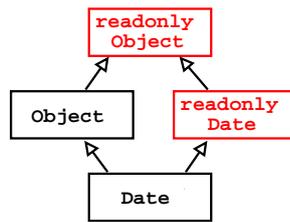
Figure 3-2: A portion of the Javari type hierarchy, which includes readonly and mutable versions of each Java reference type. Arrows connect subtypes to supertypes.

3.2 Containing object context

Javari allows fields to inherit their mutability from the context in which they are accessed. The default mutability for fields, `this-mutable`, specifies that when a field is read, it has the same mutability as the mutability of the reference through which it is read. Given a reference `x`, the expression `x.f` provides a reference to a subset of the abstract state of `x`. Thus, `x.f` should be `mutable` if and only if `x` is `mutable`. Figure 3-3 provides detailed examples of reading `this-mutable` fields.

For type-safety reasons, all `this-mutable` fields must be written to as `mutable`, as shown in Figure 3-4. Otherwise, using a `mutable` reference where a `readonly` reference is expected would allow a user to convert a `readonly` reference to a `mutable` reference, as shown in Figure 3-5.

3.3 Generic types and array types

Javari handles generic type parameters in a natural way to account for the fact that every type 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.

```
/*mutable*/ List</*mutable*/ Date> ld1; // List: may add/remove;
                                         // Date: may mutate
/*mutable*/ List<readonly    Date> ld2; // List: may add/remove
```

```
class Cell {
  Date date;
}

readonly    Cell readonlyCell;
/*mutable*/ Cell mutableCell;

readonly    Date readonlyDate;
/*mutable*/ Date mutableDate;

readonlyDate = readonlyCell.date; // legal
readonlyDate = mutableCell.date;  // legal: mutable reference
                                  //  can be read as readonly

mutableDate = readonlyCell.date;  // illegal: readonlyCell.date is readonly
mutableDate = mutableCell.date;   // legal
```

Figure 3-3: Examples of reading `this-mutable` fields. Each field is read at the same mutability as the reference used to access that field.

```
class Cell {
  Date date;
}

readonly    Cell readonlyCell;
/*mutable*/ Cell mutableCell;

readonly    Date readonlyDate;
/*mutable*/ Date mutableDate;

readonlyCell.date = readonlyDate; // illegal: must write field as mutable
mutableCell.date  = readonlyDate; // illegal: must write field as mutable

readonlyCell.date = mutableDate;  // legal
mutableCell.date  = mutableDate;  // legal
```

Figure 3-4: Examples of writing to `this-mutable` fields. A `this-mutable` field must always be written to as `mutable`.

```
 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 the type `List<? readonly Date>`. The `?` `readonly` wildcard keyword is an extension to Java's wildcard mechanism. The type argument `?` `readonly Date` specifies that `readonly`

```
class Cell {
  Date date;
}

static mutable Date convertToMutable(readonly readonlyDate) {
  /*mutable*/  Cell mutableCell = new Cell();
  readonly     Cell readonlyCell = mutableCell; // legal: mutable Cell
                                                // is a subtype of
                                                // readonly Cell

  readonlyCell.date = readonlyDate;            // illegal: cannot write to
                                               // field as mutable
  mutableDate mutableDate = mutableCell.date;
  return mutableDate;
}
```

Figure 3-5: Example of a type error that would occur if a `this-mutable` field could be written to as `readonly` through a `readonly` reference. Since `readonly` references can alias `mutable` references, writing to a `this-mutable` field through a `readonly` reference may be writing to a field that can be read as `mutable` through another (`mutable`) reference. Therefore, in order to remain sound, Javari conservatively requires that a `this-mutable` field is always written to as `mutable`.

`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`. A List declaration with 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 `Dates` in an input `List` can have a `List<? readonly Date>` parameter. If the parameter were declared as `List<readonly Date>`, then a `List</*mutable*/ Date>` argument could not be passed in, due to the invariance of type arguments. This case also demonstrates why subtyping must be invariant in terms of the mutability of type arguments. Figure 3-6 shows how a `readonly` reference could be converted to a `mutable` reference if a `List</*mutable*/ Date>` could be passed as an argument to a method with a `List<readonly Date>` parameter.

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.

3.4 Mutability polymorphism

The `polyread` qualifier expresses parametric polymorphism over mutability. (The `polyread` qualifier was previously named "`romaybe`" [30, 29].) This qualifier is useful

```
void addToList(readonly Date date, ArrayList<readonly Date> list) {
  list.add(date);
}

/*mutable*/ Date convertToMutable(readonly Date roDate) {
  ArrayList</*mutable*/ Date> mutList = new ArrayList</*mutable*/ Date>();
  addToList(roDate, mutList); // error: passing an ArrayList<mutable Date>
                              // reference into an ArrayList<readonly Date>
                              // formal parameter
  return mutList.get(0);
}
```

Figure 3-6: The `convertToMutable` method attempts to convert a `readonly Date` reference to a `mutable Date` reference by adding it to a list when the list is read as a `List<readonly Date>`, and then extracting it from the same list when the list is read as a `List<mutable Date>`. Javari prohibits this invalid conversion by making subtyping invariant in terms of type argument mutability, so that a `List<mutable Date>` can never be read as a `List<readonly Date>`.

for representing cases where certain method-local references may be either `readonly` or `mutable`, and a method must be written so that both qualifiers could be valid. The `polyread` qualifier may appear on any method-local reference: parameters, receiver, return type, local variables, and fields of method-local classes. The type checker conceptually duplicates any method containing a reference with a `polyread` qualifier. In one version of the method, all instances of `polyread` are replaced by `readonly` (or by `? readonly`, if `polyread` appeared as a type argument to a parameterized class). In the other version, all instances of `polyread` are removed, so the references are `mutable`. Both versions of the method must typecheck in Javari. Clients may use either version.

Figure 3-7 demonstrates the use of `polyread` to properly annotate an accessor method. The `polyread` qualifier permits an accessor method have the same mutability semantics as a field access of a `this-mutable` field. When the accessor `getSeat()` is called on a `mutable` reference, the returned field is read as `mutable`. When the accessor is called on a `readonly` reference, the returned field is read as `readonly`. (This similarity only applies to reading a `this-mutable` field. A method that sets a field cannot have a `polyread` receiver, since in the version of the method with `polyread` replaced with `readonly`, the assignment will modify the state of the object and thus be illegal.) Without this parametric polymorphism, an accessor method could not have the same mutability semantics as a straight forward field access. If the receiver of an accessor is `mutable`, then the accessor can never be called on a `readonly` reference. If the receiver of an accessor is `readonly`, then the accessor can only return a `readonly` reference to the internal field. Since a `polyread` receiver and return type is the natural way to annotate an accessor, `polyread` is critical for precision; in the JDK, `polyread` is needed 70% as often as `readonly` [20].

```
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);
}
```

Figure 3-7: 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.

The `polyread` semantics are not expressible in terms of Java generics. In particular, Javarifier must be able to convert existing Java programs into Javari programs by only adding qualifiers (see Chapter 4). Therefore, Javari must be able to express mutability polymorphism without using generics. The program in Figure 3-7 does not use generics, but is still able to express the mutability polymorphism required in `getSeat()`.

Neither of the `polyread` and `? readonly` qualifiers subsumes the other. The `polyread` qualifier allows a reference to be either `readonly` or `mutable` each time a type rule is checked; it can vary its mutability depending on which type rule is being checked. The `? readonly` qualifier does not have different semantics in different contexts, but rather, adds the additional constraints to the semantics of both mutabilities; some `mutable` references can only be read as `readonly` and some `readonly` references must be written to as `mutable`. The `polyread` and `? readonly` qualifiers have different goals: `polyread` is useful for expressing the precise mutability of a `this-mutable` field, whereas `? readonly` is useful for expressing the basic common features of types that differ only in mutability, which allows, for example, methods that can accept either a `List<readonly Date>` reference or a `List<mutable Date>` reference.

Wherever one of these qualifiers is used, the other cannot be used to accomplish the same goal.

For example, a general accessor method cannot have a `? readonly` qualifier in its return type because that would imply that some part of the returned value can never be mutated. If a method's return type is `mutable List<? readonly Date>`, clients could never mutate any of the `Date`s in the returned list. However, if the returned type of an accessor is `polyread List<polyread Date>`, the `Date`s could be read as

24

`mutable` by a client if the client called the accessor method on a `mutable` reference. `polyread` cannot subsume `? readonly` because it can only be used to for mutability polymorphism over method-local references, since `polyread` requires conceptually duplicating the method for typechecking. Therefore, `polyread` cannot be used on type arguments for fields, whereas `? readonly` is not restricted to methods.

3.5   Method subtyping

In Javari, method subtyping is covariant with respect to the mutability of the return type and contravariant with respect to parameter mutability (including the receiver, the implicit `this` parameter). The type hierarchy in Figure 3-2 expresses the two orthogonal dimensions in which one type may subtype another type; a type can be a subtype in terms of mutability and it can also be a subtype in terms of the Java class type. The Javari rule for method subtyping states that methods must be covariant in terms of the mutability of their return types and contravariant in terms of the mutability of their parameters. This rule is unaffected by, and does not affect, the orthogonal Java rule that method subtyping is covariant in terms of the class types of their return types and contravariant in terms of the class types of their parameters. A previous formalism required, for simplicity, method subtyping to be invariant with regards to the mutabilities of return types and parameters [29]. However, this restriction is not necessary to ensure type safety, so we removed this restriction.

The rule for enforcing that methods must be covariant in terms of the mutability of their return types states that if a method returns a `mutable` reference, all overriding methods in subclasses must return a `mutable` reference. If a method returns a `readonly` reference, an overriding method may return either a `readonly` reference or a `mutable` reference. The rule that methods are covariant in terms of the mutability of their return types is orthogonal to the rule that methods are covariant in terms of the class type of their return types. If a method in a superclass returns a `mutable` `ArrayList`, an overriding method may neither return a `readonly` `ArrayList` (which is not a subtype with respect to mutability) nor return a `mutable` `List` (which is not a subtype with respect to class type).

The rule for enforcing that methods must be contravariant in terms of the mutability of their parameters states that if a method accepts a `readonly` parameter, all overriding methods must also accept a `readonly` parameter. If a method accepts a `mutable` parameter, overriding methods may accept either a `mutable` or `readonly` parameter. In this case, an overriding method accepting a `readonly` parameter demonstrates the sometimes conservative nature of Javari's type qualifiers. The `mutable` type qualifier on the overridden method conservatively specifies that the method may mutate the argument passed in through that parameter. Javari does not require that a method actually mutate a `mutable` parameter. An overriding method that does not use that parameter for mutation may therefore safely declare that parameter to be `readonly`. Since the overridden method guarantees clients only that the method may or may not mutate its parameter, the overriding method honors this guarantee and

```
class A {
  polyread Date get() polyread;
}


class B extends A {
  polyread Date get() polyread;
}


class C extends A {
  mutable  Date get() polyread;
}
```

Figure 3-8: Examples of method subtyping for `polyread` return types (underlined for emphasis). In Javari, methods are covariant with respect to the mutability of return types. For methods with `polyread` references, the typechecker checks each subtyping relation once with each instance of `polyread` in all methods replaced with `mutable`, and once with each instance of `polyread` in all methods replaced with `readonly`. (See Section 3.5.1)

---

then strengthens it by stating that the overriding method will not mutate its parameter. Thus, Javari can allow method subtyping to be contravariant with respect to parameter mutability while still maintaining all its previous guarantees.

3.5.1 Extending the method subtyping rules to incorporate `polyread`

The Javari type hierarchy for mutability from Figure 3-2 does not incorporate the `polyread` qualifier because `polyread` does not specify a mutability type; however, `polyread` may appear as a qualifier on parameters (including the implicit receiver) and return types. Section 3.4 presented the rules for type checking the bodies and usage of methods with some `polyread` references in their signatures. This section extends the type checking rules for method subtyping from Section 3.5 to check the subtyping relations for methods that may contain some `polyread` references.

To check the subtyping relations for a pair of methods that contain `polyread` references, the type checker conceptually creates two versions of each method: a `readonly` version with all instances of `polyread` replaced by `readonly` and a `mutable` version with all instances of `polyread` replaced by `mutable`. Then, the type checker enforces that each version of the methods typecheck using the method subtyping rules from Section 3.5; the type checker checks that the `readonly` versions of the methods obey the subtyping rules and then checks that the `mutable` versions of the methods obey the subtyping rules.

The method subtyping rules enforce that methods are covariant in terms of return type as follows:

- In the `mutable` versions of a method, if the return type is `mutable` in the superclass, it must be `mutable` in the subclass. Figure 3-8 demonstrates two methods that override a method with a `polyread` return type. In the `mutable` version

26

```
class A {
  polyread Date get() polyread;
}

class B extends A {
  polyread Date get() polyread;
}

class C extends A {
  polyread Date get() readonly;
}
```

Figure 3-9: Examples of method subtyping for `polyread` parameters (including the implicit receiver, underlined for emphasis). In Javari, methods are contravariant with respect to the mutability of parameters. For methods with `polyread` references, the typechecker checks each subtyping relation once with each instance of `polyread` in all methods replaced with `mutable`, and once with each instance of `polyread` in all methods replaced with `readonly`. (See Section 3.5.1)

of `A.get()`, the method has a `mutable` return type. In `B.get()`, the `polyread` return type is `mutable` in the `mutable` version of that method. The return type of `C.get()` is `mutable` in both versions of that method.

- Analogously, in the `readonly` versions of a method, if a method has a `readonly` return type in a subclass, it must have a `readonly` return type in the superclass. If the method has a `polyread` return type in the subclass, it must have either a `readonly` or `polyread` return type in the superclass.

The logic is similar for enforcing contravariant parameter mutability. The two versions of a method with `polyread` references are checked as follows:

- In the `mutable` versions of the method, if a parameter (including the implicit receiver) is `mutable` in the superclass, it may be either `mutable` or `readonly` in the subclass. Figure 3-9 demonstrates two methods that override a method with a `polyread` parameter. In the `mutable` version of `A.get()`, it has a `mutable` receiver. In `B.get()`, the receiver is also `mutable` in the `mutable` version of the method. In `C.get()`, the receiver is `readonly` in the `mutable` version of the method, which obeys the contravariant parameter subtyping relation.

- Analogously, in the `readonly` versions of the method, if a parameter is `readonly` in the superclass, it must be `readonly` in the subclass. If the parameter has a `readonly` qualifier in the superclass, it must have a `readonly` qualifier in the subclass. If the parameter has a `polyread` qualifier in the superclass, it must have either a `readonly` or `polyread` qualifier in the subclass.

```
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);
  }
}
```

Figure 3-10: Example class with a field excluded from the abstract state. The `mutable` field annotation on the `log` field excludes the value of that field from the abstract state of a `NetworkRouter`. Therefore, adding elements to `log` inside of the `selectRoute()` method is only a mutation of the concrete state and not a mutation of the abstract state.

3.6 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 a 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 a `mutable` type (its own fields may be reassigned or mutated) even 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 Figure 3-10, the value of the `log` field is excluded from the abstract state of a `NetworkRouter`.

The (implicit, default) `mutable` type qualifier and the (explicit) `mutable` field use the same syntax, but have somewhat different semantics. The `mutable` type qualifier denotes that a reference may be used to modify its referent. The `mutable` field annotation excludes a field from the abstract state of an object by specifying that the field may always be used to modify its referent, even when the field is read through a `readonly` reference. The `mutable` field annotation thus also qualifies the field reference with all the same semantics as the `mutable` type qualifier. Therefore, the `mutable` keyword serves as both a type qualifier and field annotation when it appears on a field, and serves only as a type qualifier on any other reference.

Chapter 4

Type inference for Javari

This chapter presents a flow-insensitive and context-sensitive algorithm to statically infer reference immutability. The algorithm determines which references may be declared with `readonly` or other Javari keywords; other references are left as the default for their reference types (`this-mutable` for fields, `mutable` for all other references). The algorithm is sound: its recommendations type check under Javari's rules. Furthermore, the algorithm is precise: declaring any references in addition to its recommendations as `readonly`—without other modifications to the code—will result in the program not type checking. The algorithm is implemented as a tool, Javarifier, which also includes the toolflow for interacting with the user, as in Section 4.5.

Section 4.1 describes the core inference algorithm. The algorithm extends to handle subtyping (Section 4.1.3); unseen code and pre-existing constraints including assignable and mutable fields (Section 4.2); arrays (Section 4.3.1); Java generics (Section 4.3.2); and mutability polymorphism (Section 4.4). Furthermore, Section 4.5 presents heuristics for excluding fields from the abstract state of an object.

4.1 Core algorithm

Given as input a program, the algorithm 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 4.4 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 "y → x" states that if y is mutable, then x is mutable; again, x and y are constraint variables.

4.1.1 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 by assignments and field dereferences.

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

Figure 4-1: 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 $\mathtt{this_m}$ is the receiver of method $\mathtt{m}$; it is treated as a normal variable, except that any program that attempts to reassign $\mathtt{this_m}$ is malformed.

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

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

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

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

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

Figure 4-2: Constraint generation rules for the statements of Figure 4-1. Auxiliary functions $\mathtt{this}(\mathtt{m})$ and $\mathtt{params}(\mathtt{m})$ return the receiver reference ($\mathtt{this_m}$) and parameters of method $\mathtt{m}$, respectively. $\mathtt{retVal}(\mathtt{m})$ returns $\mathtt{ret_m}$, the constraint variable that represents the reference to $\mathtt{m}$'s return value. $type(\mathtt{x})$ returns the static type of $\mathtt{x}$.

We present constraint generation using a simple three-address core language (Figure 4-1). 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 $\mathtt{this_m}$. Static members are omitted because they do not illustrate any interesting properties. Without loss of generality, all references and methods have globally-unique names. (While the formalism in this thesis is simplified, the Javarifier implementation handles the full Java language.)

Each statement from Figure 4-1 has a constraint generation rule (Figure 4-2):

Assign The assignment of variable $\mathtt{y}$ to $\mathtt{x}$ causes the guarded constraint $\mathtt{x} \to \mathtt{y}$ to be generated because, if $\mathtt{x}$ is a $\mathtt{mutable}$ reference, $\mathtt{y}$ must also be $\mathtt{mutable}$ 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 `this`$_m$, each actual argument is assigned to the method's corresponding formal parameter, and the return value, `ret`$_m$, is assigned to `x`.

Ret The return statement `return x` adds the constraint `ret`$_m$ $\to$ `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` $\to$ `f`, is required because, if `x` is `mutable`, then the field `f` cannot be `readonly`. The second, `x` $\to$ `y`, is needed 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 4.2.)

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` $\to$ `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`.

The constraint set for a program is the union of the constraints generated for each line of the program. Figure 4-3 shows constraints for a sample program.

4.1.2 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, the consequent of the guarded constraint becomes an unguarded constraint, which can then be used to satisfy guards in other guarded constraints. The algorithm finds how many constraints can become unguarded constraints by propagating unguarded constraints throughout the set of guarded constraints. If the guarded constraints are viewed as graph edges, then the core algorithm can be viewed as graph reachability starting at the unguarded constraints. This approach can be implemented with linear time complexity in the number of constraints, and the Javarifier tool does so.

The algorithm is given in Figure 4-4. If the set of guarded constraints $\mathcal{G}$ is implemented as a hash table from each guard to the set of consequents it guards, then looking up all constraints that one unguarded constraint satisfies takes a constant amount of time, in expectation. Since every unguarded constraint is only added to the work-list once, and every guarded constraint is only satisfied by a unique guard, this process has linear time complexity. This makes the tool highly scalable in practice, as the number of constraints is linear in the size of the subject program. Specifically, the

```
class C {                                              class C {
  F f;                        field declaration          readonly F f;
  Y foo(P p) {                                           readonly Y foo(/*mutable*/ P p)
                                                             /*mutable*/ {
    X x = p;                  ASSIGN: {x→p}                  /*mutable*/ X x = p;
    Y y = x.f;                REF: {y → f, y→x}             readonly    Y y = x.f;
    Z z = x.foo(y);           INVK: {this_foo→x,            readonly    Z z = x.foo(y);
                                 p→y, z→ret_foo}
    this.f = y;               SET: {this_foo, f→y}          this.f = y;
    return y;                 RET: {ret_foo→y}              return y;
  }                                                      }
  void doNothing(P p)                                    void doNothing(readonly P p)
  {                           no constraints to generate    readonly {
  }                                                      }
}                                                      }
                             Simplified program constraints:
                               {this_foo, x, p}
```

Figure 4-3: 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, ret_{foo}, and f) can be declared `readonly`, as shown in the Javarifier output on the right side of the figure.

---

number of constraints is linear in the size of the program under analysis as measured in the three-address core language of Figure 4-1[1].

The algorithm always terminates because each constraint variable is added to the work-list $\mathcal{W}$ at most once. Only constraints that are initially in the unguarded constraint set $\mathcal{U}$ or are consequents in guarded constraints from $\mathcal{G}$ whose guard is satisfied are added to the work-list. Therefore, only unguarded constraints are added to $\mathcal{W}$, at which point they are also added to $\mathcal{U}$ if they were not previously in $\mathcal{U}$. After the work-list is initialized, only constraints that are not in $\mathcal{U}$ are added to $\mathcal{W}$. Therefore, each constraint is only added to $\mathcal{W}$ at most once, and since there are a linear number of constraint variables, the algorithm will check each constraint in $\mathcal{W}$ and will terminate in linear time.

The unguarded constraints in the simplified constraint set must be declared `mutable`, except for instance fields, which must be declared `this-mutable`. 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 4.2.2 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 field annotations, constraint solving therefore results in the maximum number of `readonly` references in the program. (Section 4.3.1 expands this argument to the other Javari qualifiers.) Since the

---

[1]A program's representation in this language is roughly the size of its class files.

The core algorithm for solving constraints uses three constraint sets:

- the set of unguarded constraints ($\mathcal{U}$), with constraints of the form a

- the set of guarded constraints ($\mathcal{G}$), with constraints of the form $a \rightarrow b$

- a work-list ($\mathcal{W}$) of unguarded constraints left to propagate

Pseudocode for the constraint solving algorithm:

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

When the algorithm terminates, $\mathcal{U}$ contains all the constraints that can be satisfied from the initial constraint sets.

Figure 4-4: Pseudocode for the constraint solving algorithm.

algorithm always terminates, constraint solving cannot fail. In the worst case, every reference will be `mutable` when the algorithm terminates.

Figure 4-3 shows the result of applying the algorithm to an example program.

4.1.3 Subtyping

Java and Javari allow subtyping polymorphism, which enables multiple implementations of a method to be specified through overriding[2], as explained in Section 3.5. 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 type 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 [29] forced the mutabilities of overriding methods to be identical to the overridden method, but that is not required for correctness.

---

[2]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.*

4.2 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 (see Section 4.5). The heuristics first infer which fields should be excluded from the abstract state of an object and then mark those fields as `assignable` and/or `mutable`. Thus, the constraint generation rules in Figure 4-5 do not need to distinguish between `assignable`/`mutable` fields supplied by the user and those supplied by the heuristics for excluding fields from the abstract state of an object.

Section 4.2.1 discusses pre-existing annotations that specify that a reference is either `readonly` or `mutable`. Section 4.2.2 discusses field annotations that exclude a field from the abstract state of an object.

4.2.1 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. Alternatively, the algorithm can recommend code changes that permit the reference to be declared `readonly`. The algorithm can track which constraints caused the reference to be declared `mutable`, by tracking, for each guarded constraint that was satisfied, which unguarded constraint satisfied it and turned the consequent into an unguarded constraint. The algorithm can then determine the initial constraint that was unguarded at the end of constraint generation. If this constraint was generated by the SET rule (from Figure 4-2), the algorithm can recommend that the field that was written to can be excluded from the abstract state. If this constraint was generated from a user's explicit `mutable` annotation (from Figure 4-5), this conflict can be pointed out to the user.

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

Closed-world versus open-world inference mode

The algorithm has two modes. In *closed-world* mode, or whole-program mode, the algorithm may infer `readonly` type qualifiers for returned/escaped references, such as public method return types and types of public fields. This ability yields more precise results — that is, more `readonly` references. In *open-world* mode, the algorithm marks (i.e., adds an unguarded constraint for) every non-private field and non-private method return value as `mutable`. If an entire package is being analyzed, package-protected (default access) methods and fields need not be so marked; they may be processed as under the closed world assumption. The open-world assumption

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

4.2.2 `assignable` and `mutable` field annotations

The inference algorithm in Section 4.1 does not handle the `assignable` and `mutable` field annotations (discussed in Section 3.6). Without handling these field annotations, Javarifier might infer many references to be `mutable` which the user expects to be `readonly`. In some cases, this type of discrepancy could expose a bug in the implementation; the specification for a reference is that it should not be used to modify its referent, yet the implementation uses that reference to perform an illegal mutation. Automatically finding unintended mutability errors by pointing out the difference between a specification and an implementation is one of the main benefits of Javarifier. (Javarifier points out errors by producing a set of Javari type qualifiers which typecheck in Javari, but which don't match a user's specification. The Javari typechecker points out errors by demonstrating that a set of Javari type qualifiers provided by the user does not conform to Javari.) However, this discrepancy may also arise if the user has marked some fields with either of the `assignable` or `mutable` field annotations, in which case Javarifier should ignore some of the mutations through those references. This section explains how Javarifier extends the core inference algorithm of Section 4.1 to understand the `assignable` and `mutable` field annotations.

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 4-5 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 field that is annotated as `mutable`.

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, $x \rightarrow f$, has already been added to the constraint set via the MUTABLE rule, and (2) the second constraint, $x \rightarrow y$, is eliminated because a `mutable` field is `mutable` regardless of how it is reached.

4.3 Arrays and generics

This section discusses how to infer immutability for arrays and generic classes. The key difficulty is inferring the `? readonly` type, which requires inferring two types (an

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

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

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

Figure 4-5: Modified constraint generation rules for `assignable` and `mutable` fields. The Set and Ref rules of Figure 4-2 are replaced by those of this figure. Mutable is new.

s	::=	...		T,S	::=	A | C	*types*		T,S	::=	C<$\overline{\text{T}}$> | 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*

Figure 4-6: Core language grammar (Figure 4-1) extended for arrays (left). Constraint generation type meta-variables extended for arrays (center) and parametric types (right).

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

4.3.1 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 4-6).

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 3). 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

$$\frac{\text{S[]} \rightarrow \text{T[]} \qquad \text{T} \subset: \text{S}}{\text{T[]} <: \text{S[]}} \qquad \frac{\text{D} \rightarrow \text{C}}{\text{C} <: \text{D}} \qquad \frac{\text{T}_\lhd <: \text{S}_\lhd \qquad \text{S}_\rhd <: \text{T}_\rhd}{\text{T} \subset: \text{S}}$$

Figure 4-7: 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 $\text{T} \subset: \text{S}$, 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 3); therefore, we use the contains relationship as Java's parametric types do.

four `Date` types corresponding to the upper/lower bounds of the upper/lower bounds[3].

We subscript upper bounds with $_\lhd$ and lower bounds with $_\rhd$. 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 4-6. 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.

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 4-7.

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 4-8.

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

---

[3] 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.

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

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

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

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

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

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

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

Figure 4-8: Constraint generation rules extended for arrays. These rules replace the constraint generation rules of Figure 4-2, where the *type*() function was not needed.

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 the corresponding guarded constraints and simplified subtyping or contains constraint (see Figure 4-7). This step is repeated until only guarded and unguarded constraints remain in the constraint set. For example, the statement $\mathtt{x} = \mathtt{y}$, where $\mathtt{x}$ and $\mathtt{y}$ have the types $\mathtt{T[]}$ and $\mathtt{S[]}$, respectively, would generate and reduce constraints as follows:

$$
\begin{aligned}
\mathtt{x} = \mathtt{y} \quad &: \quad \{type(\mathtt{y}) <: type(\mathtt{x})\} \\
&: \quad \{\mathtt{S[]} <: \mathtt{T[]}\} \\
&: \quad \{\mathtt{T[]} \rightarrow \mathtt{S[]}, \; \mathtt{S} \subset: \mathtt{T}\} \\
&: \quad \{\mathtt{T[]} \rightarrow \mathtt{S[]}, \; \mathtt{S_\triangleleft} <: \mathtt{T_\triangleleft}, \; \mathtt{T_\triangleright} <: \mathtt{S_\triangleright}\} \\
&: \quad \{\mathtt{T[]} \rightarrow \mathtt{S[]}, \; \mathtt{T_\triangleleft} \rightarrow \mathtt{S_\triangleleft}, \; \mathtt{S_\triangleright} \rightarrow \mathtt{T_\triangleright}\}
\end{aligned}
$$

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

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

Table 4.1: 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.

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. 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. Table 4.1 illustrates this mapping.

As in Section 4.1.2, given a fixed set of field annotations, the algorithm excludes the maximum number of constraint variables from the unguarded constraint set. After mapping 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` if possible, or `? readonly` otherwise.

4.3.2 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 4-6 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 $bound_\Delta$. $bound_\Delta(\texttt{T})$ returns the declared upper bound of `T` if `T` is a type variable; if `T` is not a type variable, `T` is returned

$$asType_\Delta(\texttt{C<}\overline{\texttt{T}}\texttt{>}, \texttt{C}) = \texttt{C<}\overline{\texttt{T}}\texttt{>}$$

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

Figure 4-9: *asType* returns `C<`$\overline{\texttt{T}}$`>`'s supertype of class `D`.

unchanged. In this formulation, there is a global type environment, Δ, that maps type variables to their declared bounds. *bound* ignores any upper bound ($_\lhd$) or lower bound ($_\rhd$) 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 $asType_\Delta(\texttt{C<}\overline{\texttt{T}}\texttt{>}, \texttt{D})$ function (Figure 4-9) to return `C`'s supertype of class `D`[4]. *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);
```

On the assignment of `f` to `lst`, *asType* converts `f`'s type from `Foo<Integer>` to `List<Date>` with the call: $asType_\Delta(\texttt{Foo<Integer>}, \texttt{List})$. 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 4-10) use subtyping constraints. However, the subtyping rules (shown in Figure 4-11) 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 4-12.

---

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

$$\text{x} = \text{y} : \{type(\text{y}) <: type(\text{x})\} \ (\text{Assign})$$

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

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

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

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

Figure 4-10: Constraint generation rules in the presence of parametric types.

$$\frac{\text{D} \rightarrow \text{C} \qquad \overline{\text{T}''} \subset: \overline{\text{S}'}}{\text{T} <: \text{S} \ \textbf{where} \ bound_\Delta(\text{T}) = \text{C<}\overline{\text{T}'}\text{> and} \ bound_\Delta(\text{S}) = \text{D<}\overline{\text{S}'}\text{> and}}$$
$$asType_\Delta(\text{C<}\overline{\text{T}'}\text{>}, \text{D}) = \text{D<}\overline{\text{T}''}\text{>}$$

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

Figure 4-11: Simplified subtyping rules for mutability in the presence of parametric types.

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 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.

41

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

Figure 4-12: 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`.

4.4   Inferring mutability polymorphism

This section extends the inference algorithm to infer the `polyread` keyword (previously named "`romaybe`" [30]). As described in Section 3.4 and illustrated in Figure 3-7, `polyread` enables more precise and useful immutability annotations to be expressed than if methods could not be polymorphic over mutability.

4.4.1   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`. The case where a parameter/return type is `readonly` in the `mutable` context and `mutable` in the `readonly` context cannot occur due to well-formedness constraints.

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

4.4.2   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 $\{x^{ro} \rightarrow y^{ro}, x^{mut} \rightarrow y^{mut}\}$.

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: $\{x^{?} \rightarrow y^{?}\}$.

The method invocation rule (shown from Figure 4-13) 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, which are constraints with two guards that must be satisfied before the consequent constraint can be added to the unguarded constraint set. (So the double-guarded constraint "$x \rightarrow y \rightarrow z$" states that if both `x` and `y` are `mutable`, then `z` must be `mutable` as well.) For example, consider the code in Figure 3-7, 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 4-13 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 3-7 to the invocation rule of Figure 4-13, note that the ? superscripts would be on the references `s` and `b` local to `lowerSeat()` (or `printSeat()`), whereas the explicit ^{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 ^{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 4-2 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 4-13 only generates the constraint that the return 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`.

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

Figure 4-13: The core algorithm's INVK rule (Figure 4-2) 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 $^{\text{mut}}$ contexts refer to context inside method m.

The extended algorithm for solving constraints uses four constraint sets:

- the set of unguarded constraints ($\mathcal{U}$), with constraints of the form a

- the set of guarded constraints ($\mathcal{G}$), with constraints of the form $a \rightarrow b$

- the set of double-guarded constraints ($\mathcal{D}$), with constraints of the form $a \rightarrow b \rightarrow c$

- a work-list ($\mathcal{W}$) of unguarded constraints left to propagate

Pseudocode for the constraint solving algorithm:

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

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

When the algorithm terminates, $\mathcal{U}$ contains all the constraints that can be satisfied from the initial constraint sets.

Figure 4-14: Pseudocode for the constraint solving algorithm (from Figure 4-4) extended to handle double-guarded constraints.

4.4.3 Constraint solving

This algorithm extends the solving algorithm of Section 4.1.2 in order to account for double-guarded constraints. Figure 4-14 lists the extended algorithm.

The algorithm maintains expected 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 set of constraint variables 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 4-1.

4.4.4 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.[5] Thus, in the example of Figure 3-7, 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`.

```
polyread Date getNewDate() readonly {
  polyread 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 the mutability of the receiver. If

---

[5]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.

the receiver is the only parameter with a `polyread` qualifier, it can be converted to `readonly`.

4.5   Inferring `assignable` and `mutable` field annotations

As explained in Section 3.6, the `assignable` and `mutable` field annotations override Javari's default of transitive immutability through all fields. Annotating one of an object's fields as `assignable` removes that field's identity from the abstract state of that object. Annotating one of an object's fields as `mutable` removes that field's value from the abstract state of that object.

Recall that the core inference algorithm in Section 4.1 infers `readonly` for references that are not used to mutate any part of the abstract state of their referent, where the value and identity of all fields are included in the abstract state of an object. Section 4.2.2 extended the algorithm to incorporate existing `assignable` and `mutable` field annotations so that the algorithm operates over a specific abstract state of an object and not just its concrete state. This process requires the user to manually insert `assignable` and `mutable` field annotations on select fields before running Javarifier.

This section extends the algorithm in two ways to automatically infer `assignable` and `mutable` field annotations:

- First, Section 4.5.1 informally presents a technique that leverages references that the user marked as `readonly` or `polyread` but are used to modify the concrete state of their referents; any concrete state modified through a `readonly` reference or a `polyread` receiver must not be part of the abstract state. Section 4.5.2 formally incorporates this technique into the core algorithm.

- Second, Section 4.5.3 presents heuristics to suggest fields that should be excluded from the abstract state of an object. The heuristics are based on which methods read or write to an object's fields and on other information about fields, such as whether the field is marked with Java's `transient` keyword, which indicates that a field is not part of the persistent state of an object.

4.5.1   Leveraging `readonly` and `polyread` annotations to infer which fields cannot be in the abstract state

Javarifier may infer a reference to be `mutable` or `polyread` which the user expects to be `readonly`, and may infer a reference to be `mutable` which the user expects to be `polyread`. In some cases, this type of discrepancy may be due to an implementation that does not match the user's specification (see Section 4.2.2). In other cases, this discrepancy may indicate a difference between the concrete state of an object and the intended abstract state. This section discusses how to bridge this difference between the concrete and abstract state by using existing annotations to exclude fields from the abstract state of an object. Essentially, the user provides a few hints about the

```
class BankAccount {
  int balance;

  /* Not a part of BankAccount's abstract state. */
  List<String> securityLog;

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

Figure 4-15: A user has annotated the receiver of `balance()` as `readonly`; how-ever, `balance()` mutates the field `securityLog`. Thus, Javarifier correctly infers that `securityLog` should be declared `mutable`.

semantics of the program, and Javarifier uses those hints to make logical deductions about other parts of the program.

If a user annotation conflicts with Javarifier's results — a user believes that a reference should be `readonly` but Javarifier infers that reference to be `mutable` or `polyread` (or a user believes that a reference should be `polyread` but Javarifier infers that reference to be `mutable`) — it is likely that Javarifier observed a mutation only to the concrete state of the object, and not to the abstract state. (If Javarifier knew the intended abstract state of each class, its results for the rest of the program would more closely match the user's expectations.) If the user annotates a few references as `readonly` or `polyread`, Javarifier can use these qualifiers to determine which part of the concrete state must be excluded from the abstract state in order for those references to typecheck as `readonly` or `polyread`.

This ability saves the user from the tedious task of annotating each field that is not part of the abstract state, for two reasons. First, some `readonly` and `polyread` quali-fiers, such as those on method parameters (including the receiver) are inherited from qualifiers on parent classes. If a method in a parent class, such as `Object.toString()`, is annotated with a `readonly` receiver, all subclasses that implement that method in-herit the `readonly` receiver qualifier, as per the type rules for method subtyping in Section 3.5. Second, manually examining the references in a few method signatures only requires reasoning about the specification of a method, whereas determining the correct field annotations for a set of fields requires examining the code in all methods that use those fields.

For example, consider the `balance` method shown in Figure 4-15. In this example, Javarifier infers the receiver of `balance()` to be `mutable` because it is used to mu-tate `this.securityLog`. However, the user has specified that the receiver should be `readonly`. In fact, there is no conflict: the mutation does not affect the object's ab-stract state, only its concrete state. To account for this situation, Javarifier excludes the `securityLog` field from `BankAccount`'s abstract state: Javarifier recommends that

`securityLog` be declared `mutable`.

This technique of using existing `readonly` and `polyread` annotations to infer which fields must be excluded from the abstract state is a two-step process.

- First, the user runs Javarifier, which automatically incorporates all existing `readonly` and `polyread` annotations. These annotations may be inherited from method signatures of superclasses, inserted by the programmer, or inferred by some other tool. Javarifier might infer some of the original `readonly` references to actually be `polyread` or `mutable` and may infer some of the original `polyread` references to actually be `mutable`. These sorts of discrepancies arise because existing annotations might be operating over the abstract state of the object whereas Javarifier inferred mutability over the concrete state of the object, including some fields that are not part of the abstract state. Javarifier will provide a list of fields that can be declared `mutable` or `assignable` and removed from the abstract state in order to make the references in question `readonly`. Additionally, the heuristics in Section 4.5.3 can further suggest which of these fields should be excluded from the abstract state based on how the fields are used.

- Second, the user can select which of these fields should be excluded from the abstract state. Since changing the definition of the abstract state of an object may have unintended consequences for the mutabilities of other references that the user did not manually annotate as `readonly`, the user can select which fields should be excluded from the abstract state and then run Javarifier a final time, yielding the correct results for the entire program.

  Excluding fields from the abstract state cannot cause Javarifier to infer `mutable` for a reference that was previously inferred to be `readonly`. Specifically, if a reference was originally `readonly`, the constraint solving algorithm in Section 4.1.2 did not satisfy enough guards to add the constraint variable corresponding to that reference to the unguarded constraint set. The modified constraint generation rules in Figure 4-5 for when there are existing field annotations generate a subset of the guarded and unguarded constraints from the core constraint generation rules in Figure 4-2. Therefore, if a constraint variable could not be added to the unguarded constraint set when solving the original program constraints (with no field annotations), it cannot be added to the unguarded constraint set when solving the constraints generated with the additional field annotations. Since removing fields from the abstract state cannot cause Javarifier to infer `mutable` for a reference that was previously inferred to be `readonly`, the user does not need to iterate this process multiple times. (Analogously, Javarifier cannot infer `polyread` for a reference that was previously inferred to be `readonly` and it cannot infer `mutable` for a reference that was previously inferred to be `polyread`.)

Section 4.5.2 discusses how to implement this process by extending the constraint generation and constraint solving algorithm from Section 4.1 to leverage user annotations to suggest which fields should be excluded from the abstract state.

When there are multiple field references leading to an assignment, there are multiple ways of declaring fields to be `mutable` or `assignable` that would resolve the conflict. It is preferable to modify a field of the class containing the reference in question than to modify a field of another class. For example, in the case of `BankAccount` (figure 4-15), making `List`'s `add` method `readonly` by declaring `List`'s internal fields to be `mutable` would also allow the receiver of `balance` to be `readonly`. However, changing `List`'s internals is a non-local change and, therefore, less desirable. Our technique lists possible modifications in order of their locality.

4.5.2 Incorporating existing annotations into the constraint set

Javarifier can determine which fields to exclude from the abstract state in order to satisfy other user annotations by extending the constraint set to record which fields are used to generate constraints. In the case of field reference and field assignment, Javarifier records, in the constraint variable of the enclosing class, the field that is being read or manipulated. This field is referred to as the "target field". The target field of a constraint variable is shown after the constraint variable and is separated by a ":$_a$" in the case of a field assignment and a ":$_r$" in the case of a field reference. The modified constraint generation rules are shown in Figure 4-16. The field assignment `x.f = y` produces the constraints

$$\{\texttt{x} :_a \texttt{f}, \texttt{f} \rightarrow \texttt{y}\}$$

The field `f` is the target field of the constraint variable `x` because `f` is the field that was assigned. The field reference `x = y.f` produces the constraints

$$\{\texttt{x} \rightarrow \texttt{f}, \texttt{x} \rightarrow \texttt{y} :_r \texttt{f}\}$$

The field `f` is the target field of the constraint variable `y` because `f` is the field that was referenced.

Constraint variable target fields are recorded during constraint generation as auxiliary data, but do not affect constraint solving; the auxiliary data simply propagates through the constraint solving algorithm along with its corresponding constraint variables.

Javarifier uses the solved constraint set[6] to calculate which fields need to be declared `assignable` or `mutable` by looking up, for each reference in the solved constraint set that the user explicitly marked as `readonly`, the target field that caused that reference to be inserted into the solved constraint set. Specifically, if there is a conflict about reference `x` where a user annotation states that `x` is `readonly` but Javarifier infers that `x` is `mutable`, then Javarifier can infer `x` to be `readonly` by applying the following two rules:

---

[6]We assume that satisfied guarded constraints are maintained in the constraint set throughout the solving process. The same result can be obtained by saving a copy of the unsolved constraint set then unioning the unsolved and solved constraint sets together.

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

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

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

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

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

Figure 4-16: Constraint generation rules extended to record target fields. Modifications to the constraint generation rules from Figure 4-2 are indicated by boxes.

- If the constraint $\mathtt{x} :_a \mathtt{f}$ is in the solved constraint set, then $\mathtt{f}$ should be declared `assignable`.

- If the constraint $\mathtt{x} :_r \mathtt{f}$ is in the solved constraint set, then $\mathtt{f}$ should be declared `mutable`.

Using target fields to infer `assignable` field annotations

This section demonstrates how Javarifier can use target fields for field assignments ($:_a$) to infer that some fields should be declared `assignable`. In Figure 4-17, the receiver of the method `foo`, also denoted using the shorthand $\mathtt{this_{foo}}$, was explicitly marked `readonly` by the user; however, Javarifier determined that $\mathtt{this_{foo}}$ should be declared `mutable`, as seen by the fact that the constraint variable $\mathtt{this_{foo}} :_a \mathtt{tire}$ is in the unguarded constraint set. The fact that the constraint variable $\mathtt{this_{foo}}$ is tagged by the target field $:_a \mathtt{tire}$ demonstrates that `this.tire` was assigned to within the method body of `foo`, thus modifying the concrete state of $\mathtt{this_{foo}}$. This tagged constraint variable directly suggests that $\mathtt{this_{foo}}$ could be readonly if `tire` was declared to be an `assignable` field. If `tire` was assignable, then it could be assigned to through a `readonly` reference. This observation can be generalized to the following rule: If there is a conflict about reference x where a user annotation states that x is `readonly` but Javarifier infers x is `mutable`, then x can be made `readonly` by declaring a field f to be `assignable` if the unguarded constraint $\mathtt{x} :_a \mathtt{f}$ appears in the solved constraint set.

Using target fields to infer `mutable` field annotations

This section demonstrates how Javarifier can use target fields for field references ($:_r$) to infer that some fields should be declared `mutable`. In figure 4-18, $\mathtt{this_{bar}}$

```
class Tire {
  int pressure;
  int radius;
}

class Bicycle {
  Tire tire;

  void foo() readonly {
    Tire t = new Tire();  // none
    this.tire = t;        // this_foo :_a tire, tire → t
  }
}
```

Solved constraint set:

$$\{\texttt{this}_{\texttt{foo}} :_a \texttt{tire}, \texttt{tire} \rightarrow \texttt{t}\}$$

Figure 4-17: Inferring a field to be `assignable`. The receiver of `foo` (`this`$_{\texttt{foo}}$) has been annotated explicitly by the user as `readonly`. Constraints generated for each line of code are shown after the line of code. `this`$_{\texttt{foo}}$ can be `readonly` only if `tire` is declared `assignable`.

was marked by the user to be `readonly`. However, Javarifier inferred `this`$_{\texttt{bar}}$ to be `mutable`, as indicated by the unguarded constraint variable `this`$_{\texttt{bar}}$ $:_r$ `tire` in the unguarded constraint set. The cause of this constraint variable being in the unguarded constraint set is that `tire` is read as `mutable` and must therefore be accessed through a `mutable` reference to `this`$_{\texttt{bar}}$. This cause is recorded by the fact that the `this`$_{\texttt{bar}}$ constraint variable is tagged with $:_r$ `tire`. This target field directly suggests that declaring `tire` to be a `mutable` field allows `this`$_{\texttt{bar}}$ to safely be declared `readonly`. If `tire` is a `mutable` field, then even if `this`$_{\texttt{bar}}$ is `readonly`, the reference `this.tire` is `mutable`. This observation generalizes to the following rule: If there is a conflict about a reference `x` where a user annotation states that `x` is `readonly` but Javarifier infers `x` is `mutable`, then `x` can be made `readonly` by declaring a field `f` to be `mutable` if and only if `x` $:_r$ `f` is in the unguarded constraint set.

The two rules suggested thus far must be applied together simultaneously. That is, if the code from `foo()` in Figure 4-17 and the code from `bar()` in Figure 4-18 was combined into a single method with a user annotated `readonly` receiver, then Javarifier would suggest that `tire` needs to be declared both `assignable` and `mutable`.

Leveraging existing `polyread` annotations to infer which fields cannot be in the abstract state

Javarifier also uses existing `polyread` annotations to infer fields that have to be excluded from the abstract state. The process is similar to existing `readonly` annota-

```
class Tire {
  int pressure;
  int radius;
}

class Bicycle {
  Tire tire;

  void bar() readonly {
    Tire t = this.tire;  // t → this_bar :ᵣ tire
    t.radius = 16;       // t :ₐ radius
  }
}
```

Solved constraint set:

$$\{\mathtt{t} :_a \mathtt{radius}, \mathtt{t}, \mathtt{this_{bar}} :_r \mathtt{tire}, \mathtt{t} \rightarrow \mathtt{this} :_r \mathtt{tire}\}$$

Figure 4-18: Inferring a field to be `mutable`. Constraints generated for each line of code are shown after the line of code. The receiver of `bar` ($\mathtt{this_{bar}}$) has been annotated explicitly by the user to be `readonly`. $\mathtt{this_{bar}}$ can be made `readonly` only if `tire` is declared `mutable`.

tions; for any method with `polyread` references, Javarifier will generate similar constraints (including the target fields) as in the previous explanation when it generates the constraints for that method in a `readonly` context (as explained in Section 4.4). For example, consider the class in Figure 4-19. The `NodeIterator.next()` method inherits `polyread` annotations for its receiver and return type. The version of this method with all `polyread` references replaced by `readonly` can only typecheck if the `current` field is `assignable`. When Javarifier generates the constraints for this method in a `readonly` context, the assignment to the `current` field generates the constraint variable $\mathtt{this_{next}} :_a \mathtt{current}$. Since this constraint conflicts with the `readonly` semantics of the `polyread` annotation on the receiver, Javarifier recommends `current` to be `assignable`.

4.5.3 Heuristics for excluding fields from the abstract state

Caches and other auxiliary data should often have `mutable` and `assignable` field annotations. Javarifier searches for these types of fields and suggests that they should be `assignable` or `mutable` according to the following heuristics:

1. Fields that are private and are only used in a single method. Many methods that perform a computationally expensive operation will cache their results in a private field. This cache field is essentially local to the method and not related

```
class NodeIterator implements Iterator<Node> {
  private assignable Node current;

  public polyread Node next() polyread {
    Node returnValue = current;
    current = current.next;
    return returnValue;
  }
}
```

Figure 4-19: A sample program where an existing `polyread` annotation on `Iterator.next()` propagated to the `next()` method of a class that implements `Iterator`. The `NodeIterator.next()` method must typecheck with all instances of `polyread` replaced with `readonly`, in which case the reassignment of the `current` field is only valid if that field is declared `assignable`.

to the rest of the class; it is more of a local variable that exists across multiple method calls, and the only way to express this in Java is to use a field.

2. A related field usage pattern is private fields that are never referenced as `mutable` in any method. Javarifier will infer these fields to be `readonly` rather than `this-mutable`. If, additionally, these fields are never written to as `readonly` (that is, only `mutable` references are assigned to it), then it would be valid for Javarifier to change the fields from `readonly` to `mutable` and thus exclude these fields from the abstract state. However, retaining the `readonly` qualifier might be a more useful approach, since it eliminates the need to even consider whether mutations to that field conceptually mutate the abstract state of the object, since that field is never mutated. The user ultimately decides which field annotation is most desirable.

3. Fields using the Java `transient` keyword (designed to mark fields that should not be saved during object serialization). Since serialization is intended to capture the state of the object that should persist between concrete instances of that object, fields that are not serialized are likely not part of the abstract state.

4. Fields that are not read in both the `equals()` and `hashCode()` method, if these methods are implemented in a class. `Object.equals()` is specified to check whether two objects can be considered equivalent according to the abstract specification of those objects. The specification of `hashCode()` is that it must provide identical values for objects which are equivalent according to `equals()`. Thus, fields that are in the abstract specification of a class should be read in these methods, and fields that aren't in the abstract specification should not be read in these methods.

Javarifier only infers `private` fields to be `assignable` or `mutable`; non-private fields are already exposed as part of the specification of the class.

53

Javarifier can apply these heuristics to subject programs as a pre-pass before the constraint generation step. Alternatively, Javarifier can apply these heuristics to suggest which fields should be `assignable` or `mutable`, but only apply these field annotations if some user annotations conflict with Javarifier's results and excluding these fields from the abstract state would eliminate these differences. Thus, it does not exclude fields from the abstract state if they would have no effect on other annotations. Javarifier can determine if adding a field annotation to a field will eliminate some differences between the user's annotations and Javarifier's results using the process outlined in Section 4.5.1. That is, it can only exclude a field from the abstract state if it appears as a target field for some reference that conflicts with the user annotation. In this case, Javarifier provides additional justification for excluding a field from the abstract state.

Chapter 5

Evaluation

Javarifier is the implementation of the algorithm described in Chapter 4. Javarifier reads a set of classfiles, determines the mutability of every reference in those classfiles, and inserts the inferred Javari annotations in either 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 5.1) compared Javarifier's output to manually written Javari code that had been type-checked by the Javari type-checker. The second variety (Section 5.2) compared Javarifier to Pidasa, 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.

Table 5.1 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.

5.1 Comparison to manual annotations

Before either the Javarifier implementation or the Javari type checker [9] were complete, another developer manually annotated the JOlden benchmark suite. We were

55

Program	Size		Time	Annotatable references					
	lines	classes		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

Table 5.1: 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.

able to later verify 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.

The programmer omitted 22 `readonly` qualifiers due to simple oversights, such as omitting the `readonly` qualifiers on receivers of `toString()` methods. 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 (from Section 4.5), 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.

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

Table 5.2: Reasons for differences between Javarifier and Pidasa inference results (see Section 5.2). None of the differences indicates an error in Javarifier.

5.2 Comparison to another mutability 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. An earlier analysis of these results appeared in [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 Table 5.2. 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 5-1 gives an example.

`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.

```
class TagNode {
  private List<Attribute> mAttributes;
  public /*mutable*/ List<Attribute> getAttributes() /*mutable*/ {
    return mAttributes;
  }
  public String toHtml() /*mutable*/ {
    String s = "";
    for(Attribute attr : getAttributes()) {
      s += attr.toHtml();
    }
    return s;
  }
}


class LazyTagNode extends TagNode {
  public /*mutable*/ List<Attribute> getAttributes() /*mutable*/ {
    // Actually mutates the abstract state of the object,
    // in accordance to the specification for this class.
  }
}
```

Figure 5-1: 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 because it may change the state of its receiver. The method subtyping rule thus forces `TagNode.getAttributes()` to have a `mutable` receiver. Since `TagNode.toHtml()` calls `getAttributes()`, it must also have a `mutable` receiver, even though not every call to `toHtml()` can cause a mutation.

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.

Chapter 6

Related Work

6.1 Javarifier

The full Javarifier inference algorithm, and experience with a preliminary Javarifier implementation, first appeared as part of Tschantz's thesis [29]. A previous paper, which is a summary of this thesis, built upon that work with an extensive experimental evaluation [23].

Additionally, we have eliminated the requirement in [29] that methods have to be invariant in terms of the mutability of their parameters and return types. Experimental evaluation revealed that this requirement was overly conservative and unnecessarily propagated the `mutable` qualifier on parameters and return types throughout an entire class hierarchy. The method subtyping rules outlined in Section 3.5 that allow methods to be contravariant with respect to parameter mutability and covariant with respect to return type mutability allow Javarifier to infer more `readonly` references while maintaining Javari type safety.

Furthermore, [29] contained a constraint generation rule for inferring the `polyread` qualifier which did not properly handle some cases of invocations on local variables of methods defined in the same class. (It would result in conservative `mutable` qualifiers instead of the more flexible `polyread` qualifiers.) Our `polyread` constraint generation rule in Figure 4-13 does not have this shortcoming.

6.2 JQual

JQual [15] is an immutability inference tool similar to the conceptual framework in [29]. 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.

- 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.

- 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.

- 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.

- 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 [13, 14] 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, 12], inferring non-local aliasing [1], and supporting type qualifiers dependent on flow-sensitivity (like read, write, and open) [14].

6.3 Pidasa

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.

6.4   JPPA

JPPA [28] 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 [25] takes a similar approach but computes a coarser notion of side-effect-free methods rather than per-parameter mutability.

6.5   Other reference usage analyses

Reference immutability is distinct from the related notions of object immutability and of parameter "safety" [28]; 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 [30]. A method parameter is safe if the method never modifies the object passed to the parameter during method invocation. Effect analyses [8, 27, 24, 26, 18, 17] 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. [22] and Liu and Milanova [16] propose immutability inference for fields in Java, the latter in the context of UML, but their definitions differ from ours.

6.6   Type checking

This thesis covers inference of reference immutability according to the Javari language [29]. 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 Javari implementation using the Checkers Framework [21] and did case studies involving 13,000 lines of Javari [20]. The JQual inference system [15] (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 [10], JACK [5], and Polyglot [19].

Chapter 7

Conclusion

This thesis 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, either by identity or by value. Javarifier, the tool that implements the inference algorithm, correctly infers all the advanced features of Javari. 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. Experiments have both confirmed Javarifier's correctness on real programs and have shown that Javarifier scales to handle large programs. 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/`.

The Javari type checker is also publicly available for download at: `http://pag.csail.mit.edu/jsr308/current/checkers-manual.html#javari`.

Bibliography

[1] Alex Aiken, Jeffrey S. Foster, John Kodumal, and Tachio Terauchi. Checking and inferring local non-aliasing. In *PLDI*, pages 129–140, June 2003.

[2] Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. A framework for implementing pluggable type systems. In *OOPSLA*, pages 57–74, Oct. 2006.

[3] Shay Artzi, Adam Kieżun, David Glasser, and Michael D. Ernst. Combined static and dynamic mutability analysis. In *ASE*, Nov. 2007.

[4] Shay Artzi, Jaime Quinonez, Adam Kieżun, and Michael D. Ernst. A formal definition and evaluation of parameter immutability., Dec. 2007. Under review.

[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] Adrian Birka and Michael D. Ernst. A practical type system and language for reference immutability. In *OOPSLA*, pages 35–49, Oct. 2004.

[7] Keith D. Cooper and Ken Kennedy. Interprocedural side-effect analysis in linear time. In *PLDI*, pages 57–66, June 1988.

[8] Keith D. Cooper and Ken Kennedy. Interprocedural side-effect analysis in linear time. In *PLDI*, pages 57–66, June 1988.

[9] Telmo Luis Correa Jr., Jaime Quinonez, and Michael D. Ernst. Tools for enforcing and inferring reference immutability in Java. In *OOPSLA Companion*, pages 866–867, Oct. 2007.

[10] Torbjörn Ekman and Görel Hedin. The JastAdd extensible Java compiler. In *OOPSLA*, pages 1–18, Oct. 2007.

[11] Michael D. Ernst. Annotations on Java types: JSR 308 working document. `http://pag.csail.mit.edu/jsr308/`, Nov. 12, 2007.

[12] Cormac Flanagan and Stephen N. Freund. Type inference against races. In *Static Analysis Symposium*, pages 116–132, 2004.

[13] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *PLDI*, pages 192–203, June 1999.

[14] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *PLDI*, pages 1–12, June 2002.

[15] David Greenfieldboyce and Jeffrey S. Foster. Type qualifier inference for Java. In *OOPSLA*, pages 321–336, Oct. 2007.

[16] Y. Liu and A. Milanova. Ownership and immutability inference for UML-based object access control. In *ICSE*, pages 323–332, May 2007.

[17] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *ISSTA*, pages 1–11, July 2002.

[18] Phung Hua Nguyen and Jingling Xue. Interprocedural side-effect analysis and optimisation in the presence of dynamic class loading. In *ACSC*, pages 9–18, Feb. 2005.

[19] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for Java. In *CC*, pages 138–152, Apr. 2003.

[20] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Pluggable type-checking for custom type qualifiers in Java. Technical Report MIT-CSAIL-TR-2007-047, MIT CSAIL, Sep. 17, 2007.

[21] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for Java. In *ISSTA*, July 2008.

[22] Sara Porat, Marina Biberstein, Larry Koved, and Bilba Mendelson. Automatic detection of immutable fields in Java. In *CASCON*, Nov. 2000.

[23] Jaime Quinonez, Matthew S. Tschantz, and Michael D. Ernst. Inference of reference immutability. In *ECOOP*, July 2008.

[24] Chrislain Razafimahefa. A study of side-effect analyses for Java. Master's thesis, School of Computer Science, McGill University, Montreal, Canada, Dec. 1999.

[25] Atanas Rountev. Precise identification of side-effect-free methods in Java. In *ICSM*, pages 82–91, Sep. 2004.

[26] Atanas Rountev and Barbara G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *CC*, pages 20–36, Apr. 2001.

[27] 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 TOPLAS*, 23(2):105–186, Mar. 2001.

[28] Alexandru Sălcianu and Martin C. Rinard. Purity and side-effect analysis for Java programs. In *VMCAI*, pages 199–215, Jan. 2005.

[29] Matthew S. Tschantz. Javari: Adding reference immutability to Java. Master's thesis, MIT Dept. of EECS, Aug. 2006.

[30] Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to Java. In *OOPSLA*, pages 211–230, Oct. 2005.